Université de Montréal

Deep Neural Networks for Natural Language Processing and its Acceleration

par

Zhouhan Lin

Département d'informatique et operational recherche
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en Informatique

août 2019

© Zhouhan Lin, 2019

Sommaire

Cette thèse par article comprend quatre articles qui contribuent au domaine de l'apprentissage profond, en particulier à l'accélération de l'apprentissage par le biais de réseaux à faible précision et à l'application de réseaux de neurones profonds au traitement du langage naturel.

Dans le premier article, nous étudions un schéma d'entraînement de réseau de neurones qui élimine la plupart des multiplications en virgule flottante. Cette approche consiste à binariser ou à ternariser les poids dans la propagation en avant et à quantifier les états cachés dans la propagation arrière, ce qui convertit les multiplications en changements de signe et en décalages binaires. Les résultats expérimentaux sur des jeux de données de petite à moyenne taille montrent que cette approche produit des performances encore meilleures que l'approche standard de descente de gradient stochastique, ouvrant la voie à un entraînement des réseaux de neurones rapide et efficace au niveau du matériel.

Dans le deuxième article, nous avons proposé un mécanisme structuré d'auto-attention d'enchâssement de phrases qui extrait des représentations interprétables de phrases sous forme matricielle. Nous démontrons des améliorations dans 3 tâches différentes: le profilage de l'auteur, la classification des sentiments et l'implication textuelle. Les résultats expérimentaux montrent que notre modèle génère un gain en performance significatif par rapport aux autres méthodes d'enchâssement de phrases dans les 3 tâches.

Dans le troisième article, nous proposons un modèle hiérarchique avec graphe de calcul dynamique, pour les données séquentielles, qui apprend à construire un arbre lors de la lecture de la séquence. Le modèle apprend à créer des connexions de saut adaptatives, ce qui facilitent l'apprentissage des dépendances à long terme en construisant des cellules récurrentes de manière récursive. L'entraînement du réseau peut être fait soit par entraînement

supervisée en donnant des structures d'arbres dorés, soit par apprentissage par renforcement. Nous proposons des expériences préliminaires dans 3 tâches différentes: une nouvelle tâche d'évaluation de l'expression mathématique (MEE), une tâche bien connue de la logique propositionnelle et des tâches de modélisation du langage. Les résultats expérimentaux montrent le potentiel de l'approche proposée.

Dans le quatrième article, nous proposons une nouvelle méthode d'analyse par circonscription utilisant les réseaux de neurones. Le modèle prédit la structure de l'arbre d'analyse en prédisant un scalaire à valeur réelle, soit la distance syntaxique, pour chaque position de division dans la phrase d'entrée. L'ordre des valeurs relatives de ces distances syntaxiques détermine ensuite la structure de l'arbre d'analyse en spécifiant l'ordre dans lequel les points de division seront sélectionnés, en partitionnant l'entrée de manière récursive et descendante. L'approche proposée obtient une performance compétitive sur le jeu de données Penn Treebank et réalise l'état de l'art sur le jeu de données Chinese Treebank.

Mots-clés: réseaux neuronaux quantifiés, connexion ternaire, connexion binaire, enchâssement de phrase, auto-attention, inférence en langage naturel, analyse des sentiments, graphe de calcul dynamique, réseaux récurrents, analyse syntaxique, analyseur syntaxique, réseaux neuronaux, apprentissage profond, langage naturel traitement, apprentissage automatique

Summary

This thesis by article consists of four articles which contribute to the field of deep learning, specifically in the acceleration of training through low-precision networks, and the application of deep neural networks on natural language processing.

In the first article, we investigate a neural network training scheme that eliminates most of the floating-point multiplications. This approach consists of binarizing or ternarizing the weights in the forward propagation and quantizing the hidden states in the backward propagation, which converts multiplications to sign changes and binary shifts. Experimental results on datasets from small to medium size show that this approach result in even better performance than standard stochastic gradient descent training, paving the way to fast, hardware-friendly training of neural networks.

In the second article, we proposed a structured self-attentive sentence embedding that extracts interpretable sentence representations in matrix form. We demonstrate improvements on 3 different tasks: author profiling, sentiment classification and textual entailment. Experimental results show that our model yields a significant performance gain compared to other sentence embedding methods in all of the 3 tasks.

In the third article, we propose a hierarchical model with dynamical computation graph for sequential data that learns to construct a tree while reading the sequence. The model learns to create adaptive skip-connections that ease the learning of long-term dependencies through constructing recurrent cells in a recursive manner. The training of the network can either be supervised training by giving golden tree structures, or through reinforcement learning. We provide preliminary experiments in 3 different tasks: a novel Math Expression Evaluation (MEE) task, a well-known propositional logic task, and language modelling tasks. Experimental results show the potential of the proposed approach.

In the fourth article, we propose a novel constituency parsing method with neural networks. The model predicts the parse tree structure by predicting a real valued scalar, named syntactic distance, for each split position in the input sentence. The order of the relative values of these syntactic distances then determine the parse tree structure by specifying the order in which the split points will be selected, recursively partitioning the input, in a top-down fashion. Our proposed approach was demonstrated with competitive performance on Penn Treebank dataset, and the state-of-the-art performance on Chinese Treebank dataset.

Keywords: quantized neural networks, ternary connect, binary connect, sentence embedding, self-attention, natural language inference, sentiment analysis, dynamic computational graph, recurrent networks, recursive networks, constituent parsing, syntactic parser, neural networks, deep learning, natural language processing, machine learning

Contents

Sommaire ... iii

Summary ... v

List of tables ... xiii

List of figures ... xv

Acknowledgement ... xix

Chapter 1. Machine Learning Backgrounds ... 1

1.1. Machine Learning ... 1

1.2. Types of Learning ... 2

1.3. Parametric and Non-parametric Models ... 4

1.4. Maximum Likelihood Estimation ... 5

1.5. Generalization and Model Capacity ... 6

1.6. Parameters and Hyperparameters ... 8

1.7. Regularization ... 8

Chapter 2. Neural Networks ... 11

2.1. Convolutional Neural Networks ... 11

2.2. Recurrent Neural Networks ... 13

2.2.1. Vanilla Recurrent Networks ... 14

2.2.2. Long-Short Term Memory units ... 15

2.2.3. Gated Recurrent Unit 17

2.3. Attention Mechanism 18

2.4. Transformer 20

2.5. Low-precision neural networks 24
2.5.1. Multiplication 24
2.5.2. Memory Demand 25

Chapter 3. Natural Language Processing 27

3.1. Encoding words: Neural Language Models 27

3.2. Encoding sentences: Sentence Embeddings 30

3.3. Contextualized Pre-trained Models 33

Chapter 4. Prologue to First Article 35

4.1. Article Details 35

4.2. Context 35

4.3. Contributions 36

4.4. Recent Developments 36

Chapter 5. Neural Networks with Few Multiplications 37

5.1. Introduction 37

5.2. Related work 37

5.3. Binary and ternary connect 38
5.3.1. Binary connect revisited 38
5.3.2. Ternary connect 39

5.4. Quantized back propagation 40

5.5. Experiments 42
5.5.1. General performance 42
5.5.1.1. MNIST 43
5.5.1.2. CIFAR10 44
5.5.1.3. SVHN 44
5.5.2. Convergence 45
5.5.3. The effect of bit clipping 45
5.6. Conclusion and future work 46
Chapter 6. Prologue to Second Article 49
6.1. Article Details 49
6.2. Context 49
6.3. Contributions 50
6.4. Recent Developments 50
Chapter 7. A Structured Self-Attentive Sentence Embedding 51
7.1. Introduction 51
7.2. Approach 52
7.2.1. Model 52
7.2.2. Penalization term 55
7.2.3. Visualization 56
7.3. Related work 57
7.4. Experimental results 58
7.4.1. Author profiling 58
7.4.2. Sentiment analysis 59
7.4.3. Textual entailment 62
7.4.4. Exploratory experiments 63

7.4.4.1. Effect of penalization term 63
7.4.4.2. Effect of multiple vectors 65

7.5. Conclusion and discussion 65

7.6. Pruned MLP for Structured Matrix Sentence Embedding 67

7.7. Detailed Structure of the Model for SNLI Dataset 69

Chapter 8. Prologue to Third Article 73

8.1. Article Details 73

8.2. Context 73

8.3. Contributions 74

8.4. Recent Developments 74

Chapter 9. Learning Hierarchical Structures on the Fly with a Recurrent-Recursive Model for Sequences 75

9.1. Introduction 75

9.2. Model 77

9.3. Experimental Results 78
9.3.1. Math Induction 78
9.3.2. Logical inference 81
9.3.3. Language Modeling 81

9.4. Final Considerations 82

Chapter 10. Prologue to Fourth Article 83

10.1. Article Details 83

10.2. Context 83

10.3. Contributions 84

10.4. Recent Developments 84

Chapter 11. Straight to the Tree: Constituency Parsing with Neural Syntactic Distance 85

11.1. Introduction 85

11.2. Syntactic Distances of a Parse Tree 87

11.3. Learning Syntactic Distances 90

11.3.1. Model Architecture 90

11.3.2. Objective 92

11.4. Experiments 93

11.4.1. Penn Treebank 94

11.4.2. Chinese Treebank 95

11.4.3. Ablation Study 97

11.5. Related Work 97

11.6. Conclusion 98

Chapter 12. Conclusion 101

Bibliography 103

List of tables

5.1 Performances across different datasets 42

5.2 Estimated number of multiplications in MNIST net 44

7.1 Performance Comparison of Different Models on Yelp and Age Dataset 58

7.2 Test Set Performance Compared to other Sentence Encoding Based Methods in SNLI Datset 61

7.3 Performance comparision regarding the penalization term 65

7.4 Model Size Comparison Before and After Pruning 68

9.1 Sample expressions from MEE dataset 79

9.2 Prediction accuracy on MEE dataset. 80

11.1 Results on the PTB dataset WSJ test set, Section 23. LP, LR represents labeled precision and recall respectively. 94

11.2 Test set performance comparison on the CTB dataset 96

11.3 Detailed experimental results on PTB and CTB datasets 96

11.4 Ablation test on the PTB dataset. "w/o top LSTM" is the full model without the top LSTM layer. "w Char LSTM" is the full model with the extra Character-level LSTM layer. "w. embedding" stands for the full model using the pretrained word embeddings. "w. MSE loss" stands for the full model trained with MSE loss. 97

List of figures

1.1 An illustration of underfit and overfit with respect to the change on model capacity. The vertical read line corresponds to the optimal model capacity, which corresponds to the minimum in test error. 7

2.1 A typical structure of convolutional neural networks. The first pile of rectangles (3@128x128) stands for the original image, with its 3 channels standing for RGB channels. The latter piles of rectangles stand for the hidden states in the convolutional network, with the size of the pile being the number of channels, and the size of the rectangles in the piles stand for the shape of the hidden states. The last layer is a dense layer with flat outputs, which represented by a long rectangle strip to the right of the figure. 12

2.2 Structure of a simple single layer, unidirectional recurrent neural network. (a) The folded diagram of the RNN, showing its structure as a directed cyclic graph. The black rectangle stands for a one step time delay. (b) The unfolded diagram of the same RNN, which unfolds the RNN in the time direction by repeatedly drawing the same cell over all time steps. The unfolded structure should always be acyclic. 15

2.3 Structure of bidirectional recurrent neural network. For simplicity we omitted all the notations on weights and hidden states. 16

2.4 Attention mechanism in a machine translation context. The upper unidirectional RNN is a decoder that is trying to infer the next word y_j, and the lower bidirectional RNN is an encoder that encodes source sentence tokens into a sequence of hidden states. 19

2.5 Structure of the Transformer model. For simplicity we merge all the individual tokens in a sequence and represent them as a whole, which is notated as X and Y

in the figure. Within each of the components in the encoder and decoder block, there is a layer normalization step included. Please refer to the text for details. . 21

5.1 Test set error rate at each epoch for ordinary back propagation, binary connect, binary connect with quantized back propagation, and ternary connect with quantized back propagation. Vertical axis is represented in logarithmic scale. 45

5.2 Model performance as a function of the maximum bit shifts allowed in quantized back propagation. The dark blue line indicates mean error rate over 10 independent runs, while light blue lines indicate their corresponding maximum and minimum error rates. 46

5.3 Histogram of representations at each layer while training a fully connected network for MNIST. The figure represents a snap-shot in the middle of training. Each subfigure, from bottom up, represents the histogram of hidden states from the first layer to the last layer. The horizontal axes stand for the exponent of the layers' representations, i.e., $\log_2 \mathbf{x}$. 47

7.1 A sample model structure showing the sentence embedding model combined with a fully connected and softmax layer for sentiment analysis (a). The sentence embedding M is computed as multiple weighted sums of hidden states from a bidirectional LSTM $(\mathbf{h_1}, ..., \mathbf{h_n})$, where the summation weights $(A_{i1}, ..., A_{in})$ are computed in a way illustrated in (b). Blue colored shapes stand for hidden representations, and red colored shapes stand for weights, annotations, or input/output. 53

7.2 Heatmap of Yelp reviews with the two extreme score. 60

7.3 Heat maps for 2 models trained on Age dataset. The left column is trained without the penalization term, and the right column is trained with 1.0 penalization. (a) and (b) shows detailed attentions taken by 6 out of 30 rows of the matrix embedding, while (c) and (d) shows the overall attention by summing up all 30 attention weight vectors. 64

7.4 Attention of sentence embedding on 3 different Yelp reviews. The left one is trained without penalization, and the right one is trained with 1.0 penalization. . 64

7.5 Effect of the number of rows (r) in matrix sentence embedding. The vertical axes indicates test set accuracy and the horizontal axes indicates training epochs. Numbers in the legends stand for the corresponding values of r. (a) is conducted in Age dataset and (b) is conducted in SNLI dataset. 66

7.6 Hidden layer with pruned weight connections. M is the matrix sentence embedding, M^v and M^h are the structured hidden representation computed by pruned weights. 68

7.7 Model structure used for textual entailment task. 70

9.1 (a) - (c) are the 3 different cells. (d) is a sample model structure resulted from a sequence of decisions. "R", "S" and "M" stand for recurrent cell, split cell, and merge cell, respectively. Note that the "S" and "M" node can take inputs in datasets where splitting and merging signals are part of the sequence. (e) is the tree inferred from (d). 75

9.2 Test accuracy of the models, trained on sequences of length ≤ 6 in logic data. The horizontal axis indicates the length of the sequence, and the vertical axis indicates the accuracy of model's performance on the corresponding test set. 80

11.1 An example of how syntactic distances ($d1$ and $d2$) describe the structure of a parse tree: consecutive words with larger predicted distance are split earlier than those with smaller distances, in a process akin to divisive clustering. 86

11.2 Inferring the parse tree with Algorithm 3 given distances, constituent labels, and POS tags. Starting with the full sentence, we pick split point 1 (as it is assigned to the larger distance) and assign label S to span (0,5). The left child span (0,1) is assigned with a tag PRP and a label NP, which produces an unary node and a terminal node. The right child span (1,5) is assigned the label ∅, coming from implicit binarization, which indicates that the span is not a real constituent and

all of its children are instead direct children of its parent. For the span (1,5), the split point 4 is selected. The recursion of splitting and labeling continues until the process reaches a terminal node. 88

11.3 The overall visualization of our model. Circles represent hidden states, triangles represent convolution layers, block arrows represent feed-forward layers, arrows represent recurrent connections. The bottom part of the model predicts unary labels for each input word. The ∅ is treated as a special label together with other labels. The top part of the model predicts the syntactic distances and the constituent labels. The inputs of model are the word embeddings concatenated with the POS tag embeddings. The tags are given by an external Part-Of-Speech tagger. 91

Acknowledgement

There are a lot of people who helped me in various aspects of the research I conducted, as well as being supportive during the past five years.

I'd especially like to thank my supervisor, Yoshua Bengio, for taking me onto the boat of deep learning, and for his supervision on my research. His insights towards various research fields have always shed light on my vision towards the problems in those fields. His perseverance and enthusiasm have encouraged me as well. I'd also like to thank him for always keeping the Mila lab an open, inclusive, and collaborative research lab. It provides such a great environment that allows students, professors, and even external collaborators to freely explore various ideas out of their curiosity. I'd also like to thank Roland Memisevic, who was also my supervisor and worked with me in my early days of Ph.D. study.

I'd also like to thank all of my collaborators without whom this thesis won't be possible. Specifically, I'd like to thank Aaron Courville for his discussions and advise. Alessandro Sordoni for hosting me at Microsoft Research Montreal, and provides precious feedback and critiques on my proposed research. Yikang Shen for the close collaboration, and the discussions and debates we had. Athul Paul Jacob for the work we've done together at Microsoft Research Montreal. Matthieu Courbariaux for the collaborations on low precision networks. Minwei Feng, Mo Yu, Bowen Zhou and Bing Xiang for the work we've done at IBM. Also, I'd like to thank Samuel Lavoie for translating the summary in this thesis into French for me.

There are several people who were supportive to my life outside of research. I'd like to thank Kyunghyun Cho for always organizing those beer parties when he was here, which helped me get involved into the lab more quickly. Athul Paul Jacob for always keeping his pranks happening in its surprising way. Shawn Tan for his memes, jokes, and his extended

vocabulary in multiple languages that make us happy, even when facing tightest deadlines. Cheng Li and Guoxin Gu for the various kinds of sports we had played together.

There are also a lot of people I want to thank, who helped me in various ways. These include Jie Fu, Jian Tang, Jae Hyun Lim, Min Lin, Ziwei He, Sarath Chandar, Dmitriy Serdyuk, Sandeep Subramanian, Vincent Michalski, Julian Vlad Serban, Dendi Suhubdy, Chinnadhurai Sankar, Chin-Wei Huang, Mathieu Germain, Saizheng Zhang, Dong-Hyun Lee, Yuhuai Wu, Joachim Ott, Ying Zhang, Adam Trischler, Frederic Bastien, and Guillaume Alain.

Lastly, I would like to thank my mother, my aunt, and my grand parents for their support during my academic career.

Chapter 1

Machine Learning Backgrounds

This thesis focuses on several topics around deep learning, which is a machine learning approach based on neural networks. We'll provide some background knowledge in machine learning and neural networks in this chapter, and in subsequent chapters we are going to dive deep into more specific topics that are related to the work being introduced in the thesis. The remainder of the thesis presents the articles. For the articles to be presented, we will first introduce low precision networks. Then we will introduce an early version of self attention which is used in sentiment analysis and natural language inference. Finally we will introduce a neural parser where a neural network is used to learn grammar trees out from its hidden states.

In this chapter I will give a brief introduction to some of the important basic aspects and concepts of machine learning that will be used and studied in the subsequent chapters. As supervised learning will be studied a lot in subsequent articles, I will use this learning scheme as a main example while introducing various machine learning concepts. More details on other learning schemes and the concepts being introduced here can be found in [**13**] and [**51**].

1.1. Machine Learning

Machine learning is a sub-field in computer science that seeks to enable computer systems to learn knowledge through data, observations, and interactions with the world. The acquired knowledge should be general enough so that it also allows the computer systems to correctly generalize to new observations or even new settings.

Seen as a subset of a much broader field named artificial intelligence, machine learning is of important merits in both theoretical and practical aspects. Theoretically it has developed

into several branches in theoretical computer science, such as computational learning theory and statistical learning theory. Practically, benefiting from the increase in data production and computational power, the recent resurgence of the neural network approach has revolutionized, and become the dominant technical approach, in a wide variety of application fields, such as computer vision [**93**], speech recognition [**65**], and natural language processing [**5**]. These achievements in the past several decades has made machine learning an important part of computer science.

Early in the 1950s Alan Turing has studied in his paper the question of "Can machines do what we (as thinking entities) can do?" [**160**]. In his proposal he discussed and exposed various characteristics that an intelligent machine should possess, and some implications in constructing these machines. Later in 1959, the phrase "machine learning" was invented by Arthur Samuel [**142**]. In the following 3 decades after that, the field of machine learning and artificial intelligence has branched out into several different sub-fields before the resurgence of neural networks approach has reunited them under the name of artificial intelligence in this recent decade. In the 1980s the statistical approach had developed itself into fields under the names of pattern recognition and information retrieval. And due to the dominance of those approaches in those years, the term "artificial intelligence" was more associated with those approaches based on rule-based systems such as expert systems. Although almost abandoned by the mainstream of artificial intelligence community since the publication of the book "Perceptrons" [**120**], persistent researchers such as Hopfield, Rumelhart, Hinton, Bengio, and LeCun were still conducting research under the name of "connectionism." Since 2006 [**10, 93**], with the company of a series of breakthroughs in several important fields [**93, 65, 5**], this neural network approach has become the dominant method in the machine learning community, and changed the field of artificial intelligence to lean more on machine learning.

1.2. Types of Learning

The academic study of machine learning could be clustered into several different learning schemes, and the most widely accepted taxonomy consists of three major learning types, i.e., supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning is a learning type which includes learning an input-to-output mapping with a dataset containing labeled examples of the mapping. The desired output of the model is required to be provided during training. It is worth noting that the crucial property of success of this learning process is generalization. During training, the model learns a mapping function that is able to infer correctly in the training data, while during test time the function is expected to perform well on unseen, test data. Generalization cares about the model's predictive performance on the unseen test set, rather than the training set.

According to the type of the label, supervised learning can further be split into two types. With the provided labels belonging to a finite set of discrete labels, we usually refer to the learning task as classification. On the other hand, with the labels being continuous numbers, such as stock price, the learning task is called regression instead. Supervised learning could get into some more sophisticated learning settings beyond classification and regression. For example, structured prediction [**6**] involves predicting structured objects such as trees, rather than scalar discrete or real values. Curriculum learning [**11**] feeds gradually more difficult examples to speed up training and get better generalization.

Unsupervised learning, on the other hand, doesn't require the training data to provide labeled examples. It is particularly interesting when the collection of ground truth labels is very expensive. Unlike supervised learning whose goal is pretty clear, which is to learn the mapping from data to labels, the goal of unsupervised learning is more nuanced. In some cases it is to discover some meaningful structure from the data (such as clustering or manifold learning), while in some other cases it is to learn the data distribution, either implicitly (such as generative adversarial networks) or explicitly (such as Gaussian mixture models).

There are many different learning schemes in unsupervised learning as well. The most common scheme is called clustering, where the algorithm learns to group the examples in a dataset into several discrete groups. Data samples that fall into a same group are expected to be more similar than those falling into different groups. Density estimation is also considered as unsupervised learning. In density estimation, we train a model to approximate the underlying probability function from which the data samples are drawn. More broadly there are a wide set of algorithms that fall into this category, such as generative adversarial

networks [**63**], self-organized maps [**91**], non-linear independent components estimation [**48**], variational autoencoders [**88**], etc.

Reinforcement learning is another type of learning which concerns how a learner (usually called agent in this setting) could optimize its reactions (called actions) through its interaction with an environment to maximize some predefined reward. Reinforcement learning is quite different from the previous two types of learning schemes since it introduces interaction which makes the environment states correlated with a series of previous actions, while in the former two cases samples in the dataset are mostly under the i.i.d. assumption (independent and identically distributed). Apart from the focuses that we introduced for supervised and unsupervised learning, reinforcement learning has a unique focus in finding a balance between exploration and exploitation, which corresponds to exploring the unknown states in the environment, and utilizing the current knowledge about the environment, respectively.

1.3. Parametric and Non-parametric Models

From another viewpoint, the models used to do the learning task in the aforementioned various schemes can generally be split into two types as parametric and non-parametric models. The main difference between these two groups of models are the way they model the data.

For parametric models, one defines a parametric family of functions that are controlled by a fixed number of parameters θ. For example, in the discriminative supervised learning case, a parametric model defines a set of functions $\mathscr{F} = \{f_\theta : x \to y\}$, where for each set of parameter $\theta \in \Theta$ it corresponds to a function f_θ that maps input $x \in \mathbb{D}$ to the corresponding label $y \in \mathbb{Y}$. The form of the function f_θ could vary a lot from simple linear regression models to complex multi-layer neural networks with millions of parameters. Despite the great variations the parameter set θ could provide, the family of functions are pre-defined by the form of f_θ. The capacity of a given parametric model is bounded by the function family f. Thus, the form of the parametric family of models is usually called the model's *inductive bias.*

On the contrary, non-parametric models assume that the data distribution cannot be defined in terms of a finite set of parameters. The number of parameters is not fixed, and usually grows with the dataset size. For example, some classical non-parametric models

such as K nearest neighbor classifiers and decision trees, they directly memorize the entire dataset. This will make very high demand in computation and memory consumption when the model is applied to a modern-sized dataset.

Neural networks usually involves a fixed set of parameters at inference time. But since the hyperparameters can be tuned through a validation set (as we will describe in the following section), the model size could effectively change according to the dataset. In regard of that, neural networks cannot simply be classified as parametric models, and it should be considered as a hybrid approach between parametric and non-parametric method.

1.4. Maximum Likelihood Estimation

Maximum likelihood estimation (MLE) plays an important role in various types of learning, such as classification, density estimation, etc. It is used to estimate the parameters in a parametric model by maximizing the probability of observed data. From a Bayesian inference point of view, maximum likelihood estimation corresponds to a special case of maximum a posteriori estimation in which a uniform prior is assumed.

We will take the supervised learning case as example to elaborate a conditional version of this method. For a general non-conditional version of the MLE method, please refer to **[13]** for details. In a supervised setting, we have access to a set D of example pairs (x, y) that are sampled under an i.i.d. (independently and identically distributed) assumption. Our goal is to estimate the optimal parameter set θ^* that maximizes the conditional probability $p(y|x)$:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \prod_{x,y \in D} p(y|x;\theta). \tag{1.4.1}$$

Since the samples are i.i.d., and the computation leads to a lot of overflow/underflow problems due to limited numerical precision in practical cases, we usually use its logarithm form instead:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \sum_{x,y \in D} \log p(y|x;\theta) \tag{1.4.2}$$

We can also view the maximum likelihood estimation as minimizing the Kullback-Leibler (KL) divergence between the data distribution and model distribution. The conditional distribution $q(y|x)$ is deemed as one-hot with all probability mass concentrating on the ground truth label, while the model output $p(y|x;\theta)$ is a distribution over all possible labels.

Thus, the KL divergence between the model estimated distribution and the data distribution becomes:

$$D_{KL}(q||p) = \mathbb{E}_{(x,y)\sim D_{data}}[\log q(y|x) - \log p(y|x;\theta)] \tag{1.4.3}$$

Here we use D_{data} to represent the unknown data generating distribution. Since the conditional log likelihood $\log q(y|x)$ is a constant which doesn't depend on θ, we can safely remove this term. Also since the D_{data} remains unknown, and we can only have access to a set of examples draw in an i.i.d. fashion to represent it, we substitute the expectation term with a summation over all drawn examples. Thus we get the negative log likelihood (NLL) loss:

$$L_{NLL} = -\sum_{x,y\in D} [\log p(y|x;\theta)] \tag{1.4.4}$$

From this we can see that minimizing the NLL loss corresponds to maximizing the conditional likelihood of the data. In the process of optimization for models such as deep neural networks, modern optimizers are conventionally minimizing a loss function, and this loss is widely used in almost all the supervised learning tasks.

1.5. Generalization and Model Capacity

In the last subsection we introduced an optimization objective that could be used in optimization to find the best parameter set. However, what makes machine learning different from optimization is that it cares about generalization, as mentioned in Section 1.2. In a general setting, we usually have two separate sets of samples, the training set D_{train} and the test set D_{test}. The model only has access to the training set during training by minimizing a training loss such as the negative log likelihood, while its performance will be evaluated on the unseen test set. We want both the training error and test error to be low in order to show that the model generalizes well to new, unseen examples. Otherwise, one can easily think of a model that "cheats" the learning scheme. For example, a model could do pretty well in predicting training set examples by just memorizing the training samples and predict those memorized samples with a look-up table during training. Thus it easily gets zero training error. However, it is not actually learning anything interesting, and will not be useful since

it has no capability in correctly predicting new samples that are sampled from the same distribution.

In real cases, machine learning models don't get into so extreme case as explicitly memorizing the training set, but the problem of generalization still exists. Models usually exhibit a better performance in their training set than test set. Figure 1.1 shows a model's typical performance on training and test sets as the model size varies.

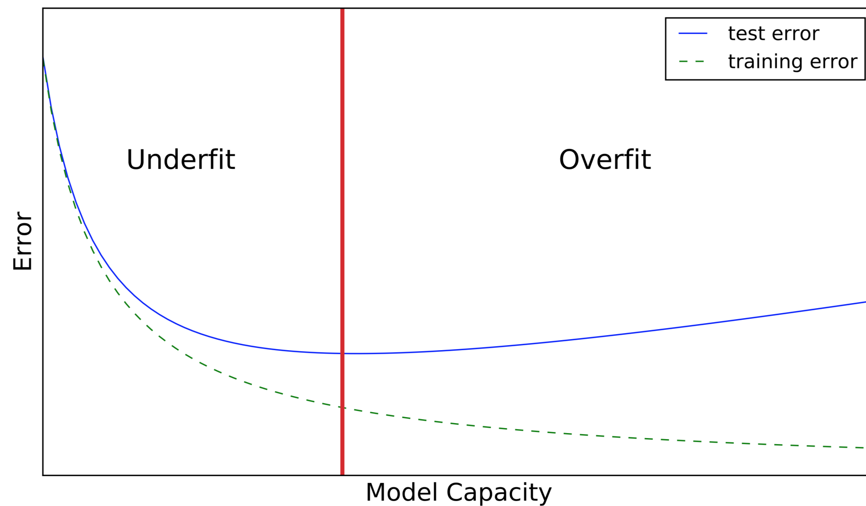


Fig. 1.1. An illustration of underfit and overfit with respect to the change on model capacity. The vertical read line corresponds to the optimal model capacity, which corresponds to the minimum in test error.

Underfitting refers to the situation that the model's capacity is not even large enough to satisfyingly model the training set. This corresponds to a high training error as well as a high test error. On the contrary, when the model is in an overfitting regime, the model has managed to get a low training error while the test error is still high. Typically we refer to the gap between the training error and test error as the *generalization gap*.

One of the key element in the transition between underfitting and overfitting is the model's capacity. Model capacity depicts the group of function it can fit onto. The larger that group of function is, the larger the capacity. The model capacity could be changed in various ways. For example, increasing a feed-forward neural network's hidden layer sizes will increase the model capacity; Adding quadratic terms in a linear regression model will also increase the model capacity.

1.6. Parameters and Hyperparameters

For most of the parametric machine learning models, there are usually two types of variables that is going to be decided during the training phase, which are called parameters and hyperparameters, respectively. The term parameter typically refers to the variables that are adapted during a single training process. Typical parameters are, for example, weights and biases in a neural network. On the other hand, hyperparameters refer to variables that have to be specified before a single training process. For example, in a neural network setting, the number of layers, the hidden layer sizes, and the softmax temperature, etc. are considered as hyperparameters.

1.7. Regularization

In subsection 1.5 we introduced the relation between the model capacity and its generalization gap. We stated that a model with the right capacity performs the best on the test set. In practice, as the practical problem could get very complicated, such as image recognition or text classification, we almost will never know the true data generating process. Moreover, we don't even have a clue about if our assumed model family includes the true data generating process. Thus in practice, almost all models are assuming a wrong family of functions to approximate the real data generation process. The mismatch between the model's assumed family of function and the true data generation process is called *bias*. As a result, instead of just using the model with the right capacity, we almost always prefer a model with larger capacity, which provides more *variance* to the model, that could narrow the gap between the model assumed function and the true data generation process.

Larger models comes with a lower training error and a larger generalization gap. To compensate for the drop on test error, a bunch of methods have been proposed that aim at reducing this generalization gap, thus reducing the test error. These methods are referred to as *regularization* techniques.

Adding a regularization term into the learning objective is one of the standard ways of regularizing a machine learning model. A regularization term is usually an extra term added to the original loss function. Again let's take the loss in Eq. 1.4.4 as example, The final loss with the extra regularization term becomes

$$L = L_{NLL} + \lambda\Omega(\theta) \tag{1.7.1}$$

where λ is a positive real number that serves as a hyperparameter, which remains fixed during the training process. A common choice of the form of $\Omega(\cdot)$ is some norm of the parameter, e.g.,

$$\Omega(\theta) = \|\theta\|_p^2. \tag{1.7.2}$$

For example, if $p = 2$, then we call this term the L^2 regularization term. It is also referred to as *weight decay* in deep neural networks since it constrains the parameters to have a smaller norm. If $p = 1$, then the norm represents the sum of the absolute values of all the parameters. This is called L^1 regularization. An L^1 regularization will constrain the model parameters to be sparse [**62**].

Apart from adding an extra term in the loss function, regularization methods could take other forms. *Early stopping* is such an example. Early stopping prevents overfitting by stopping an iterative training process at an earlier stage. It keeps monitoring the model's performance on a hold-out validation set, and stops the training algorithm when the loss on the validation set starts increasing.

In deep learning, *Dropout* is yet another popular regularization technique. As its name indicates, dropout randomly sets to zero hidden activation at each neuron during the forward propagation. [**151**] provides more details about dropout.

Apart from the aforementioned 3 approaches, there are various other approaches in regularization. All these different methods aims at reducing the test error, sometimes even at the cost of increasing the training error. Regularization is playing an important role in the learning process of machine learning models.

Chapter 2

Neural Networks

In this chapter we are going to introduce some important building bricks of modern neural architectures. These components forms the basis of the models we are going to introduce in the following articles.

2.1. Convolutional Neural Networks

The convolutional neural network (CNN) [**98**] was proposed in the 1990s to analyze visual imagery. It is biologically inspired by research from Hubel and Wiesel's pioneering work, which revealed the connectivity pattern of neurons in cats' visual cortex [**79**]. The CNN has been one of the core neural network architectures in the resurgence in deep learning due to its success in computer vision. After the recent several years of development, the CNN has emerged into a series of variants, including Residual Networks (ResNet) [**69**], densely connected networks (DenseNet) [**77**], dilated convolutional networks [**182**], etc. The application of CNNs has also expanded beyond the field of computer vision to various other fields such as natural language processing.

In this section we are going to introduce the structure of a classical CNN in an image recognition scenario. It consists of three different types of layers: the convolution layer, pooling layer, and fully connected layer. The convolutional and pooling layers correspond to simple and complex cells in the visual cortex, while the fully connected layers make classification decisions by looking at the hidden representation learned by the last convolutional layer. Figure 2.1 shows the overall structure of a simple CNN.

The convolutional layers (the first and third layer in Figure 2.1) consist of a set of local receptive filters. Each of the filters takes as input hidden states within a rectangular receptive

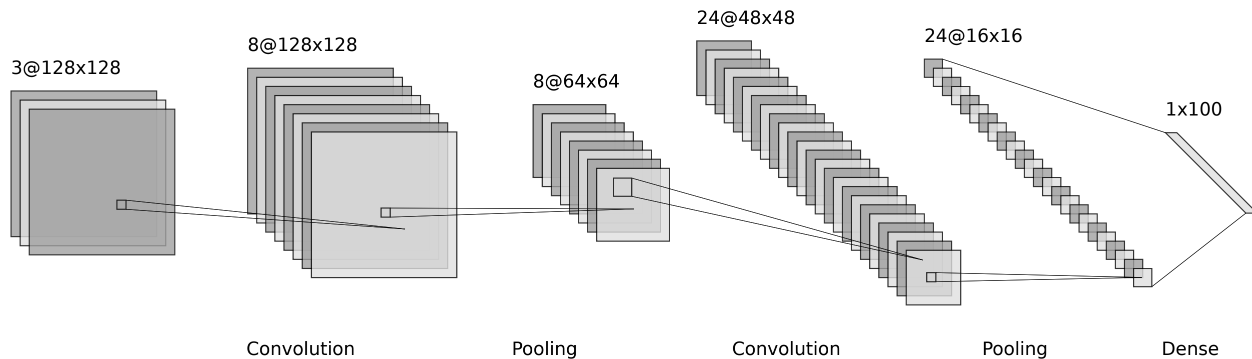


Fig. 2.1. A typical structure of convolutional neural networks. The first pile of rectangles (3@128x128) stands for the original image, with its 3 channels standing for RGB channels. The latter piles of rectangles stand for the hidden states in the convolutional network, with the size of the pile being the number of channels, and the size of the rectangles in the piles stand for the shape of the hidden states. The last layer is a dense layer with flat outputs, which represented by a long rectangle strip to the right of the figure.

field from its previous layer, multiply them by the filter's weights, sum up the products, pass them through an activation function, and output a single value as the hidden activation. The resulting output thus has a 2-D structure, and is called a *feature map*. These feature maps correspond to the big rectangles in the piles of hidden states in the figure. For each of the convolutional layers, there are usually multiple filters, as a result each layer will output multiple feature maps. The number of feature maps in a layer is also called the number of *channels* in the literature.

Figure 2.1 shows a typical 2-D convolution applied on image recognition, while there are other variants of convolution for other types of data. For example, in natural language processing (NLP), 1-D convolution over the sequence of words is the most popular type of convolution. In this case, the receptive filters are convolving on the representation of words sequentially on the input sequence. In some other computer vision tasks [**83, 158**], 3-D convolutions are needed, in which case the convolution happens not only on the width and height dimensions of the input image, but also on a third spatial or temporal dimension.

The pooling layers (the second and fourth layer in Figure 2.1) function like complex cells in the convolutional network architecture. They down-sample the resulting feature maps by a pooling operation. There are a lot of ways of doing the pooling operation, which include

mean-pooling, sum-pooling, and max-pooling, etc. Their meanings are straight forward from their names. Among all of them, the most popular and widely used is max-pooling. It takes the largest value among its receptive field as output and discards all other values. The receptive field of a pooling filter is also a small rectangular block, which is referred to as *pooling size*. In addition to the pooling size, the *stride* of pooling is another important hyperparameter. It indicates that, while the receptive field of pooling is scanning over the feature map, the number of pixels each step of pooling operation moves. If the stride is equal to the pooling size, then the pooling receptive fields are not overlapped with each other, and each activation gets pooled exactly once. Otherwise, if the strides are smaller than the pooling size, then the receptive fields are partially overlapping, thus some of the activation get pooled in multiple pooling receptive fields.

The last component in a CNN is a series of fully connected layers (the fifth layer in Figure 2.1). Fully connected layers take as input the last layer of feature maps, and map it to a flat hidden activation with a simple matrix multiplication. The last layer has softmax as its activation function, thus it outputs a set of probabilities indicating to which class an input image belongs. The probability can be used to match the one-hot target values with the negative log likelihood loss indicated in Section 1.4.

2.2. Recurrent Neural Networks

Recurrent neural networks (RNNs) are a group of models that have shown great promise in various sequential data processing tasks, including natural language processing. The main difference that distinguishes it from feed forward neural network is that its output not only depends on the input, but also depends on the internal hidden states, or hidden states in its history, forming a memory or summary of the past. These dependencies form loops in the diagram of RNN and enable the model to make use of sequential inputs.

The invention of RNN can be dated back to 1980s [**141**]. Various forms of RNNs have been proposed since then, among which the most popular nowadays include the vanilla RNN [**12**], Long-Short-term Memory networks (LSTM) [**73**], and more recently Gated recurrent units (GRU) [**31**].

The recurrent connections can be visualized in two ways. One illustrates the model as a directed cyclic graph, which denotes the recurrent dependencies as loops with delay

units (Figure 2.2), the other is the unfolded form, where the RNN is unfolded in the time dimension, thus illustrating the model in the form of an infinite directed acyclic graph. Note that the recurrent connections in an RNN could become very complicated, such as covering multiple time steps or involving multiple groups of hidden states, but only a small part of them are explored and widely used. The most common way of scaling up RNNs is using a relatively simple recurrent cell and stack them into layers. For a more fundamental analysis on the connecting architecture of RNNs, please refer to [**183**].

RNNs can process the sequential data in a chronological order, in which case it is called uni-directional RNN. A lot of models such as language models, real-time machine translation models fall into this case. On the other hand, if the sequential data is finite and the whole sequence could be accessed at once (i.e., the sequential model does not necessarily need to be a causal system), then it will also be possible to apply another RNN that is applied in a reverse-chronological order. These recurrent models are called bi-directional RNNs. Bi-directional RNNs are a natural extension to uni-directional RNNs, and has found wide usage in applications such as sentence embedding, dialog systems, etc.

In the following subsections we are going to introduce the canonical vanilla RNN [**12**], Long-Short-term Memory networks (LSTM) [**73**], and Gated recurrent units (GRU) [**31**].

2.2.1. Vanilla Recurrent Networks

Vanilla RNN was introduced in [**74**], which is considered the most basic RNN structure these days. It is a single layer, unidirectional RNN with $\tanh(\cdot)$ as its activation function. Vanilla RNN extends feed forward networks by simply making its hidden state updates both related to the current input and the state at the previous time step:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h) \tag{2.2.1}$$

where $\tanh(\cdot)$ denotes the nonlinear activation function. Figure 2.2 depicts the two ways of visualization of this model. The hidden state h_t can further be connected to some output layers to yield an output at each time step. For example, in character-level language modeling, h_t at each time step is mapped by a softmax layer to an output, which yields the probability of each character appearing at the next time step.

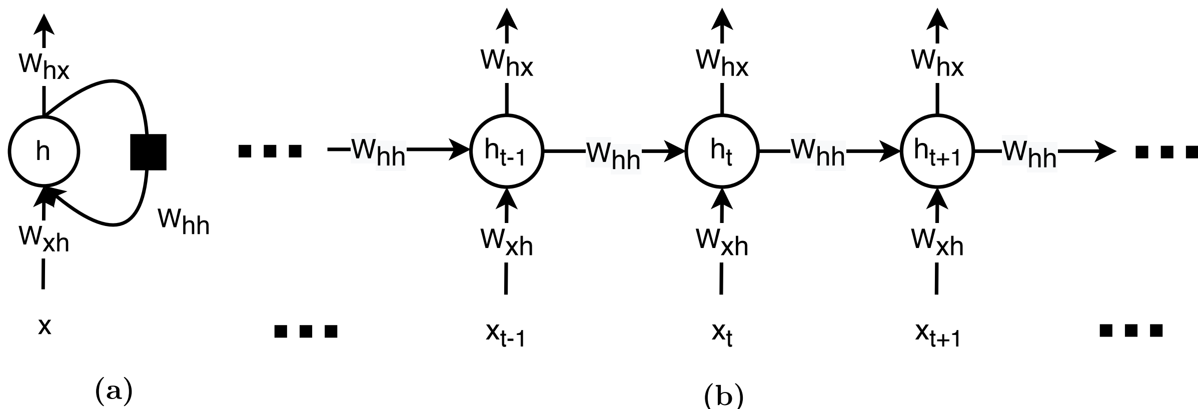


Fig. 2.2. Structure of a simple single layer, unidirectional recurrent neural network. (a) The folded diagram of the RNN, showing its structure as a directed cyclic graph. The black rectangle stands for a one step time delay. (b) The unfolded diagram of the same RNN, which unfolds the RNN in the time direction by repeatedly drawing the same cell over all time steps. The unfolded structure should always be acyclic.

$$p_t = softmax(W_{hx}h_t + b_x) \tag{2.2.2}$$

The bi-directional version of the vanilla RNN is a natural extension to Figure 2.2, which applies another RNN in the reverse order, and the hidden states of both RNNs are concatenated Figure 2.3.

One of the draw-backs of vanilla RNNs is its instability due to vanishing or exploding gradients, and people have found it hard for the network to remember contents after a long period of time steps [**12**]. These problems are explored in depth by Hochreiter [**73**] and Bengio, et al. [**12**]. In practical applications, there are two successful RNN variants that alleviate this problem by introducing *gates* to the connections: Long-Short Term Memory units and Gated Recurrent Units. We will elaborate on them in the next two subsections.

2.2.2. Long-Short Term Memory units

The Long-Short Term Memory units (LSTM) is a variation of recurrent networks proposed by Sepp Hochreiter in early 1990s [**73**]. LSTM can be viewed as an extension to vanilla RNN by adding a series of *gates* and an internal cell state. The update equations at time step t for the LSTM are as follows

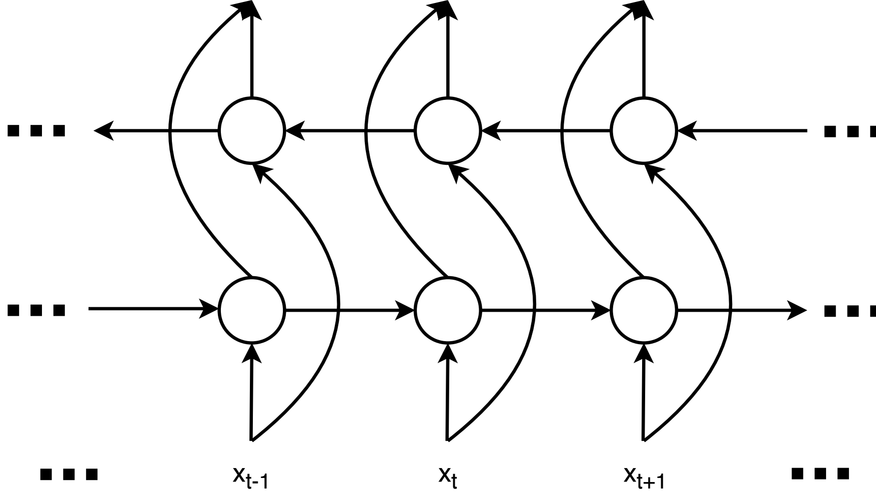


Fig. 2.3. Structure of bidirectional recurrent neural network. For simplicity we omitted all the notations on weights and hidden states.

$$
\begin{aligned}
i_t &= \sigma(U_i x_t + W_i s_{t-1} + b_i) \\
f_t &= \sigma(U_f x_t + W_f s_{t-1} + b_f) \\
o_t &= \sigma(U_o x_t + W_o s_{t-1} + b_o) \\
g_t &= \tanh(U_g x_t + W_g s_{t-1} + b_g) \\
c_t &= c_{t-1} \odot f + g \odot i \\
h_t &= \tanh(c_t) \odot o_t
\end{aligned}
\tag{2.2.3}
$$

Here $\odot$ stands for element-wise multiplication, and $\sigma(\cdot)$ is the sigmoid activation function. $W_.$ and $b_.$ are the corresponding weights and biases. The i_t, f_t, o_t are the input gate, forget gate, and output gates, respectively. c_t is the internal cell state, and h_t is the output hidden state.

The three gates control the flow of information in the network. All of i_t, f_t, o_t are outputs of sigmoid functions, thus their values resides between 0 and 1. While getting element-wise multiplied with another vector, it can be seen as a *gate* controlling all the dimensions of that vector, with 0 being *closed* since it zeros out whatever value in that dimension, while 1 being *open* since it lets the original value pass through the gate. Input gate (i_t), forget gate (f_t), and output gate (o_t) are controlling the inputs to the cell state (g), cell state inherited from the last time step (c_{t-1}), and cell state at the current time step (c_t), respectively.

These gating mechanisms are explicitly designed to enable the LSTM to memorize contents in the cell states for long time by controlling the input and forget gates [**73**]. Experimentally it is found to work tremendously well on a large variety of problems, and is now widely used.

2.2.3. Gated Recurrent Unit

The Gated Recurrent Unit (GRU) [**31, 35**] is another successful variant of RNN that processes sequential data. It can be viewed as a simplified version of LSTM without the cell state and output gate.

At time step t, GRU first computes two sets of gates, i.e., the update gate z_t and the reset gate r_t:

$$\begin{aligned} z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \end{aligned} \tag{2.2.4}$$

where x_t is the input at time step t, $W_.$ and $b_.$ are the corresponding weights and biases. h_{t-1} is its hidden state at $t-1$, which we will elaborate later. Then a candidate activation $\widetilde{h_t}$ is computed with the reset gate r_t and hidden state h_{t-1}:

$$\widetilde{h_t} = \tanh(Wx_t + U(r_t \odot h_{t-1})) \tag{2.2.5}$$

The intuition of this step is that the reset gate r_t effectively makes the unit to forget the previous hidden states h_{t-1}, just like the forget gate in the LSTM. The final hidden state of GRU is then specified by

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h_t} \tag{2.2.6}$$

where the hidden state h_t is a linear interpolation between the previous activation h_{t-1} and the candidate activation $\widetilde{h_t}$, where the interpolation is determined by the update gate z_t.

Note that there is another variant for Eq. 2.2.5, which was proposed in [**35**], where the order of multiplication between U and r_t is slightly different [**30**]:

$$\widetilde{h_t} = \tanh(Wx_t + r_t \odot (Uh_{t-1})) \tag{2.2.7}$$

Experimentally there is no big difference between the two variants, but the latter one is preferred due to computational reasons.

The GRU effectively removes one fourth of the parameters in the LSTM, thus allows more hidden states with a same budget of model size. This advantage allows the GRU to be able to yield better results on tasks with large datasets such a machine translation. It achieved state-of-the-art performance on several machine translation benchmarks when it was published, while later on the more parallizable Transformer model is proposed and achieved a new set of state-of-the-arts. We will introduce the Transformer model in Section 2.4.

2.3. Attention Mechanism

Attention mechanisms are proposed in the research on neural machine translation [**5**], which substantially improved the performance of neural machine translation models and make it surpass the traditional phrase-based methods. Nowadays attention is used in various problems, expanding from the field of machine translation to computer vision, speech recognition, and so on.

In this section, we are going to introduce a soft-attention mechanism in the context of machine translation, which is most related to the articles included in this thesis. A neural machine translation model usually involves an encoder which encodes the source sentence to a series of hidden states, and a decoder which generates the target sentence given the source sentence hidden states as context (Figure 2.4). The encoder can be represented as a bidirectional RNN, i.e.,

$$h_i = [\overrightarrow{h_i}, \overleftarrow{h_i}]^T \tag{2.3.1}$$

Here we use $\overleftarrow{h_i}$ and $\overrightarrow{h_i}$ to represent the forward and backward RNNs' hidden states at time step i, and we omit the details inside the RNNs. Note that h_i is dependent on the all the inputs $x = \{x_1, ..., x_i, ..., x_N\}$.

The decoder at decoding time step j predicts the probability of the next word given the current word y_{j-1}, the decoder state s_j, and the context c_j:

$$p(y_j|y_1, ..., y_{j-1}, x) = D(y_{j-1}, s_j, c_j) \tag{2.3.2}$$

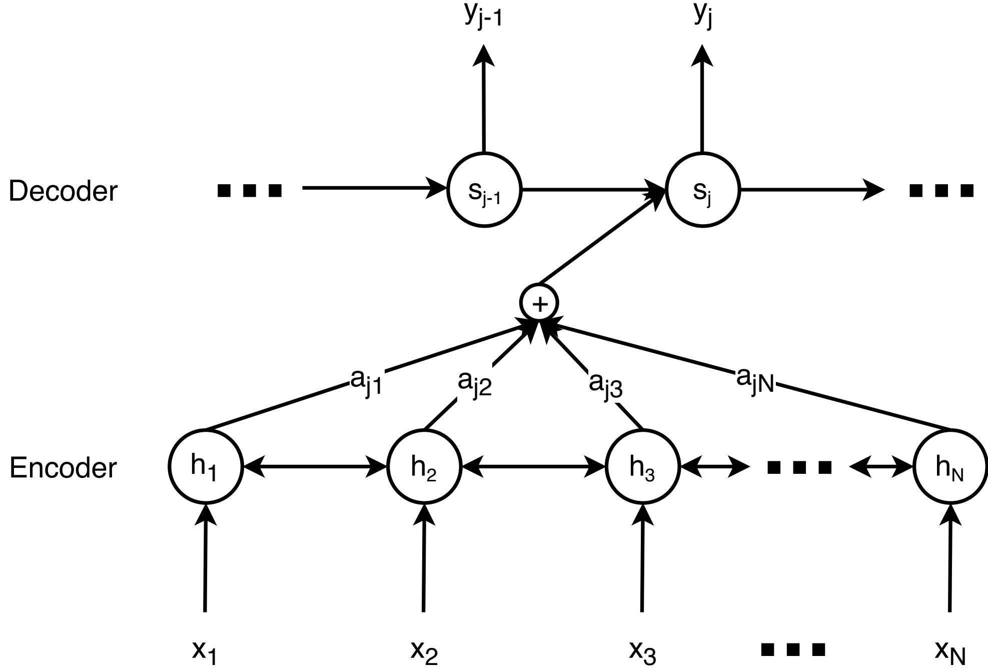


Fig. 2.4. Attention mechanism in a machine translation context. The upper unidirectional RNN is a decoder that is trying to infer the next word y_j, and the lower bidirectional RNN is an encoder that encodes source sentence tokens into a sequence of hidden states.

Here we use $D(\cdot)$ to represent the RNN computations inside the decoder. s_j is simply the RNN hidden states at time step j. The attention mechanism resides in the way we factorize the context vector c_j.

$$c_j = \sum_{k=1}^{N} a_{jk} h_k \tag{2.3.3}$$

where a_{jk} is called the *attention weight*, which is the softmax output over all the N positions in the encoder, whose input is dependent on the current decoder state s_{j-1} and the encoder states h_i:

$$e_{jk} = A(s_{j-1}, h_k) \tag{2.3.4}$$

$$a_{jk} = \frac{\exp(e_{jk})}{\sum_{n=1}^{N} \exp(e_{jn})} \tag{2.3.5}$$

Here we use $A(\cdot)$ to denote the neural network that computes e_{jk} from the inputs s_{j-1} and h_k). Usually this is implemented as a multi-layer feed-forward network, or simply a dot product.

The intuition of this attention mechanism is that during decoding, it allows the decoder to focus on part of the input sentences that are relevant to the current decoding step. This allows the model to find the relevant input positions and align them for the decoding position j. Since the alignments appears very similar to human's attention on different words while doing translation, it is called *attention mechanism*.

2.4. Transformer

There have been different variants of attention mechanisms since they are first proposed, among which self-attention becomes the most influential variant, which are proposed under the name of self-attention [**103**] or intra-attention [**178**]. Later on the Transformer model [**162**] significantly developed the self-attention mechanism, which resulted in a series of the state-of-the-art performances. Since then the Transformer has become a popular model in a lot of tasks, such as machine translation [**162**], language modelling [**42**], etc. More importantly, recent advances on pre-trained contextualized embeddings, such as BERT [**47**], GPT [**139**], RoBERTa [**108**], etc., has made Transformer in a central role of various NLP tasks. In this section, we are going to introduce the Transformer model, also in a machine translation context. We will cover the BERT mode in Chapter 3 as well.

The Transformer still follows the encoder-decoder framework as depicted in Section 2.3. The difference lies in the form of the encoder and decoder, and the attention mechanism between them. Following the notations, let $X = \{x_1, ..., x_i, ..., x_N\}$ be the input tokens and $IE = \{IE_1, ..., IE_i, ..., IE_N\}$ be their corresponding word embeddings of e dimensions, which is retrieved through a loo-up table of parameters. The encoder maps the inputs to a sequence of representations $H = \{h_1, ..., h_i, ..., h_N\}$, each is of d dimensions, and the decoder sequentially generates an output sequence $Y = \{y_1, ..., y_i, ..., y_M\}$ given H.

Since the Transformer model uses a set of attentions to process the sequential input, in order to make the model aware of the position of each token in the sequence, it introduces *positional encodings* in addition to the input embeddings. The positional encodings are a set of sine and cosine functions of different frequencies. i.e., for the $2i$-th and $2i+1$-th dimension

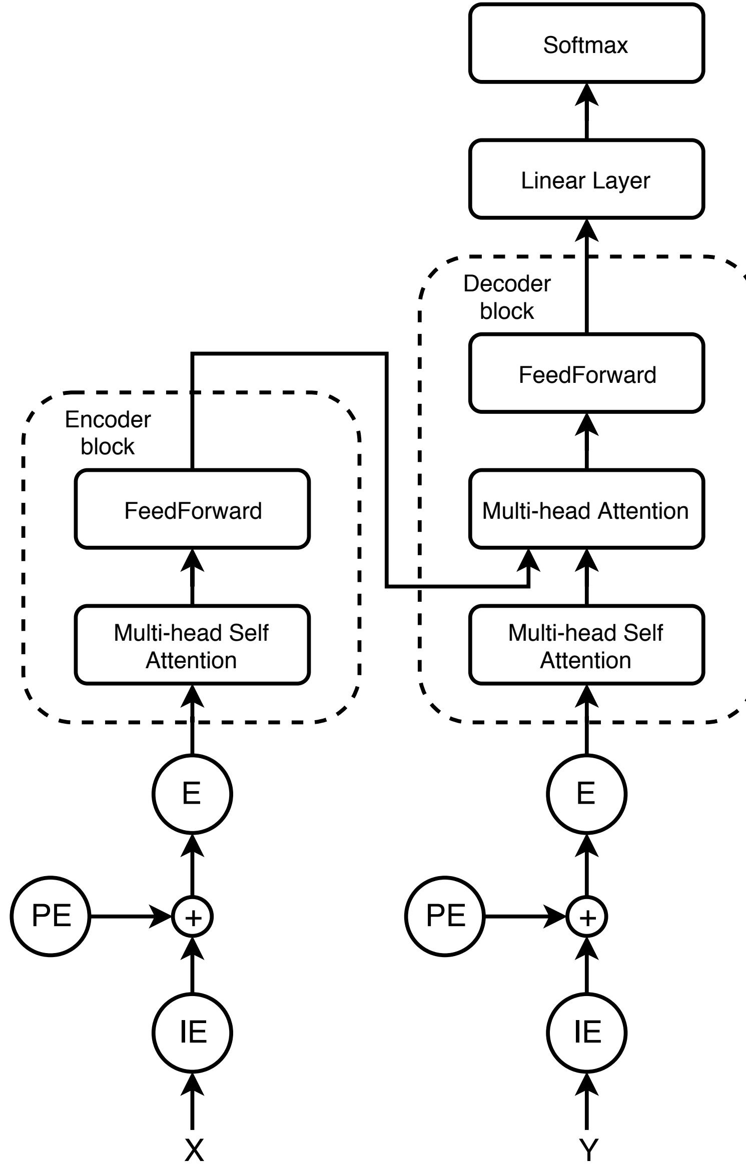


Fig. 2.5. Structure of the Transformer model. For simplicity we merge all the individual tokens in a sequence and represent them as a whole, which is notated as X and Y in the figure. Within each of the components in the encoder and decoder block, there is a layer normalization step included. Please refer to the text for details.

of the positional encoding of the j-th position of the sequence, their corresponding positional encodings (PE) are given by

$$PE_{j,2i} = \sin \frac{j}{10000^{2i/e}} \tag{2.4.1}$$

$$PE_{j,2i} = \cos \frac{j}{10000^{2i/e}} \tag{2.4.2}$$

where e here stands for the number of dimensions of the positional encoding, which is the same as that of input embeddings IE. The positional encoding PE_j is then summed up with its corresponding input embedding (IE_j) to form the input representations (E_j) of the model.

$$E_j = PE_j + IE_j. \tag{2.4.3}$$

For simplicity of notation, here we use PE_j to represent the whole d-dimensional positional encoding vector $PE_{j,\cdot}$, so as to E_j and IE_j. We note that instead of the sinusoidal functions, the positional encoding can also use randomly initialized parameters and learn their values through the training process.

Now let's get into the encoder. The encoder consists of several replicas of the same block structure, with its first block taking E as input. For each of the blocks, it consists of 2 components, which are a multi-head self-attention layer followed by a position-wise fully connected layer. Moreover, a residual connection followed by layer normalization is used for both of the components in the block. This is depicted in the left half of Figure 2.5. Different blocks are vertically stacked together and share the same structure, but their parameters are not tied. We note the input to the i-th block as H^{in}, and its output as H^{out}.

Within each block in the encoder, the multi-head attention component first maps its input H^{in} through a set of linear projections, which will later be utilized as inputs to the multiple hops of attentions. Thus for the i-th attention, we have

$$L_i = W_i H^{in} \tag{2.4.4}$$

where W_i is the linear projection. Since there are multiple hops, we note them all together as 3-dimensional tensor L.

Each single attention maps a query and a set of key-value pairs to an output. The output is a weighted sum of the values in the key-value pairs, where the summing weights (namely attention weights) are computed from the query and the corresponding keys. Formally, each hop of attention is computed as follows:

$$A = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.4.5}$$

where Q, K and V are called query, key and value of the attention, and d_k stands for the dimension of the keys. In the self-attention case, the Q, K and V are all from the same source, which is a linear projection of the block's inputs. In the case of encoder, the Q, K and V are all the same, which is the block's projected input L_i. The output of each of the attention heads is then concatenated and linearly projected to a d dimensional space.

$$\tilde{M} = \text{concat}(A_1, A_2, ..., A_i, ..., A_m)W_o \tag{2.4.6}$$

where A_i stands for the attention output of the i-th attention hop, and W_o is the linear projection matrix. Since layer normalization [**4**] is used, the actual output of the multi-head attention component M is the output of the LayerNorm$(\cdot)$ function:

$$M = \text{LayerNorm}(\tilde{M}, H^{in}) \tag{2.4.7}$$

The second component, the position-wise fully connected layer is simply two linear transformations with a ReLU activation function (noted as $r(\cdot)$ in the equation) in between. i.e.,

$$\tilde{H^{out}} = FFN(M) = W_2 r(W_1 M + b_1) + b_2 \tag{2.4.8}$$

where W_1, W_2, b_1 and b_2 are all parameters. And again, the actual of the output is the layer normalized output:

$$H^{out} = \text{LayerNorm}(\tilde{H^{out}}, M) \tag{2.4.9}$$

This block-wise structure is repeated several times in the vertical direction to form a stack of blocks, which forms the encoder.

As for decoder, in addition to the two components in the encoder, the decoder block has one extra multi-head attention component that performs attention between the encoder and decoder (See Figure 2.5). In this component, the multi-head attention structure is still the same, except that the input to the attention is different. The query Q is from the decoder self-attention component, while the key K and value V are from the last block's output of the encoder H^{out}.

The decoder takes its last blocks output, and pass it through a softmax layer to yield a probability distribution of the next token in the decoded sequence. Since this will be introduced in Section 3.1, we are not going into the details here.

2.5. Low-precision neural networks

Computations in biological neural networks are parallel. Most of the computations in artificial neural networks are matrix multiplications, and this type of computation can easily be parallelized. Moreover, the heavily used matrix multiplications could be further accelerated by incorporating low precision computation. In all, there are a lot of aspects in neural network models that could be improved from the perspective of computational efficiency. We will review some of them in this section, and discuss previous works in these threads of research.

2.5.1. Multiplication

Training deep neural networks has long been computational demanding and time consuming. For some state-of-the-art architectures, it can take weeks to get models trained [**93**]. Most of the computation performed in training a neural network are floating point multiplications with 32-bit numerical accuracy. A multiplication takes 32 times more computation than an addition.

Several approaches have been proposed in the past to simplify computations in neural networks. Some of them try to restrict weight values to be an integer power of two, thus to reduce all the multiplications to be binary shifts [**94, 112**]. In this way, multiplications are eliminated in both training and testing time. The disadvantage is that model performance can be severely reduced, and convergence of training can no longer be guaranteed.

[**85**] introduces a completely Boolean network, which simplifies the test time computation at an acceptable performance hit. The approach still requires a real-valued, full precision training phase, however, so the benefits of reducing computations does not apply to training. Similarly, [**111**] manage to get acceptable accuracy on sparse representation classification by replacing all floating-point multiplications by integer shifts. Bit-stream networks [**17**] also provide a way of binarizing neural network connections, by substituting weight connections

with logical gates. Similar to that, [**28**] proves that deep neural networks with binary weights can be trained to distinguish between multiple classes with Expectation Back-propagation.

There are some other techniques which focus on reducing the training complexity. For instance, instead of reducing the precision of weights, [**147**] quantizes states, learning rates, and gradients to powers of two. This approach manages to eliminate multiplications with negligible performance reduction.

2.5.2. Memory Demand

The demand for memory in practical networks can be huge. For example, many common models in speech recognition or machine translation need 12 Gigabytes or more of storage [**67**]. However, we know that neural networks usually doesn't need that much accuracy, a network with *float16* will generally work equally well as *float32*. The redundancy also exists in the number of parameters [**46**].

There are two ways to deal with this. We can use more computational power, and develop specialized algorithms for distributed computing, thus make it possible to train larger networks. It is common to train deep neural networks by resorting to GPU or CPU clusters and to well designed parallelization strategies [**96**]. There have been several researches which propose an approach for a distributed asynchronous stochastic gradient descent algorithm [**9, 45, 29**]. Stochastic gradient descent with parameter synchronization are also studied in [**24, 44**]. Alain et al. [**1**] even studied a more specialized way designed for training neural networks in a distributed setting.

On the other hand, we can compress the current model to fit a larger model into the current available memory. [**68**] managed to reduce the memory requirements by several hundred times, while still retaining the same performance. [**3**] quantized the neural network using L2 error minimization and achieved better accuracy on MNIST and CIFAR-10 datasets. HashNets [**26**] uses a hash function to separate weights into different groups, and share weight values across all weight that fall into a same hash bucket. [**61**] uses vector quantization to compress deep CNNs, without significant performance loss.

As we see above, further development and optimization of deep learning techniques requires the combination of both algorithm design and hardware realization.

Chapter 3

Natural Language Processing

In this chapter, we are going to introduce three basic building blocks that set foundations for the application of neural networks in Natural Language Processing (NLP). They are neural language models, sentence embedding models and the more recent pre-trained Transformer models. The language models and sentence embedding models correspond to learning to encode distributed representations for words and sentences respectively in a continuous space, while the pre-trained Transformer models learn contextualized representations for sentences. We are also going to show relevant downstream tasks that could benefit from these models. Most of the tasks to be mentioned in this chapter are going to be referred to in the following chapters of articles.

3.1. Encoding words: Neural Language Models

Language model serves as a central problem in a large range of natural language processing tasks such as machine translation [**5**], image captioning [**177**], text summarization [**127**], speech recognition [**65**], and so on.

Language models are probabilistic models that are able to estimate the probability of the appearance of a given sentence in a language. Suppose we have a sentence S with k words $S = (w_1, w_2, ..., w_k)$, the language model estimates the joint probability

$$p(S) = p(w_1, w_2, ..., w_k) \tag{3.1.1}$$

Most language models estimate the probability by exploiting the chain rule of probabilities by factorizing the joint probability into a sequence of conditional probabilities. This

allows the language model to assign a series probabilities for the likelihood of each of the words in the sequence, given its preceding words as context, i.e.,

$$p(S) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)...p(w_k|w_1, w_2, ..., w_{k-1}). \tag{3.1.2}$$

Again for computational issues, the probability is always computed in an logarithmic manner:

$$\log p(S) = \sum_{i=1}^{k} \log p(w_i|w_1, w_2, ..., w_{i-1}) \tag{3.1.3}$$

Since the context k could grow very big as the sentence becomes longer, which may make it hard to estimate the conditional probability, one approach is to approximate it by simplifying the dependencies. We can truncate the dependencies at a small fixed number (N) of tokens , i.e.,

$$p(w_i|w_1, w_2, ..., w_{i-1}) \approx p(w_i|w_{i-N+1}, ..., w_{i-1}) \tag{3.1.4}$$

This leads to the assumption of the *N-gram* models. N-gram models approximate the conditional probability with N words, and estimate these probability by counting the occurrence of such examples in a big corpus:

$$p(w_i|w_{i-N+1}, ..., w_{i-1}) \approx \frac{count(w_{i-N+1}, ..., w_{i-1}, w_i)}{count(w_{i-N+1}, ..., w_{i-1})} \tag{3.1.5}$$

One of the central problem that comes with N-gram models is data sparsity. As N grows, the counts in Eq. 3.1.5 becomes very small, or even zero, for most of the word combinations. It then becomes very hard to estimate the conditional probability. There are a whole series of research in discussing how to distribute the probability onto these rare conditional probabilities, which is referred to as smoothing techniques [**130, 25**]. We are not going to dive into these techniques, but instead introduce neural language modelling, which completely gets rid of this problem by encoding words into continuous space and using neural networks to estimate candidate probabilities.

Neural language models [**8**] estimate $p(w_i|w_1, w_2, ..., w_{i-1})$ by calculating probability through dot products between a hidden state and a big table of continuous vectors. The continuous vectors are distributed representations of words, whose parameters are learned

during the training process. These distributed representations are also called word embeddings, which serve as a central role in various downstream tasks in NLP. Note that there are other kinds of models that learns word embeddings, such as GloVe [**136**], Skip-Gram and CBOW [**118**]. We are not going to dive into those models since they are not closely related to the articles in the following chapters.

A neural language model first maps all the words to a big word embedding matrix through a look-up table,

$$e_i = W_e[w_i] \tag{3.1.6}$$

where e_i is the vector-valued embedding of word symbol w_i. In a typical neural language model, a recurrent neural network, such as the LSTM or GRU that we introduced in Chapter 2, is used to model the dependency between words. These word embeddings are used as input to the recurrent neural network. Here we use a function $f(\cdot)$ to represent the recurrent network:

$$h_{i+1} = f(e_i, h_i) \tag{3.1.7}$$

The h_i and h_{i+1} are hidden states in the network. h_{i+1} is then used to compute the probability of a certain word being the next word, for each of the words in the word embedding matrix. This is done via a dot product between h_{i+1} and the word embedding matrix W, and pass them through a softmax function:

$$p(w_i|w_{i-N+1}, ..., w_{i-1}) = \frac{exp(e_i^T h_{i+1})}{\sum_{k=1}^{V} exp(e_k^T h_{i+1})} \tag{3.1.8}$$

where V stands for the total number of words in the vocabulary.

During learning, this recurrent step iterates through the whole training corpus, while at the same time it keeps inheriting the hidden states h_i through the whole iteration. By inheriting the hidden states, recurrent networks could learn the relevant context over a much longer sequences than N-grams. In the output layer, it tries to predict correctly the next word by maximizing the likelihood of each of the words given its context through a cross-entropy loss:

$$p(w_i|w_{i-N+1},...,w_{i-1}) = \frac{exp(e_i^T h_{i+1})}{\sum_{k=1}^{V} exp(e_k^T h_{i+1})} \tag{3.1.9}$$

The evaluation of language models are based on the likelihood the model would give on an unseen test corpus. At test time, the model predicts the next token word-by-word and iterates through the whole dataset. For each of the tokens w_i, it predicts the conditional probability that the target word appears as the next word, given its previous context. According to Eq. 3.1.3, we can define a per-word entropy H which represents the average amount of non-redundant information provided by each new word.

$$H = -\frac{1}{k} \log p(S) \tag{3.1.10}$$

Since a good enough language model should be able to predict pretty well on the next tokens, this entropy is getting minimized during the training process. The *perplexity per word* (PPW) is then defined as

$$PPW = e^H \tag{3.1.11}$$

One of the good feature about PPW is its intuitiveness. Suppose we have a random model that always predict a uniform distribution among all the words in the vocabulary, Then Eq. 3.1.11 will yield V, where V is the vocabulary size. On the opposite side, suppose we have an ideal language model that always knows the next token exactly, then Eq. 3.1.11 will yield a value of 1. Intuitively, the value of PPW corresponds to "the performance of the evaluated model is equivalent to a model that is choosing randomly between PPW words at each time step".

Experimentally, people have found that continuous neural models generalize much better than N-gram models, and scale better with respect to the vocabulary and corpus size. Thus state-of-the-art language models nowadays are all neural language models.

3.2. Encoding sentences: Sentence Embeddings

As neural language models that provide one way to embed words into a continuous space turn out to be successful and become the dominant method in language modelling, sentence embeddings, which try to encode sentences into a continuous space that could reflect the semantic relatedness of the sentences, have also been explored in various directions in

the literature. Unlike the consensus of learning and using word embeddings as distributed representations for word tokens, efforts in encoding sentences into a fixed size vector in a continuous space has encountered much more challenges.

There are essentially two groups of approaches in learning sentence embeddings. One tries to learn a general sentence encoder from a raw corpus, which corresponds to unsupervised sentence embeddings. These sentence embeddings are deemed as having the ability to boost the performance of various downstream tasks, so that the parameters in these models usually serve as pre-trained parameters to initialize models for downstream tasks. The other approach learns the sentence through some specific task, and thus usually the resulting sentence embedding is only effective for that task. That corresponds to supervised sentence embeddings.

Unsupervised sentence embeddings are learned from a raw corpus, by learning to reconstruct the sentence itself, learning to predict the surrounding sentences or just adding word embeddings in a sentence etc. For example, [**100**] encodes sentences by feeding word embeddings into an LSTM and then max-pooling over all of its hidden states. Sequential denoising autoencoders [**71**] incorporate more sentence information by making the model sensitive to the ordering of the words. It first corrupts a sentence by randomly deleting or swapping words, and then learns to reconstruct the original sentences by reading the corrupted sentence. Skip-Thought vectors [**89**] incorporate the coherence between consecutive sentences. It learns to encode a sentence in a manner pretty similar to Skip-Gram [**118**], where it encodes a certain sentence into a fixed size embedding vector and tries to reconstruct its previous and next sentence with a separate decoder given the embedding vector. DiscSent [**81**] takes more inter-sentence relations into learning. It encodes a pair of sentences into vector representations, and predicts 3 different discriminative tasks that are related to their positions in the context and their semantics. The first two tasks are classification tasks on if one sentence shows up before or after the other. The third task is for sentence pairs with the second sentence starting with a conjunction phrase. A classifier is trained to tell between 9 different conjunction types such as “addition”, “contrast” or “strengthen”, depending on the conjunction phrase.

Unsupervised sentence embeddings can be evaluated in various ways. A simple qualitative evaluation could be simply visualize the sentence embedding by visualization methods

such as t-SNE [**110**] to see how good the representation space reflects human intuitions of the semantics of sentences. Quantitative measures also exist. For example, we can compute the cosine distance between sentence embedding pairs, and compute the resulting score with gold standard scores provided by external datasets, such as the SICK corpus [**114**], which consists of sentence pairs with human labeled judgments on their relatedness. There are also supervised evaluations, which combine the trained sentence embedding with a downstream task, and evaluate the sentence embeddings by looking at their performance on the downstream tasks. Widely used downstream tasks for this purpose include paraphrase identification [**171**], natural language inference [**37**], sentiment analysis [**20**], question type classification [**20**], etc.

Supervised sentence embeddings, on the other hand, rely on supervised training on a specific downstream task to capture the semantics of sentences. For example, [**72**] learns sentence embeddings by learning to map dictionary definitions to words defined by those definitions. Natural language Inference (NLI) is another important task for learning supervised sentence embeddings. In NLI, the model is given a pair of sentences and is asked to tell if the content in the sentences are contradicting, entailing, or irrelevant to each other. There are a bunch of sentence embedding models that are built around this task, such as NSE [**126**], SPINN-PI [**15**], InferSent [**37**], Tree-based CNN [**122**], Gumbel TreeLSTM [**33**] etc. Other downstream tasks such as those aforementioned in the evaluation methods for unsupervised sentence embedding could also be applied as supervised training signals for supervised sentence embeddings. Among all the downstream tasks, research [**37**] has shown that NLI seems to be the best supervised learning task that is able to extract the most semantic information from sentences, since sentence embeddings learned under this task show better transfer ability towards other tasks.

Evaluating supervised sentence embeddings is much more straight forward. The most important evaluation metric naturally becomes the downstream task that it is trained on. Apart from the supervised task, other evaluations that apply on unsupervised sentence embeddings could also be used.

3.3. Contextualized Pre-trained Models

Instead of trying to encode sentences into a fixed size embedding, more recent approaches reveal that encoding sentences into a sequence of vector representations by some unsupervised learning objective could yield a pre-trained model with promising results on various tasks [**47, 139, 137**]. While some of them are based on bi-directional LSTM such as the ELMo model [**137**], most of these models are based on the Transformer model [**162**], and pre-trained on a large amount of unlabelled data. We will introduce the BERT model as an example here.

The BERT model inherits the Transformer (detailed in Section 2.4) model structure, while proposed several pre-training tasks that can be performed on large scale corpus of plain text. These pre-training tasks enable the model to learn a general, contextualized representation for each of the tokens in the sentence.

The first pre-training task is masked language modelling. Instead of using an auto-regressive approach like the neural language model introduced in Section 3.1 which can only learn a left-to-right dependency, the BERT model uses a Cloze style task to encourage the model to learn a bidirectional, contextualized representation for each of the input tokens. It randomly chooses 15% of the tokens as masked tokens, and let the model to predict the actual token through a softmax output. The chosen tokens are not always replaced by the dedicated `[MASK]` token; they could also be replaced by a random word or kept unchanged.

The second pre-training task is next sentence prediction. The model encodes two sentences and then asked if the two input sentences are consecutive or not. This is a plain text classification task with only two classes. The input sequences could also easily be constructed from plain corpus as well, with the negative examples (the ones who don't consist of consecutive sentences) being constructed by putting two random sentences together.

The pre-training tasks are performed on a huge plain corpus, which is the concatenation of BookCorpus [**190**] and English Wikipedia, reaching a total word count of over 3 billion. The bulky pre-training corpus makes the training computationally heavy.

As for fine-tuning, the pre-trained model is then connected to an extra task-specific component. The fine-tuning procedure is performed on both the pre-trained parameters as well as those in the task-specific component. Since most of the times the extra parameters introduced for downstream tasks are several orders of magnitude fewer than the pre-trained

parameters, and the dataset used for fine-tuning task is usually not so large as that of pre-training, fine-tuning the model is not so cumbersome as the pre-training.

The pre-training and fine-tuning scheme of BERT has shown its effectiveness on a wide spectrum of NLP tasks, such as sentiment analysis, text classification, paraphrase identification, language inference, question answering, etc. Other models that follow a similar scheme are also shown to be successful, such as the earlier ELMo [**137**] and GPT [**139**] models. More recently, various newer models that follow this scheme have been proposed, mostly based on the transformer model. Some of them use a different training procedure and fine-tune the hyperparameters over BERT, such as RoBERTa [**108**]. Some of them use a larger model and larger training dataset over their predecessor, such as GPT-2 [**140**]. And some of them try to shrink the model size, such as ALBERT [**95**]. These follow-ups achieved impressive improvements over their non-pre-trained counterparts in different tasks.

Although models like BERT are beyond the classical scope of sentence embeddings since they are not encoding sentences into fixed size vectors anymore, their effectiveness is pushing them towards a central role in modern NLP research.

Chapter 4

Prologue to First Article

4.1. Article Details

Neural Networks with Few Multiplications. Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, Yoshua Bengio, *4th International Conference on Learning Representations, 2016*

Personal Contribution. The idea of ternary connect is an extension to Matthieu Courbariaux's BinaryConnect paper [**39**], Yoshua Bengio suggested using quantization during backpropagation to convert multiplications to binary shifts. I implemented the ternary connect and quantized backpropagation, conducted all the experiments, and rendered all the figures and tables. I wrote most parts of the paper, with significant contributions to the writing from Roland Memisevic and Yoshua Bengio.

4.2. Context

The computation and memory demands in training deep neural networks had been an issue that drew a lot of attention and critiques [**96**] during the time the paper was published. Most of the computation burdens in training neural networks are in floating-point multiplications. In a previous work [**39**], Courbariaux et al. have proposed to sample binarized weights during the forward pass to convert the multiplications to sign changes, which could significantly reduce the forward computation time. Several other approaches have been proposed before this publication, but either not focusing on training time [**85**], or significantly reduced the performance of the network [**94, 112**].

4.3. Contributions

Our method consists of two parts. First, weight values are stochastically binarized or ternarized in the forward propagation, which converts multiplications to sign changes. Second, during backward propagation, in addition to binarizing the weights, the hidden states at each layer are quantized so that multiplications are converted to binary shifts. We found that our method not only does not hurt the model's performance but results in even better performance than standard, full-precision training.

4.4. Recent Developments

As of writing, the paper has accumulated 185 citations. Binarizing or ternarizing weights have been used in various works since then. For example, [**40**] has made binarization work for both weights and activations. Ternary weight networks [**101**] focus on evaluation time by minimizing the Euclidean distance between the original weights and the ternarized weights. Trained Ternary Quantization [**188**] extends the weight ternarization by making the model able to learn an extra scaling factor that decides the ternarized weights' values. Reducing computation in the backward process was explored more extensively in various forms as well. DoReFa-Net [**187**] has proposed to low-bitwidth gradients, [**176**] proposes low-bitwidth integers to represent gradients.

Chapter 5

Neural Networks with Few Multiplications

5.1. Introduction

Training deep neural networks has long been computational demanding and time consuming. For some state-of-the-art architectures, it can take weeks to get models trained [**93**]. Another problem is that the demand for memory can be huge. For example, many common models in speech recognition or machine translation need 12 Gigabytes or more of storage [**67**]. To deal with these issues it is common to train deep neural networks by resorting to GPU or CPU clusters and to well designed parallelization strategies [**96**].

Most of the computation performed in training a neural network are floating point multiplications. In this paper, we focus on eliminating most of these multiplications to reduce computation. Based on our previous work [**39**], which eliminates multiplications in computing hidden representations by binarizing weights, our method deals with both hidden state computations and backward weight updates. Our approach has 2 components. In the forward pass, weights are stochastically binarized using an approach we call *binary connect* or *ternary connect*, and for back-propagation of errors, we propose a new approach which we call *quantized back propagation* that converts multiplications into bit-shifts. [1]

5.2. Related work

Several approaches have been proposed in the past to simplify computations in neural networks. Some of them try to restrict weight values to be an integer power of two, thus to reduce all the multiplications to be binary shifts [**94, 112**]. In this way, multiplications are

[1]The codes for these approaches are available online at `https://github.com/hantek/BinaryConnect`

eliminated in both training and testing time. The disadvantage is that model performance can be severely reduced, and convergence of training can no longer be guaranteed.

[**85**] introduces a completely Boolean network, which simplifies the test time computation at an acceptable performance hit. The approach still requires a real-valued, full precision training phase, however, so the benefits of reducing computations does not apply to training. Similarly, [**111**] manage to get acceptable accuracy on sparse representation classification by replacing all floating-point multiplications by integer shifts. Bit-stream networks [**17**] also provides a way of binarizing neural network connections, by substituting weight connections with logical gates. Similar to that, [**28**] proves deep neural networks with binary weights can be trained to distinguish between multiple classes with expectation back propagation.

There are some other techniques, which focus on reducing the training complexity. For instance, instead of reducing the precision of weights, [**147**] quantizes states, learning rates, and gradients to powers of two. This approach manages to eliminate multiplications with negligible performance reduction.

5.3. Binary and ternary connect

5.3.1. Binary connect revisited

In [**39**], we introduced a weight binarization technique which removes multiplications in the forward pass. We summarize this approach in this subsection, and introduce an extension to it in the next.

Consider a neural network layer with N input and M output units. The forward computation is $\mathbf{y} = h(W\mathbf{x} + \mathbf{b})$ where W and $\mathbf{b}$ are weights and biases, respectively, h is the activation function, and $\mathbf{x}$ and $\mathbf{y}$ are the layer's inputs and outputs. If we choose ReLU as h, there will be no multiplications in computing the activation function, thus all multiplications reside in the matrix product $W\mathbf{x}$. For each input vector x, NM floating point multiplications are needed.

Binary connect eliminates these multiplications by stochastically sampling weights to be -1 or 1. Full precision weights $\bar{w}$ are kept in memory as reference, and each time when y is needed, we sample a stochastic weight matrix W according to $\bar{w}$. For each element of the sampled matrix W, the probability of getting a 1 is proportional to how "close" its corresponding entry in $\bar{w}$ is to 1. i.e.,

$$P(W_{ij} = 1) = \frac{\bar{w}_{ij} + 1}{2}; \;\; P(W_{ij} = -1) = 1 - P(W_{ij} = 1) \tag{5.3.1}$$

It is necessary to add some edge constraints to $\bar{w}$. To ensure that $P(W_{ij} = 1)$ lies in a reasonable range, values in $\bar{w}$ are forced to be a real value in the interval [-1, 1]. If during the updates any of its value grows beyond that interval, we set it to be its corresponding edge values -1 or 1. That way floating point multiplications become sign changes.

A remaining question concerns the use of multiplications in the random number generator involved in the sampling process. Sampling an integer has to be faster than multiplication for the algorithm to be worth it. To be precise, in most cases we are doing mini-batch learning and the sampling process is performed only once for the whole mini-batch. Normally the batch size B varies up to several hundreds. So, as long as one sampling process is significantly faster than B times of multiplications, it is still worth it. Fortunately, efficiently generating random numbers has been studied in [**80, 161**]. Also, it is possible to get random numbers according to real random processes, like CPU temperatures, etc. We are not going into the details of random number generation as this is not the focus of this paper.

5.3.2. Ternary connect

The binary connect introduced in the former subsection allows weights to be -1 or 1. However, in a trained neural network, it is common to observe that many learned weights are zero or close to zero. Although the stochastic sampling process would allow the mean value of sampled weights to be zero, this suggests that it may be beneficial to explicitly allow weights to be zero.

To allow weights to be zero, some adjustments are needed for Eq. 5.3.1. We split the interval of [-1, 1], within which the full precision weight value $\bar{w}_{ij}$ lies, into two sub-intervals: $[-1, 0]$ and $(0, 1]$. If a weight value $\bar{w}_{ij}$ drops into one of them, we sample $\bar{w}_{ij}$ to be the two edge values of that interval, according to their distance from $\bar{w}_{ij}$, i.e., if $\bar{w}_{ij} > 0$:

$$P(W_{ij} = 1) = \bar{w}_{ij}; \;\; P(W_{ij} = 0) = 1 - \bar{w}_{ij} \tag{5.3.2}$$

and if $\bar{w}_{ij} <= 0$:

$$P(W_{ij} = -1) = -\bar{w}_{ij}; \;\; P(W_{ij} = 0) = 1 + \bar{w}_{ij} \tag{5.3.3}$$

Like binary connect, ternary connect also eliminates all multiplications in the forward pass.

5.4. Quantized back propagation

In the former section we described how multiplications can be eliminated from the forward pass. In this section, we propose a way to eliminate multiplications from the backward pass.

Suppose the i-th layer of the network has N input and M output units, and consider an error signal δ propagating downward from its output. The updates for weights and biases would be the outer product of the layer's input and the error signal:

$$\Delta W = \eta \left[\delta \odot h^{'} \left(W\mathbf{x} + b\right)\right] \mathbf{x}^T \tag{5.4.1}$$

$$\Delta b = \eta \left[\delta \odot h^{'} \left(W\mathbf{x} + b\right)\right] \tag{5.4.2}$$

where η is the learning rate, and $\mathbf{x}$ the input to the layer. The operator $\odot$ stands for element-wise multiply. While propagating through the layers, the error signal δ needs to be updated, too. Its update taking into account the next layer below takes the form:

$$\delta = \left[W^T \delta\right] \odot h^{'} \left(W\mathbf{x} + b\right) \tag{5.4.3}$$

There are 3 terms that appear repeatedly in Eqs. 5.4.1 to 5.4.3: $\delta, h^{'} \left(W\mathbf{x} + b\right)$ and $\mathbf{x}$. The latter two terms introduce matrix outer products. To eliminate multiplications, we can quantize one of them to be an integer power of 2, so that multiplications involving that term become binary shifts. The expression $h^{'} \left(W\mathbf{x} + b\right)$ contains downflowing gradients, which are largely determined by the cost function and network parameters, thus it is hard to bound its values. However, bounding the values is essential for quantization because we need to supply a fixed number of bits for each sampled value, and if that value varies too much, we will need too many bits for the exponent. This, in turn, will result in the need for more bits to store the sampled value and unnecessarily increase the required amount of computation.

While $h^{'} \left(W\mathbf{x} + b\right)$ is not a good choice for quantization, $\mathbf{x}$ is a better choice, because it is the hidden representation at each layer, and we know roughly the distribution of each layer's activation.

Our approach is therefore to eliminate multiplications in Eq. 5.4.1 by quantizing each entry in $\mathbf{x}$ to an integer power of 2. That way the outer product in Eq. 5.4.1 becomes a

series of bit shifts. Experimentally, we find that allowing a maximum of 3 to 4 bits of shift is sufficient to make the network work well. This means that 3 bits are already enough to quantize $\mathbf{x}$. As the float32 format has 24 bits of mantissa, shifting (to the left or right) by 3 to 4 bits is completely tolerable. We refer to this approach of back propagation as "quantized back propagation."

If we choose ReLU as the activation function, and since we are reusing the $(W\mathbf{x}+b)$ that was computed during the forward pass, computing the term $h^{'}(W\mathbf{x}+b)$ involves no additional sampling or multiplications. In addition, quantized back propagation eliminates the multiplications in the outer product in Eq. 5.4.1. The only places where multiplications remain are the element-wise products. In Eq. 5.4.2, multiplying by η and σ requires $2 \times M$ multiplications, while in Eq. 5.4.1 we can reuse the result of Eq. 5.4.2. To update δ would need another M multiplications, thus $3 \times M$ multiplications are needed for all computations from Eqs. 5.4.1 through 5.4.3. Pseudo code in Algorithm 1 outlines how quantized back propagation is conducted.

Algorithm 1 Quantized Back Propagation (QBP). C is the cost function. binarize(W) and clip(W) stands for binarize and clip methods. L is the number of layers.

Require: a deep model with parameters W, b at each layer. Input data x, its corresponding targets y, and learning rate η.

1: **procedure** QBP(model, x, y, η)
2:     **1. Forward propagation:**
3:     **for** each layer i in range$(1, L)$ **do**
4:         $W_b \leftarrow$ binarize(W)
5:         Compute activation a_i according to its previous layer output a_{i-1}, W_b and b.
6:     **2. Backward propagation:**
7:     Initialize output layer's error signal $\delta = \frac{\partial C}{\partial a_L}$.
8:     **for** each layer i in range$(L, 1)$ **do**
9:         Compute ΔW and Δb according to Eqs. 5.4.1 and 5.4.2.
10:         Update W: $W \leftarrow \text{clip}(W - \Delta W)$
11:         Update b: $b \leftarrow b - \Delta b$
12:         Compute $\frac{\partial C}{\partial a_{k-1}}$ by updating δ according to Eq. 5.4.3.

Like in the forward pass, most of the multiplications are used in the weight updates. Compared with standard back propagation, which would need $2MN + 3M$ multiplications at least, the amount of multiplications left is negligible in quantized back propagation. Our experiments in Section 5.5 show that this way of dramatically decreasing multiplications does not necessarily entail a loss in performance.

5.5. Experiments

We tried our approach on both fully connected networks and convolutional networks. Our implementation uses Theano [**7**]. We experimented with 3 datasets: MNIST, CIFAR10, and SVHN. In the following subsection we show the performance that these multiplier-light neural networks can achieve. Note that we didn't tune the hidden layer sizes heavily since our focus is to compare the performance of a binarized or ternerized model with its full precision counterparts, while strictly keeping the number of parameters exactly the same. In the subsequent subsections we study some of their properties, such as convergence and robustness, in more detail.

5.5.1. General performance

We tested different variations of our approach, and compare the results with [**39**] and full precision training (Table 5.1). All models are trained with stochastic gradient descent (SGD) without momentum. We use batch normalization for all the models to accelerate learning. At training time, binary (ternary) connect and quantized back propagation are used, while at test time, we use the learned full resolution weights for the forward propagation. For each dataset, all hyper-parameters are set to the same values for the different methods, except that the learning rate is adapted independently for each one.

Tab. 5.1. Performances across different datasets

	Full precision	Binary connect	Binary connect + Quantized backprop	Ternary connect + Quantized backprop
MNIST	1.33%	1.23%	1.29%	1.15%
CIFAR10	15.64%	12.04%	12.08%	12.01%
SVHN	2.85%	2.47%	2.48%	2.42%

5.5.1.1. *MNIST*

The MNIST dataset [**99**] has 50000 images for training and 10000 for testing. All images are grey value images of size 28×28 pixels, falling into 10 classes corresponding to the 10 digits. The model we use is a fully connected network with 4 layers: 784-1024-1024-1024-10. At the last layer we use the hinge loss as the cost. The training set is separated into two parts, one of which is the training set with 40000 images and the other the validation set with 10000 images. Training is conducted in a mini-batch way, with a batch size of 200.

With ternary connect, quantized backprop, and batch normalization, we reach an error rate of 1.15%. This result is better than full precision training (also with batch normalization), which yields an error rate 1.33%. If without batch normalization, the error rates rise to 1.48% and 1.67%, respectively. We also explored the performance if we sample those weights during *test time*. With ternary connect at test time, the same model (the one reaches 1.15% error rate) yields 1.49% error rate, which is still fairly acceptable. Our experimental results show that despite removing most multiplications, our approach yields a comparable (in fact, even slightly higher) performance than full precision training. The performance improvement is likely due to the regularization effect implied by the stochastic sampling.

Taking this network as a concrete example, the actual amount of multiplications in each case can be estimated precisely. Multiplications in the forward pass is obvious, and for the backward pass section 5.4 has already given an estimation. Now we estimate the amount of multiplications incurred by batch normalization. Suppose we have a pre-hidden representation h with mini-batch size B on a layer which has M output units (thus h should have shape $B \times M$), then batch normalization can be formalized as $\gamma \frac{h - mean(h)}{std(h)} + \beta$. One need to compute the $mean(h)$ over a mini-batch, which takes M multiplications, and $BM + 2M$ multiplication to compute the standard deviation $std(h)$. The fraction takes BM divisions, which should be equal to the same amount of multiplication. Multiplying that by the γ parameter, adds another BM multiplications. So each batch normalization layer takes an extra $3BM + 3M$ multiplications in the forward pass. The backward pass takes roughly twice as many multiplications in addition, if we use SGD. These amount of multiplications are the same no matter we use binarization or not. Bearing those in mind, the total amount of multiplications invoked in a mini-batch update are shown in Table 5.2. The last column

lists the ratio of multiplications left, after applying ternary connect and quantized back propagation.

Tab. 5.2. Estimated number of multiplications in MNIST net

	Full precision	Ternary connect + Quantized backprop	ratio
without BN	1.7480×10^9	1.8492×10^6	0.001058
with BN	1.7535×10^9	7.4245×10^6	0.004234

5.5.1.2. *CIFAR10*

CIFAR10 [**92**] contains images of size 32×32 RGB pixels. Like for MNIST, we split the dataset into 40000, 10000, and 10000 training-, validation-, and test-cases, respectively. We apply our approach in a convolutional network for this dataset. The network has 7 convolution/pooling layers, 1 fully connected layer and 1 classification layer. We use the hinge loss for training, with a batch size of 100. We also tried using ternary connect at test time. On the model trained by ternary connect and quantized back propagation, it yields 13.54% error rate. Similar to what we observed in the fully connected network, binary (ternary) connect and quantized back propagation yield a slightly higher performance than ordinary SGD.

5.5.1.3. *SVHN*

The Street View House Numbers (SVHN) dataset [**129**] contains RGB images of house numbers. It contains more than 600,000 images in its extended training set, and roughly 26,000 images in its test set. We remove 6,000 images from the training set for validation. We use 7 layers of convolution/pooling, 1 fully connected layer, and 1 classification layer. Batch size is also set to be 100. The performances we get is consistent with our results on CIFAR10. Extending the ternary connect mechanism to its test time yields 2.99% error rate on this dataset. Again, it improves over ordinary SGD by using binary (ternary) connect and quantized back propagation.

5.5.2. Convergence

Taking the convolutional networks on CIFAR10 as a test-bed, we now study the learning behaviour in more detail. Figure 5.1 shows the performance of the model in terms of test set errors during training. The figure shows that binarization makes the network converge slower than ordinary SGD, but yields a better optimum after the algorithm converges. Compared with binary connect (red line), adding quantization in the error propagation (yellow line) doesn't hurt the model accuracy at all. Moreover, having ternary connect combined with quantized back propagation (green line) surpasses all the other three approaches.

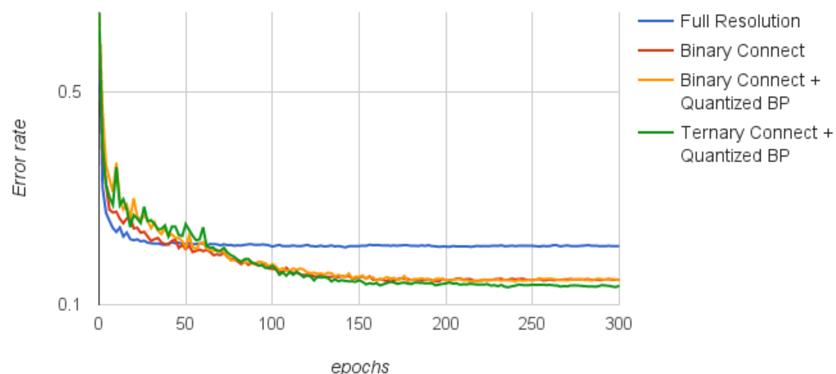


Fig. 5.1. Test set error rate at each epoch for ordinary back propagation, binary connect, binary connect with quantized back propagation, and ternary connect with quantized back propagation. Vertical axis is represented in logarithmic scale.

5.5.3. The effect of bit clipping

In Section 5.4 we mentioned that quantization will be limited by the number of bits we use. The maximum number of bits to shift determines the amount of memory needed, but it also determines in what range a single weight update can vary. Figure 5.2 shows the model performance as a function of the maximum allowed bit shifts. These experiments are conducted on the MNIST dataset, with the aforementioned fully connected model. For each case of bit clipping, we repeat the experiment for 10 times with different initial random instantiations.

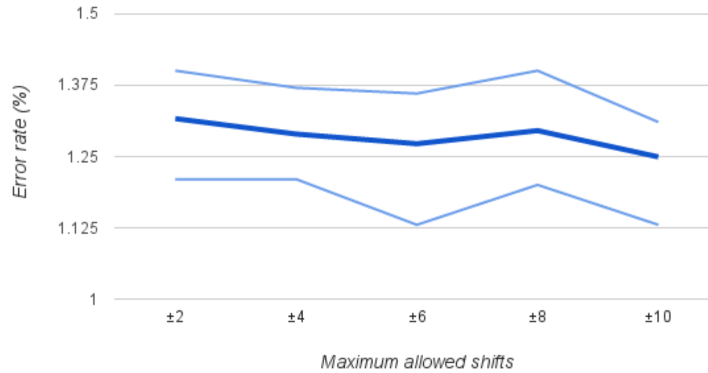


Fig. 5.2. Model performance as a function of the maximum bit shifts allowed in quantized back propagation. The dark blue line indicates mean error rate over 10 independent runs, while light blue lines indicate their corresponding maximum and minimum error rates.

The figure shows that the approach is not very sensible to the number of bits used. The maximum allowed shift in the figure varies from 2 bits to 10 bits, and the performance remains roughly the same. Even by restricting bit shifts to 2, the model can still learn successfully. The fact that the performance is not very sensitive to the maximum of allowed bit shifts suggests that we do not need to redefine the number of bits used for quantizing $\mathbf{x}$ for different tasks, which would be an important practical advantage.

The $\mathbf{x}$ to be quantized is not necessarily distributed symmetrically around 2. For example, Figure 5.3 shows the distribution of $\mathbf{x}$ at each layer in the middle of training. The maximum amount of shift to the left does not need to be the same as that on the right. A more efficient way is to use different values for the maximum left shift and the maximum right shift. Bearing that in mind, we set it to 3 bits maximum to the right and 4 bits to the left.

5.6. Conclusion and future work

We proposed a way to eliminate most of the floating point multiplications used during training a feedforward neural network. This could make it possible to dramatically accelerate the training of neural networks by using dedicated hardware implementations.

A somewhat surprising fact is that instead of damaging prediction accuracy the approach tends improve it, which is probably due to several facts. First is the regularization effect

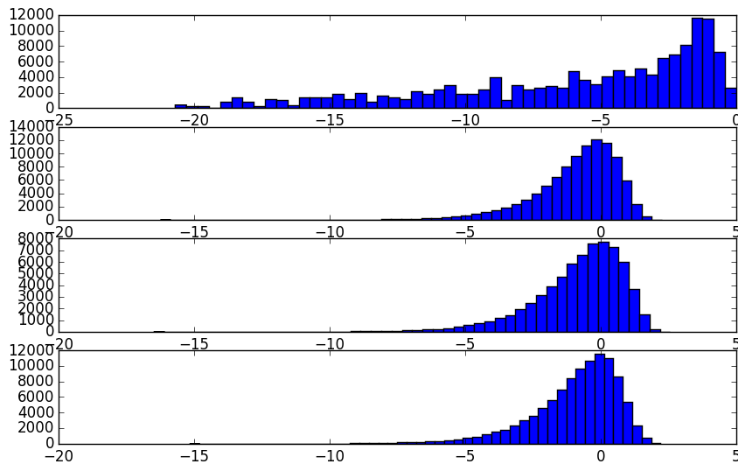


Fig. 5.3. Histogram of representations at each layer while training a fully connected network for MNIST. The figure represents a snap-shot in the middle of training. Each subfigure, from bottom up, represents the histogram of hidden states from the first layer to the last layer. The horizontal axes stand for the exponent of the layers' representations, i.e., $\log_2 \mathbf{x}$.

that the stochastic sampling process entails. Noise injection brought by sampling the weight values can be viewed as a regularizer, and that improves the model generalization. The second fact is low precision weight values. Basically, the generalization error bounds for neural nets depend on the weights precision. Low precision prevents the optimizer from finding solutions that require a lot of precision, which correspond to very thin (high curvature) critical points, and these minima are more likely to correspond to overfitted solutions then broad minima (there are more functions that are compatible with such solutions, corresponding to a smaller description length and thus better generalization). Similarly, [**128**] adds noise into gradients, which makes the optimizer prefer large-basin areas and forces it to find broad minima. It also lowers the training loss and improves generalization.

Directions for future work include exploring actual implementations of this approach (for example, using FPGA), seeking more efficient ways of binarization, and the extension to recurrent neural networks.

Chapter 6

Prologue to Second Article

6.1. Article Details

A Structured Self-Attentive Sentence Embedding. Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, Yoshua Bengio, *5th International Conference on Learning Representations, 2017*

Personal Contribution. I came up with the idea of introducing the attention mechanism for sentence embedding as well as the multi-row matrix representation. I received lots of important suggestions and discussions from Yoshua Bengio, Bowen Zhou, Bing Xiang and Mo Yu. I wrote up the code base, and Minwei Feng helped me run experiments on Yelp and Age datasets. I rendered all the figures and tables in the paper. The paper was largely written by me, and the others have helped improving and polishing the paper.

6.2. Context

Obtaining satisfying representations of phrases and sentences remained a research front when the paper was published. There were in general two categories of models, one is through unsupervised learning [**89, 71, 149, 150**]. The other category focuses on obtaining sentence embeddings from models trained specifically for a certain task [**75, 35, 150, 84**]. A common approach in all of the methods is that they try to create a fixed-size vector representation from the sentence. Sentence representations with a more sophisticated structure was not extensively explored yet. In other NLP tasks such as language modelling [**27**], question answering [**102**], and language inference [**134**], various forms of attention variations have been proposed, mostly under the name of "intra-attention".

6.3. Contributions

Our model exhibits a way to extract structured, interpretable sentence embeddings by introducing self-attention. The name of "self-attention" was new, but related works under similar idea have shown up in other NLP tasks such as question answering [**102**] and word embedding [**104**]. The matrix structure of the sentence embedding as well as the penalization term has shed light on alternative forms for sentence embedding.

6.4. Recent Developments

As of writing, the paper has accumulated 437 citations. Self-attention continues to evolve into more advanced forms in later publications, resulting in some very successful models such as the Transformer [**162**]. It was also found effective in other settings such as graph attention networks [**163**]. As for structured representations for sentences, there are two trends going in different directions. The first is using sequential models such as LSTM or attention based models such as Transformer to build flat representations with a variable size, usually through unsupervised learning. Models that fall into this category include ELMo [**137**] and BERT [**47**]. The other direction tries to learn or incorporate the tree structure of natural language syntax through various losses, some of them are supervised [**15, 155**], while others are unsupervised [**144, 173**].

Chapter 7

A Structured Self-Attentive Sentence Embedding

7.1. Introduction

Much progress has been made in learning semantically meaningful distributed representations of individual words, also known as word embeddings [**8, 118**]. On the other hand, much remains to be done to obtain satisfying representations of phrases and sentences. Those methods generally fall into two categories. The first consists of universal sentence embeddings usually trained by unsupervised learning [**71**]. This includes SkipThought vectors [**89**], ParagraphVector [**97**], recursive auto-encoders [**149, 150**], Sequential Denoising Autoencoders (SDAE), FastSent [**71**], etc.

The other category consists of models trained specifically for a certain task. They are usually combined with downstream applications and trained by supervised learning. One generally finds that specifically trained sentence embeddings perform better than generic ones, although generic ones can be used in a semi-supervised setting, exploiting large unlabeled corpora. Several models have been proposed along this line, by using recurrent networks [**75, 35**], recursive networks [**150**] and convolutional networks [**84, 49, 86**] as an intermediate step in creating sentence representations to solve a wide variety of tasks including classification and ranking [**179, 133, 156, 56**]. A common approach in previous methods consists in creating a simple vector representation by using the final hidden state of the RNN or the max (or average) pooling from either RNNs hidden states or convolved n-grams. Additional works have also been done in exploiting linguistic structures such as parse and dependence trees to improve sentence representations [**109, 123, 155**].

For some tasks people propose to use attention mechanism on top of the CNN or LSTM model to introduce extra source of information to guide the extraction of sentence embedding [**50**]. However, for some other tasks like sentiment classification, this is not directly applicable since there is no such extra information: the model is only given one single sentence as input. In those cases, the most common way is to add a max pooling or averaging step across all time steps [**100**], or just pick up the hidden representation at the last time step as the encoded embedding [**115**].

A common approach in many of the aforementioned methods consists of creating a simple vector representation by using the final hidden state of the RNN or the max (or average) pooling from either RNNs hidden states or convolved n-grams. We hypothesize that carrying the semantics along all time steps of a recurrent model is relatively hard and not necessary. We propose a self-attention mechanism for these sequential models to replace the max pooling or averaging step. Different from previous approaches, the proposed self-attention mechanism allows extracting different aspects of the sentence into multiple vector representations. It is performed on top of an LSTM in our sentence embedding model. This enables attention to be used in those cases when there are no extra inputs. In addition, due to its direct access to hidden representations from previous time steps, it relieves some long-term memorization burden from LSTM. As a side effect coming together with our proposed self-attentive sentence embedding, interpreting the extracted embedding becomes very easy and explicit.

Section 7.2 details on our proposed self-attentive sentence embedding model, as well as a regularization term we proposed for this model, which is described in Section 7.2.2. We also provide a visualization method for this sentence embedding in section 7.2.3. We then evaluate our model in author profiling, sentiment classification and textual entailment tasks in Section 7.4.

7.2. Approach

7.2.1. Model

The proposed sentence embedding model consists of two parts. The first part is a bidirectional LSTM, and the second part is the self-attention mechanism, which provides a set of summation weight vectors for the LSTM hidden states. These set of summation weight vectors are dotted with the LSTM hidden states, and the resulting weighted LSTM hidden

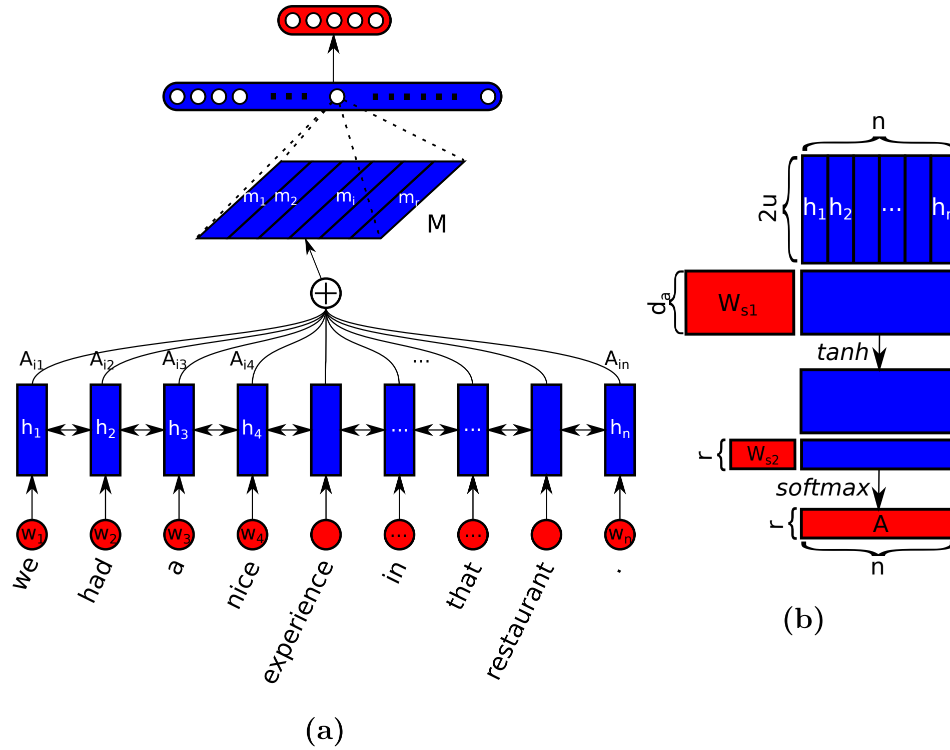


Fig. 7.1. A sample model structure showing the sentence embedding model combined with a fully connected and softmax layer for sentiment analysis (a). The sentence embedding M is computed as multiple weighted sums of hidden states from a bidirectional LSTM $(\mathbf{h_1}, ..., \mathbf{h_n})$, where the summation weights $(A_{i1}, ..., A_{in})$ are computed in a way illustrated in (b). Blue colored shapes stand for hidden representations, and red colored shapes stand for weights, annotations, or input/output.

states are considered as an embedding for the sentence. It can be combined with, for example, a multilayer perceptron to be applied on a downstream application. Figure 7.1 shows an example when the proposed sentence embedding model is applied to sentiment analysis, combined with a fully connected layer and a softmax layer. Besides using a fully connected layer, we also proposes an approach that prunes weight connections by utilizing the 2-D structure of matrix sentence embedding, which is detailed in Appendix 7.6. For this section, we will use Figure 7.1 to describe our model.

Suppose we have a sentence, which has n tokens, represented in a sequence of word embeddings.

$$S = (\mathbf{w_1}, \mathbf{w_2}, \cdots \mathbf{w_n}) \tag{7.2.1}$$

Here w_i is a vector standing for a d dimentional word embedding for the i-th word in the sentence. S is thus a sequence represented as a 2-D matrix, which concatenates all the word embeddings together. S should have the shape n-by-d.

Now each entry in the sequence S are independent with each other. To gain some dependency between adjacent words within a single sentence, we use a bidirectional LSTM to process the sentence:

$$\overrightarrow{h_t} = \overrightarrow{LSTM}(w_t, \overrightarrow{h_{t-1}}) \tag{7.2.2}$$

$$\overleftarrow{h_t} = \overleftarrow{LSTM}(w_t, \overleftarrow{h_{t+1}}) \tag{7.2.3}$$

And we concatenate each $\overrightarrow{h_t}$ with $\overleftarrow{h_t}$ to obtain a hidden state h_t. Let the hidden unit number for each unidirectional LSTM be u. For simplicity, we note all the n h_ts as H, who have the size n-by-$2u$.

$$H = (\mathbf{h_1}, \mathbf{h_2}, \cdots \mathbf{h_n}) \tag{7.2.4}$$

Our aim is to encode a variable length sentence into a fixed size embedding. We achieve that by choosing a linear combination of the n LSTM hidden vectors in H. Computing the linear combination requires the self-attention mechanism. The attention mechanism takes the whole LSTM hidden states H as input, and outputs a vector of weights $\mathbf{a}$:

$$\mathbf{a} = softmax\left(\mathbf{w_{s2}} tanh\left(W_{s1} H^T\right)\right) \tag{7.2.5}$$

Here W_{s1} is a weight matrix with a shape of d_a-by-$2u$. and $\mathbf{w_{s2}}$ is a vector of parameters with size d_a, where d_a is a hyperparameter we can set arbitrarily. Since H is sized n-by-$2u$, the annotation vector a will have a size n. the $softmax()$ ensures all the computed weights sum up to 1. Then we sum up the LSTM hidden states H according to the weight provided by $\mathbf{a}$ to get a vector representation $\mathbf{m}$ of the input sentence.

This vector representation usually focuses on a specific component of the sentence, like a special set of related words or phrases. So it is expected to reflect an aspect, or component of the semantics in a sentence. However, there can be multiple components in a sentence that together forms the overall semantics of the whole sentence, especially for long sentences. (For example, two clauses linked together by an "and.") Thus, to represent the overall semantics of the sentence, we need multiple $\mathbf{m}$'s that focus on different parts of the sentence. Thus we need to perform multiple hops of attention. Say we want r different parts to be extracted

from the sentence, with regard to this, we extend the $\mathbf{w_{s2}}$ into a r-by-d_a matrix, note it as W_{s2}, and the resulting annotation vector $\mathbf{a}$ becomes annotation matrix A. Formally,

$$A = softmax\left(W_{s2} tanh\left(W_{s1} H^T\right)\right) \tag{7.2.6}$$

Here the $softmax(\dot{)}$ is performed along the second dimension of its input. We can deem Equation 7.2.6 as a 2-layer MLP without bias, whose hidden unit numbers is d_a, and parameters are $\{W_{s2}, W_{s1}\}$.

The embedding vector m then becomes an r-by-$2u$ embedding matrix M. We compute the r weighted sums by multiplying the annotation matrix A and LSTM hidden states H, the resulting matrix is the sentence embedding:

$$M = AH \tag{7.2.7}$$

7.2.2. Penalization term

The embedding matrix M can suffer from redundancy problems if the attention mechanism always provides similar summation weights for all the r hops. Thus we need a penalization term to encourage the diversity of summation weight vectors across different hops of attention.

The best way to evaluate the diversity is definitely the Kullback – Leibler divergence between any 2 of the summation weight vectors. However, we found that not very stable in our case. We conjecture it is because we are maximizing a set of KL divergence (instead of minimizing only one, which is the usual case), we are optimizing the annotation matrix A to have a lot of sufficiently small or even zero values at different softmax output units, and these vast amount of zeros is making the training unstable. There is another feature that KL doesn't provide but we want, which is, we want each individual row to focus on a single aspect of semantics, so we want the probability mass in the annotation softmax output to be more focused. but with KL penalty we can't encourage that.

We hereby introduce a new penalization term which overcomes the aforementioned shortcomings. Compared to the KL divergence penalization, this term consumes only one third of the computation. We use the dot product of A and its transpose, subtracted by an identity matrix, as a measure of redundancy.

$$P = \left\|\left(AA^T - I\right)\right\|_F^{\,2} \tag{7.2.8}$$

Here $\|\bullet\|_F$ stands for the Frobenius norm of a matrix. Similar to adding an L2 regularization term, this penalization term P will be multiplied by a coefficient, and we minimize it together with the original loss, which is dependent on the downstream application.

Let's consider two different summation vectors $\mathbf{a^i}$ and $\mathbf{a^j}$ in A. Because of the softmax, all entries within any summation vector in A should sum up to 1. Thus they can be deemed as probability masses in a discrete probability distribution. For any non-diagonal elements $a_{ij}(i \neq j)$ in the AA^T matrix, it corresponds to a summation over elementwise product of two distributions:

$$0 < a_{ij} = \sum_{k=1}^{n} a_k^i a_k^j < 1 \tag{7.2.9}$$

where a_k^i and a_k^j are the k-th element in the $\mathbf{a^i}$ and $\mathbf{a^j}$ vectors, respectively. In the most extreme case, where there is no overlap between the two probability distributions $\mathbf{a^i}$ and $\mathbf{a^j}$, the correspond a_{ij} will be 0. Otherwise, it will have a positive value. On the other extreme end, if the two distributions are identical and all concentrates on one single word, it will have a maximum value of 1. We subtract an identity matrix from AA^T so that forces the elements on the diagonal of AA^T to approximate 1, which encourages each summation vector $\mathbf{a^i}$ to focus on as few number of words as possible, forcing each vector to be focused on a single aspect, and all other elements to 0, which punishes redundancy between different summation vectors.

7.2.3. Visualization

The interpretation of the sentence embedding is quite straight forward because of the existence of annotation matrix A. For each row in the sentence embedding matrix M, we have its corresponding annotation vector $\mathbf{a^i}$. Each element in this vector corresponds to how much contribution the LSTM hidden state of a token on that position contributes to. We can thus draw a heat map for each row of the embedding matrix M This way of visualization gives hints on what is encoded in each part of the embedding, adding an extra layer of interpretation. (See Figure 7.3a and 7.3b).

The second way of visualization can be achieved by summing up over all the annotation vectors, and then normalizing the resulting weight vector to sum up to 1. Since it sums up all aspects of semantics of a sentence, it yields a general view of what the embedding mostly

focuses on. We can figure out which words the embedding takes into account a lot, and which ones are skipped by the embedding. See Figure 7.3c and 7.3d.

7.3. Related work

Various supervised and unsupervised sentence embedding models have been mentioned in Section 7.1. Different from those models, our proposed method uses a new self-attention mechanism that allows it to extract different aspects of the sentence into multiple vector-representations. The matrix structure together with the penalization term gives our model a greater capacity to disentangle the latent information from the input sentence. We also do not use linguistic structures to guide our sentence representation model. Additionally, using our method we can easily create visualizations that can help in the interpretation of the learned representations.

Some recent work have also proposed supervised methods that use intra/self-sentence attention. [**104**] proposed an attention based model for word embedding, which calculates an attention weight for each word at each possible position in the context window. However this method cannot be extended to sentence level embeddings since one cannot exhaustively enumerate all possible sentences. [**107**] proposes a sentence level attention which has a similar motivation but done differently. They utilize the mean pooling over LSTM states as the attention source, and use that to re-weight the pooled vector representation of the sentence.

Apart from the previous 2 variants, we want to note that [**102**] proposed a same self attention mechanism for question encoding in their factoid QA model, which is concurrent to our work. The difference lies in that their encoding is still presented as a vector, but our attention produces a matrix representation instead, with a specially designed penalty term. We applied the model for sentiment anaysis and entailment, and their model is for factoid QA.

The LSTMN model [**27**] also proposed a very successful intra-sentence level attention mechanism, which is later used by [**134**]. We see our attention and theirs as having different granularities. LSTMN produces an attention vector for each of its hidden states during the recurrent iteration, which is sort of an "online updating" attention. It's more fine-grained, targeting at discovering lexical correlations between a certain word and its previous words.

On the contrary, our attention mechanism is only performed once, focuses directly on the semantics that makes sense for discriminating the targets. It is less focused on relations between words, but more on the semantics of the whole sentence that each word contributes to. Computationally, our method also scales up with the sentence length better, since it doesn't require the LSTM to compute an annotation vector over all of its previous words each time when the LSTMN computes its next step.

7.4. Experimental results

We first evaluate our sentence embedding model by applying it to several different tasks. Specifically, we choose 3 different tasks that can be framed as a classification task, namely author profiling, sentiment analysis, and textual entailment. The 3 datasets are the Age dataset, the Yelp dataset, and the Stanford Natural Language Inference (SNLI) Corpus, respectively. Then we also perform a set of exploratory experiments to validate properties of various aspects for our sentence embedding model.

7.4.1. Author profiling

The Author Profiling dataset[1] consists of Twitter tweets in English, Spanish, and Dutch. For some of the tweets, it also provides an age and gender of the user when writing the tweet. The age range are split into 5 classes: 18-24, 25-34, 35-49, 50-64, 65+. We use English tweets as input, and use those tweets to predict the age range of the user. Since we are predicting the age of users, we refer to it as Age dataset in the rest of our paper. We randomly selected 68485 tweets as training set, 4000 for development set, and 4000 for test set. Performances are also chosen to be classification accuracy.

Tab. 7.1. Performance Comparison of Different Models on Yelp and Age Dataset

Models	Yelp	Age
BiLSTM + Max Pooling + MLP	61.99%	77.40%
CNN + Max Pooling + MLP	62.05%	78.15%
Our Model	**64.21%**	**80.45%**

[1] http://pan.webis.de/clef16/pan16-web/author-profiling.html

We compare our model with two baseline models: biLSTM and CNN. The hyperparameters are tuned separately for our model and the two baseline models, except that we force the biLSTM part the same size in our model and the biLSTM baseline. Our hyperparameter search mainly covers the hidden layer size in the output MLP, the dropout rate, attention hidden layer size, the number of attention hops, and L2 regularization. The biLSTM hidden state size is set to be 300D in each dimension, and we didn't heavily search over it. The biLSTM model uses a bidirectional LSTM, and use max pooling across all LSTM hidden states to get the sentence embedding vector, then use a 2-layer ReLU output MLP with 3000 hidden states to output the classification result. The CNN model uses the same scheme, but substituting biLSTM with 1 layer of 1-D convolutional network. During training we use 0.5 dropout on the MLP and 0.0001 L2 regularization. We use stochastic gradient descent as the optimizer, with a learning rate of 0.06, batch size 16. For biLSTM, we also clip the norm of gradients to be between -0.5 and 0.5. We searched hyperparameters in a wide range and find the aforementioned set of hyperparameters yields the highest accuracy.

For our model, we use the same settings as what we did in biLSTM. We also use a 2-layer ReLU output MLP, but with 2000 hidden units. In addition, our self-attention MLP has a hidden layer with 350 units (the d_a in Section 7.2), we choose the matrix embedding to have 30 rows (the r), and a coefficient of 1 for the penalization term.

We train all the three models until convergence and select the corresponding test set performance according to the best development set performance. Our results show that the model outperforms both of the biLSTM and CNN baselines by a significant margin.

7.4.2. Sentiment analysis

We choose the Yelp dataset[2] for sentiment analysis task. It consists of 2.7M yelp reviews, we take the review as input and predict the number of stars the user who wrote that review assigned to the corresponding business store. We randomly select 500K review-star pairs as training set, and 2000 for development set, 2000 for test set. As preprocessing steps, we first lowercase the initial character in each of the sentences, remove the punctuation from the text. We then tokenize and stem the review texts with Stanford tokenizer [3] and stemmer

[2] https://www.yelp.com/dataset_challenge

[3] https://nlp.stanford.edu/static/software/tokenizer.shtml.

- if I can give this restaurant a 0 I will we be just ask our waitress leave because someone with a reservation be wait for our table my father and father-in-law be still finish up their coffee and we have not yet finish our dessert I have never be so humiliated do not go to this restaurant their food be mediocre at best if you want excellent Italian in a small intimate restaurant go to dish on the South Side I will not be go back
- this place suck the food be gross and taste like grease I will never go here again ever sure the entrance look cool and the waiter can be very nice but the food simply be gross taste like cheap 99cent food do not go here the food shot out of me quick then it go in
- everything be pre cook and dry its crazy most Filipino people be used to very cheap ingredient and they do not know quality the food be disgusting I have eat at least 20 different Filipino family home this not even mediocre
- seriously f *** this place disgust food and shitty service ambience be great if you like dine in a hot cellar engulf in stagnate air truly it be over rate over price and they just under deliver forget try order a drink here it will take forever get and when it finally do arrive you will be ready pass out from heat exhaustion and lack of oxygen how be that a head change you do not even have pay for it I will not disgust you with the detailed review of everything I have try here but make it simple it all suck and after you get the bill you will be walk out with a sore ass save your money and spare your self the disappointment
- i be so angry about my horrible experience at Medusa today my previous visit be amaze 5/5 however my go to out of town and I land an appointment with Stephanie I go in with a picture of roughly what I want and come out look absolutely nothing like it my hair be a horrible ashy blonde not anywhere close to the platinum blonde I request she will not do any of the pop of colour I want and even after specifically tell her I do not like blunt cut my hair have lot of straight edge she do not listen to a single thing I want and when I tell her I be unhappy with the colour she basically tell me I be wrong and I have do it this way no no I do not if I can go from Little Mermaid red to golden blonde in 1 sitting that leave my hair fine I shall be able go from golden blonde to a shade of platinum blonde in 1 sitting thanks for ruin my New Year's with 1 the bad hair job I have ever have

(a) 1 star reviews

- i really enjoy Ashley and Ami salon she do a great job be friendly and professional I usually get my hair do when I go to MI because of the quality of the highlight and the price the price be very affordable the highlight fantastic thank Ashley i highly recommend you and ill be back
- love this place it really be my favorite restaurant in Charlotte they use charcoal for their grill and you can taste it steak with chimichurri be always perfect Fried yucca cilantro rice pork sandwich and the good tres lech I have had.The desert be all incredible if you do not like it you be a mutant if you will like diabeetus try the Inca Cola
- this place be so much fun I have never go at night because it seem a little too busy for my taste but that just prove how great this restaurant be they have amazing food and the staff definitely remember us every time we be in town I love when a waitress or waiter come over and ask if you want the cab or the Pinot even when there be a rush and the staff be run around like crazy whenever I grab someone they instantly smile acknowlegde us the food be also killer I love when everyone know the special and can tell you they have try them all and what they pair well with this be a first last stop whenever we be in Charlotte and I highly recommend them
- great food and good service .... what else can you ask for everything that I have ever try here have be great
- first off I hardly remember waiter name because its rare you have an unforgettable experience the day I go I be celebrate my birthday and let me say I leave feel extra special our waiter be the best ever Carlos and the staff as well I be with a party of 4 and we order the potato salad shrimp cocktail lobster amongst other thing and boy be the food great the lobster be the good lobster I have ever eat if you eat a dessert I will recommend the cheese cake that be also the good I have ever have it be expensive but so worth every penny I will definitely be back there go again for the second time in a week and it be even good ...... this place be amazing

(b) 5 star reviews

Fig. 7.2. Heatmap of Yelp reviews with the two extreme score.

[4]. We use 100 dimensional word2vec as initialization for word embeddings, and tune the embedding during training across all of our experiments. The target number of stars is an integer number in the range of $[1, 5]$, inclusive. We are treating the task as a classification task, i.e., classify a review text into one of the 5 classes. We use classification accuracy as a measurement.

---

[4]https://github.com/stanfordnlp/CoreNLP

For the two baseline models, we use the same setting as what we used for Author Profiling dataset, except that we are using a batch size of 32 instead. For our model, we are also using the same setting, except that we choose the hidden unit numbers in the output MLP to be 3000 instead. We also observe a significant performance gain comparining to the two baselines. (Table 7.1)

As an interpretation of the learned sentence embedding, we use the second way of visualization described in Section 7.2.3 to plot heat maps for some of the reviews in the dataset. We randomly select 5 examples of negative (1 star) and positive (5 stars) reviews from the test set, when the model has a high confidence (> 0.8) in predicting the label. As shown in Figure 7.2, we find that the model majorly learns to capture some key factors in the review that indicate strongly on the sentiment behind the sentence. For most of the short reviews, the model manages to capture all the key factors that contribute to an extreme score, but for longer reviews, the model is still not able to capture all related factors. For example, in the 3rd review in Figure 7.2b), it seems that a lot of focus is spent on one single factor, i.e., the "so much fun", and the model puts a little amount of attention on other key points like "highly recommend", "amazing food", etc.

Tab. 7.2. Test Set Performance Compared to other Sentence Encoding Based Methods in SNLI Datset

Model	**Test Accuracy**
300D LSTM encoders [**15**]	80.6%
600D (300+300) BiLSTM encoders [**107**]	83.3%
300D Tree-based CNN encoders [**122**]	82.1%
300D SPINN-PI encoders [**15**]	83.2%
300D NTI-SLSTM-LSTM encoders [**125**]	83.4%
1024D GRU encoders with SkipThoughts pre-training [**164**]	81.4%
300D NSE encoders [**124**]	**84.6%**
Our method	84.4%

7.4.3. Textual entailment

We use the biggest dataset in textual entailment, the SNLI corpus [**14**] for our evaluation on this task. SNLI is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels entailment, contradiction, and neutral. The model will be given a pair of sentences, called premise and hypothesis respectively, and asked to tell if the semantics in the hypothesis are contradicting with that in the premise or not. It is also a classification task, so we measure the performance by accuracy.

We process the hypothesis and premise independently, and then extract the relation between the two sentence embeddings by using multiplicative interactions proposed in [**116**] (see Appendix 7.7 for details), and use a 2-layer ReLU output MLP with 4000 hidden units to map the hidden representation into classification results. Parameters of biLSTM and attention MLP are shared across hypothesis and premise. The biLSTM is 300 dimension in each direction, the attention MLP has 150 hidden units instead, and both sentence embeddings for hypothesis and premise have 30 rows (the r). The penalization term coefficient is set to 0.3. We use 300 dimensional GloVe [**136**] word embedding to initialize word embeddings. We use AdaGrad as the optimizer, with a learning rate of 0.01. We don't use any extra regularization methods, like dropout or L2 normalization. Training converges after 4 epochs, which is relatively fast.

As for hyperparameter tuning, we first did a hyperparameter sweep on each of the following hyperparameters: the hidden sizes of LSTM, attention MLP, and output MLP; the penalization term in Section 7.2.2; and the number of attention hops. We then train a model from scratch with its hyperparameters set to be the best performing values in each of the individual hyperparameter sweeps.

This task is a bit different from previous two tasks, in that it has 2 sentences as input. There are a bunch of ways to add inter-sentence level attention, and those attentions bring a lot of benefits. To make the comparison focused and fair, we only compare methods that fall into the sentence encoding-based models. i.e., there is no information exchanged between the hypothesis and premise before they are encoded into some distributed encoding.

We find that compared to other published approaches, our method shows a significant gain ($\geq 1\%$) to them, except for the 300D NSE encoders, which is the state-of-the-art in this

category. However, the 0.2% different is relatively small compared to the differences between other methods.

7.4.4. Exploratory experiments

In this subsection we are going to do a set of exploratory experiments to study the relative effect of each component in our model.

7.4.4.1. *Effect of penalization term*

Since the purpose of introducing the penalization term P is majorly to discourage the redundancy in the embedding, we first directly visualize the heat maps of each row when the model is presented with a sentence. We compare two identical models with the same size as detailed in Section 7.4.1 trained separately on Age dataset, one with this penalization term (where the penalization coefficient is set to 1.0) and the other with no penalty. We randomly select one tweet from the test set and compare the two models by plotting a heat map for each hop of attention on that single tweet. Since there are 30 hops of attention for each model, which makes plotting all of them quite redundant, we only plot 6 of them. These 6 hops already reflect the situation in all of the 30 hops.

From the figure we can tell that the model trained without the penalization term have lots of redundancies between different hops of attention (Figure 7.3a), resulting in putting lot of focus on the word "it" (Figure 7.3c), which is not so relevant to the age of the author. However in the right column, the model shows more variations between different hops, and as a result, the overall embedding focuses on "mail-replies spam" instead. (Figure 7.3d)

For the Yelp dataset, we also observe a similar phenomenon. To make the experiments more explorative, we choose to plot heat maps of overall attention heat maps for more samples, instead of plotting detailed heat maps for a single sample again. Figure 7.4 shows overall focus of the sentence embedding on three different reviews. We observe that with the penalization term, the model tends to be more focused on important parts of the review. We think it is because that we are encouraging it to be focused, in the diagonals of matrix AA^T (Equation 7.2.8).

To validate if these differences result in performance difference, we evaluate four models trained on Yelp and Age datasets, both with and without the penalization term. Results are

shown in Table 7.3. Consistent with what expected, models trained with the penalization term outperforms their counterpart trained without.

In SNLI dataset, although we observe that introducing the penalization term still contributes to encouraging the diversity of different rows in the matrix sentence embedding, and forcing the network to be more focused on the sentences, the quantitative effect of this

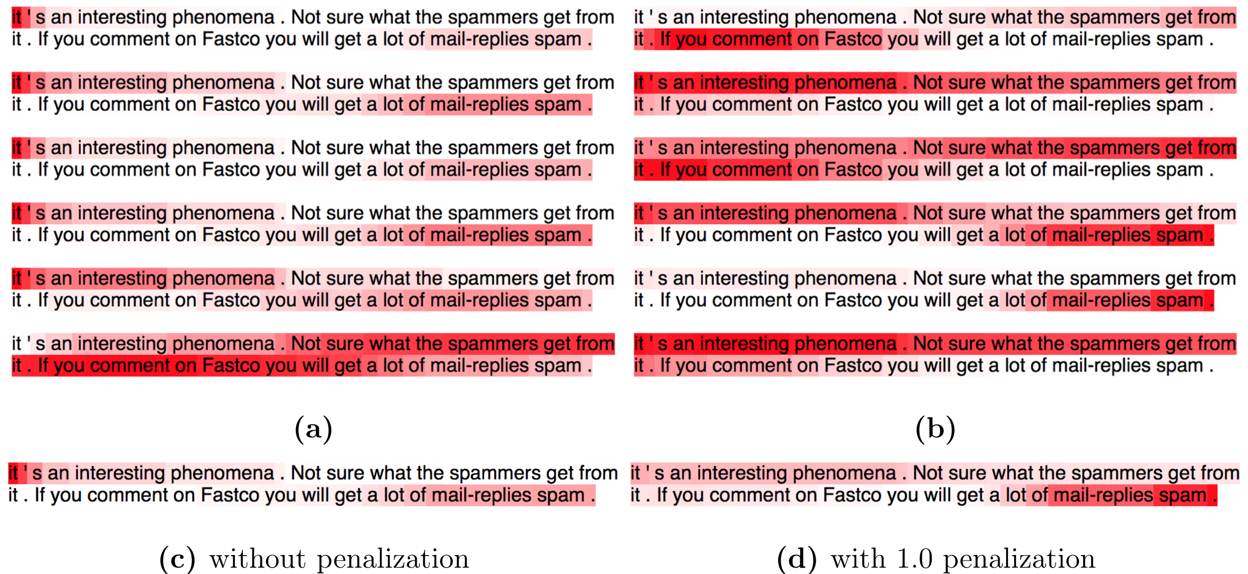


Fig. 7.3. Heat maps for 2 models trained on Age dataset. The left column is trained without the penalization term, and the right column is trained with 1.0 penalization. (a) and (b) shows detailed attentions taken by 6 out of 30 rows of the matrix embedding, while (c) and (d) shows the overall attention by summing up all 30 attention weight vectors.

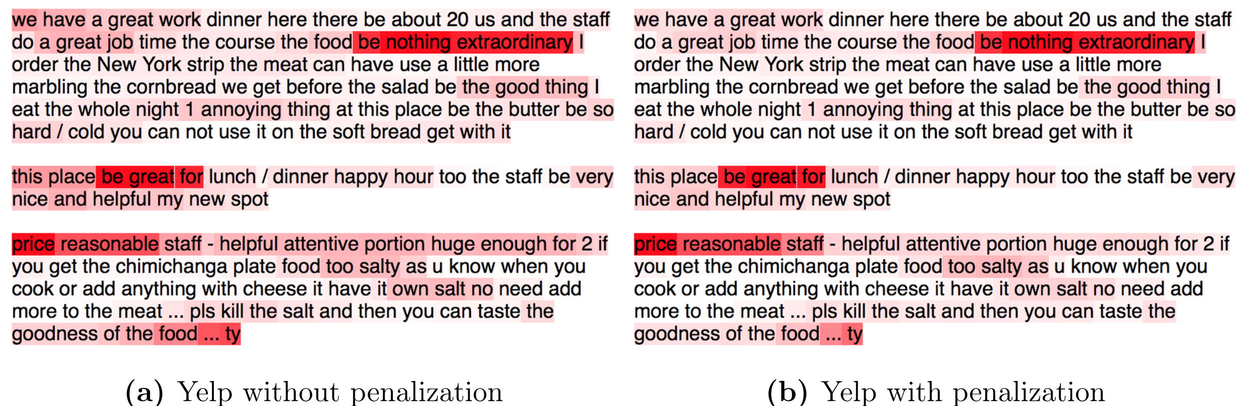


Fig. 7.4. Attention of sentence embedding on 3 different Yelp reviews. The left one is trained without penalization, and the right one is trained with 1.0 penalization.

Tab. 7.3. Performance comparision regarding the penalization term

Penalization coefficient	**Yelp**	**Age**
1.0	64.21%	80.45%
0.0	61.74%	79.27%

penalization term is not so obvious on SNLI dataset. Both models yield similar test set accuracies.

7.4.4.2. *Effect of multiple vectors*

Having multiple rows in the sentence embedding is expected to provide more abundant information about the encoded content. It makes sence to evaluate how significant the improvement can be brought by r. Taking the models we used for Age and SNLI dataset as an example, we vary r from 1 to 30 for each task, and train the resulting 10 models independently (Figure 7.5). Note that when $r = 1$, the sentence embedding reduces to a normal vector form.

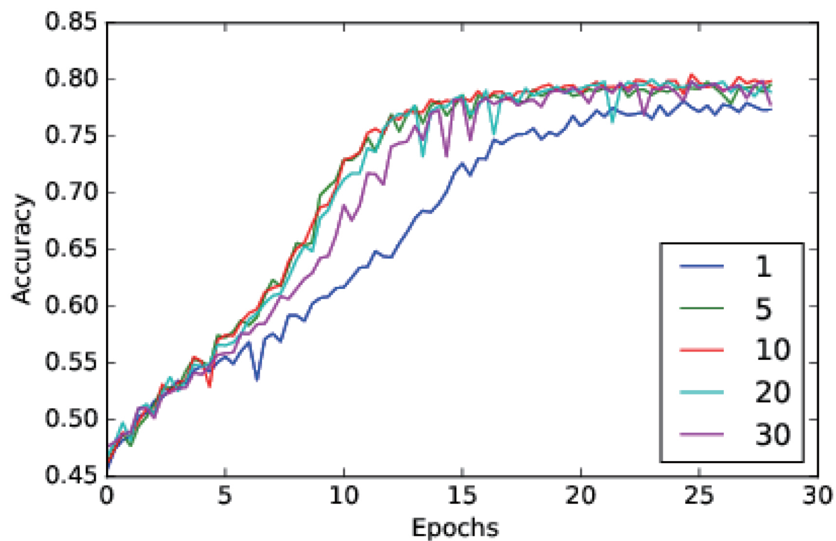
From this figure we can find that, without having multiple rows, the model performs on-par with its competitiors which use other forms of vector sentence embeddings. But there is significant difference between having only one vector for the sentence embedding and multiple vectors. The models are also quite invariant with respect to r, since in the two figures a wide range of values between 10 to 30 are all generating comparable curves.

7.5. Conclusion and discussion

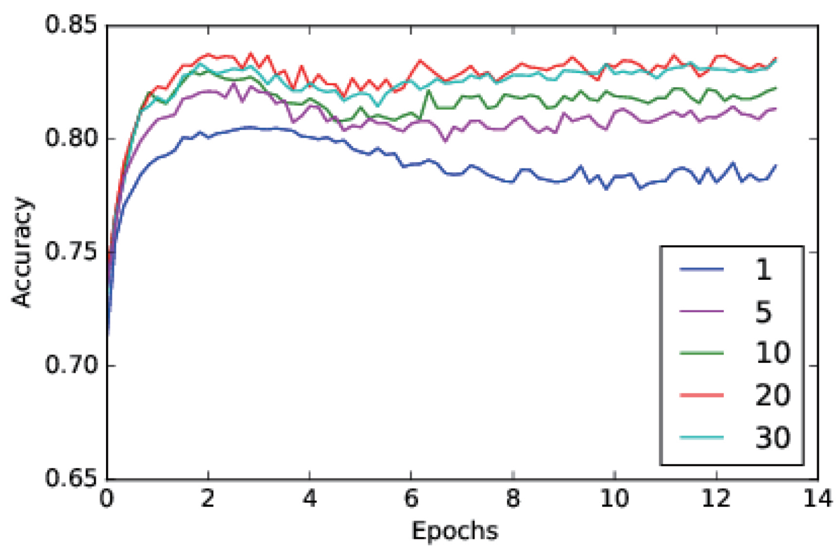
In this paper, we introduced a fixed size, matrix sentence embedding with a self-attention mechanism. Because of this attention mechanism, there is a way to interpret the sentence embedding in depth in our model. Experimental results over 3 different tasks show that the model outperforms other sentence embedding models by a significant margin.

Introducing attention mechanism allows the final sentence embedding to directly access previous LSTM hidden states via the attention summation. Thus the LSTM doesn't need to carry every piece of information towards its last hidden state. Instead, each LSTM hidden state is only expected to provide shorter term context information around each word, while the higher level semantics, which requires longer term dependency, can be picked up directly

by the attention mechanism. This setting reliefs the burden of LSTM to carry on long term dependencies. Our experiments also support that, as we observed that our model has a



(a)



(b)

Fig. 7.5. Effect of the number of rows (r) in matrix sentence embedding. The vertical axes indicates test set accuracy and the horizontal axes indicates training epochs. Numbers in the legends stand for the corresponding values of r. (a) is conducted in Age dataset and (b) is conducted in SNLI dataset.

bigger advantage when the contents are longer. Further more, the notion of summing up elements in the attention mechanism is very primitive, it can be something more complex than that, which will allow more operations on the hidden states of LSTM.

The model is able to encode any sequence with variable length into a fixed size representation, without suffering from long-term dependency problems. This brings a lot of scalability to the model: without any modification, it can be applied directly to longer contents like paragraphs, articles, etc. Though this is beyond the focus of this paper, it remains an interesting direction to explore as a future work.

As a downside of our proposed model, the current training method heavily relies on downstream applications, thus we are not able to train it in an unsupervised way. The major obstacle towards enabling unsupervised learning in this model is that during decoding, we don't know as prior how the different rows in the embedding should be divided and reorganized. Exploring all those possible divisions by using a neural network could easily end up with overfitting. Although we can still do unsupervised learning on the proposed model by using a sequential decoder on top of the sentence embedding, it merits more to find some other structures as a decoder.

7.6. Pruned MLP for Structured Matrix Sentence Embedding

As a side effect of having multiple vectors to represent a sentence, the matrix sentence embedding is usually several times larger than vector sentence embeddings. This results in needing more parameters in the subsequent fully connected layer, which connects every hidden units to every units in the matrix sentence embedding. Actually in the example shown in Figure 7.1, this fully connected layer takes around 90% percent of the parameters. See Table 7.4. In this appendix we are going to introduce a weight pruning method which, by utilizing the 2D structure of matrix embedding, is able to drastically reduce the number of parameters in the fully connected hidden layer.

Inheriting the notation used in the main paper, let the matrix embedding M has a shape of r by u, and let the fully connected hidden layer has b units. The normal fully connected hidden layer will require each hidden unit to be connected to every unit in the matrix embedding, as shown in Figure 7.1. This ends up with $r \times u \times b$ parameters in total.

However there are 2-D structures in the matrix embedding, which we should make use of. Each row (m_i in Figure 7.1) in the matrix is computed from a weighted sum of LSTM hidden states, which means they share some similarities

To reflect these similarity in the fully connected layer, we split the hidden states into r equally sized groups, with each group having p units. The i-th group is only fully connected to the i-th row in the matrix representation. All connections that connects the i-th group hidden units to other rows of the matrix are pruned away. In this way, Simillarity between different rows of matrix embedding are reflected as symmetry of connecting type in the hidden layer. As a result, the hidden layer can be interperated as also having a 2-D structute, with

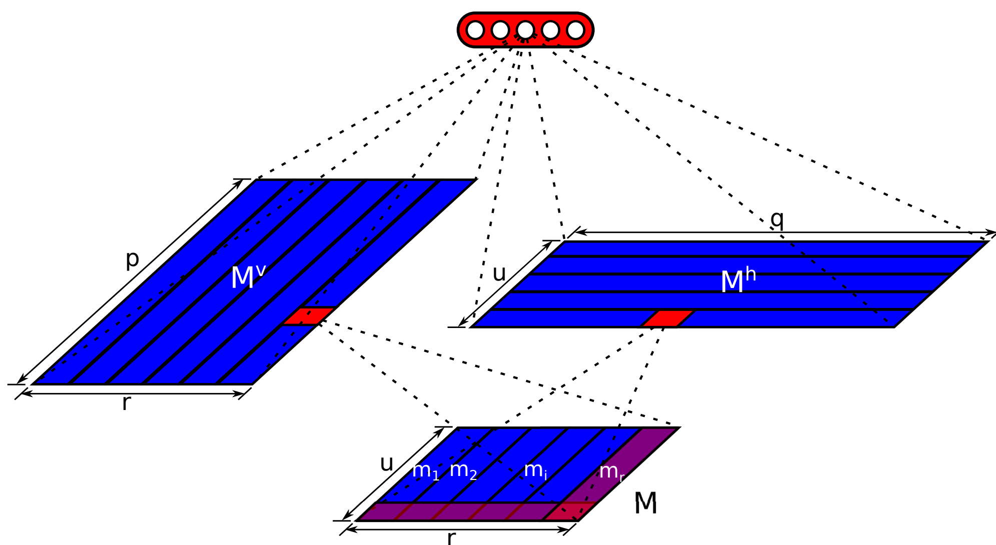


Fig. 7.6. Hidden layer with pruned weight connections. M is the matrix sentence embedding, M^v and M^h are the structured hidden representation computed by pruned weights.

Tab. 7.4. Model Size Comparison Before and After Pruning

	Hidden layer	**Softmax**	**Other Parts**	**Total**	**Accuracy**
Yelp, Original, b=3000	54M	15K	1.3M	55.3M	64.21%
Yelp, Pruned, p=150, q=10	2.7M	52.5K	1.3M	4.1M	63.86%
Age, Original, b=4000	72M	20K	1.3M	73.2M	80.45%
Age, Pruned, p=25, q=20	822K	63.75K	1.3M	2.1M	77.32%
SNLI, Original, b=4000	72M	12K	22.9M	95.0M	84.43%
SNLI, Pruned, p=300, q=10	5.6M	45K	22.9M	28.6M	83.16%

the number (r) and size (p) of groups as its two dimensions (The M^v in Figure 7.6). When the total number of hidden units are the same (i.e., $r \times p = b$), this process prunes away $(r-1)/r$ of weight values, which is a fairly large portion when r is large.

On the other dimension, another form of similarity exists too. For each vector representation m_i in M, the j-th element m_{ij} is a weighted sum of an LSTM hidden unit at different time steps. And for a certain j-th element in all vector representations, they are summed up from a same LSTM hidden unit. We can also reflect this similarity into the symmetry of weight connections by using the same pruning method we did above. Thus we will have another 2-D structured hidden states sized u-by-q, noted as M^h in Figure 7.6.

Table 7.4 takes the model we use for yelp dataset as a concrete example, and compared the number of parameters in each part of the model, both before and after pruning. We can see the above pruning method drastically reduces the model size. Note that the p and q in this structure can be adjusted freely as hyperparameters. Also, we can continue the corresponding pruning process on top of M^v and M^h over and over again, and end up with having a stack of structured hidden layers, just like stacking fully connected layers.

The subsequent softmax layer will be fully connected to both M_v and M_h, i.e., each unit in the softmax layer is connected to all units in M_v and M_h. This is not a problem since the speed of softmax is largely dependent of the number of softmax units, which is not changed.In addition, for applications like sentiment analysis and textural entailment, the softmax layer is so tiny that only contains several units.

Experimental results in the three datasets has shown that, this pruning mechanism lowers performances a bit, but still allows all three models to perform comparable or better than other models compared in the paper.

7.7. Detailed Structure of the Model for SNLI Dataset

In Section 7.4.3 we tested our matrix sentence embedding model for the textual entailment task on the SNLI dataset. Different from the former two tasks, the textual entailment task consists of a pair of sentences as input. We propose to use a set of multiplicative interactions to combine the two matrix embeddings extracted for each sentence. The form of multiplicative interaction is inspired by *Factored Gated Autoencoder* [**116**].

The overall structure of our model for SNLI is dipicted in Figure 7.7. For both hypothesis and premise, we extract their embeddings (M_h and M_p in the figure) independently, with a same LSTM and attention mechanism. The parameters of this part of model are shared (rectangles with dashed orange line in the figure).

Comparing the two matrix embeddings corresponds to the green dashed rectangle part in the figure, which computes a single matrix embedding (F_r) as the factor of semantic relation between the two sentences. To represent the relation between M_h and M_p, F_r can be connected to M_h and M_p through a three-way *multiplicative interaction*. In a three-way multiplicative interaction, the value of anyone of F_r, M_h and M_p is a function of the product of the others. This type of connection is originally introduced to extract relation between images [**116**]. Since here we are just computing the factor of relations (F_r) from M_h and M_p, it corresponds to the encoder part in the Factored Gated Autoencoder in [**116**]. We call it Gated Encoder in Figure 7.7.

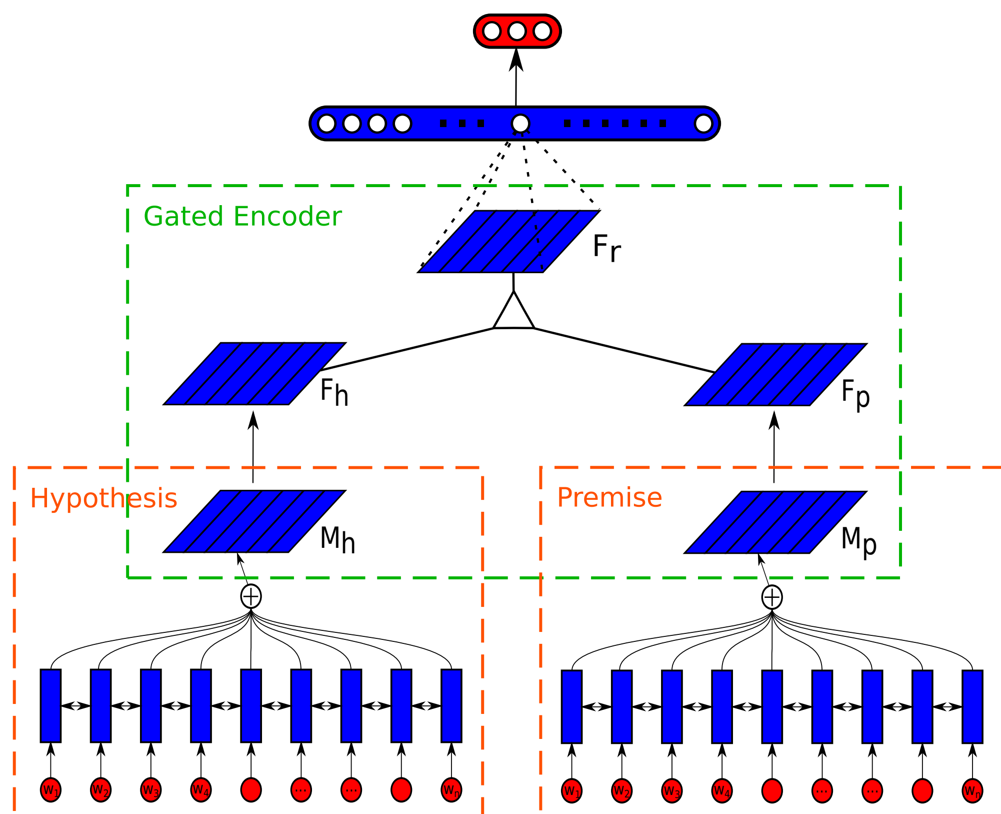


Fig. 7.7. Model structure used for textual entailment task.

First we multiply each row in the matrix embedding by a different weight matrix. Repeating it over all rows, corresponds to a batched dot product between a 2-D matrix and a 3-D weight tensor. Inheriting the name in [**116**], we call the resulting matrix as *factor*. Doing the batched dot for both hypothesis embedding and premise embedding, we have F_h and F_p, respectively.

$$F_h = batcheddot(M_h, W_{fh}) \tag{7.7.1}$$

$$F_p = batcheddot(M_p, W_{fp}) \tag{7.7.2}$$

Here W_{fh} and W_{fp} are the two weight tensors for hypothesis embedding and premise embedding.

The factor of the relation (F_r) is just an element-wise product of F_h and F_p (the triangle in the middle of Figure 7.7):

$$F_r = F_h \odot F_p \tag{7.7.3}$$

Here $\odot$ stands for element-wise product. After the F_r layer, we then use an MLP with softmax output to classify the relation into different categlories.

Chapter 8

Prologue to Third Article

8.1. Article Details

Learning Hierarchical Structures on the Fly with a Recurrent-Recursive Model for Sequences. Athul Paul Jacob*, Zhouhan Lin*, Alessandro Sordoni, Yoshua Bengio, *Assosiation of computational linguistics, 3rd workshop on representation learning for natural language processing, 2018*

Personal Contribution. (*) denotes co-first authorship. I came up with the idea of using dynamic neural network to learn a recursive structure. I implemented the prototype of the model. Athul Paul Jacob and I iterated on the prototype to significantly improve its performance. We receive important feedback from Alessandro Sordoni and Yoshua Bengio during the whole project. Athul Paul Jacob, Alessandro Sordoni and I have wrote up the paper, while Yoshua Bengio helped improve its presentation.

8.2. Context

RNNs are among the most popular models for modelling sequential data at the time when the paper was published. However, RNNs lead to the difficult-to-learn long-term dependencies because of their sequential nature [**12, 54**]. On the other hand, in many cases data comes with intrinsic structures that wasn't well utilized in sequential models. For example in language, syntax is usually analyzed in the form of a grammar tree [**34**] and in compositional semantics, single terms aggregate recursively into larger units of meaning, such as phrases and sentences [**135**].

Hierarchical structures such as trees could alleviate this problem by creating shortcuts between distant inputs, as well as simulating compositionality of the sequence. Tree structure could be used as prior knowledge [**150, 155, 15**], as supervised signal [**53, 148, 185, 184**]. More recent works have been able to learn tree structures through unsupervised learning. Structures of various quality could emerge from minimizing the negative log likelihood of the observed corpus [**172**], or by optimizing over a downstream task [**32, 180**].

On another line of research, models that learn to online-adapt their structure with respect to the data had been an emerging area of interest after Adaptive Computation Time (ACT) [**64**] was proposed. This has been found useful in convolutional networks [**57**] and recurrent networks [**82**].

8.3. Contributions

We explored the possibility of using an adaptive neural network to learn the underlying structure of data. The adaptive structure could either be inferred without supervision through reinforcement learning, or learned in a supervised manner. We also contributed a new Math Expression Evaluation (MEE) task that can be used to evaluate the performance of models learned from hierarchical data.

8.4. Recent Developments

Adaptive computation time have been explored more extensively since the publication of the paper. For example, [**57**] has explored its application on image classification. [**181**] learns to adaptively skip some of the tokens in reading text. On another thread, unsupervised learning of syntactic structures of language has been explored as well. These models took various forms, from discrete models such as unsupervised recurrent network grammar (URNNG) [**87**], to continuous models such as Parsing-Reading-Predicting networks [**144**].

Chapter 9

Learning Hierarchical Structures on the Fly with a Recurrent-Recursive Model for Sequences

9.1. Introduction

Many kinds of sequential data such as language or math expressions naturally come with a hierarchical structure. Sometimes the structure is hidden deep in the semantics of the sequence, like the syntax tree in natural language; Other times the structure is more explicit, as in math expressions, where the tree is determined by the parentheses.

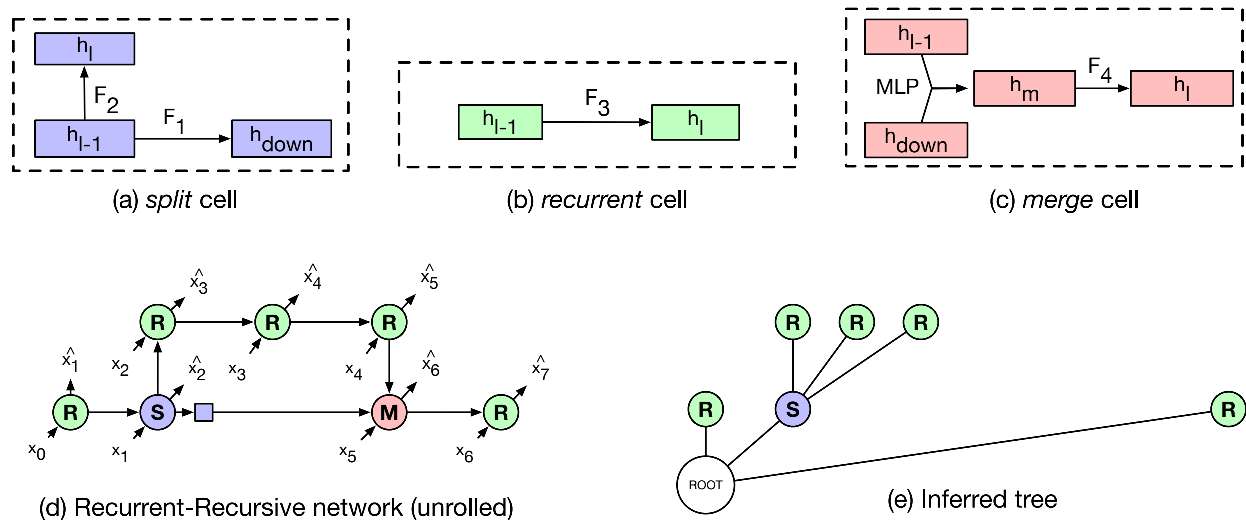


Fig. 9.1. (a) - (c) are the 3 different cells. (d) is a sample model structure resulted from a sequence of decisions. "R", "S" and "M" stand for recurrent cell, split cell, and merge cell, respectively. Note that the "S" and "M" node can take inputs in datasets where splitting and merging signals are part of the sequence. (e) is the tree inferred from (d).

Recurrent neural networks (RNNs) have shown tremendous success in modeling sequential data, such as natural language [**119, 117**]. However, RNNs process the observed data as a linear sequence of observations: the length of the computational path between any two words is a function of their position in the observed sequence, instead of their semantic or syntactic roles, leading to the appearance of difficult-to-learn long-term dependencies and stimulating research on strategies to deal with that [**12, 54, 75**]. Hierarchical, tree-like structures may alleviate this problem by creating shortcuts between distant inputs and by simulating compositionality of the sequence, compressing the sequence into higher-level abstractions. Models that use tree as prior knowledge (e.g. [**150, 155, 15**]) have shown improved performances over sequential models, validating the value of tree structure. For example, TreeLSTM [**155**] learns a bottom-up encoder, but requires the model to have access to the entire sentence as well as its parse tree before encoding it, which limits its application in some cases, e.g. language modeling. There has been various efforts to learn the tree structure as a supervised training target [**53, 148, 185, 184**], which free the model from relying on an external parser.

More recent efforts learn the best tree structure without supervision, by minimizing the log likelihood of the observed corpus, or by optimizing over a downstream task [**172**]. These models usually take advantage of a binary tree assumption on the inferred tree, for example, the Gumbel TreeLSTM [**32, 180**]. However, it also means that these models are not able to directly infer non-binary trees. Since the grammar trees are not originally binary, this is a considerable limitation.

We propose a model that reads sequences using a hierarchical, tree-structured process (Fig. 9.1): it creates a tree on-the-fly, in a top-down fashion. Our model sits in between fully recursive models that have access to the whole sequence, such as TreeLSTMs [**155**], and vanilla recurrent models that "flatten" input sequence, such as LSTMs [**143**]. At each time-step in the sequence, the model chooses either to create a new sub-tree (split), to return and merge information into the parent node (merge), or to predict the next word in the sequence (recur). On split, a new sub-tree is created which takes control on which operation to perform. Merge operations end the current computation and return a representation of the current sub-tree to the parent node, which composes it with the previously available information on the same level. Recurrent operations use information from the siblings to

perform predictions. Operations at every level in the tree use shared weights, thus sharing the recursive nature of TreeLSTMs. In contrast to TreeLSTMs however, the tree is created on-the-fly, which establishes skip-connections with previous tokens in the sequence and forms compositional representations of the past. The branching decisions can either be trained through supervised learning, by providing the true branching signals, or by policy gradients [**174**] which maximizes the log-likelihood of the observed sequence. As opposed to previous models, these three operations constantly change the structure of the model in an online manner.

Experimental evaluation aims to analyze various aspects of the model such as: how does the model generalize on sequences of different lengths than those seen during training? how hard is the tree learning problem? To answer those questions, we propose a novel multi-operation math expression evaluation (MEE) dataset, with a standard set of tasks with varying levels of difficulty, where the difficulty scales up with respect to the length of the sequence.

9.2. Model

Similar to a standard RNN, our model modifies a hidden state h_l for each step of the input sequence $x = \{x_1, \ldots, x_N\}$ by means of split, merge and recurrent operations. Denote the sequence of operations by $z = \{z_1, \ldots, z_L\}$, where L may be greater than the number of tokens N since only recurrent operations consume input tokens (see Fig. 9.1). Each operation is parametrized by a different "cell". A policy network controls which operation to perform during sequence generation.

split (S). The split cell creates a sub-tree by taking the previous state h_{l-1} as input and generating two outputs h_l and h_{down}. h_l is used for further computation, while h_{down} (the small blue rectangle in Fig. 9.1(d)) is pushed into a stack for future use. In our model, $h_{down} = F_1(h_{l-1}, x_t)$ and $h_l = F_2(h_{l-1}, x_t)$ where the F_1 and F_2 are LSTM units [**75**], and x_t is the current input.

recurrent (R). This cell is a standard LSTM unit that takes as input the previous state h_{l-1} and the current token x_t, and outputs the hidden state h_l, which will be used to predict the next output $\hat{x}_{t+1}$. After application of this cell, the counter t is incremented and input x_t is consumed.

merge (M). The merge cell closes a sub-tree and returns control to its parent node. It does so by merging the previous hidden state h_{l-1} with the top of the stack h_{down} into a new hidden state $h_m = MLP(h_{l-1}, h_{down})$ (Fig. 9.1(c)). h_m is then used as input to another LSTM unit (F_4) to yield $h_l = F_4(x_t, h_m)$, the new hidden state of the overall network. Intuitively, h_{l-1} summarizes the contents within the sub-tree. This is merged with information obtained before the model entered the sub-tree h_{down} into the new state h_l.
Policy Network. We consider the decision at each timestep $z_t \in \{S, M, R\}$ as a categorical variable sampled from a policy network p_π, conditioned on the hidden state h_t and the input embedding e_t of the current input x_t. In the supervised setting, $p_\pi(z_t|e_t, h_t)$ is trained by maximizing the likelihood of the true branching labels, while in the unsupervised setting, we resort to the REINFORCE algorithm using $-\log p(y_t|C)$ as a reward, where y_t is the task target (i.e. the next word in language modeling), and C is the representation learnt by the model up until time t.

The main network being fully-backpropable, it can be trained using gradient descent. However, the policy network has a discrete sampling step and its gradient cannot be obtained simply by backprop. We therefore obtain the gradient on the output of the policy network using REINFORCE [**175**], and then back-propagated that into the whole policy network.

9.3. Experimental Results

We conduct our experiments on a math induction task, a propositional logic inference task [**15**] and language modeling. First of all, we aim to investigate whether a) our hierarchical model may help in tasks that explicitly exhibit hierarchical structure, and then b) whether the trees learned without supervision correspond to the ground-truth trees, c) how our model fare with respect to hierarchical models that have access to the whole sequence with a pre-determined tree structure and finally, d) are there any limitations for models that are not capable of learning hierarchical structures on-the-fly.

9.3.1. Math Induction

Our math expression evaluation dataset (MEE) consists of parenthesized mathematical expressions and their corresponding evaluations. The math expressions contain bracketing

Length	Expression	Value
4	((9+(2+6))+(1*3))	20
5	(((7-2)%((3%1)+6))*9)	45
6	((((3-0)+(7-6))*(0+9))-7)	29
7	((4*(6+(7*(2*8))))%(9+(3+7)))	16

Tab. 9.1. Sample expressions from MEE dataset

symbols ("()"), four different kinds of operations, "+-*%", where "%" is the modulo operation, and digits from 0 to 9. The "length" of an expression is the number of operations in the expression and its result is restricted to be a positive, two-digit integer (Table 9.1). We randomly generate expressions of different lengths and for each length the resulting sub-dataset is divided into 100,000, 10,000 and 10,000 expressions as training, valid and test sets. We make sure that there is no overlap between the splits and every expression is made unique across the whole dataset.

We use an encoder-decoder approach where the encoder reads the characters in the expression and produces the encoding as input to the decoder, which in turn sequentially generates 2 digits as the predicted value. We experiment on various encoders, including our model, and compare their performances. We use the same decoder architecture to ensure a fair comparison. The output of the encoder is provided as the initial hidden state of the decoder LSTM. To test the generalization of our model, for all the experiments shown in this subsection, we train the model on expressions of length 4 and 5, and evaluate on expressions of length 4 to 7 in the test sets.

For this task, our baseline is a simple *LSTM* encoder (which corresponds to our model with only recur operations). We compare two versions of our *RRNet* encoder. In the supervised setting, we force the model to split and merge when it reads "(" and ")", respectively, and recur otherwise. This gives us an idea on how well the model would perform if it had access to the ground-truth tree. In the unsupervised setting, we learn the tree using policy gradient, where the reward is the accuracy of the math result prediction.

The results are in Table 9.2. The *supervised RRNet* yields the best performance showing that (a) exploiting the hierarchical structure of the observed data either in supervised or unsupervised ways could benefit the performance, which corroborates previous work [**172**],

Model	Train	Test			
		L=4	**L=5**	**L=6**	**L=7**
LSTM	75.80	81.32	67.65	52.70	41.35
Uns RRNet	86.00	89.42	77.96	61.34	50.46
Sup RRNet	93.70	93.28	86.69	79.09	72.70

Tab. 9.2. Prediction accuracy on MEE dataset.

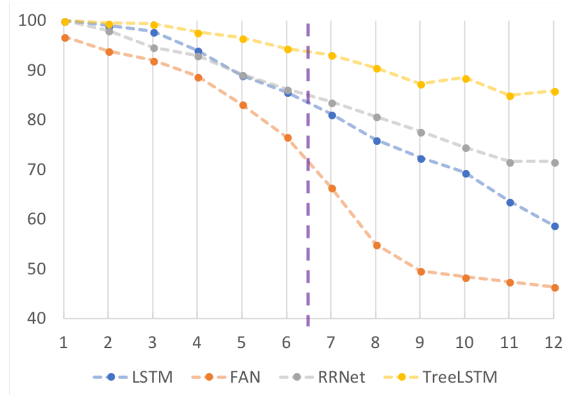


Fig. 9.2. Test accuracy of the models, trained on sequences of length ≤ 6 in logic data. The horizontal axis indicates the length of the sequence, and the vertical axis indicates the accuracy of model's performance on the corresponding test set.

and (b) our model is effective at capturing that information. The *unsupervised RRNet* model also outperforms the baseline LSTM: the model learn to exploit branching operations to achieve better performance. We observe that the trees produced by the model do not correspond to the ground-truth trees. In order to assess whether the additional parameters of split and merge operations, rather than the learned tree structure, produce better results, we measured the performance of our model trained with "random" deterministic policies (associating each of the input characters to either a split, merge or recur operation). We see that "random" policies perform worse than a "learnt" policy on the task, effectively similar to the baseline LSTM. In turn, the model with the learnt policy underperforms the model trained with ground-truth trees. Even in this seemingly easy task, it has appeared difficult for the model to learn the optimal branching policy.

9.3.2. Logical inference

In the next task, we analyze performance on the artificial language as described in [**16**]. This language has six word types $\{a,\ b,\ c,\ d,\ e,\ f\}$ and three logical operations $\{$*or*, *and*, *not*$\}$. There are seven mutually exclusive logical relations that describe the relationship between two sentences: entailment ($\sqsubset$, $\sqsupset$), equivalence ($\equiv$), exhaustive and non-exhaustive contradiction (^, $|$), and two types of semantic independence ($\#$, $\smile$). The train/dev/test dataset ratios are set to 0.8/0.1/0.1 as described [1] with the number of logical operations ranging from 1 to 12.

From Figure 2, we report the performance of our model when trained with ground-truth trees as input. It is encouraging to see that our recurrent-recursive encoder improves performance over Transformer (FAN) [**159**] and LSTM, especially for long sequences. The best performance on this dataset is given by TreeLSTM [**16**], which has access to the whole sequence and does not encode sequences on-the-fly. TreeLSTM significantly outperforms the other models, especially in the generalization tasks, i.e., those tasks when sequence lengths are bigger than 6. Note that TreeLSTM here utilizes extra information at test time, i.e., the ground truth parse tree. On the other hand, if we consider from the optimization point of view, our model is facing a much harder optimization problem than TreeLSTM, in trade for being auto-regressive and a learnable structure. The TreeLSTM is utterly a recursive network. Thus it takes approximately $O(\log n)$ steps from the leaf nodes to the final root encoding, which makes it easy for the gradients to flow back to every nodes. However, for our model the gradients are not that easy to propagate. As although the shortest path from a certain node to the final node is $O(\log n)$ as well (travelling through the shortcuts), the longest path could be $O(n)$ (traversing all the leaf nodes).

9.3.3. Language Modeling

In language modeling, architectures such as TreeLSTM aren't directly applicable since their structure isn't computed on-the-fly, while reading the sentence. We perform preliminary experiments using the Penn Treebank Corpus dataset [**113**], which has a vocabulary of 10,000 unique words and 929k, 73k and 82k words in training, validation and test set respectively. Our cells use one layer and the hidden dimensionality is 350. Our model yields test perplexity

[1] `https://github.com/sleepinyourhat/vector-entailment`

of 107.28 as compared to the LSTM baseline which gets 113.4 [**53**]. This preliminary result shows that the endeavor to exploit explicit hierarchical structures for language modeling, although challenging, may be promising.

9.4. Final Considerations

In this work, we began exploring properties of a recurrent-recursive neural network architecture that learns to encode the sequence on-the-fly, i.e. while reading. We argued this may be an important feature for tasks such as language modeling. We additionally proposed a new mathematical expression evaluation dataset (MEE) as a toy problem for validating the performance of sequential models to learn from hierarchical data. We empirically observed that, in this task, our model performs better than a standard LSTM architecture with no explicit structure and also outperforms the baseline LSTM and FAN architectures on the propositional logic task.

We hope to further study the properties of this model by either more thorough architecture search (recurrent dropouts, layer norm, hyper-parameter sweeps), different variation of RL algorithms such as deep Q-learning [**121**] and employing this model on various other tasks such as SNLI [**14**] and semi-supervised parsing.

Chapter 10

Prologue to Fourth Article

10.1. Article Details

Straight to the Tree: Constituency Parsing with Neural Syntactic Distance. Yikang Shen*, Zhouhan Lin*, Athul Paul Jacob, Alessandro Sordoni, Aaron Courville, Yoshua Bengio, *56th Annual Meeting of the Association for Computational Linguistics (ACL), 2018*

Personal Contribution. (*) denotes co-first authorship. Yikang Shen and I came up with the idea of using the syntactic distance for supervised learning. I implemented the prototype of the model. Yikang Shen and I iterated on the prototype to significantly improve its performance. Athul Paul Jacob also iterated on the prototype as well, and has run some of the experiments. We receive important feedback from Alessandro Sordoni, Aaron Courville, and Yoshua Bengio during the whole project. Yikang Shen, Alessandro Sordoni and I have wrote up the paper, while Athul Paul Jacob, Aaron Courville, and Yoshua Bengio helped improve its presentation.

10.2. Context

Parsing natural language with neural network models was attracting growing attention when the paper is published. Various works have shown the effectiveness of neural networks in both dependency parsing [**23**] and constituency parsing [**53, 41, 36**]. Early works proposed to use a feed-forward network to predict parse trees [**23**], or use a sequence-to-sequence framework to predict a linearized version of the parse tree [**165**].

There are different kinds of parsing schemes as well. Transition-based parsers such as [**106, 23, 168, 41**] generally suffer from compounding errors due to exposure bias, since the model is expected to predict multiple steps during test time while only asked to predict one step ahead during training time. Chart-based parsers such as [**52, 152**] ensure structural consistency and offer exact inference with the CYK algorithm, however the computational complexity remains in $O(n^3)$, which is relatively high.

10.3. Contributions

We presented a novel constituency parsing scheme based on predicting real-valued scalars, named syntactic distances, whose ordering identify the sequence of top-down split decisions. Differently from both transition based parsers, our parsing scheme doesn't suffer from exposure bias and compounding errors. Also, since the prediction of syntactic distances can be done in parallel, our parsing scheme could be much faster than previous models. We achieved competitive performance in the Penn Treebank (PTB) dataset and our method was the state-of-the-art in the Chinese Treebank (CTB) dataset.

10.4. Recent Developments

The syntactic distance idea was extensively inspected and extended in various follow-up works such as [**76, 145**]. [**70**] has defined another form of syntactic distance which requires supervised learning on a mapping from word representations to syntactic distances. There are various forms of newer parsers with more sophisticated structures proposed after the publication of this work. These models include transition based parsers [**105**], or utilizing pre-trained models like ELMo [**90**] or BERT [**186**].

Chapter 11

Straight to the Tree: Constituency Parsing with Neural Syntactic Distance

11.1. Introduction

Devising fast and accurate constituency parsing algorithms is an important, long-standing problem in natural language processing. Parsing has been useful for incorporating linguistic prior in several related tasks, such as relation extraction, paraphrase detection [**19**], and more recently, natural language inference [**15**] and machine translation [**55**].

Neural network-based approaches relying on dense input representations have recently achieved competitive results for constituency parsing [**165, 41, 106, 152**]. Generally speaking, either these approaches produce the parse tree sequentially, by governing the sequence of transitions in a transition-based parser [**131, 189, 23, 41**], or use a chart-based approach by estimating non-linear potentials and performing exact structured inference by dynamic programming [**58, 52, 152**].

Transition-based models decompose the structured prediction problem into a sequence of local decisions. This enables fast greedy decoding but also leads to compounding errors because the model is never exposed to its own mistakes during training [**43**]. Solutions to this problem usually complexify the training procedure by using structured training through beam-search [**169, 2**] and dynamic oracles [**60, 41**]. On the other hand, chart-based models can incorporate structured loss functions during training and benefit from exact inference via the CYK algorithm but suffer from higher computational cost during decoding [**52, 152**].

In this paper, we propose a novel, fully-parallel model for constituency parsing, based on the concept of “syntactic distance”, recently introduced by [**144**] for language modeling. To

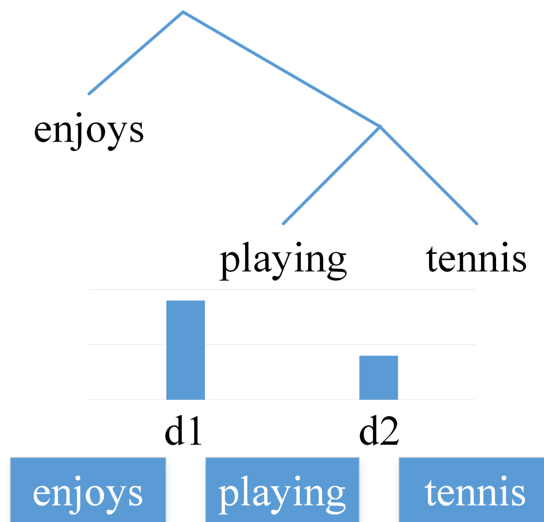


Fig. 11.1. An example of how syntactic distances ($d1$ and $d2$) describe the structure of a parse tree: consecutive words with larger predicted distance are split earlier than those with smaller distances, in a process akin to divisive clustering.

construct a parse tree from a sentence, one can proceed in a top-down manner, recursively splitting larger constituents into smaller constituents, where the order of the splits defines the hierarchical structure. The syntactic distances are defined for each possible split point in the sentence. The order induced by the syntactic distances fully specifies the order in which the sentence needs to be recursively split into smaller constituents (Figure 11.1): in case of a binary tree, there exists a one-to-one correspondence between the ordering and the tree. Therefore, our model is trained to reproduce the ordering between split points induced by the ground-truth distances by means of a margin rank loss [**170**]. Crucially, our model works *in parallel*: the estimated distance for each split point is produced independently from the others, which allows for an easy parallelization in modern parallel computing architectures for deep learning, such as GPUs. Along with the distances, we also train the model to produce the constituent labels, which are used to build the fully labeled tree.

Our model is fully parallel and thus does not require computationally expensive structured inference during training. Mapping from syntactic distances to a tree can be efficiently done in $\mathcal{O}(n \log n)$, which makes the decoding computationally attractive. Despite our strong conditional independence assumption on the output predictions, we achieve good performance for single model discriminative parsing in PTB (91.8 F1) and CTB (86.5 F1)

matching, and sometimes outperforming, recent chart-based and transition-based parsing models.

11.2. Syntactic Distances of a Parse Tree

In this section, we start from the concept of syntactic distance introduced in [**144**] for unsupervised parsing via language modeling and we extend it to the supervised setting. We propose two algorithms, one to convert a parse tree into a compact representation based on distances between consecutive words, and another to map the inferred representation back to a complete parse tree. The representation will later be used for supervised training. We formally define the syntactic distances of a parse tree as follows:

Definition 11.2.1. Let $\mathbf{T}$ be a parse tree that contains a set of leaves $(w_0, ..., w_n)$. The height of the lowest common ancestor for two leaves (w_i, w_j) is noted as $\tilde{d}_j^i$. The syntactic distances of $\mathbf{T}$ can be any vector of scalars $\mathbf{d} = (d_1, ..., d_n)$ that satisfy:

$$\text{sign}(d_i - d_j) = \text{sign}(\tilde{d}_i^{i-1} - \tilde{d}_j^{j-1}) \tag{11.2.1}$$

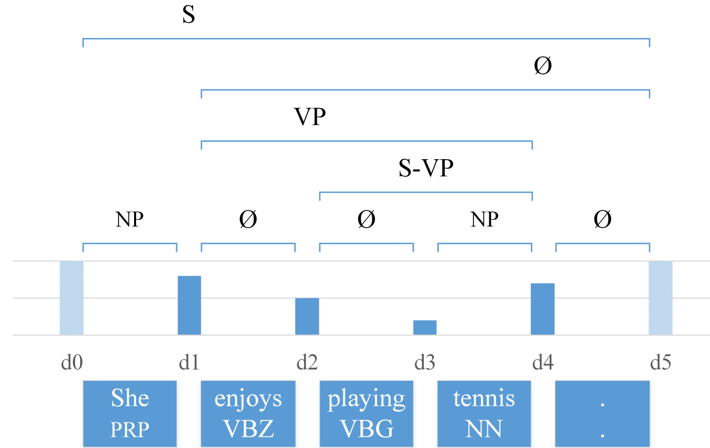
In other words, $\mathbf{d}$ induces the same ranking order as the quantities $\tilde{d}_i^j$ computed between pairs of consecutive words in the sequence, i.e. $(\tilde{d}_1^0, ..., \tilde{d}_n^{n-1})$. Note that there are $n-1$ syntactic distances for a sentence of length n.

Example 11.2.1. Consider the tree in Fig. 11.1 for which $\tilde{d}_1^0 = 2$, $\tilde{d}_2^1 = 1$. An example of valid syntactic distances for this tree is any $\mathbf{d} = (d_1, d_2)$ such that $d_1 > d_2$.

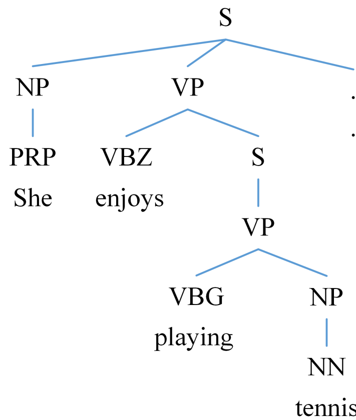
Given this definition, the parsing model predicts a sequence of scalars, which is a more natural setting for models based on neural networks, rather than predicting a set of spans. For comparison, in most of the current neural parsing methods, the model needs to output a sequence of transitions [**41, 23**].

Let us first consider the case of a binary parse tree. Algorithm 2 provides a way to convert it to a tuple $(\mathbf{d}, \mathbf{c}, \mathbf{t})$, where $\mathbf{d}$ contains the height of the inner nodes in the tree following a left-to-right (in order) traversal, $\mathbf{c}$ the constituent labels for each node in the same order and $\mathbf{t}$ the part-of-speech (POS) tags of each word in the left-to-right order. $\mathbf{d}$ is a valid vector of syntactic distances satisfying Definition 11.2.1.

Once a model has learned to predict these variables, Algorithm 3 can reconstruct a unique binary tree from the output of the model $(\hat{\mathbf{d}}, \hat{\mathbf{c}}, \hat{\mathbf{t}})$. The idea in Algorithm 3 is similar to the top-down parsing method proposed by [**152**], but differs in one key aspect: at each recursive



(a) Boxes in the bottom are words and their corresponding POS tags predicted by an external tagger. The vertical bars in the middle are the syntactic distances, and the brackets on top of them are labels of constituents. The bottom brackets are the predicted unary label for each words, and the upper brackets are predicted labels for other constituent.



(b) The corresponding inferred grammar tree.

Fig. 11.2. Inferring the parse tree with Algorithm 3 given distances, constituent labels, and POS tags. Starting with the full sentence, we pick split point 1 (as it is assigned to the larger distance) and assign label S to span (0,5). The left child span (0,1) is assigned with a tag PRP and a label NP, which produces an unary node and a terminal node. The right child span (1,5) is assigned the label ∅, coming from implicit binarization, which indicates that the span is not a real constituent and all of its children are instead direct children of its parent. For the span (1,5), the split point 4 is selected. The recursion of splitting and labeling continues until the process reaches a terminal node.

Algorithm 2 Binary Parse Tree to Distance

(∪ represents the concatenation operator of lists)

1: **function** DISTANCE(node)
2:     **if** node **is** leaf **then**
3:         $\mathbf{d} \leftarrow []$
4:         $\mathbf{c} \leftarrow []$
5:         $\mathbf{t} \leftarrow [\text{node.tag}]$
6:         $h \leftarrow 0$
7:     **else**
8:         $\text{child}_l, \text{child}_r \leftarrow$ children of node
9:         $\mathbf{d}_l$, $\mathbf{c}_l$, $\mathbf{t}_l$, $h_l \leftarrow$ Distance(child_l)
10:         $\mathbf{d}_r$, $\mathbf{c}_r$, $\mathbf{t}_r$, $h_r \leftarrow$ Distance(child_r)
11:         $h \leftarrow \max(h_l, h_r) + 1$
12:         $\mathbf{d} \leftarrow \mathbf{d}_l \cup [h] \cup \mathbf{d}_r$
13:         $\mathbf{c} \leftarrow \mathbf{c}_l \cup [\text{node.label}] \cup \mathbf{c}_r$
14:         $\mathbf{t} \leftarrow \mathbf{t}_l \cup \mathbf{t}_r$
15:     **return** $\mathbf{d}$, $\mathbf{c}$, $\mathbf{t}$, h

Algorithm 3 Distance to Binary Parse Tree

1: **function** TREE($\mathbf{d}$,$\mathbf{c}$,$\mathbf{t}$)
2:     **if** $\mathbf{d} = []$ **then**
3:         node $\leftarrow$ Leaf($\mathbf{t}$)
4:     **else**
5:         $i \leftarrow \arg\max_i(\mathbf{d})$
6:         $\text{child}_l \leftarrow$ Tree($\mathbf{d}_{<i}$, $\mathbf{c}_{<i}$, $\mathbf{t}_{<i}$)
7:         $\text{child}_r \leftarrow$ Tree($\mathbf{d}_{>i}$, $\mathbf{c}_{>i}$, $\mathbf{t}_{\geq i}$)
8:         node $\leftarrow$ Node(child_l, child_r, $\mathbf{c}_i$)
9:     **return** node

call, there is no need to estimate the confidence for every split point. The algorithm simply chooses the split point i with the maximum $\hat{d}_i$, and assigns to the span the predicted label $\hat{c}_i$. This makes the running time of our algorithm to be in $\mathcal{O}(n \log n)$, compared to the $\mathcal{O}(n^2)$ of

the greedy top-down algorithm by [**152**]. Figure 11.2 shows an example of the reconstruction of parse tree. Alternatively, the tree reconstruction process can also be done in a bottom-up manner, which requires the recursive composition of adjacent spans according to the ranking induced by their syntactic distance, a process akin to agglomerative clustering.

One potential issue is the existence of unary and n-ary nodes. We follow the method proposed by [**152**] and add a special empty label $\varnothing$ to spans that are not themselves full constituents but simply arise during the course of implicit binarization. For the unary nodes that contains one nonterminal node, we take the common approach of treating these as additional atomic labels alongside all elementary nonterminals [**152**]. For all terminal nodes, we determine whether it belongs to a unary chain or not by predicting an additional label. If it is predicted with a label different from the empty label, we conclude that it is a direct child of a unary constituent with that label. Otherwise if it is predicted to have an empty label, we conclude that it is a child of a bigger constituent which has other constituents or words as its siblings.

An n-ary node can arbitrarily be split into binary nodes. We choose to use the leftmost split point. The split point may also be chosen based on model prediction during training. Recovering an n-ary parse tree from the predicted binary tree simply requires removing the empty nodes and split combined labels corresponding to unary chains.

Algorithm 3 is a divide-and-conquer algorithm. The running time of this procedure is $\mathcal{O}(n \log n)$. However, the algorithm is naturally adapted for execution in a parallel environment, which can further reduce its running time to $\mathcal{O}(\log n)$.

11.3. Learning Syntactic Distances

We use neural networks to estimate the vector of syntactic distances for a given sentence. We use a modified hinge loss, where the target distances are generated by the tree-to-distance conversion given by Algorithm 2. Section 11.3.1 will describe in detail the model architecture, and Section 11.3.2 describes the loss we use in this setting.

11.3.1. Model Architecture

Given input words $\mathbf{w} = (w_0, w_1, ..., w_n)$, we predict the tuple $(\mathbf{d}, \mathbf{c}, \mathbf{t})$. The POS tags $\mathbf{t}$ are given by an external Part-Of-Speech (POS) tagger. The syntactic distances $\mathbf{d}$ and

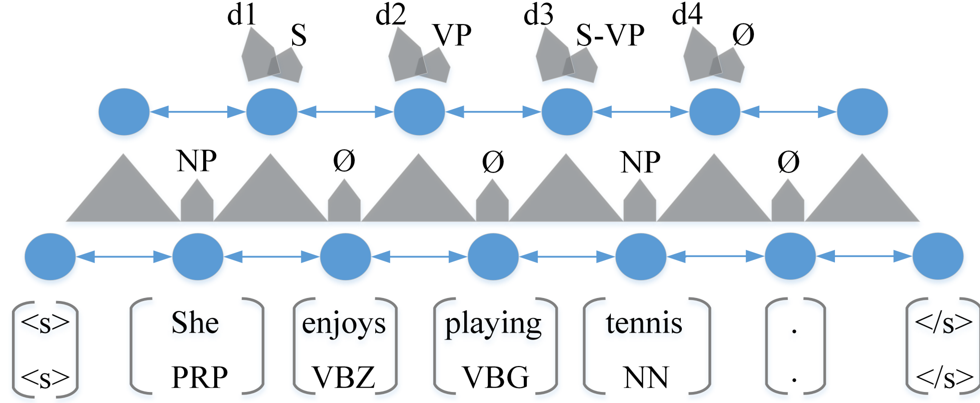


Fig. 11.3. The overall visualization of our model. Circles represent hidden states, triangles represent convolution layers, block arrows represent feed-forward layers, arrows represent recurrent connections. The bottom part of the model predicts unary labels for each input word. The $\emptyset$ is treated as a special label together with other labels. The top part of the model predicts the syntactic distances and the constituent labels. The inputs of model are the word embeddings concatenated with the POS tag embeddings. The tags are given by an external Part-Of-Speech tagger.

constituent labels $\mathbf{c}$ are predicted using a neural network architecture that stacks recurrent (LSTM [**75**]) and convolutional layers.

Words and tags are first mapped to sequences of embeddings $\mathbf{e}_0^{\text{w}}, ..., \mathbf{e}_n^{\text{w}}$ and $\mathbf{e}_0^{\text{t}}, ..., \mathbf{e}_n^{\text{t}}$. Then the word embeddings and the tag embeddings are concatenated together as inputs for a stack of bidirectional LSTM layers:

$$\mathbf{h}_0^{\text{w}}, ..., \mathbf{h}_n^{\text{w}} = \text{BiLSTM}_{\text{w}}([\mathbf{e}_0^{\text{w}}, \mathbf{e}_0^{\text{t}}], ..., [\mathbf{e}_n^{\text{w}}, \mathbf{e}_n^{\text{t}}]) \tag{11.3.1}$$

where $\text{BiLSTM}_{\text{w}}(\cdot)$ is the word-level bidirectional layer, which gives the model enough capacity to capture long-term syntactical relations between words.

To predict the constituent labels for each word, we pass the hidden states representations $\mathbf{h}_0^{\text{w}}, ..., \mathbf{h}_n^{\text{w}}$ through a 2-layer network FF_c^{w}, with softmax output:

$$p(c_i^{\mathbf{w}}|\mathbf{w}) = \text{softmax}(\text{FF}_c^{\text{w}}(\mathbf{h}_i^{\text{w}})) \tag{11.3.2}$$

To compose the necessary information for inferring the syntactic distances and the constituency label information, we perform an additional convolution:

$$\mathbf{g}_1^{\text{s}}, \ldots, \mathbf{g}_n^{\text{s}} = \text{CONV}(\mathbf{h}_0^{\text{w}}, ..., \mathbf{h}_n^{\text{w}}) \tag{11.3.3}$$

where $\mathbf{g}_i^{\mathrm{s}}$ can be seen as a draft representation for each split position in Algorithm 3. Note that the subscripts of g_i^ss start with 1, since we have $n-1$ positions as non-terminal constituents. Then, we stack a bidirectional LSTM layer on top of $\mathbf{g}_i^{\mathrm{s}}$:

$$\mathbf{h}_1^{\mathrm{s}}, ..., \mathbf{h}_n^{\mathrm{s}} = \mathrm{BiLSTM}_{\mathrm{s}}(\mathbf{g}_1^{\mathrm{s}}, \ldots, \mathbf{g}_n^{\mathrm{s}}) \tag{11.3.4}$$

where $\mathrm{BiLSTM}_{\mathrm{s}}$ fine-tunes the representation by conditioning on other split position representations. Interleaving between LSTM and convolution layers turned out empirically to be the best choice over multiple variations of the model, including using self-attention [**162**] instead of LSTM.

To calculate the syntactic distances for each position, the vectors $\mathbf{h}_1^{\mathrm{s}}, \ldots, \mathbf{h}_n^{\mathrm{s}}$ are transformed through a 2-layer feed-forward network FF_d with a single output unit (this can be done in parallel with 1x1 convolutions), with no activation function at the output layer:

$$\hat{d}_i = \mathrm{FF}_d(\mathbf{h}_i^s), \tag{11.3.5}$$

For predicting the constituent labels, we pass the same representations $\mathbf{h}_1^{\mathrm{s}}, \ldots, \mathbf{h}_n^{\mathrm{s}}$ through another 2-layer network $\mathrm{FF}_c^{\mathrm{s}}$, with softmax output.

$$p(c_i^{\mathrm{s}}|\mathbf{w}) = \mathrm{softmax}(\mathrm{FF}_c^{\mathrm{s}}(\mathbf{h}_i^{\mathrm{s}})) \tag{11.3.6}$$

The overall architecture is shown in Figure 11.2a. Since the output $(\mathbf{d}, \mathbf{c}, \mathbf{t})$ can be unambiguously transfered to a unique parse tree, the model implicitly makes all parsing decisions inside the recurrent and convolutional layers.

11.3.2. Objective

Given a set of training examples $\mathcal{D} = \{\langle \mathbf{d}_k, \mathbf{c}_k, \mathbf{t}_k, \mathbf{w}_k \rangle\}_{k=1}^K$, the training objective is the sum of the prediction losses of syntactic distances $\mathbf{d}_k$ and constituent labels $\mathbf{c}_k$.

Due to the categorical nature of variable $\mathbf{c}$, we use a standard softmax classifier with a cross-entropy loss L_{label} for constituent labels, using the estimated probabilities obtained in Eq. 11.3.2 and 11.3.6.

A naïve loss function for estimating syntactic distances is the mean-squared error (MSE):

$$L_{\mathrm{dist}}^{\mathrm{mse}} = \sum_i (d_i - \hat{d}_i)^2 \tag{11.3.7}$$

The MSE loss forces the model to regress on the exact value of the true distances. Given that only the *ranking* induced by the ground-truth distances in $\mathbf{d}$ is important, as opposed to the absolute values themselves, using an MSE loss over-penalizes the model by ignoring ranking equivalence between different predictions.

Therefore, we propose to minimize a pair-wise learning-to-rank loss, similar to those proposed in [**18**]. We define our loss as a variant of the hinge loss as:

$$L_{\text{dist}}^{\text{rank}} = \sum_{i,j>i} [1 - \text{sign}(d_i - d_j)(\hat{d}_i - \hat{d}_j)]^+ \tag{11.3.8}$$

$$\text{sign}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases} \tag{11.3.9}$$

where $[x]^+$ is defined as $\max(0, x)$. This loss encourages the model to reproduce the full ranking order induced by the ground-truth distances. The final loss for the overall model is just the sum of individual losses $L = L_{\text{label}} + L_{\text{dist}}^{\text{rank}}$.

11.4. Experiments

We evaluate our model described above on 2 different datasets, the standard Wall Street Journal (WSJ) part of the Penn Treebank (PTB) dataset, and the Chinese Treebank (CTB) dataset.

For evaluating the F1 score, we use the standard `evalb`[1] tool. We provide both labeled and unlabeled F1 score, where the former takes into consideration the constituent label for each predicted constituent, while the latter only considers the position of the constituents. In the tables below, we report the labeled F1 scores for comparison with previous work, as this is the standard metric usually reported in the relevant literature. We tuned in a wide range for the hidden state and embedding size, as well as the convolutional window size. The dropout in different parts of the model are tuned separately as well.

[1] `http://nlp.cs.nyu.edu/evalb/`

Model	Labeled Precision	Labeled Recall	F1
Single Model			
Vinyals et al. [**165**]	-	-	88.3
Zhu et al. [**189**]	90.7	90.2	90.4
RNNG [**53**]	-	-	89.8
Watanabe et al. [**168**]	-	-	90.7
Cross et al. [**41**]	92.1	90.5	91.3
Liu et al. [**106**]	92.1	91.3	91.7
Stern et al. [**152**]	93.2	90.3	91.8
Liu et al. [**105**]	-	-	91.8
Gaddy et al. [**59**]	-	-	92.1
Stern et al. [**153**]	92.5	92.5	92.5
Our Model	92.0	91.7	91.8
Ensemble			
Shindo et al. [**146**]	-	-	92.4
Vinyals et al. [**165**]	-	-	90.5
Semi-supervised			
Zhu et al. [**189**]	91.5	91.1	91.3
Vinyals et al. [**165**]	-	-	92.8
Re-ranking			
Charniak et al. [**22**]	91.8	91.2	91.5
Huang et al. [**78**]	91.2	92.2	91.7
RNNG [**53**]	-	-	93.3

Tab. 11.1. Results on the PTB dataset WSJ test set, Section 23. LP, LR represents labeled precision and recall respectively.

11.4.1. Penn Treebank

For the PTB experiments, we follow the standard train/valid/test separation and use sections 2-21 for training, section 22 for development and section 23 for test set. Following this split, the dataset has 45K training sentences and 1700, 2416 sentences for valid/test

respectively. The placeholders with the `-NONE-` tag are stripped from the dataset during preprocessing. The POS tags are predicted with the Stanford Tagger [**157**].

We use a hidden size of 1200 for each direction on all LSTMs, with 0.3 dropout in all the feed-forward connections, and 0.2 recurrent connection dropout [**117**]. The convolutional filter size is 2. The number of convolutional channels is 1200. As a common practice for neural network based NLP models, the embedding layer that maps word indexes to word embeddings is randomly initialized. The word embeddings are sized 400. Following [**117**], we randomly swap an input word embedding during training with the zero vector with probability of 0.1. We found this helped the model to generalize better. Training is conducted with Adam algorithm with l2 regularization decay 1×10^{-6}. We pick the result obtaining the highest labeled F1 on the validation set, and report the corresponding test F1, together with other statistics. We report our results in Table 11.1. Our best model obtains a labeled F1 score of 91.8 on the test set (Table 11.1). Detailed dev/test set performances, including label accuracy is reported in Table 11.3.

Our model performs achieves good performance for single-model constituency parsing trained without external data. The best result from [**153**] is obtained by a generative model. Very recently, we came to knowledge of [**59**], which uses character-level LSTM features coupled with chart-based parsing to improve performance. Similar sub-word features can be also used in our model. We leave this investigation for future works. For comparison, other models obtaining better scores either use ensembles, benefit from semi-supervised learning, or recur to re-ranking of a set of candidates.

11.4.2. Chinese Treebank

We use the Chinese Treebank 5.1 dataset, with articles 001-270 and 440-1151 for training, articles 301-325 as development set, and articles 271-300 for test set. This is a standard split in the literature [**106**]. The `-NONE-` tags are stripped as well. The hidden size for the LSTM networks is set to 1200. We use a dropout rate of 0.4 on the feed-forward connections, and 0.1 recurrent connection dropout. The convolutional layer has 1200 channels, with a filter size of 2. We use 400 dimensional word embeddings. During training, input word embeddings are randomly swapped with the zero vector with probability of 0.1. We also apply a l2 regularization weighted by 1×10^{-6} on the parameters of the network. Table 11.2

Model	Labeled Precision	Labeled Recall	F1
Single Model			
Charniak et al. [**21**]	82.1	79.6	80.8
Zhu et al. [**189**]	84.3	82.1	83.2
Wang et al. [**166**]	-	-	83.2
Watanabe et al. [**168**]	-	-	84.3
RNNG [**53**]	-	-	84.6
Liu et al. [**106**]	85.9	85.2	85.5
Liu et al. [**105**]	-	-	86.1
Our Model	86.6	86.4	86.5
Semi-supervised			
Zhu et al. [**189**]	86.8	84.4	85.6
Wang et al. [**167**]	-	-	86.3
Wang et al. [**166**]	-	-	86.6
Re-ranking			
Charniak et al. [**22**]	83.8	80.8	82.3
RNNG [**53**]	-	-	86.9

Tab. 11.2. Test set performance comparison on the CTB dataset

reports our results compared to other benchmarks. To the best of our knowledge, we set a new state-of-the-art for single-model parsing achieving 86.5 F1 on the test set. The detailed statistics are shown in Table 11.3.

dev/test result		Prec.	Recall	F1	label accuracy
PTB	labeled	91.7/92.0	91.8/91.7	91.8/91.8	94.9/95.4%
	unlabeled	93.0/93.2	93.0/92.8	93.0/93.0	
CTB	labeled	89.4/86.6	89.4/86.4	89.4/86.5	92.2/91.1%
	unlabeled	91.1/88.9	91.1/88.6	91.1/88.8	

Tab. 11.3. Detailed experimental results on PTB and CTB datasets

11.4.3. Ablation Study

Model	Labeled Precision	Labeled Recall	F1
Full model	92.0	91.7	91.8
w/o top LSTM	91.0	90.5	90.7
w. Char LSTM	92.1	91.7	91.9
w. embedding	91.9	91.6	91.7
w. MSE loss	90.3	90.0	90.1

Tab. 11.4. Ablation test on the PTB dataset. "w/o top LSTM" is the full model without the top LSTM layer. "w Char LSTM" is the full model with the extra Character-level LSTM layer. "w. embedding" stands for the full model using the pretrained word embeddings. "w. MSE loss" stands for the full model trained with MSE loss.

We perform an ablation study by removing/adding components from a our model, and retrain the ablated version from scratch. This gives an idea of the relative contributions of each of the components in the model. Results are reported in Table 11.4. It seems that the top LSTM layer has a relatively big impact on performance. This may give additional capacity to the model for capturing long-term dependencies useful for label prediction. We used an extra 1-layer character-level BiLSTM to compute an extra word level embedding vector as input of our model. It's seems that character-level features give marginal improvements in our model. We also experimented by using 300D GloVe [**136**] embedding for the input layer but this didn't yield improvements over the model's best performance. Unsurprisingly, the model trained with MSE loss underperforms considerably a model trained with the rank loss.

11.5. Related Work

Parsing natural language with neural network models has recently received growing attention. These models have attained state-of-the-art results for dependency parsing [**23**] and constituency parsing [**53, 41, 36**]. Early work in neural network based parsing directly use a feed-forward neural network to predict parse trees [**23**]. [**165**] use a sequence-to-sequence framework where the decoder outputs a linearized version of the parse tree given an input

sentence. Generally, in these models, the correctness of the output tree is not strictly ensured (although empirically observed).

Other parsing methods ensure structural consistency by operating in a transition-based setting [**23**] by parsing either in the top-down direction [**53, 106**], bottom-up [**189, 168, 41**] and recently in-order [**105**]. Transition-based methods generally suffer from compounding errors due to exposure bias: during testing, the model is exposed to a very different regime (i.e. decisions sampled from the model itself) than what was encountered during training (i.e. the ground-truth decisions) [**43, 60**]. This can have catastrophic effects on test performance but can be mitigated to a certain extent by using beam-search instead of greedy decoding. [**153**] proposes an effective inference method for generative parsing, which enables direct decoding in those models. More complex training methods have been devised in order to alleviate this problem [**60, 41**]. Other efforts have been put into neural chart-based parsing [**52, 152**] which ensure structural consistency and offer exact inference with CYK algorithm. [**59**] includes a simplified CYK-style inference, but the complexity still remains in $O(n^3)$.

In this work, our model learns to produce a particular representation of a tree in parallel. Representations can be computed in parallel, and the conversion from representation to a full tree can efficiently be done with a divide-and-conquer algorithm. As our model outputs decisions in parallel, our model doesn't suffer from the exposure bias. Interestingly, a series of recent works, both in machine translation [**66**] and speech synthesis [**132**], considered the sequence of output variables conditionally independent given the inputs.

11.6. Conclusion

We presented a novel constituency parsing scheme based on predicting real-valued scalars, named syntactic distances, whose ordering identify the sequence of top-down split decisions. We employ a neural network model that predicts the distances $\mathbf{d}$ and the constituent labels $\mathbf{c}$. Given the algorithms presented in Section 11.2, we can build an unambiguous mapping between each $(\mathbf{d}, \mathbf{c}, \mathbf{t})$ and a parse tree. One peculiar aspect of our model is that it predicts split decisions *in parallel*. Our experiments show that our model can achieve strong performance compare to previous models.

In terms of computational complexity, Unlike the $\mathcal{O}(n^3)$ complexity of the classical CYK algorithm, our distance to tree conversion is a $\mathcal{O}(n \log n)$ divide-and-conquer algorithm. (n stand for the number of words in the input sentence.) Since the LSTM and convolutional layers all have a complexity of $\mathcal{O}(n)$, the overall complexity remains in $\mathcal{O}(n \log n)$. In our experiments, we reached a 111.1 sentences per second speed on the PTB dataset, using an NVIDIA TITAN Xp graphics card for running the neural network part, and the distance to tree inference is run on an Intel Core i7-6850K CPU, with 3.60GHz clock speed. [2]

Since the architecture of model is no more than a stack of standard recurrent and convolution layers, which are essential components in most academic and industrial deep learning frameworks, the deployment of this method would be straightforward.

[2] Please refer to [**138, 189, 106, 152**] for parsing speed on other published works. However these parsing speed baselines are not rigorously comparable since some of the reported results are using very different hardware settings. Unfortunately we couldn't find their source code to re-run them on our hardware.

Chapter 12

Conclusion

This thesis has touched various topics around neural networks for natural language processing and low precision networks.

The work on eliminating multiplications in neural networks had shed light on the possibility of significantly reducing the computational demand of neural networks from a perspective of using low-precision numbers. The research front has developed a lot since the publication of our paper. Various approaches has been proposed to convert different aspects of neural networks into using low-precision numbers, covering weights, activations, error signals, and gradients. As the research on algorithms for low-precision networks has become more abundant and developed, it spurs possibilities in realizing them on various types of hardware, ranging from FPGAs to bottom-up ASIC designs. Although there is already some successful dedicated hardware realizations [**154**], the co-design and optimization between hardware and algorithms is going to be an active topic.

The second and third work are exploring how to encode structured representations for natural language, as well as other kinds of sequential data with implicit structure. In the third work, we use attention mechanism to learn sentence embeddings, and we tried to represent different aspects of the sentence semantics in different rows in the matrix embedding. By incorporating multiple hops of attention and a specified penalty term, different rows in the matrix embeddings keep a diversity, and are shown to be more informative for downstream tasks. Generally we found that this type of flat structures are more successful in capturing the keywords in the text that are crucial for the task, but less advantageous in tracing relations between constituents. Thus the third work tries to attack this problem from a different setting. We explored properties of a dynamic, recurrent-recursive neural network architecture

that learns to encode the sequence while reading it. The discrete structure of the sequence is reflected in the structure of the dynamic networks. Experimentally, both in the case of supervised learning and unsupervised learning, our model outperforms a standard LSTM architecture with no explicit structure, which validates the effectiveness of the introduced recursive structure in the model. However, the discrete nature of the model structure makes the model only able to resort to reinforcement learning when the ground truth structures are not provided during training, which makes learning unstable and ineffective. In the mean time, syntactic distances proposed in [**144**] offer a way to learn syntactic structure in a continuous way. We are hoping to either use more advanced reinforcement learning approaches that could overcome these difficulties in recurrent-recursive network, or explore the possibility of devising a continuous way of representing recursive network that makes backpropagation possible.

The fourth work more rigorously defines the notion of syntactic distance, and verifies the effectiveness of it from a canonical NLP perspective. It was successfully applied to parsing. One specific advantage of our parsing scheme is that is is highly parallelizable, which makes our model able to achieve high parsing speed. Follow-up works on this thread could get deep into the field of linguistics and parsing, by studying the difference between learned syntax and linguist-tagged syntax, and the possibility to incorporate them in a unified representation, possibly through a multi-task setting.

More broadly, it is reasonable to believe that learning and modelling the structure of natural language could enhance the neural network to exploit the compositionality of natural language. Works in this thesis has made several attempts in the first step, which is representing and learning the structure. One interesting next step is to devise mechanisms that could utilize these structures for language generation or understanding. For example, non-autoregressive language generation through a learned branching structure could be an exciting topic to try in the future. Also, as for language understanding, modelling the structure helps the model to understand the compositionality of natural language. We might need interactive environments such as TextWorld [**38**] to learn generalizable and compositional structures.

Bibliography

[1] Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. Variance reduction in sgd by distributed importance sampling. *arXiv preprint arXiv:1511.06481*, 2015.

[2] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2442–2452. Association for Computational Linguistics, August 2016.

[3] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1131–1135. IEEE, 2015.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[6] Gökhan BakIr, Thomas Hofmann, Bernhard Schölkopf, Alexander J Smola, and Ben Taskar. *Predicting structured data*. MIT press, 2007.

[7] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop, 2012.

[8] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. In *Advances in Neural Information Processing Systems*, pages 932–938, 2001.

[9] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007.

[11] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[12] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[13] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[14] Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*, 2015.

[15] Samuel R Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D Manning, and Christopher Potts. A fast unified model for parsing and sentence understanding. *arXiv preprint arXiv:1603.06021*, 2016.

[16] Samuel R. Bowman, Christopher D. Manning, and Christopher Potts. Tree-structured composition in neural networks without tree-structured architectures. *CoRR*, abs/1506.04834, 2015.

[17] Peter S Burge, Max R van Daalen, Barry JP Rising, and John S Shawe-Taylor. Stochastic bit-stream neural networks. In *Pulsed neural networks*, pages 337–352. MIT Press, 1999.

[18] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22Nd International Conference on Machine Learning*, pages 89–96, 2005.

[19] Chris Callison-Burch. Syntactic constraints on paraphrases extracted from parallel corpora. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 196–205. Association for Computational Linguistics, 2008.

[20] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, et al. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*, 2018.

[21] Eugene Charniak. A maximum-entropy-inspired parser. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 132–139. Association for Computational Linguistics, 2000.

[22] Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, pages 173–180. Association for Computational Linguistics, 2005.

[23] Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 740–750. Association for Computational Linguistics, 2014.

[24] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[25] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999.

[26] Wenlin Chen, James T Wilson, Stephen Tyree, Kilian Q Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. *arXiv preprint arXiv:1504.04788*, 2015.

[27] Jianpeng Cheng, Li Dong, and Mirella Lapata. Long short-term memory-networks for machine reading. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2016.

[28] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhenzhong Lan. Training binary multilayer neural networks for image classification using expectation backpropagation. *arXiv preprint arXiv:1503.03562*, 2015.

[29] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[30] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[31] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[32] Jihun Choi, Kang Min Yoo, and Sang-goo Lee. Learning to compose task-specific tree structures. *arXiv preprint arXiv:1707.02786*, 2017.

[33] Jihun Choi, Kang Min Yoo, and Sang-goo Lee. Learning to compose task-specific tree structures. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[34] Noam Chomsky. *Syntactic Structures*. Mouton and Co., The Hague, 1957.

[35] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[36] Maximin Coavoux and Benoit Crabbé. Neural greedy constituent parsing with dynamic oracles. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 172–182. Association for Computational Linguistics, 2016.

[37] Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*, 2017.

[38] Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Tavian Barnes, Emery Fine, James Moore, Matthew Hausknecht, Layla El Asri, Mahmoud Adada, et al. Textworld: A learning environment for text-based games. *arXiv preprint arXiv:1806.11532*, 2018.

[39] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[40] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[41] James Cross and Liang Huang. Span-based constituency parsing with a structure-label system and provably optimal dynamic oracles. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1—11. Association for Computational Linguistics, 2016.

[42] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[43] Hal Daumé, John Langford, and Daniel Marcu. Search-based structured prediction. *Machine learning*, 75(3):297–325, 2009.

[44] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[45] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[46] Misha Denil, Babak Shakibi, Laurent Dinh, Nando de Freitas, et al. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156, 2013.

[47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[48] Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

[49] Cicero dos Santos and Maira Gatti. Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78, 2014.

[50] Cicero dos Santos, Ming Tan, Bing Xiang, and Bowen Zhou. Attentive pooling networks. *arXiv preprint arXiv:1602.03609*, 2016.

[51] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[52] Greg Durrett and Dan Klein. Neural crf parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 302–312. Association for Computational Linguistics, 2015.

[53] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199—-209. Association for Computational Linguistics, 2016.

[54] Salah El Hihi and Yoshua Bengio. Hierarchical recurrent neural networks for long-term dependencies. In *Proceedings of NIPS*, 1996.

[55] Akiko Eriguchi, Yoshimasa Tsuruoka, and Kyunghyun Cho. Learning to parse and translate improves neural machine translation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 72–78. Association for Computational Linguistics, 2017.

[56] Minwei Feng, Bing Xiang, Michael R. Glass, Lidan Wang, and Bowen Zhou. Applying deep learning to answer selection: a study and an open task. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU 2015, Scottsdale, AZ, USA, December 13-17, 2015*, pages 813–820, 2015.

[57] Michael Figurnov, Maxwell D Collins, Yukun Zhu, Li Zhang, Jonathan Huang, Dmitry Vetrov, and Ruslan Salakhutdinov. Spatially adaptive computation time for residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1039–1048, 2017.

[58] Jenny Rose Finkel, Alex Kleeman, and Christopher D. Manning. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL*, pages 959–967. Association for Computational Linguistics, 2008.

[59] David Gaddy, Mitchell Stern, and Dan Klein. What's going on in neural constituency parsers? an analysis. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018.

[60] Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *COLING 2012, 24th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers*, pages 959–976, 2012.

[61] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

[62] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[63] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[64] Alex Graves. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

[65] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[66] Jiatao Gu, James Bradbury, Caiming Xiong, Victor OK Li, and Richard Socher. Non-autoregressive neural machine translation. In *Proceedings of International Conference on Learning Representations*, 2018.

[67] Caglar Gulcehre, Orhan Firat, Kelvin Xu, Kyunghyun Cho, Loic Barrault, Huei-Chi Lin, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. On using monolingual corpora in neural machine translation. *arXiv preprint arXiv:1503.03535*, 2015.

[68] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[69] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[70] John Hewitt and Christopher D Manning. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, 2019.

[71] Felix Hill, Kyunghyun Cho, and Anna Korhonen. Learning distributed representations of sentences from unlabelled data. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1367–1377, San Diego, California, June 2016. Association for Computational Linguistics.

[72] Felix Hill, Kyunghyun Cho, Anna Korhonen, and Yoshua Bengio. Learning to understand phrases by embedding the dictionary. *Transactions of the Association for Computational Linguistics*, 4:17–30, 2016.

[73] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *diploma thesis, Institute of Computer Science*, 1991.

[74] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[75] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[76] Phu Mon Htut, Kyunghyun Cho, and Samuel R Bowman. Grammar induction with neural language models: An unusual replication. *arXiv preprint arXiv:1808.10000*, 2018.

[77] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[78] Liang Huang. Forest reranking: Discriminative parsing with non-local features. In *Proceedings of ACL-08: HLT*, pages 586–594. Association for Computational Linguistics, 2008.

[79] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[80] Peter Jeavons, David A. Cohen, and John Shawe-Taylor. Generating binary sequences for stochastic computing. *Information Theory, IEEE Transactions on*, 40(3):716–720, 1994.

[81] Yacine Jernite, Samuel R Bowman, and David Sontag. Discourse-based objectives for fast unsupervised sentence representation learning. *arXiv preprint arXiv:1705.00557*, 2017.

[82] Yacine Jernite, Edouard Grave, Armand Joulin, and Tomas Mikolov. Variable computation in recurrent neural networks. *arXiv preprint arXiv:1611.06188*, 2016.

[83] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2012.

[84] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

[85] Minje Kim and Smaragdis Paris. Bitwise neural networks. In *Proceedings of The 31st International Conference on Machine Learning*, pages 0–0, 2015.

[86] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, Doha, Qatar, October 2014. Association for Computational Linguistics.

[87] Yoon Kim, Alexander M Rush, Lei Yu, Adhiguna Kuncoro, Chris Dyer, and Gábor Melis. Unsupervised recurrent neural network grammars. *arXiv preprint arXiv:1904.03746*, 2019.

[88] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[89] Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *Advances in neural information processing systems*, pages 3294–3302, 2015.

[90] Nikita Kitaev and Dan Klein. Constituency parsing with a self-attentive encoder. *arXiv preprint arXiv:1805.01052*, 2018.

[91] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[92] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009.

[93] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[94] Hon Keung Kwan and CZ Tang. Multiplierless multilayer feedforward neural network design suitable for continuous input-output mapping. *Electronics Letters*, 29(14):1259–1260, 1993.

[95] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[96] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[97] Quoc V Le and Tomas Mikolov. Distributed representations of sentences and documents. In *ICML*, volume 14, pages 1188–1196, 2014.

[98] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404, 1990.

[99] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[100] Ji Young Lee and Franck Dernoncourt. Sequential short-text classification with recurrent and convolutional neural networks. *arXiv preprint arXiv:1603.03827*, 2016.

[101] F. Li, B. Zhang, and B. Liu. Ternary Weight Networks. *arXiv e-prints*, May 2016.

[102] Peng Li, Wei Li, Zhengyan He, Xuguang Wang, Ying Cao, Jie Zhou, and Wei Xu. Dataset and neural recurrent sequence labeling model for open-domain factoid question answering. *arXiv preprint arXiv:1607.06275*, 2016.

[103] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

[104] Wang Ling, Lin Chu-Cheng, Yulia Tsvetkov, and Silvio Amir. Not all contexts are created equal: Better word representations with variable attention. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1367–1372, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[105] Jiangming Liu and Yue Zhang. In-order transition-based constituent parsing. *Transactions of the Association of Computational Linguistics*, 5(1):413–424, 2017.

[106] Jiangming Liu and Yue Zhang. Shift-reduce constituent parsing with neural lookahead features. *Transactions of the Association for Computational Linguistics*, 5:45–58, 2017.

[107] Yang Liu, Chengjie Sun, Lei Lin, and Xiaolong Wang. Learning natural language inference using bidirectional lstm model and inner-attention. *arXiv preprint arXiv:1605.09090*, 2016.

[108] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[109] Mingbo Ma, Liang Huang, Bing Xiang, and Bowen Zhou. Dependency-based convolutional neural networks for sentence embedding. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, volume 2, pages 174–179, 2015.

[110] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[111] Emerson Lopes Machado, Cristiano Jacques Miosso, Ricardo von Borries, Murilo Coutinho, Pedro de Azevedo Berger, Thiago Marques, and Ricardo Pezzuol Jacobi. Computational cost reduction in learned transform classifications. *arXiv preprint arXiv:1504.06779*, 2015.

[112] Michele Marchesi, Gianni Orlandi, Francesco Piazza, and Aurelio Uncini. Fast neural networks without multipliers. *Neural Networks, IEEE Transactions on*, 4(1):53–62, 1993.

[113] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[114] Marco Marelli, Luisa Bentivogli, Marco Baroni, Raffaella Bernardi, Stefano Menini, and Roberto Zamparelli. Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. In *Proceedings of the 8th international workshop on semantic evaluation (SemEval 2014)*, pages 1–8, 2014.

[115] Horia Margarit and Raghav Subramaniam. A batch-normalized recurrent network for sentiment classification. In *Advances in Neural Information Processing Systems*, 2016.

[116] Roland Memisevic. Learning to relate images. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1829–1846, 2013.

[117] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[118] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[119] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[120] Marvin Minsky and Seymour Papert. Perceptrons: An introduction to computational geometry, 1961.

[121] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[122] Lili Mou, Rui Men, Ge Li, Yan Xu, Lu Zhang, Rui Yan, and Zhi Jin. Natural language inference by tree-based convolution and heuristic matching. *arXiv preprint arXiv:1512.08422*, 2015.

[123] Lili Mou, Hao Peng, Ge Li, Yan Xu, Lu Zhang, and Zhi Jin. Discriminative neural sentence modeling by tree-based convolution. *arXiv preprint arXiv:1504.01106*, 2015.

[124] Tsendsuren Munkhdalai and Hong Yu. Neural semantic encoders. *arXiv preprint arXiv:1607.04315*, 2016.

[125] Tsendsuren Munkhdalai and Hong Yu. Neural tree indexers for text understanding. *arXiv preprint arXiv:1607.04492*, 2016.

[126] Tsendsuren Munkhdalai and Hong Yu. Neural semantic encoders. In *Proceedings of the conference. Association for Computational Linguistics. Meeting*, volume 1, page 397. NIH Public Access, 2017.

[127] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, Bing Xiang, et al. Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*, 2016.

[128] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint arXiv:1511.06807*, 2015.

[129] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*. Granada, Spain, 2011.

[130] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994.

[131] Joakim Nivre. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics, 2004.

[132] Aaron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George van den Driessche, Edward Lockhart, Luis C Cobo, Florian Stimberg, et al. Parallel wavenet: Fast high-fidelity speech synthesis. *arXiv preprint arXiv:1711.10433*, 2017.

[133] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(4):694–707, 2016.

[134] Ankur P. Parikh, Oscar Tackstrom, Dipanjan Das, and Jakob Uszkoreit. A decomposable attention model for natural language inference. In *Proceedings of EMNLP*, 2016.

[135] Francis Jeffry Pelletier. The principle of semantic compositionality. *Topoi*, 13(1):11–24, 1994.

[136] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *EMNLP*, volume 14, pages 1532–43, 2014.

[137] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[138] Slav Petrov and Dan Klein. Improved inference for unlexicalized parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 404–411, 2007.

[139] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

[140] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

[141] AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering, 1987.

[142] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM JOURNAL OF RESEARCH AND DEVELOPMENT*, pages 71–105, 1959.

[143] Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2), 1992.

[144] Yikang Shen, Zhouhan Lin, Chin-Wei Huang, and Aaron Courville. Neural language modeling by jointly learning syntax and lexicon. In *Proceedings of the International Conference on Learning Representations*, 2017.

[145] Yikang Shen, Shawn Tan, Alessandro Sordoni, and Aaron Courville. Ordered neurons: Integrating tree structures into recurrent neural networks. *arXiv preprint arXiv:1810.09536*, 2018.

[146] Hiroyuki Shindo, Yusuke Miyao, Akinori Fujino, and Masaaki Nagata. Bayesian symbol-refined tree substitution grammars for syntactic parsing. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Volume 1, Long Papers*, pages 440–448. Association for Computational Linguistics, 2012.

[147] Patrice Y Simard and Hans Peter Graf. Backpropagation without multiplication. In *Advances in Neural Information Processing Systems*, pages 232–239, 1994.

[148] Richard Socher, Christopher D Manning, and Andrew Y Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, pages 1–9, 2010.

[149] Richard Socher, Jeffrey Pennington, Eric H. Huang, Andrew Y. Ng, and Christopher D. Manning. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 151–161, Edinburgh, Scotland, UK., July 2011. Association for Computational Linguistics.

[150] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[151] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[152] Mitchell Stern, Jacob Andreas, and Dan Klein. A minimal span-based neural constituency parser. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 818–827. Association for Computational Linguistics, 2017.

[153] Mitchell Stern, Daniel Fried, and Dan Klein. Effective inference for generative neural parsing. *arXiv preprint arXiv:1707.08976*, 2017.

[154] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25. ACM, 2016.

[155] Kai Sheng Tai, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[156] Ming Tan, Cicero dos Santos, Bing Xiang, and Bowen Zhou. Improved representation learning for question answer matching. In *Proceedings of ACL*, pages 464–473, Berlin, Germany, August 2016. Association for Computational Linguistics.

[157] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 173–180. Association for Computational Linguistics, 2003.

[158] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4489–4497, 2015.

[159] K. Tran, A. Bisazza, and C. Monz. The Importance of Being Recurrent for Modeling Hierarchical Structure. *ArXiv e-prints*, March 2018.

[160] Alan M Turing. Computing machinery and intelligence. In *Parsing the Turing Test*, pages 23–65. Springer, 2009.

[161] Max van Daalen, Pete Jeavons, John Shawe-Taylor, and Dave Cohen. Device for generating binary sequences for stochastic computing. *Electronics Letters*, 29(1):80–81, 1993.

[162] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

[163] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[164] Ivan Vendrov, Ryan Kiros, Sanja Fidler, and Raquel Urtasun. Order-embeddings of images and language. *arXiv preprint arXiv:1511.06361*, 2015.

[165] Oriol Vinyals, Łukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, and Geoffrey Hinton. Grammar as a foreign language. In *Advances in Neural Information Processing Systems*, pages 2773–2781, 2015.

[166] Zhiguo Wang, Haitao Mi, and Nianwen Xue. Feature optimization for constituent parsing via neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, volume 1, pages 1138–1147, 2015.

[167] Zhiguo Wang and Nianwen Xue. Joint pos tagging and transition-based constituent parsing in chinese with non-local features. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 733–742, 2014.

[168] Taro Watanabe and Eiichiro Sumita. Transition-based neural constituent parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing: Volume 1, Long Papers*, pages 1169–1179, 2015.

[169] David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. Structured training for neural network transition-based parsing. *arXiv preprint arXiv:1506.06158*, 2015.

[170] Jason Weston, Samy Bengio, and Nicolas Usunier. Wsabie: Scaling up to large vocabulary image annotation. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2764–2770, 2011.

[171] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Towards universal paraphrastic sentence embeddings. *arXiv preprint arXiv:1511.08198*, 2015.

[172] Adina Williams, Andrew Drozdov, and Samuel R Bowman. Learning to parse from a semantic objective: It works. is it syntax? *arXiv preprint arXiv:1709.01121*, 2017.

[173] Adina Williams, Andrew Drozdov*, and Samuel R Bowman. Do latent tree learning models identify meaningful structure in sentences? *Transactions of the Association for Computational Linguistics*, 6:253–267, 2018.

[174] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[175] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, pages 229–256, 1992.

[176] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*, 2018.

[177] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[178] Han Yang, Marta R Costa-jussà, and José AR Fonollosa. Character-level intra attention network for natural language inference. *arXiv preprint arXiv:1707.07469*, 2017.

[179] Wenpeng Yin and Hinrich Schütze. Convolutional neural network for paraphrase identification. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 901–911, 2015.

[180] Dani Yogatama, Phil Blunsom, Chris Dyer, Edward Grefenstette, and Wang Ling. Learning to compose words into sentences with reinforcement learning. *arXiv preprint arXiv:1611.09100*, 2016.

[181] Adams Wei Yu, Hongrae Lee, and Quoc V Le. Learning to skim text. *arXiv preprint arXiv:1704.06877*, 2017.

[182] Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*, 2015.

[183] Saizheng Zhang, Yuhuai Wu, Tong Che, Zhouhan Lin, Roland Memisevic, Ruslan Salakhutdinov, and Yoshua Bengio. Architectural complexity measures of recurrent neural networks. *arXiv preprint arXiv:1602.08210*, 2016.

[184] Xingxing Zhang, Liang Lu, and Mirella Lapata. Top-down tree long short-term memory networks. *arXiv preprint arXiv:1511.00060*, 2015.

[185] Ganbin Zhou, Ping Luo, Rongyu Cao, Yijun Xiao, Fen Lin, Bo Chen, and Qing He. Generative neural machine for tree structures. *arXiv preprint arXiv:1705.00321*, 2017.

[186] Junru Zhou and Hai Zhao. Head-driven phrase structure grammar parsing on penn treebank. *arXiv preprint arXiv:1907.02684*, 2019.

[187] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[188] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

[189] Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 434–443, 2013.

[190] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.