Université de Montréal

Distributed Conditional Computation

par Nicholas Léonard

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Août, 2014

Résumé

L'objectif de cette thèse est de présenter différentes applications du programme de recherche de calcul conditionnel distribué. On espère que ces applications, ainsi que la théorie présentée ici, mènera à une solution générale du problème d'intelligence artificielle, en particulier en ce qui a trait à la nécessité d'efficience. La vision du calcul conditionnel distribué consiste à accélérer l'évaluation et l'entraînement de modèles profonds, ce qui est très différent de l'objectif usuel d'améliorer sa capacité de généralisation et d'optimisation. Le travail présenté ici a des liens étroits avec les modèles de type mélange d'experts.

Dans le chapitre 2, nous présentons un nouvel algorithme d'apprentissage profond qui utilise une forme simple d'apprentissage par renforcement sur un modèle d'arbre de décisions à base de réseau de neurones. Nous démontrons la nécessité d'une contrainte d'équilibre pour maintenir la distribution d'exemples aux experts uniforme et empêcher les monopoles. Pour rendre le calcul efficient, l'entrainement et l'évaluation sont contraints à être éparse en utilisant un routeur échantillonnant des experts d'une distribution multinomiale étant donné un exemple.

Dans le chapitre 3, nous présentons un nouveau modèle profond constitué d'une représentation éparse divisée en segments d'experts. Un modèle de langue à base de réseau de neurones est construit à partir des transformations éparses entre ces segments. L'opération éparse par bloc est implémentée pour utilisation sur des cartes graphiques. Sa vitesse est comparée à deux opérations denses du même calibre pour démontrer le gain réel de calcul qui peut être obtenu. Un modèle profond utilisant des opérations éparses contrôlées par un routeur distinct des experts est entraîné sur un ensemble de données d'un milliard de mots. Un nouvel algorithme de partitionnement de données est appliqué sur un ensemble de mots pour hiérarchiser la couche de sortie d'un modèle de langage, la rendant ainsi beaucoup plus efficiente.

Le travail présenté dans cette thèse est au centre de la vision de calcul conditionnel distribué émis par Yoshua Bengio. Elle tente d'appliquer la recherche dans le domaine des mélanges d'experts aux modèles profonds pour améliorer leur vitesse ainsi que leur capacité d'optimisation. Nous croyons que la théorie et les expériences de cette thèse sont une étape importante sur la voie du calcul conditionnel distribué car elle cadre bien le problème, surtout en ce qui concerne la compétitivité des systèmes d'experts.

Mots-clés: calcul conditionnel distribué, réseau de neurones, apprentissage profond, apprentissage supervisé, apprentissage par renforcement, arbres de décisions, modèle de langage, softmax hierarchique, bloc éparse, mélange d'experts, torch

Summary

The objective of this paper is to present different applications of the distributed conditional computation research program. It is hoped that these applications and the theory presented here will lead to a general solution of the problem of artificial intelligence, especially with regard to the need for efficiency. The vision of distributed conditional computation is to accelerate the evaluation and training of deep models which is very different from the usual objective of improving its generalization and optimization capacity. The work presented here has close ties with mixture of experts models.

In Chapter 2, we present a new deep learning algorithm that uses a form of reinforcement learning on a novel neural network decision tree model. We demonstrate the need for a balancing constraint to keep the distribution of examples to experts uniform and to prevent monopolies. To make the calculation efficient, the training and evaluation are constrained to be sparse by using a gater that samples experts from a multinomial distribution given examples.

In Chapter 3, we present a new deep model consisting of a sparse representation divided into segments of experts. A neural network language model is constructed from blocks of sparse transformations between these expert segments. The block-sparse operation is implemented for use on graphics cards. Its speed is compared with two dense operations of the same caliber to demonstrate and measure the actual efficiency gain that can be obtained. A deep model using these block-sparse operations controlled by a distinct gater is trained on a dataset of one billion words. A new algorithm for data partitioning (clustering) is applied to a set of words to organize the output layer of a language model into a conditional hierarchy, thereby making it much more efficient.

The work presented in this thesis is central to the vision of distributed conditional computation as issued by Yoshua Bengio. It attempts to apply research in the area of mixture of experts to deep models to improve their speed and their optimization capacity. We believe that the theory and experiments of this thesis are an important step on the path to distributed conditional computation because it provides a good framework for the problem, especially concerning competitiveness inherent to systems of experts.

Keywords: distributed conditional computation, neural network, deep learning, supervised learning, reinforcement learning, decision tree, language model, hierarchical softmax, block-sparse, mixture of experts, torch

Contents

List of Figures

List of Tables

List of Abbreviations

BP	Back-Propagation
BLAS	Basic Linear Algebra Subprograms
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture
DCC	Distributed Conditional Computation
DL	Deep Learning
EM	Expectation Maximization
ESSRL	Equanimous Sparse Supervised Reinforcement Learning
GCN	Global Contrast Normalization
GPU	Graphics Processing Unit
HLBL	Hierarchical Log-Bilinear
KL	Kullback-Leibler
LCN	Local Contrast Normalization
LISA	Laboratoire d'Informatique des SystÃĺmes Adaptatifs
LM	Language Model
MAP	Maximum a posteriori
ML	Machine Learning
MLP	Multi-Layer Perceptron
MOE	Mixture of Experts
MNIST	Mixed National Institute of Standards and Technology
MSE	Mean Square Error
NDT	Neural Decision Tree
NLL	Negative Log Likelihood
NN	Neural Network
NNLM	Neural Network Language Model
RL	Reinforcement Learning
RBM	Restricted Boltzmann Machine
RDBMS	Relational Database Management System
SGD	Stochastic Gradient Descent
ZCA	Zero Component Analysis whitening

Acknowledgments

I'd like to thank many people who helped me on my journey resulting in this thesis.

I would like to thank my co-directors Yoshua Bengio and Aaron Courville for giving me their support and trust in pursuing my research, for their much appreciated guidance during the process, and for the great working environment.

I would like to thank my professors from the Royal Military College of Canada, in particular Robert Gervais, Lucien Haddad, Louis Massey and Alain Beaulieu, for teaching me mathematics and computer science in such an excellent way.

I would like to thank the Canadian Forces for providing me with education and employment experience in various management positions, for teaching me self-confidence, leadership, discipline and how to survive in the office. In particular I would like to thank Major Dany Boivin, Lcol Ross Ermel and Capt Luke Williams.

I'd like to thank my professors at Université de Montréal, in particular Pascal Vincent, Yoshua Bengio, Jian-Yun Nie and Roland Memisevic for teaching me machine learning, deep learning, information retrieval and computer vision, respectively.

I'd like to thank Ubisoft for providing me a practical research opportunity in the field of deep learning. I would also like to thank then for awarding me a research grant.

I'd like to thank Frédéric Bastien for keeping all of the computing and software infrastructure at LISA running smoothly, being patient with my Theano code contributions, and keeping an open mind to the alternative use of Torch7 in the lab.

I'd like to thank Zhen Zhou Wu for helping me implement some of the models used in this thesis.

I'd like to thank all my LISA Lab colleagues for providing such a nice workplace experience, in particular Xavier Bouthillier, Mehdi Mirza, Pierre-Luc Carrier, Yann Dauphin, Kyunghyun Cho, Vincent Dumoulin, Guillaume Desjardins, Raul Chandias Ferrari, Gauthier Viau, Bing Xu, David Ward-Farley and Ian Goodfellow. I learned from each of them, and had great discussions.

I'd like to thank my parents, Lucy-Ann Adam and Yves Roy, for raising me to value hard work, for always being there for me, and for teaching me family values.

I would also like to thank my girlfriend, Shilian Delvas, for supporting me throughout my many hours, days, weeks and months spent in front of my computer programming.

1 Machine Learning

Machine Learning (ML) is a vast field emanating from the field of Artificial Intelligence. It exists at the intersection of many fields including: probability and statistics, computer science and engineering, calculus, algebra, as well as many others. Its vastness makes it a difficult and time-consuming field to master. Any prospecting machine learning scientist must keep up with the many techniques available, or notwithstanding, learn enough of these to be able to function as a researcher or scientific programmer. The entire field is joined by a shared approach to problems, which is to build algorithms that have the ability to learn a desired behavior from data. The data itself is chosen to represent the target problem.

An alternative approach is to do away with data and use teams of experts to code a detailed algorithm to solve the particular problem explicitly, which involves no training of an algorithm using data. Instead, this learning process is abstracted away into the minds of the experts who produce the algorithm. On the other hand, the approach of machine learning is to artificially reproduce aspects of biological learning by implementing synthetic (or artificial) alternatives that can be algorithmically codified.

Algorithms that learn, or conversely, that can be trained, see the resulting learned behaviors more difficult to understand. Or at least we cannot understand them as directly as the more transparent algorithms used in symbolic AI, which often result in a kind of symbolic explanation. In the case of ML, the result is a kind of artificial *behavior*, which consists in reactions or actions that can be observed directly, while nevertheless having an internal process that cannot always be observed or understood explicitly. This hidden aspect of machine learning stems from its use of many free parameters, which often consist in real-valued numbers, that cannot be so easily interpreted or understood as leading to the observed behavior. The same is true for humans where the processes of the brain or mind cannot be easily understood as causing or operating the observed behaviors.

1.1 Types of Learning

To begin to understand ML, we should discern the different types of learning:

- Supervised Learning : roughly speaking, model $P(Y|X)$;
- Unsupervised Learning : roughly speaking, model $P(X)$;
- Reinforcement Learning : maximize reward R given environment states S and available actions A.

Supervised learning uses labeled data. It consists of random variables X and Y, respectively the input and the target, or label. Tasks are often divided into classification and regression (or prediction). Classification is about predicting one or many classes Y given inputs X. In other words, Y is a discrete random variable. Regression on the other hand consists in predicting continuous random variables Y given inputs X. The supervised learning problem space can often be modeled as a conditional probability distribution $P(Y|X)$.

Supervised learning problems, by their very nature, presuppose the existence of labeled data, which isn't always possible in practice. For unlabeled data, random variables X can be modeled through unsupervised learning. Where supervised learning tries to model causal relationships between inputs and targets, its unsupervised counterpart tries to find hidden structure in the data. Approaches include clustering, feature extraction, dimensionality reduction and hidden Markov models. Unsupervised learning can be used to model probability distribution $P(X)$. This can be done by introducing latent variables H. The problem space is then modeled as $P(X, H)$ where the objective is to synthesize random variables H where $P(X, H) \neq P(X)P(H)$. These can be of lower dimensionality than X, therefore simplifying X through abstractions. The opposite case, where the objective is to synthesize higher dimensional variables H, can also be sought out using techniques like denoising auto-encoders (Vincent et al. (2008)).

Reinforcement learning is very different from the above two approaches. In this scheme, the problem consists in optimizing the interactions of an agent with its environment states S. The agent can only learn from a very sparse reinforcement signal that usually takes the form of a reward R following a sequence of states and actions. The agent's interactions with its environment are limited to a set of actions A. The environment is often modeled as a Markov decision process and we use dynamic programming to learn to optimize the agent behavior to maximize reward.

1.2 Theory of Learning

In each of these algorithms, there are different trainable parameters θ. These are adapted to solve the problem, i.e. to fit the data (which frames the problem). These parameters often take the form of a set of tensors. Tensors are multivariate data like vectors (a 1-dimension tensor), and matrices (a 2-dimension tensor). The same is true for inputs x and y. For example, images usually take the form of 3D tensors, where dimensions represent colors (or channels), like red, green and blue (RGB), and the width and height of the image. Through predefined transformations $\hat{y} = F(\theta, x)$ on trainable parameters θ, where x is an instance or sample from random variable X, F is the parameterized transformation, and $\hat{y}$ is the prediction. Of course, this particular definition is limited to supervised learning, as it presupposes labels Y, which is the main concern of this thesis.

To train a transformation $F(\theta, X)$ is to search for the optimal values of θ that minimize the empirical risk:

$$\hat{R}(F_\theta, D_t) = \frac{1}{N} \sum_{x,y \in D_t} L\left(y, F(\theta, x)\right) \tag{1.1}$$

where D_t is the training set, N is the number of samples in D_t, and $L\left(y, F(\theta, x)\right)$ measures the error of the model in its prediction $\hat{y} = F(\theta, x)$ compared to the target y. Even though the θ, x and y often take the form of multi-dimensional tensors, this loss must be reduced to a single value (or scalar). This allows the scientist to measure the progress of learning as the minimization of risk $\hat{R}(F_\theta, D_t)$. Furthermore, if the transformation F is differentiable, it can be used for learning through gradient descent using the chain rule (or backpropgation). In any case, the basic learning problem can be reduced to the following:

$$\theta^\star = \operatorname*{argmin}_\theta \hat{R}(F_\theta, D_t) \tag{1.2}$$

In laymen terms, the objective is to find the parameters $\theta^\star$ that minimize the empirical risk over the training set. As we will see later, this does not cover the entirety of the learning tasks, as it does not consider the generalization ability of the model (or learning agent).

In machine learning, we often divide our data set D into 3 distinct sets:

$$D = D_t \cup D_v \cup D_e \tag{1.3}$$

where D_t, D_v and D_e are respectively the training, validation and test sets, and D is sampled from the problem space $P(X, Y)$. The training set is used for minimizing the empirical risk by optimizing θ. However, models with a great many degrees of freedom can over-fit the training set. This can be caused by a θ containing many variables, or by a transformation F with powerful non-linearities, such as high-order polynomials.

As is often the case with machine learning problems, the problem space $P(Y|X)$ cannot be sufficiently sampled, or represented by D. This is either due to the curse of dimensionality, or to the real-world cost of obtaining enough samples. This leads to problems of generalization, where the model cannot perform so well on data that it hasn't yet learned, i.e. that wasn't part of D_t.

A common technique to deal with this issue, is to withhold a portion of the available data D such that it can be used to approximate the expected risk (or generalization error):

$$R(\theta, F) = \mathbb{E}\left[L\left(Y, F(\theta, X)\right)\right] = \int\limits_{X,Y} L\left(Y, F(\theta, X)\right) p(X, Y) d(X, Y) \tag{1.4}$$

Unlike the empirical risk, which is measured over a sample of the problem distribution, the expected risk is over the true distribution, which is almost always unknown. The true objective of most supervised learning problems is to minimize the expected risk. This implies that the transformation F_θ, while being trained to minimize an empirical risk, must be constrained in such a way that it generalizes to the entire problem distribution $P(X, Y)$, which D only approximates.

We monitor the generalization ability of our model F_θ by measuring its risk (mean loss) over some held-out data (D_v and D_e). The validation set D_v is used for cross-validation. We can train several models with different hyper-parameters controlling various aspects of its learning, such as its capacity and its rate of convergence, and compare their mean loss (or validation error) over the validation set

D_v. We can also use the risk over D_v to keep only the version of the model with the lowest validation error (i.e. *early-stopping*) — as high capacity models tend to over-fit D_t after some training iterations. To over-fit is to minimize the empirical risk at the expense of the expected risk. To under-fit is the opposite.

The test set D_e is used for evaluation purposes. It shouldn't be used in any way that would bias the model in its favor. If we were to use D_e to perform cross-validation or early-stopping, then the model would become biased. So we hold-out D_e for the purpose of publishing results, comparing different approaches, or more generally, to provide equal unbiased grounds for evaluating the generalization ability of different approaches.

1.3 Artificial Neural Networks

Neural Networks are a very popular computational model used for solving complex supervised learning tasks like image classification and language modeling, and they are the basis of all models used in this thesis. Neural networks are modular in that they are divided into parameterized layers that can be assembled into a directed graph. Each layer can further be reduced to small units of computation called *artificial neurons* (figure 1.1).

The simplest layer is a linear affine transformation of the form:

$$y = h(x) = Wx + b \tag{1.5}$$

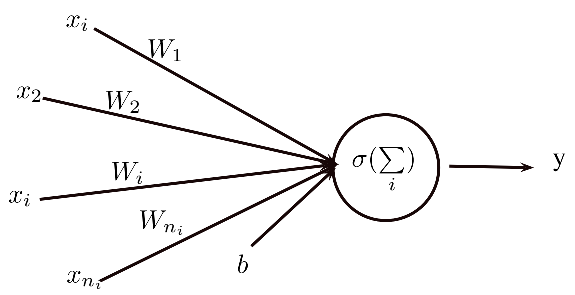


Figure 1.1: Graphical representation of an Artificial Neuron.

where b is a n_j-element bias vector, x is a n_i-element input vector, W is an $n_j \times n_i$ weight matrix. In this simplified model, parameters $\theta = \{W, b\}$. We can use this for linear regression, which is one of the simplest kind of supervised learning predictors.

1.3.1 Logistic Regression and Gradient Descent

If we wish to bound it between 0 and 1, we can introduce a non-linear transfer (or activation) function:

$$y = f(x) = \sigma(h(x)) \tag{1.6}$$

where σ is the logistic function:

$$\sigma(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{1.7}$$

and $h(x)$ is a linear transformation:

$$h(x) = Wx + b \tag{1.8}$$

Equation 1.6 is from a family of sigmoidal (S-shaped) functions. It can be used along with a quadratic loss function for the prediction of multi-dimensional bounded continuous targets:

$$L_q(y, \hat{y}) = \frac{1}{n_j} \sum_{j=1}^{n_j} (y_j - \hat{y}_j)^2 \tag{1.9}$$

where y is the target vector, and $\hat{y} = f(x)$ is the prediction. The empirical risk function that uses loss L_q is called the mean squared error (MSE). MSE is used for regression tasks, and can work directly on linear transformation (equation 1.5).

For binary classification we can use the cross-entropy loss function :

$$L_{ce}(y, \hat{y}) = \begin{cases} -\log(\hat{y}_j) & \text{for} \quad y = 1 \\ -\log(1 - \hat{y}_j) & \text{for} \quad y = 0 \end{cases} \tag{1.10}$$

Combining equation 1.6 (the transformation F) with the cross-entropy loss brings us closer to logistic regression. All that is missing is a learning algorithm. If we can obtain the gradient of the loss with respect to (w.r.t.) the parameters, we can perform iterative updates of the form:

$$\theta \leftarrow \theta - \eta \frac{\partial L_{ce}(y, \hat{y})}{\partial \theta} \tag{1.11}$$

where η is the learning rate, such that this simple model can be trained iteratively by stochastic gradient descent (SGD) (algorithm 1).

Algorithm 1 Stochastic Gradient descent algorithm for minimizing loss function L. Inputs include learning rate η, maximum iterations $\max_k$, and empirical risk (measured using ϵ_t) threshold $\min_\epsilon$ below which training is suspended.

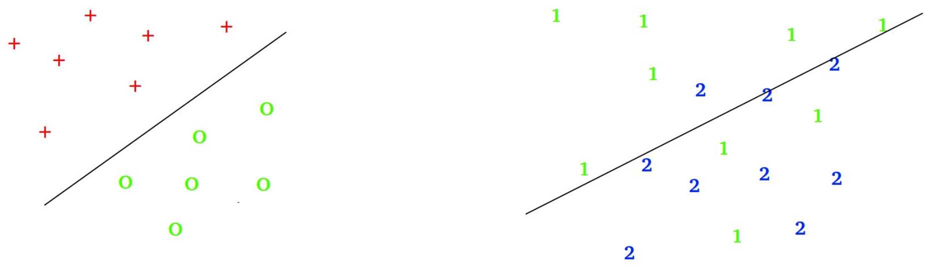
```
function SGD($D_t$, $\theta$, $\eta$, $\max_k$, $\min_\epsilon$, $L$)
    $k \leftarrow 0$
    repeat
        $\epsilon_t \leftarrow 0$
        for $(x, y) \in \boldsymbol{D_t}$ do
            $\hat{y} \leftarrow F(\theta, x)$
            $\epsilon_t \leftarrow \epsilon_t + \frac{1}{N} L(y, \hat{y})$        ▷ $\epsilon_t$ approximates empirical risk
            $\theta \leftarrow \theta - \eta \frac{\partial L(y,\hat{y})}{\partial \theta}$
        end for
        Shuffle $\boldsymbol{D_t}$        ▷ The shuffling makes it stochastic.
        $k \leftarrow k + 1$
    until $\epsilon_t < \min_\epsilon$ or $k = \max_k$
    return $\epsilon_t, \theta$
end function
```

In the particular case of logistic regression, parameter gradients $\frac{\partial L_{ce}(y,\hat{y})}{\partial \theta}$ can be obtained by a simple application of the chain rule:

$$\frac{\partial L_{ce}}{\partial \theta} = \frac{\partial L_{ce}}{\partial f(x)} \frac{\partial f(x)}{\partial h(x)} \frac{\partial h(x)}{\partial \theta} \tag{1.12}$$

where the gradient of the cross-entropy loss w.r.t. the output $f(x)$ is:

$$\frac{\partial L_{ce}}{\partial f(x)} = \begin{cases} \frac{-1}{f(x)} & \text{for} \quad y = 1 \\ \frac{1}{f(x)-1} & \text{for} \quad y = 0 \end{cases} \tag{1.13}$$



(a) Linear separable **(b)** Non-linear separable

Figure 1.2: Linearly separable (a), and a non-linearly separable (b) sets. While the first dataset can be separated by a straight line, the second cannot (it would require a curvy line, or multiple joined straight lines).

the gradient of the logistic function w.r.t. the affine transformation is $h(x)$:

$$\frac{\partial f(x)}{\partial h(x)} = f(x)(1 - f(x)) \tag{1.14}$$

and the gradient of the affine transformation w.r.t the parameters $\theta = \{W, b\}$: is

$$\frac{\partial h(x)}{\partial W} = x, \qquad \frac{\partial h(x)}{\partial b} = 1 \tag{1.15}$$

1.3.2 Multi-Layer Perceptron

Logistic regression for binary classification is however not very powerful. In particular, while it can learn to discriminate linearly separable data, it cannot do so for non-linearly separable data (see fig 1.2). This is due to the limited modeling capacity of the transformation F (equation 1.6). In particular, the linear transformation is limited to generating hyperplanes for dividing the data space into two regions (2-class discrimination). Even though this is followed by a logistic function (equation 1.7), this non-linearity only serves to bound the distance between each input x to the linear transformation's hyper-plane.

To overcome this limitation, neural networks[1] stack layers — each consisting of a linear transformation followed by non-linear activation function — to obtain a

1. Artificial neural networks, or neural networks, are also generally known as multi-layer perceptrons (MLP) for its relationship to the original Perceptron (Rosenblatt (1958)).

non-linear discriminator or predictor F. We have already seen the logistic (or sigmoid) activation function. There are other alternatives like the hyperbolic tangent (tanh), which is also sigmoidal yet bounded between -1 and 1:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{1.16}$$

and the rectified linear unit (Glorot et al. (2011)), which outputs (and has a gradient of) 0 for all non-positive inputs x.

$$\mathrm{relu}(x) = \max(0, x) \tag{1.17}$$

These are placed between linear transformations in order to introduce a non-linearity to F. Without these activation functions, stacked linear transformations could be simplified to a single linear transformation, which can only perform linear discrimination. The actual parameters are still to be found in these linear transformations. As in logistic regression, these are updated through gradient descent using the chain rule. We can divide our neural network into modular transformations (or layers) M:

$$M(\theta, x) = \sigma(Wx + b) \tag{1.18}$$

where $\sigma()$ is a differentiable activation function, including, but not limited to, any of the above non-linearities. Since we stack such layers, they can be indexed, along with their inputs and parameters, by their order ℓ in the sequence. The chain rule can then be applied to obtain the gradient of a loss L w.r.t. parameters θ_ℓ:

$$\frac{\partial L(F_\theta(x), y)}{\partial \theta_\ell} = \frac{\partial L(F_\theta(x), y)}{\partial M_\ell(x_\ell)} \frac{\partial M_\ell(x_\ell)}{\partial \theta_\ell} \tag{1.19}$$

There are however difficulties in applying this method to networks with many layers (or deep networks). For each layer M_ℓ the gradient passes through, a portion of the information found in the original gradient (w.r.t loss L) is lost.[2]

The gradient, by its very nature, is subdivided, aggregated, and non-linearly

2. We hypothesize that this loss is due to two factors: the activation and the linear transformation. The first is especially prevalent in networks that make use of sigmoidal activation functions as these tend to generate smaller gradients near the boundaries of the activation function, which has the effect of dissipating the gradients. As for the linear transformations, the loss is not so obvious. It occurs in the division of gradients through the incoming weights of an output neuron, and its converse, the aggregation of such signals from outgoing weights of input neurons.

scaled at various points in the computation graph. Undergoing a sequence of such transformations makes the causal relationship between the gradient and the loss decay as it moves down through the graph. This is even more true since all parameters are normally updated concurrently, as opposed to one at a time, which would be ideal (yet intractable) as the gradient w.r.t. to a parameter doesn't fully take into account that the other parameters will be updated as well.

1.3.3 Loss Functions

Neural networks have also been associated with other loss functions, such as the negative log likelihood (NLL) and the Kullback—Leibler divergence (KL-divergence). The first is almost exclusively used with the softmax activation function, which differs from those seen so far in that it does not operate in an element-wise fashion (each x_i computed independently of other $x_{i'}$) :

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum\limits_{i'}^{n_i} e^{x_{i'}}} \tag{1.20}$$

where n_i is the number of elements in the vector x. Since these are normalized to sum to one, it amounts to transforming the input x into a valid probability distribution $P(Y|X)$ over n_i categories (or classes).

The NLL loss is similar to cross-entropy (equation 1.10), but is used for multi-class classification problems (as opposed to just binary classification).

$$L_{NLL}(y, \hat{y}) = -\log P(y|x) = -\log(\hat{y}_y) \tag{1.21}$$

where $P(y|x)$ is the probability of class y (the target class) given input x, as predicted by the model. Its empirical risk is equivalent to maximum likelihood estimation, where the objective is to maximize the model's prediction of the likelihood of target classes given the inputs. In practice, using softmax for the output layer, the loss amounts to indexing (and negating) the output of the softmax at index $i = y$, where y is the target class.

The NLL can also be trivially extended for use with many-class classification problems. In such problems, the input x is associated to target y, where y is formulated as a vector of probabilities who's ith element contains the target probability of class i : $y_i = Q(Y{=}i|x)$. If we think of the original multi-class NLL in this way, then target y is a one-hot vector with target class y_i (same as original y in equation 1.21) having a $p(Y{=}y|x) = 1$, the remaining elements being zeros, such that :

$$L_{\text{NLL}}(y, \hat{y}) = -\sum_{i}^{n_i} y_i \log(\hat{y}_i) \tag{1.22}$$

The KL-divergence is different in that it can be used for a wider range of probability distributions: e.g. for predicting multinomial probabilities as described in equation 1.22 :

$$L_{\text{KL}}(y, \hat{y}) = \sum_i \log\left(\frac{Q(y|x)}{P(y|x)}\right) Q(y|x) = \sum_i \log\left(\frac{y_i}{\hat{y}_i}\right) y_i \tag{1.23}$$

Interestingly, the gradient of both L_{NLL} and L_{KL} w.r.t. $\hat{y}_i$ are the same:

$$\frac{\partial L_{\text{KL}}}{\partial \hat{y}_i} = \frac{\partial}{\partial \hat{y}_i} y_i \left(\log(y_i) - \log(\hat{y}_i)\right) = \frac{\partial}{\partial \hat{y}_i} y_i \log(\hat{y}_i) = \frac{\partial L_{\text{NLL}}}{\partial \hat{y}_i} \tag{1.24}$$

1.3.4 Weight Decay Regularization

When training supervised learning algorithms for the purpose of generalization, we often need to regularize the training of the model by introducing additional information in order to solve an ill-posed problem or to prevent overfitting. This can be accomplished by using techniques such as weight decay or a hard constraint on the maximum norm of weights. Weight decay is a very popular technique that involves modifying the loss function to include the L2-norm of the weights:

$$\Omega(\theta) = \Omega(W_1, W_2, ..., W_\ell, ...W_{N_\ell}) = \sum_\ell ||W||^2 = \sum_{\ell ij} W_{\ell ij}^2 \tag{1.25}$$

The regularized empirical risk then becomes:

$$\hat{R}(F_\theta, D_t) = \left(\frac{1}{N} \sum_{x,y \in D_t} L\left(y, F(\theta, x)\right)\right) + \lambda \Omega(\theta) \tag{1.26}$$

where λ is a coefficient modulating the importance of the regularization. It is a hyper-parameter that must be chosen through cross-validation as it affects the expected risk (equation 1.4). The gradient of the regularization term w.r.t. to weight matrix W_ℓ is then:

$$\frac{\partial \lambda \Omega(\theta)}{\partial W_\ell} = \lambda 2 W_\ell \tag{1.27}$$

During training, the effect is to keep the norm of the weight matrix small, such that the model does not over-fit the training set by using large weights. We will consider regularization using a hard constraint on the maximum norm of weights in section 1.4.6.

1.4 Deep Learning

The depth of a neural network is measured by the number of layers between input and output. Training MLPs with one, two or even three layers isn't very difficult. In most cases, the techniques presented in the last section can be applied to get such a model working. However, in many cases, using three or more layers isn't so easy as the limitations of gradient descent become more apparent, such as poor optimization and generalization (Bengio et al. (2007)) and (Erhan et al. (2009)). In this section, we explore some techniques used to facilitate the training of *deep* neural networks. Most of the techniques presented here are detailed, explained and consolidated in (Bengio (2012)), (Bengio et al. (2013)) and (Bengio (2009a)).

1.4.1 Activation Function

As discussed in section 1.3, different activation functions can be used to introduce a non-linearity into the network, thereby allowing the modeling of more complex boundaries. While it was once very popular, the logistic function (equation 1.7) has some issues. Unlike the tanh function (equation 1.16), although both are bounded, it can only output positive values.
Both these functions have a weakness in that the gradient resulting from activation values close to the boundaries are much smaller. This slows down learning and makes it more difficult for low and high activations to be modified in the opposite direction, thereby contributing to learning plateaus.
An alternative is to use the rectified linear unit (ReLU) (equation 1.17) (Glorot et al. (2011)). The gradient of activations below the threshold of zero is simply

zero, which means that such an activation never receives any gradient signal. As for the gradient of activations over zero — the linear part —, the gradient is passed backward as is. For the latter case, the gradient isn't scaled by any non-linearity. This part of the activation behaves like a simple linear activation function. This allows the training value of the gradient signal to be preserved for many more layers.

Although paradoxical, the ReLU is a non-linear activation function due to all negative inputs having an output value of zero. In most cases, ReLU learns much faster than the tanh or logistic functions. A caveat is that it requires that constraints be put to limit the size of weights or activations since it lacks an upper bound. The max norm constraint (section 1.4.6) works particularly well in this respect.

1.4.2 Parameter Initialization

All neural networks need to have their parameters initialized before training can commence. The biases can usually be initialized to zero and so can the weights of the parameterized output layer. The remaining weights on the other hand are more tricky. The simplest approach is to initialize weights using small values sampled from a normal or a uniform distribution.

Another simple technique, useful for layers that use ReLUs ((section 1.4.1), is to impose a very sparse weight matrix (with many zeros), by initializing only 15 random weights for each output neuron by sampling values from a normal distribution with mean 0 and standard deviation 1 (Martens (2010)). This technique, commonly known as *sparse initialization* (in the Pylearn2 and dp libraries), is the one that we most often use in practice, as it is simple to implement and is supported by empirical evidence.

This section wouldn't be complete without a technique based on the idea that units with more inputs should have smaller weights. Note that the above sparse initialization is based on this same principle, except that it initializes less weights. Scaling by the inverse of the square root of the fan-in (number of input neurons connected by weights to each output neuron) is a good practice (Orr and Müller (1998)). (Glorot and Bengio (2010)) use a combination of fan-in and fan-out where each weight is sampled from a uniform distribution between $-r$ and r where

$$r = \sqrt{\frac{6}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}} \tag{1.28}$$

is used for tanh activation functions and a variation can be used for the logistic function.

Pre-Training

A more complicated technique used for initialization is *pre-training*, where the initial network is constructed in a layer-wise fashion. Each layer is pre-trained using the outputs of the last layer (or in the case of the first layer, the inputs x) to perform unsupervised learning. Learning approaches include using restricted Boltzman machines (RBM) (Hinton et al. (2006)) — an energy-based model, denoising auto-encoders (DAE) (Vincent et al. (2008)) or contractive auto-encoders (CAE) (Rifai et al. (2011)). These techniques initialize the weights in a region which can lead to better generalization and which may be less prone to being caught in local minima. In cases were pre-training doesn't seem to provide better results than random initialization, it rarely yields worse results. While these techniques were more popular following 2006, they have recently somewhat fallen out of use due in part to recent developments, such as rectified linear units (section 1.4.1), which reduce their benefits, and also because the technique is more involved and introduces more hyper-parameters.

1.4.3 Preprocessing

Training can often benefit from performing some preprocessing on the raw data. In language modeling, the raw data is often a corpus of texts, where these need to be tokenized, rare words must be agglomerated into a single placeholder token, and each token can be translated into a number for reducing the memory footprint. Bag-of-word approaches have even more preprocessing techniques that can be used to reduce the importance of frequent tokens (stop-words), such as *tf-idf* (Jones (1972)), although neural networks can often learn these heuristics implicitly.
As for datasets that use images as inputs, preprocessing techniques are very dif-

ferent. Techniques include standardization which involves subtracting the mean and dividing the standard deviation of each feature. Global contrast normalization (GCN) optionally subtracts the mean across features and normalizes by either the vector norm or the standard deviation (across features, for each example) (Coates et al. (2011)). Zero-Component Analysis (ZCA) whitening (Bell and Sejnowski (1997)) is a computationally expensive technique used to making the input less redundant (whitening). The resulting features are less correlated with each other, and all have the same variance.[3] Local Contrast Normalization (LCN) (Pinto et al. (2008)) is another rather expensive technique used for performing local and subtractive normalizations which enforces a kind of local competition between adjacent features in a feature map and between features in the same spatial location in different feature maps, which is particularly useful for object recognition (Jarrett et al. (2009)).
Different combinations of preprocessing steps can be used. These should be determined during hyper-optimization as it isn't always obvious beforehand which combination works best.

1.4.4 Loss Function and Output Layer

In section 1.3.3, we explored different loss functions such as KL-divergence, cross-entropy, NLL and the quadratic error (or MSE). Each criterion is particularly suited for its range of problems which requires experience with different models and datasets in order to be applied correctly. Even with experience, finding the right combination of loss function and output layer will often follow some trial and error. Not all problems can make use of the loss functions presented here, but they can often make use of variations thereof.
An example would be facial keypoint detection, which requires predicting the coordinate of a number of keypoints given an input image. The initial reflex might be to use an MSE criterion on linear outputs, or maybe bound these using a logistic function and scaled targets.
However, this doesn't work well in practice as it doesn't correctly capture the spatial localities. An alternative solution is to model the output space as a vectors of approximately 100 values (a hyper-parameter) for each keypoint's x and y coordi-

3. see http://ufldl.stanford.edu/wiki/index.php/Whitening for a detailed overview.

nate. Each target value can be translated to a small (standard deviation of about 1) Gaussian blur centered at the keypoint coordinate. The blur increases the precision of the new targets as compared to just using a one-hot vector (a vector with one 1, the rest being zeros). The use of a gaussian blur centered on the target, which amounts to predicting multinomial probabilities, can be combined with the KL-divergence criterion to train a softmax output for each keypoint coordinate. This alternative approach actually works well in practice[4] and is a great example of the necessity of choosing the right loss function (or criterion) and commensurate output layer.

1.4.5 Dropout

The technique known as *dropout*, presented in (Hinton et al. (2012)), immediately became very popular, particularly in the field of computer vision where its benefits are more pronounced (Krizhevsky et al. (2012)). Dropout is extremely simple to implement as it basically involves multiplying samples (0 or 1) from a binomial distribution to the input of parameterized layers. During training, each hidden neuron is dropped (set to 0) with a probability of $p \approx 0.5$ and the output of this process (the non-dropped activations) are scaled by $\frac{1}{(1-p)}$. During testing, all neurons are used, and no scaling factor is applied. Empirically, the effects are a kind of regularization where the binomial noise tends to curtail over-fitting. The hypothesis presented in the original paper is that it effectively prevents the co-adaptation of feature detectors (neurons) and performs a kind of model averaging. The technique is known to work particularly well with ReLU activation functions (section 1.4.1) and Maxout Network (Goodfellow et al. (2013)).

1.4.6 Max Norm Constraint

The max norm constraint is a suitable replacement for weight decay regularization (section 1.3.4). It consist of a hard constraint on the norm of the incoming or outgoing weight of a neuron presented in (Hinton et al. (2012)) for use with dropout (section 1.4.5). However, the technique can be used to regularize the weights of

4. see https://github.com/nicholas-leonard/dp/blob/master/doc/facialkeypointstutorial.md for a concrete implementation used for submitting top 3 entries to the Kaggle Facial Keypoints Detection challenge.

any neural network. The constraint acts on the weights tensors of layers following a parameter update. When the L2-norm of incoming weights of a neuron is greater than threshold m, these are rescaled to have norm m while the remainder are left untouched.[5]. This is unlike weight decay, were all weights are decayed during each update. The m value should to be chosen such that it isn't so big that it has no effect, and not so small that it leaves no maneuvering room for learning. A good place to start is around the mean norm of incoming weights during training. In my experience, a value of 2, 4 or 8 will provide good results. The hyper-parameter m can of course be fine-tuned for each layer and through hyper-optimization.

1.4.7 Learning Rate Schedule

The learning rate η (equation 1.11) is the most important hyper-parameter as it controls the size of the gradient descent step. Choose a value too large and the model will diverge to generate very large errors which may eventually introduce NaN values throughout the system. Choose a value too small and training will advance too slowly. High learning rates can help the model get out of local minima and explore widely different configurations in the parameter space. Lower values allow training to work out the dents in the model yet make it more prone to getting stuck in local minima.
This is why it is often a good idea to use the highest possible values (without diverging) at the beginning of training. Different schedules can be used to achieve this. These include exponential and linear decays of the learning rate. Our own experience has however led us to prefer a step schedule where the learning rate is decayed instantaneously at different points in training (e.g. between specific epochs). The points at which learning rate is decayed, and the amount of decay is a hyper-parameter to optimize. The main reason why this approach is preferred is that the resulting learning curves are more informative. The different errors can often be seen to instantaneously fall at such points, thereby making the effect of the decay more clear. And when the decay has no visible effect, we can conclude that any further reduction will have no or little effect, and that the learning rate may in fact be too low.

5. The max norm operation is available in optimized C and CUDA code for Torch7 : https://github.com/torch/torch7/blob/master/doc/maths.md#torch.renorm

1.4.8 Momentum

Momentum is a technique that involves keeping an exponentially moving average of parameter gradients and use these for updates instead of the current gradients themselves (Hinton (1977)):

$$v_t = \beta v_{t-1} + (1-\beta)\frac{\partial L}{\partial \theta} \tag{1.29}$$

$$\theta_t = \theta_t - \eta v_t \tag{1.30}$$

where β is the momentum factor, η is the learning rate and t indexes the current training iteration. We can think of this as a kind of second-order gradient descent method that can help both accelerate learning and overcome local minima in stochastic gradient descent (algorithm 1). Many variations of the approach exist, including task-tailoring the initialization for momentum learning (Sutskever et al. (2013)), or a accumulating velocity vectors in the directions of persistent reduction across iterations (Nesterov (1983)). Higher β values should likely be used with lower η. In any case, both these hyper-parameters tend to interact, thus requiring that a change in one be accompanied by a complementary shift in the value of the other.

1.4.9 Convolutional Neural Networks

The earliest successful application of what is now known as deep learning can likely be attributed to the discovery of convolutional neural networks (CNN) (Fukushima (1980)) and its application to hand-written character recognition (LeCun et al. (1998)). A parameterized convolution layer is usually divided into three steps:

1. Convolution
2. Activation
3. Pooling (or sub-sampling)

The first step involves convolving a set of small feature detectors over an input feature map. If the input is a 3D tensor — an image with several channels (e.g. an RGB image of dimensions $3 \times 32 \times 32$) —, each feature detector takes the form of a 3D tensor, although much smaller (e.g. $3 \times 5 \times 5$). These feature detectors (or kernels) are convolved over the image's height and width dimensions (a 2D convolution). This means that every s pixels, where s is the stride of the convolution, the kernel is compared to a patch of the same size around the current pixel:

$$y_{kl} = h_{kl}(x) = \sum_{ij \in \text{patch}(kl)} W_{ij} x_{k+i,l+j} + b_{kl} \tag{1.31}$$

where i, j are the coordinate of pixels in the input patch centered on the commensurate coordinates k, l of the output feature map. So the parameters of the convolution are *reused* (or shared) at many positions of the input image. This parameter sharing does wonders for learning as gradient signals originate from many different positions in the convolution for each example. Even without this learning advantage, using such feature maps has many statistical advantages as features tend to be extracted from natural images more easily by first detecting small patterns like those obtained from Gabor filters (Marčelja (1980), Field (1987)), and detecting more and more abstract features as more layers of convolutions are stacked.

The activation serves its usual function of introducing a non-linearity into the network thus providing more modeling capacity (section 1.3.2). For the purpose of convolutions, ReLUs with dropout tend to work really well (Krizhevsky et al. (2012)). An expansion on this idea is Maxout networks (Goodfellow et al. (2013)) which have been very successful in image classification tasks using CNNs.

The output of the 2D convolution is a 3D Tensor with a depth that is equal to the number of kernels used. Height and width on the other hand, when using a stride $s = 1$ will remain approximately constant, loosing only a portion due to border effects, which varies depending on the size of the kernel. This results in very large output representations where much of the contained information may be redundant for contiguous pixels. This brings us to the pooling layer whose task it is to reduce the redundancy and size of the representation. Different pooling techniques can be used, the most common being *max-pooling*, which takes the maximum value in a patch of the convolution output. The width and height of this pooling patch is another hyper-parameter. Furthermore, the pooling operation is also performed as a convolution — this time on the output representation — which means that another stride can be specified. The larger the stride and dimensions of the patch, the greater the reduction in the representation. Other pooling operations include averaging, and stochastic pooling (Zeiler and Fergus (2013)), among others.

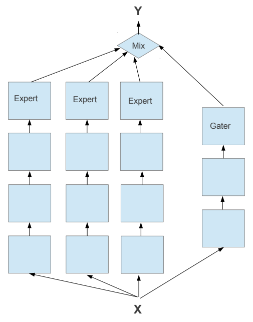
Many layers of parameterized convolutions, as presented above, can be stacked to obtain higher-level feature maps. The output of this process can then be flattened to a vector of features and fed into an MLP up to the output layer. CNNs of this form are very successful and have been used to solve a variety a tasks, breaking many image recognition records.

1.5 Mixture of Experts

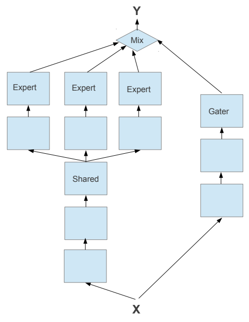
A mixture of experts (MOE) is an application of the principle of "divide and conquer". The standard model consists of multiple expert networks and a gater network (Jacobs et al. (1991)), (Nowlan and Hinton (1991)), as illustrated in fig. 1.3a. The gater takes an input x to generate an output for each of the experts. The purpose of the gater is to distribute importance g among experts given an input x. The purpose of experts is to predict y given x (Xu and Amari (2009)). Gater and experts work together by dividing the problem space into simpler sub-problems for which the experts are to be made specialists.

The main difficulty in learning MOEs is expressed by the *chicken and egg* paradox. In order to train effective experts, the gater must first learn to divide the problem into simpler sub-problems for the experts to learn to solve. Yet what makes a sub-problem simple, in the context of gradient descent, is that the experts can solve it with greater ease than a single *generalist* that learns to solve all sub-problems as a singular whole, as in non-MOE models. In laymen terms, the experts and gater are all co-dependent. This makes it difficult for the gradient descent to find a suitable division of labor.

One approach to this problem is to separate the training of the gater from that of the experts. This can be accomplished using Expectation-Maximization (EM) (Dempster et al. (1977)) in order to divide the learning into two phases that iterate until convergence (Jordan and Jacobs (1994)). Another approach is to train the experts $P_k(Y|X)$ on different subsets of the dataset and to train local unsupervised gaters $P(X|E = k)$ using Gaussian Mixture Models (GMM), which are then used with Bayes theorem to estimate an "easier" posterior distribution $P(E = k|X)$ of the dataset among experts. (Collobert et al. (2003)) iterate this process until convergence and show how this method can be used to create hard parallel mixtures



(a) Mixture of Experts



(b) Shared Mixture of Experts

Figure 1.3: A mixture of three four-layer experts and a three-layer gater. In *b*, these 2 layers are shared among experts, yet separate from the gater.

of experts.

The earliest means by which to train a mixture of experts is through simple back-propagation. The gater network uses a softmax activation function on its final layer, thus ensuring that all expert coefficients sum to one:

$$g_k = \text{softmax}\left(\sum_i w_{ki}x_i + b_k\right) \tag{1.32}$$

where g_k is the relative weight of expert k, x_i is the output of the i-th unit of the previous layer of the gater, b_k is the bias of unit k, w_{ki} is the strength of the connection from unit i to unit k. The output of each expert k is multiplied by the corresponding expert coefficient g_k provided by the gater, and the resulting gater-weighted expert outputs are summed element-wise to get the final network prediction $\hat{y}$. This process of accumulation is illustrated as the diamond **MIX** in figure 1.3a), and takes the following form:

$$h_j = \sum_k e_{kj}g_k \tag{1.33}$$

e_{kj} is the output of unit j of expert k and in the context of classification, it is calculated as

$$e_{kj} = \text{softmax}\left(\sum_i w_{kij}r_{ki} + b_{kj}\right) \tag{1.34}$$

where r_{ki} is the output of the i-th unit of the previous layer of expert k, b_{kj} is the bias of unit j, w_{kij} is the strength of the connection from unit i to j for expert k. The usual NLL loss function can be used on the output of the gater-weighted accumulation of these expert outputs as the result will still sum to one.

A hard mixture of experts is a most promising application of the divide and conquer principle since it may provide great speedups during both training and testing (Collobert et al. (2003)). The gater chooses only a subset of all experts to be propagated through, thus allowing a great number of weights without the usual implication that this results in a greater strain on processing resources such as GPU and CPU time. Our primary motivation in focusing our research on mixtures of experts was this potential efficiency gain.

Variants on MOEs can also be exploited where the first layers are actually shared by all experts (figure 1.3b), or where each layer has its own dedicated gater and experts (Eigen et al. (2013)). In the chapter 2, we will explore MOE in the context of the more general family of conditional models.

2 Image Classification using DCC

2.1 Introduction

In the beginnings of the Internet, TCP/IP networks were made out of hubs, which worked on layer 1 of the OSI model (Braden (1989)). Yet a network hub is an unsophisticated device in comparison with, for example, a switch. A hub does not examine or manage any of the traffic that comes through it: any packet entering any port is rebroadcast on all other ports. A switch processes data at layer 2, while a router processes it at layer 3 of the OSI model. This makes switches and routers much smarter than hubs, such that the latter have fallen out of use. The state of the neural network today is that of a hub: all input signals are broadcast to all output neurons.
Distributed Conditional Computing (DCC) is analogous to a switch in this metaphor. The basic idea is to route neural signals to the appropriate neurons, such that each prediction requires processing but a small fraction of the model's capacity. In this thesis, we will attempt to demonstrate the efficiency and effectiveness of DCC. We first discuss the components of conditional models, provide a concrete implementation of a training algorithm for image classification using DCC, and conclude by presenting experimental results.

2.2 Conditional Models

The vision of DCC is where a gater, formed of *sparse stochastic* output units, generates a distributed representation of decisions z_k that can turn off experts in combinatorially many ways, thus doing away with large chunks of the computation associated to these experts. Previous experiments have show that networks can be trained successfully in spite of such noise and hard non-linearities (Bengio et al.

(2013)).
Gater G is responsible for choosing experts E_k for the task of processing X into Y. The purpose of the gater is to provide *guidance* for the flow of signal propagations through experts. While that of the experts is to provide the *expertise* to model the target data distribution $P(Y|X)$.

2.2.1 Experts: Pieces of the Puzzle

The experts E_k can be anything from a full blown multi-layer perceptron (MLP) as in section 1.3, to a single-output linear transformation as in equation 1.5.
In this chapter, we define experts $E_k(Y|X)$ as the following function:

$$y_k = \sigma(xW_k + b_k) \tag{2.1}$$

where y_k is an output vector of size N_j, x is an input vector of size N_i, W_k is weight matrix of shape $N_i \times N_j$, b_k is a bias vector of size N_j, and $\sigma()$ is an optional activation function, like the logistic or hyperbolic tangent functions, etc.
N_j is an important hyper-parameter for it controls the granularity of the gater's ability to coordinate the expertise (the set S_e of all experts E_k). A small value of N_j is the equivalent of micro-management, where the gater gets to choose exactly which individual units to activate in the concatenated output representation:

$$y = y_1 \parallel y_2 \parallel \cdots \parallel y_{N_k} \tag{2.2}$$

where $\parallel$ is the concatenation operator. While this ability to micro-manage might allow the conditional model to optimize a cost function, it may have negative effects on generalization such that the gater has too much *switching capacity*. Furthermore, gater granularity has a cost in terms of the increased complexity of the gating task. Given a fixed size of $N_kN_j = N_{out}$ for vector y, the gater must generate an output of size $N_k = \frac{N_{out}}{N_j}$, and thus $N_k \propto \frac{1}{N_j}$. In simpler terms, again for a fixed y bandwidth N_{out}, the gater will also have a smaller bandwidth N_k for larger values of the y_k bandwidth N_j. Smaller values of N_k reduce the complexity of the gating problem [1]

1. The more classes to discriminate, the harder the multi-class classification problem.

2.2.2 Gater : A Second Order of Intelligence

A gater $G(E|X)$ is assigned with the task of choosing the optimal combination of experts for any example x, y sampled from $P(X, Y)$ such that the empirical risk $\hat{R}(F_\theta, D_t)$ (see equation 1.1) is minimized. We model a prototypical gater as a multinomial distribution [2], where each class represents an expert. Given x, experts might be sampled (with or without replacement) from the gater, or we might take the *maximum-a-posteriori* (MAP) experts, or we might multiply each probability $z_k = G(E = k|X)$ by the corresponding expert output $y_k = E_k(Y|X)$ and pool the expert outputs element-wise, etc.

The task of gating is not an easy one to learn through backpropagation (BP) (Rumelhart et al. (2002)), or deep learning (DL) (Bengio (2009b)) approaches. This is in part because we constrain its probability distribution $G(E|X)$ to be sparse (sample a small subset of experts), and yet balanced (each expert gets sampled almost as often). The main difficulty with BP/DL approaches could also reside in stabilizing $P(Y|X)$ for all k. During training, the nature (or definition) of k expert distributions

$$P(Y|X) = \sum_k G(E{=}k|X)E_k(Y|X) \tag{2.3}$$

tend to fluctuate a great deal.

Yet even this simpler problem would prove difficult to solve in practice, for learning must still be coordinated between gater and experts. Initialized with random parameters, the conditional model must learn to separate the task $P(Y|X)$ into specialized sub-components that must be learned by experts $E_k(Y|X)$. Yet at the same time, the gater must learn how to divide the task into sub-tasks, and how to assign them to experts given x. Thus we are dealing with a chicken and egg paradox, where the experts require a fully trained gater to learn (or fit) $E_k(Y|X)$, and where the gater requires fully trained experts to know how to best distribute examples to them.

Given the complexity of the task, there is great risk that the model falls prey to the simplest of solutions, which might be to always choose the same best experts, or to use all equally to form a kind of model averaging.

2. A bernouilli distribution could also be used, such that each expert is sampled independently, but previous work (Bengio et al. (2013)) has shown that enforcing a sparsity constraint on these is not so easy as with a multinomial, where we need only sample experts without replacement until the desired sparsity is reached.

The ideal situation is for the gater to choose the optimal configuration of experts K^+ such that the resulting loss L can be minimized over samples x, y. Ideally, a gater would be sophisticated enough to predict the L of each combination of experts given x, such that the optimal combination could be chosen. This means that the gater is a kind of second order intelligence able to predict which combination of experts will minimize the error. This might be an indication why training a conditional model using BP doesn't work as well as one would expect.

During training, the sampling of experts from a gater distribution given x is a form of competitive learning since only the sampled experts have the opportunity to learn $P(Y|X)$. The stochastic gater takes on the form of a sampling probability distribution $G(E|X)$, such that only a subset of experts are sampled given an example. Since $G(E|X)$ influences which experts get an opportunity to learn, and since we would like each expert to have an equal opportunity to learn, as well as equal utility, the gater must be constrained such that

$$P(E = k) \approx \frac{1}{N_k}, \text{ for } 1 \leq k \leq N_k \tag{2.4}$$

where N_k is the number of experts, and $P(E)$ is the sampling frequency of experts from $G(E|X)$. We call this constraint *equanimity*.

While the distribution of experts must be equanimous (balanced), the gater must also sample experts such that their contribution to the error is minimized[3]. In order for our gater to be effective, we need a means of measuring each sampled expert's contribution to the error. The gater must then make use of these metrics to increase the probability of sampling the winning experts[4]. But never to the point that the same experts get sampled for all examples[5]. The solution to this problem implies a constraint on the distribution $P(E)$ to be uniform across E. The difficulty lies in implementing such a constraint.

One possible implementation involves sampling examples (from the current batch) for each expert from $P(X|E) = \frac{P(E|X)P(X)}{P(E)}$. This hard-constraint on the sampling

3. These two objectives will sometimes pull the training forces in different directions, resulting in a kind of flux during training.

4. Winning experts: those experts that contributed most to minimizing the cost of the prediction, for a given example.

5. Experts are randomly initialized and the ordering of examples is randomized, such that some experts will have a random hebbian advantage. They will be quicker to learn $P(Y|X)$, and if the gater catches-on to this, they will be sampled more often.

distribution $G(E|X)$ allows each expert an equal opportunity to specialize, for they are guaranteed a chance at learning at least one example. But a chance to learn may not be enough. If we would like to guarantee that each expert gets to learn something from a mini-batch, we need to sample at least two examples for each expert, such that their contribution to the error can be compared, and the expert can learn to model the least error-prone example $x \rightarrow y$.[6]

2.2.3 Accumulators : Pooling and Concatenation

The output of experts can either be pooled, concatenated or a combination thereof. Various types of accumulators A_* exist such as a the weighted mean used in mixtures of experts (MoE) pooling:

$$A_{moe}(z_{1\rightarrow N_k}, y_{1\rightarrow N_k}) = \sum_{k \in K^+} f\left(\frac{z_k}{\sum\limits_{k' \in K^+} z_{k'}}, y_k\right) \tag{2.5}$$

where $f(\alpha, y)$ is an element-wise function that scales y according to α, such as $f(\alpha, y) = \alpha y$ or y^α, etc., and $a \rightarrow b$ is the set of all integer values $x \; \forall \; a \leq x \leq b$. Pooling may also take the form of a product of experts (PoE):

$$A_{poe}(z_{1\rightarrow N_k}, y_{1\rightarrow N_k}) = \prod_{k \in K^+} f\left(\frac{z_k}{\sum\limits_{k' \in K^+} z_{k'}}, y_k\right) \tag{2.6}$$

Accumulators may also be stacked, as in Maxout pooling (Goodfellow et al. (2013)). In this model the gater and the experts are fully integrated. Each dimension of a layer's output y, or lane, is represented by its own array of approximately $N_k \approx 5$ experts. Each such lane pools its experts using a max pooling accumulator:

$$A_{max}(z_{1\rightarrow N_k}, y_{1\rightarrow N_k}) = \max_k(y_k) \tag{2.7}$$

6. Yet different examples have different losses, so how can we compare expert gradients? A proposed solution is to divide each example's gradient amplitude by its sum of gradient amplitudes such that the relative error is normalized to sum to one. Comparison can then be made between examples, and the example with the lowest relative gradient amplitude will learn. An thus, each example and expert gets to learn. We can then train the winning expert by allowing the flow of gradients backwards to its parameters, while the loser(s) learn nothing. We have not tried this.

where y_k is obtained from eq. 1.5. These units are linear-activated before any gating can occur, such that no immediate efficiency is gained from the gater[7]. In the framework of our conditional model, the gater and experts of a lane share the same parameters from eq. 1.5. The gater imposes an output sparsity of $\frac{1}{N_k}$ by use of the heuristic $\max\limits_k()$. In each lane (set of N_k experts), the only experts that get to learn are those having maximum output y_k for example x. We can thus imagine the experts as having individual weights of $w_k = 1$ with no bias connecting the output y_k to output y of each lane, such that the gater selects which of these connections to use. Each output y_l for lane l is accumulated through concatenation:

$$A_{cat}(y_{1\to N_l}) = y_1 \parallel y_2 \parallel \cdots \parallel y_{N_l} \tag{2.8}$$

where N_l is the number of lanes. This is the second accumulator (over lanes) used for Maxout pooling.

2.3 Equanimous Sparse Supervised Reinforcement Learning

In this section, we discuss a concrete training algorithm for DCC aiming to solve the difficulties encountered with the use of stochastic units, in training conditional models composed of inter-dependent gaters and experts. Recall $P(X,Y) = P(X)P(Y|X)$ is our target data distribution. Let $E_k(Y|X)$ be our $1 \leq k \leq N_k$ expert functions taking the form of eq. 2.1. Let $G(E|X)$ be our gater distribution taking on the form of an MLP of one or many parametrized non-linearities. Its output non-linearity is a sigmoid $g_k = \frac{1}{1+e^{-h_k}}$, where h_k is the result of the final linear transformation of the MLP. These sigmoid activations are then softmaxed:

$$z_k = \frac{\exp(g_k)}{\sum\limits_{k'}^{N_k} \exp(g_{k'})} \tag{2.9}$$

7. Although one might argue that efficiency is obtained for the next layer due to the pooling of expert representations y_k. Nevertheless, this type of accumulator doesn't qualify as conditional computation as all experts need to be computed for each example.

to form the probabilities of a multinomial distribution [8]. The gater is thus a conditional parametrized multinomial probability distribution. $M(Y|X)$ is our conditional model:

$$M(Y|X) = \underset{k:1\rightarrow N_k}{A_*}[G(E{=}k|X), E_k(Y|X)] \tag{2.10}$$

where A_* is an accumulator from section 2.2.3. Eq. 2.10 has a constraint for sparsity, and another for equanimity. The first is a sparsity constraint on $G(E|X)$ such that a majority of experts $K^-(x)$ have $G(E \in K^-(x)|X = x) \approx 0$, and very few $K^+(x)$ have $G(E \in K^+(x)|X = x) \approx \frac{1}{n_{test}}$, for x sampled from $P(X)$, where n_{test} is the number of desired experts per example for test runs. This constraint allows for speedups in test-processing, as only a small fraction of top experts are required for approximating 2.10:

$$M(Y|X{=}x) \approx \underset{k\in K^+(x)}{A_*}[G(E{=}k|X{=}x)E_k(Y|X{=}x)] \tag{2.11}$$

$$\text{where } K^+(x) = \underset{n_{test},k}{\text{argsmax}}[G(E{=}k|X{=}x)] \tag{2.12}$$

and where n_{test} is the quantity of *maximum-a-posteriori* (MAP) experts used per example (during testing), and $\underset{n,k}{\text{argsmax}}[x_k]$ is a function taking the n arguments k having the maximum x_k.

During training, we sample $n_{train} > n_{test}$ experts per example to allow for stochastic exploration of $E_k(Y|X)$ for $1 \leq k \leq N_k$, such that the empirical risk can be minimized. The sparsity constraint is such that only n_{test} of n_{train} of N_k expert distributions $G(E = k|X = x)E_k(Y|X = x)$ will be reinforced on example x, y sampled from $P(X, Y)$. Reinforcement occurs in both gater and experts. For a given example, the gater increases the sampling probability of n_{test} winning experts, while these same experts learn $P(Y|X)$ through BP/DL.

The gater increases the sampling probability of an expert by targeting the n_{test} commensurate sigmoid outputs to one, and the $|V^-| = n_{train}$-n_{test} losers to zero

8. Note that the sigmoid limits the range of outputs allowed for the softmax, which isn't optimal.

using the mean-squared-error criteria[9]:

$$\frac{\sum_{k \in V^+} (g_k - 1)^2 + \sum_{k \in V^-} (g_k - 0)^2}{|V^+ \cup V^-|} \tag{2.13}$$

For a very uniform distribution of $G_{train}(E|X)$, this would cause problems in terms of the desired sparsity of $G_{test}(E|X)$[10], since the pull towards 1 and the pull towards 0 wouldn't agree with the desired sparsity. However, we reinforce experts having the least error for a given example x:

$$V^+(x, y) = \underset{n_{test},k}{\text{argsmin}}\ C_k(x, y) \tag{2.14}$$

such that $G_{train}(E|X{=}x)$ is expected to favor $V^+(x, y) \approx K^+(x)$ samples. In which case, we say that experts $V^+(x, y)$ *monopolize* $G(E|X{=}x)$ for a given x. Therefore, the sparsity constraint will be respected.
Yet some experts may come to quickly monopolize $G(E|X)P(X)$ such that other experts will not be given a chance to learn (and catch up), and will see their capacity underutilized. A kind of socialist, egalitarian, balancing, or for lack of better word, *equanimity* constraint, is required. This constraint must ensure that each expert monopolizes an approximately equal share of $P(X, Y)$, as described in eq. 2.4. To train our conditional model, we explored different training algorithms which incorporate these two constraints while minimizing empirical risk.
We found that the most effective way of implementing the equanimity constraint was by inserting a simple step after the gater softmax and before the multinomial sampling.[11] It uses a small cache keeping track of the most recent examples seen to re-normalize the current softmax output y such that the probability of popular experts is reduced, or conversely, the probability of under-utilized experts is increased:

This works well in practice as the function can be inserted after any softmax, it is simple, and does indeed have the desired balancing effect. It can also be easily backpropagated through by caching some of the intermediate values (please refer

9. Unsampled experts are not reinforced in gater. Note also that the softmax and NLL combination was also explored, offering very similar experimental results.

10. Unless $\frac{n_{test}}{n_{train}} \approx \frac{n_{test}}{N_k}$

11. The code for this has been made available online as its own Torch Module https://github.com/clementfarabet/lua—nnx/blob/master/Balance.lua

Algorithm 2 Algorithm for constraining the $y = P(E|X)$ output of a softmax with N_k categories to have uniform probability $P(E) = \frac{1}{N_k}$. Note that k indexes categories (in this case, experts), $0 < \alpha \leq 1$ is the weight of the past in the moving average s (a vector of length N_k). The comments outline what probabilities each step approximates.

```
1: function EquanimityContraint($y,\ \alpha,\ s$)
2:     $p = \frac{s}{\sum_k s_k}$        ▷ $\approx P(E) = \sum_x (P(E|X{=}x)P(X{=}x))$
3:     $l = \frac{y}{p}$        ▷ $\approx P(X|E) = P(E|X)P(X)/P(E)$
4:     $z = \frac{l}{\sum_k l_k}$        ▷ $\approx P(Z|X) = P(X|E)\sum_k (P(X|E{=}k))$ where $P(Z) = \frac{1}{N_k}$
5:     $s = \alpha s + (1 - \alpha)y$
6:     return $z$
7: end function
8:
```

to source code). The α should be set to a higher value when $P(E)$ seems to remain unbalanced.

As for the sparsity constraint, sampling without replacement from a multinomial distribution with probabilities equal to z (from algorithm 2) is a hard constraint which guarantees the desired sparsity. The entire training algorithm can be summarized as follows:

Algorithm 3 Algorithm for training conditional model $M(Y|X)$ (equation 2.10) using sparse approximation (equation 2.12). Arguments include gater $G(E|X)$ and N_k experts $E_k(Y|X)$ (which are components of $M(Y|X)$), training set D_t, number of training samples n_{train} and number of reinforced experts n_{test}. The gater uses the equanimity constraint as defined in algorithm 2 to draw samples from a multinomial probability distribution *without replacement.*

```
 1: procedure ESSRL($G(E|X),\ E_k(Y|X)\ \forall\ 1 \leq k \leq N_k,\ D_t,\ n_{train},\ n_{test}$)
 2:     repeat
 3:         Sample example $x, y$ from $D_t \approx P(X, Y)$
 4:         Sample $n_{train}$ experts from gater $G(E|X{=}x)$
 5:         for sampled experts $E_k$ do
 6:             Forward propagate $x$ to obtain likelihood vector $\hat{y}_k = E_k(Y|X{=}x)$
 7:             Measure loss $L_k(\hat{y}_k, y)$
 8:         end for
 9:         for $k \in \underset{n_{test},k}{\text{argsmin}}\ L_k(\hat{y}_k, y)$ do        ▷ Reinforce $n_{test}$ winners
10:             Increase sampling probability $G(E = k|X{=}x)$ as per equation 2.13
11:             Backpropagate through $E_k(Y|X{=}x)$
12:             Update $E_k(Y|X{=}x)$ w.r.t. $L_k(\hat{y}_k, y)$
13:         end for
14:     until Training converges
15:     return $z$
16: end procedure
17:
```

2.3.1 Tree Architecture

The idea of organizing an MLP as a tree and training it through BP/DL was first described in (Ornstein (1996)). An attempt was made to perform a kind of unsupervised BP classification, where a decision tree is built iteratively (one layer at a time). We build on this idea by organizing our conditional model as a tree of expert arrows and gater nodes, which we call a Neural Decision Tree (NDT). The input of the tree is near the root while leaf-expert output layers are shared [12], and fed into a pooling accumulator to get the final prediction $\hat{y}$ (see fig. 2.1). The tree is composed of N_ℓ layers of $(N_k)^\ell$ experts each.

We modify algorithm 3) such that each gater-node $G_z^\ell(E^\ell|X, \text{parent}(z))$ samples n_{train} of its N_k child expert-branches E_k^ℓ for each of its input examples, where z and k index the gaters and experts of layer ℓ. [13]. This would allow each example to consider $(n_{train})^{N_\ell}$ experts for potential reinforcement, where only n_{test} experts are reinforced.

During testing, an example is propagated through the tree by taking n_{test} best local MAP expert as determined by each gater-node G_z^ℓ . For example, a tree of depth $N_\ell = 3$ where each parent has $N_k = 8$ children could choose $n_{test} = 2$ experts at each gater-node. Since each gater-node can choose from $N_k = 8$ experts, and we have a depth $N_\ell = 3$, we end up with $\text{pow}(n_{test}, N_\ell) = 2^3 = 8$ expert prediction $E_k^3(Y|X)$. This is still much less than the total amount of leaf-experts $\text{pow}(N_k, N_\ell) = 8^3 = 512$. This makes sense since at each gater-node, $N_k - n_{test}$ expert-branches (and child gater-nodes), and all their dependents are dropped from the computation graph.

As for accumulation, the different gater predictions $G_z^\ell(E^\ell{=}k|X{=}x)$ can be combined to obtain the probability of *leaf* expert predictions E_k^N:

$$G(E_z^N{=}k|X{=}x) = \prod_{k' \in \text{a}(k)} G_{z'}^\ell(E^\ell{=}k'|X{=}x) \tag{2.15}$$

where $N = N_\ell$ (for clarity), $\text{a}(k)$ is the set of all ancestors to expert k and z' indexes

12. Each leaf-expert outputs a feature vector. Each class is a parameter vector of the same length *shared* by all leaf-experts.

13. The combinatorial problem space is small since each layer ℓ is divided into $(N_k)^{\ell-1}$ independent gater-decisions involving discriminating n_{train} of N_k expert-branches : $(N_k)^{\ell-1}\binom{N_k}{n_{train}} < \binom{(N_k)^\ell}{N_k n_{train}}$

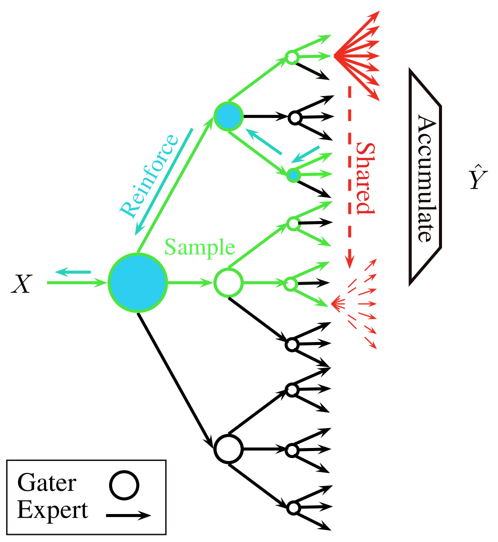


Figure 2.1: Tree Architecture and Training. Green experts are sampled by green gaters. Cyan experts and gaters are reinforced on winning expert-paths. Class experts (in red) are shared by all sampled leaf-experts, the results of which are accumulated to generate predictions $\hat{Y}$ for X. Each expert has 3 child experts. A gater-wise $n_{train} = 2$ and global $n_{test} = 1$ are represented.

the gater responsible for predicting the conditional probability of expert k'. From expert weights $G(E_z^N{=}k|X{=}x)$, the expert predictions E_k^N are accumulated using equation 2.5.

We eventually lose more than we gain from adding too many neurons to MLP layers (Dauphin and Bengio (2013)), which may be an indication that BP loses less gradient information through sparser weight matrices. Hence, in order to facilitate deeper learning, and because successive expert-branches will only receive a fraction of examples thanks to the sparsity and equanimity constraints, successive E_k^ℓ will require less and less capacity:

$$N_{j0} > N_{j1} > ... > N_{j\ell} > ... > N_{jN_\ell} \tag{2.16}$$

where $N_{j\ell}$ is the size of expert output representations for layer ℓ and thus each expert in a layer has the same such size. While we could divide each node into two branches to get a binary tree, this would ultimately require more depth for more breadth. In order to reduce depth, we allow a hyper-parameter to determine the number of branches per node: $N_k \approx 9$. Previous work has demonstrated that keeping this bellow 10 allows for a good trade-off between depth-related learning

issues, and discrimination of branches (Ornstein (1996)).
The outputs of the tree are representations y_o for each leaf expert-branch, for a total of $N_o = \text{pow}(N_k, N_\ell)$ leafs. We may be tempted to have each class c of our multi-class classification problem have its own dedicated weight vector W_{oc} and bias scalar b_{oc} for processing each y_o. But this might cause problems in terms of generalization as experts closest to the leafs could become entirely dedicated to predicting inputs from a single class. The capacity of such an expert would be wasted as it could be replaced by a constant that always outputs the same class. In such a case, the classification is being performed by the gater. This over-specialization behavior can be seen in the works of (Eigen et al. (2013)).
In order to counter this tendency, we force a common output representation space for all y_o for $1 < o < N_o$ by sharing the parameters of the last layer of experts (red arrows in figure 2.1). The resulting predictions of each expert can then be pooled with any of the accumulators discussed in section 2.2.3 to generate the prediction $\hat{y}$ (figure 2.1), for any class.

2.4 Experimental Analysis

In this section, we present our the results of our experiments in applying the ESSRL training algorithm (algorithm 3) to an NDT model and the MNIST dataset. The MNIST dataset consists of 28×28 gray-scale digit images; 60,000 for training and 10,000 for testing. The objective is to classify the digit images into their correct digit class.

2.4.1 Implementation

The NDT model was implemented using Torch. Efficiency was obtained by creating new sub-set batches for every expert sampled from a gater. In other words, after sampling from each gater-node in the tree, advanced indexing would be used create a mini-batch for each expert containing only the necessary examples. Each expert-local mini-batch would carry around the identity of its examples so that they could be joined at the output of the tree for accumulation. By ensuring that redundant memory copies weren't performed, the entire propagation of a batch was only 5 times slower than an MLP with the same quantity of layers and which utilized an equivalent number of multiplication-addition operations. This is understandable as the efficiency of the matrix-matrix multiplication operation is accelerated by intelligent use of the memory cache. This optimization cannot be performed as well for the NDT as the mini-batches are split into distinct memory segments, each using a *different* weight matrix, such that the effect of these BLAS optimizations aren't as pronounced. In any case, the speed w.r.t an MLP utilizing the same quantity of parameters as that in the NDT will still be much slower given enough sparsity. So the issue then lies in determining if the NDT can utilize this extra capacity.

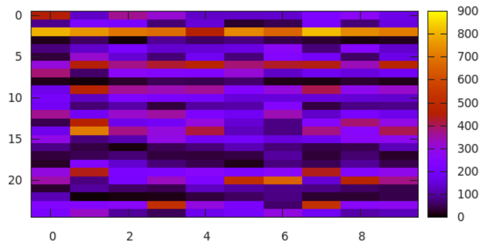
2.4.2 Failure Modes

Failure modes are hypothesized circumstances where the model could fail. The possibility of expert monopolies, where a monopoly (or cartel) forms over a large portion of the distribution of examples, has already been established. Because the ESSRL algorithm is based on the principle of competition, where only the more successful experts get to learn, experts with a head start (due to random initialization) can learn faster and therefore monopolize the distribution.

In order to diagnose this failure mode, we use a *specialization matrix* where rows and columns respectively represent leaf-experts and dataset classes. A portion of the data set, usually the training set, is propagated through the model using local-MAP experts at each gater-node. Each variable in the specialization matrix identifies the count of examples of the commensurate class that ended up using the commensurate leaf-expert. The classes are used as an indicator of the kind of specialization occurring in the leafs.

Monopolization

In figure 2.2, we present some example specialization matrices for NDT that, while providing good generalization performance ($\approx 1.3\%$ test classification error on MNIST), weren't using all of the capacity available. The fact that performance of these models is still very high — even though the equanimity constraint and other exploration techniques like ϵ-greedy sampling for multinomial —, demonstrates how


(a) Monopoly on all Examples

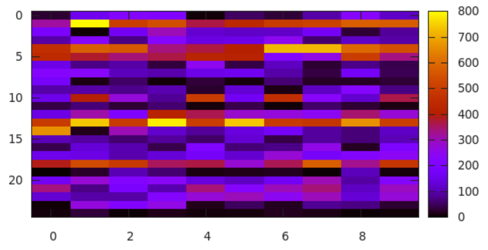

(b) Monopoly on some Classes

Figure 2.2: Specialization matrices for $5 \times 5 = 25$ experts (y-axis) and the 10 MNIST classes (x-axis) using the training set. Two common mixture of expert failure modes are presented. In *a* a single expert (row 2) is monopolizing all examples while in *b* a cartel of experts each monopolize a sub-set of classes. In both cases, the remaining experts are used for very few examples, and many of these see little to none.

difficult it is to divide the problem space among experts. The tendency for such competitive systems is indeed to monopolize, and introducing balancing constraints to utilize all available capacity is very difficult. Some of the causes may be the poor initialization of some experts over others, how the gater parameters are initialized to distribute examples in such a way as to favor some experts over others, and the nature of the problem space itself. In some cases, the problem space can't easily be divided among so many or so little experts.

Over-Specialization

In both specialization matrices presented in figure 2.2, the distribution of examples to non-monopoly experts is sometimes distributed uniformly where each class seems to be approximately equally represented in the expert. This indicates that the expert was deemed to be of some use — after all, the gater-nodes use it for evaluation —, and that it doesn't seem to be responsible for discriminating a subset of all classes. This isn't necessarily bad, as specialization doesn't imply each expert limiting itself to a subset of classes.

Another type of failure mode is for the division of experts to be trivial. This happens when each leaf-expert is responsible for processing examples for but one class. This is exemplified in figure 2.3 were the specialization matrix shows each expert having but one class, even though each class of example can find itself

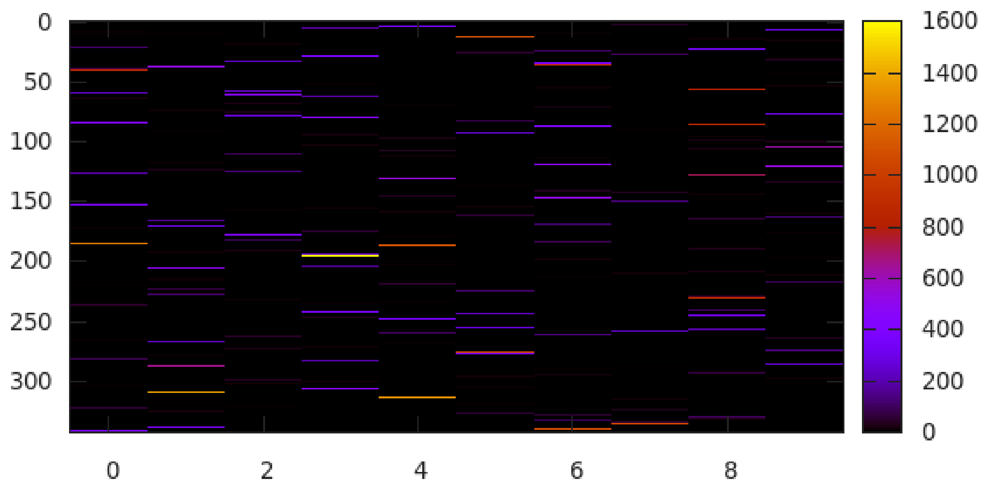


Figure 2.3: Specialization matrix for $7 \times 7 \times 7 = 343$ experts (y-axis) and the 10 MNIST classes (x-axis) using the training set. During evaluation, 3 gater-local MAP experts are chosen at each branch and the $3 \times 3 \times 3 = 27$ resulting leaf outputs are accumulated. Training was performed by sampling 4 experts at each gater-node. Although difficult to see, each leaf-expert is responsible for processing examples from but one class. This is an example of *over-specialization*.

in more than one expert. Therefore, it is the gaters which are responsible for performing the discrimination. This often happens when too many leaf-experts are used, or the hierarchy has too many levels. Each expert's discrimination task is trivial as it could be replaced by a constant one-hot vector. It is interesting to note that this particular NDT achieved a noteworthy test error of 1.32% on MNIST after 755 epochs of training. Therefore, we can conclude that an over-specialization of this nature doesn't necessarily mean that the network is over-fitting. It just means that our model has too much capacity.

2.4.3 Results

For our actual experiment, we ran hundreds of models using different configurations. All final experiments presented here use a batch size of 128. Each gater-node is a 2 parameterized layer MLP. The NDT shared a root layer (or trunk) whose output representation is shared by all later layers in the tree. This is to reduce the size of the first layer of expert-branches and the first gater-node. We present the results in table 2.1. Learning rate is decayed by a factor of 10 after epoch 500 and 700 using a linear decay schedule.

Baselines

For our baselines, we used MLPs of different depth and breadth. We chose these such that, in a single example propagation, they used approximately the same quantity of multiply-additions as the NDT. The baseline is trained using the usual techniques, including max norm regularization, a learning rate schedule (we choose the one that works best), momentum and tanh activation units (like the NDT) (section 1.4). We found that, all things being equal, using models with a depth of 4 worked best, while the NDT has a depth of 4: the trunk layer, 2 levels of one-layer experts and the final shared layer. Although since the last layer is shared and very small, we could think of the NDTs presented here as having a dept of 3, such that an MLP of depth 4 has an advantage.

In table 2.1, we identify baselines using a nomenclature to keep the description concise. For example, description **MLP-4-1024** means a depth of 4 parameterized layers, and width of 1024 for all hidden representations.

Neural Decision Trees

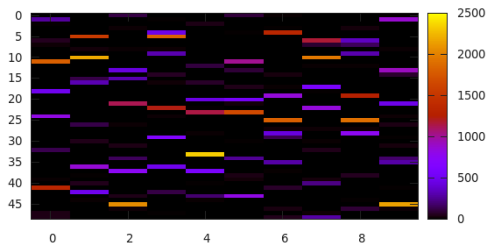
Thousands of experiments were performed using a variety of hyper-parameters — the ESSRL-NDT training algorithm has many —. Experiments where divided into different groups with fixed hyper-parameters. For example, we would choose the number of experts per node, the depth of the tree, the number of reinforced and sampled experts per node and explore variations thereof. The best results of each of the most successful groups of experiments are presented in table 2.1 where the description also follows a particular nomenclature. Using the first row of the table as an example, **NDT-2-7-3** means that the NDT has 2 layers of experts (the trunk layer is a given), where each gater-node chooses 3 from 7 possible local-MAP branch-experts. The **e:1024x175x30** means that the representation between trunk and the first level of experts is 1024, the next representation between first and second level experts is 175 and the size of the input to the shared output layer is 30. The **g:256-20** means that the first level gater-node has a hidden representation of size 256 (between the 1024-unit trunk output and its 7 outputs), and the second level gater-nodes each have a hidden representation of 20 (between the 175-unit first-level expert output and the 7 outputs).

Description	Epoch	Valid	Test
NDT-2-7-3 e:1024x175x30 g:256-20	463	1.25	1.32
NDT-2-7-2 e:1024x175x30 g:128-15	477	1.29	1.35
NDT-2-5-1 e:1024x146x21 g:256-36	478	1.56	1.36
NDT-2-5-1 e:200x200x200 g:50-50	409	1.55	1.47
MLP-4-1024	601	1.50	1.50
MLP-4-768	323	1.59	1.53

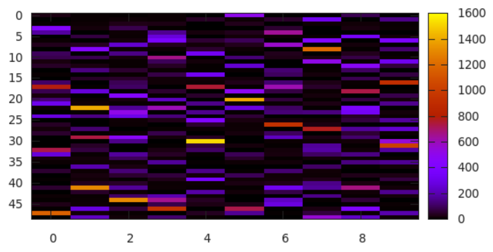
Table 2.1: NDT-ESSRL Experiments

The specialization matrices for the first two rows of NDT results are presented in figure 2.4 with a description of the training performed. These matrices look very good as a majority of active leaf-expert process 2 or more classes of examples. There are very few under-utilized and over-specialized experts, and no example monopolies. In all cases, the equanimity constraint wasn't enough as it only affected the sampling of experts and examples (equal opportunity). While this was necessary to obtain a balanced distribution of examples to experts, we also needed to apply the same principles for the reinforcement of under-performing experts. This was implemented by making a small change to the original ESSRL algorithm (algorithm 3, line 9), where instead of just reinforcing the experts with the least error for each example, we also reinforce the examples with the least error for each (sampled) expert. This guarantees that each expert learns a component of the problem space. We also explored different variations on the original algorithm where instead of reinforcing the experts and examples with the least error — which we can think of as a discrete technique —, we reinforce all sampled experts and examples in inverse proportion to their relative error — which we can think of as an analogue or continuous technique. The results of both approaches were however very similar, which is understandable since they tend to have the same effect over many training examples, especially when the continuous version uses exponential functions like softmax to weigh the reinforcement signals.

In figure 2.5, we can see the effect of an excellent specialization. Even though the last 5 experts are rather under-utilized — which means that the last expert-



(a) NDT-2-7-2 e:1024x175x30 g:128-15



(b) NDT-2-7-3 e:1024x175x30 g:256-20

Figure 2.4: Specialization matrices for $7 \times 7 = 49$ experts (y-axis) and the 10 MNIST classes (x-axis) using the training set. During evaluation, *a* chooses 2 while *b* chooses 3 gater-local MAP experts at each branch and the resulting leaf outputs are accumulated. Training was performed by reinforcing 3 of 4 sampled experts at each sampled gater-node. A majority of active leaf-expert process 2 or more classes of examples. There are very few under-utilized and over-specialized experts, and no example monopolies.

branch of the first level of the tree is —, this specialization matrix is near-perfect as similar classes have clustered together:

- 1,3 and 8;
- 0 and 6;
- 4, 7 and 9; and
- 2 and 8.

This is exactly the kind of natural distribution of examples we are looking for as it seems best to form experts that can discriminate between similar items, rather than the simpler alternative. This also indicates that the task requires only a limited amount of experts — there are only so many clusters —, and its strong links with the field of cluster analysis.

2.4.4 Conclusion

Although the results presented here are very promising, the fact of the matter is that these models were very difficult to hyper-optimize. In many cases they become unstable. We also tried it on other image datasets like CIFAR-10, CIFAR-100 and NotMNIST to find that it did not work so well in these cases. The tendency of experts to over-specialize is great. The potential for using expert at the input of the network seems quite low as the gater would require greater capacity. There

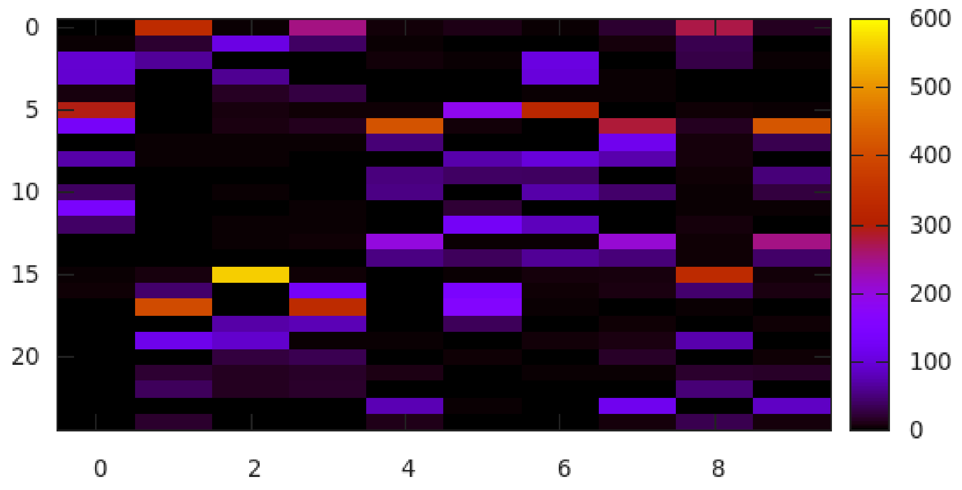


Figure 2.5: Specialization matrix for $5 \times 5 = 25$ experts (y-axis) and the 10 MNIST classes (x-axis) using the test set (row **NDT-2-5-1 e:1024x146x21 g:256-36** in table 2.1). During evaluation, 1 gater-local MAP expert is chosen at each branch such that the last expert output is used as-is for prediction. Training was performed by reinforcing 2 of 3 experts sampled at each gater-node. Even though the last 5 experts are rather under-utilized, this specialization matrix is near-perfect as similar classes have clustered together: 1,3 and 8; 0 and 6; 4, 7 and 9; and 2 and 8, etc.

are also strong tendencies for gaters to route examples from just one class to leaf-experts. Although we have tried many different variations, including using different learning rates, regularization constraints and momentum for the gaters and experts, we found no clear cut indication that any combination worked best. In the end, the particular random initialization of the network seems to be the most important factor. Another difficulty of conditional models is to keep them from becoming pure ensemble methods, where utilizing more experts during evaluation would seem to help and the gaters are essentially useless. More tests would be required in this respect. Of course, using more experts during evaluation often does provide better results, as is evidenced by table 2.1, where the top performing models use more local-MAP experts per gater-node. In the next chapter, we will see a completely different approach which is more combinatorial in nature. In the sense that experts from later layers aren't associated to experts in previous layers, as is the case with trees.

3 Language Models using DCC

3.1 One Billion Words Benchmark

For our experiments, we use the one billion words benchmark used for measuring progress in language models (Chelba et al. (2013)). The dataset contains one billion words. The task consists in using the previous n words (the context) to predict the next word (the target word). All sentences are shuffled such that only the words from the target's sentence can be used for prediction. The end of the sentence, identified by token "</S>", must also be predicted. To predict the first n words of a sentence, padding is added. Using sentence "<S> Nicholas is writing</S>" as an example, each input -> target word would have the following contexts of 3 words:

1. "</S> </S> <S>" -> "Nicholas" ;
2. "</S> <S> Nicholas" -> "is" ;
3. "<S> Nicholas is" -> "writing" ; and
4. "Nicholas is writing" -> "</S>".

The entire dataset is divided into 100 partitions of equal size. 99 of these are used for training. The remaining partition is further divided into 50 partitions, one of which is used for testing, while the remaining 49 are reserved for cross-validation. All words with less then 3 occurrences in the training set are replaced with the "<UNK>" token. This is the same split used in (Chelba et al. (2013)).

This dataset was chosen mainly for its size. We hypothesized that DCC would do best in an under-fitting regime, which is to say a regime that is very difficult to over-fit. Our past experience has shown us that conditional models like mixture of experts have a tendency to over-fit small datasets, which leads to bad generalization. We also chose this dataset in order to move away from image datasets that require convolutions to attain state of the art performance. Images thus require that the conditional computations be either integrated into the convolutions, or limited to the final dense layers.

3.2 Neural Network Language Model

A neural network language model (NNLM) uses a neural network to model language. There are various approaches to building NNLMs. The first NNLM was (Yoshua Bengio and Vincent (2001)). (Bengio et al. (2006)) concatenates n context words embeddings at the input layer to form an input of size $n \times n_i$, where n_i is the number of units per word embedding. This input layer takes the form of a *lookup table*. We can think of it as a weight matrix W of size $N_t \times n_i$ where N_t is the number of words in our vocabulary. In the case of the billion words dataset, we have approximately 800,000 words. Each word is assigned a single row of weight matrix W which will serve as its embedding. These can be learned through BP where the input of the table is a vector x of dimension n where each variable x_i contains the index of the word at position i of the context. These are used to extract all embeddings of the lookup table that correspond to the context words:

$$y = W_{x_1} \parallel W_{x_2} \parallel W_{x_3} \parallel \ldots \parallel W_{x_n} \tag{3.1}$$

where $\parallel$ corresponds to the concatenation operator. The gradient can be calculated as follows:

$$\frac{\partial y}{\partial W_j} = \begin{cases} 1 & \text{for} \quad j \in x \\ 0 & \text{for} \quad j \notin x \end{cases} \tag{3.2}$$

which makes this layer efficient for both forward and backward propagation since only the n context words need to be queried, concatenated and updated.

The resulting concatenation of embeddings can be forwarded through n_ℓ parameterized hidden layers having the following form:

$$y = \sigma(Wx + b) \tag{3.3}$$

where $\sigma()$ is the usual element-wise activation function. These are often shallow networks having no more than 1 or 2 parameterized hidden layers (Schwenk and

Gauvain (2005)), (Le et al. (2011)).

In its simplest form, the output layer uses a normalizing non-linearity like softmax:

$$y_i = \frac{e^{x_i}}{\sum_{i'} e^{x_{i'}}} \tag{3.4}$$

Its use of the exponential function has a tendency of increasing the weight of the highest input values, thus forming a kind of soft version of the max function. By dividing by the sum of the exponential of each variable in the vector x, it has a normalizing effect in that $\sum_i y_i = 1$, thus making it useful for generating multinomial probabilities $P(Y|X)$. The forward and backward propagation of this layer is extremely costly in terms of processing time for large vocabularies. This inefficiency is due to the normalization which requires calculation of all x_i for $1 \geq i \geq N_t$. Alternatives to using a pure softmax exist, which will be discussed in the next section.

The empirical risk function of the model is the ubiquitous mean negative log-likelihood:

$$R_{NLL} = -\sum_k \frac{\log(y_{k,t_k})}{N_k} \tag{3.5}$$

where k indexes examples, N_k is the total number of examples, and t_k is the target word of example k. Finally, y_{k,t_k} is the likelihood of word t_k for example k, where y_k is the output of the NNLM given example k. To evaluate our NNLMs, we use perplexity (PPL) as this is the standard metric used in the field natural language processing (NLP) for language modeling:

$$R_{PPL} = e^{R_{NLL}} \tag{3.6}$$

where e is the same logarithm basis as the log in R_{NLL}.

3.3 Output Layer Optimizations

The issue with the last layer, the parameterized softmax, is that it can become a serious performance bottleneck during both training and evaluation. This is due to the large output representation, spanning the entire vocabulary of the corpus.

3.3.1 Prior Work

Various solutions have been proposed to circumvent the issue. All of these are variants of the original class decomposition idea (Bengio and de MontrÑĽeal (2002)). Although other similar approaches exist, including importance sampling (Bengio et al. (2003)) and uniform sampling of ranking criterion (Collobert et al. (2011)), we chose to focus our attention on the following:

1. hierarchical softmax : Morin and Bengio (2005);
2. hierarchical log-bilinear model : Mnih and Hinton (2009);
3. structured output layer : Le et al. (2011);
4. noise-constrastive estimation : Mnih and Teh (2012);

The first method builds a binary hierarchy of words from expert knowledge like WordNet (Fellbaum (1998)). Each output word has its own binary representation : the sequence of binary decisions from the root leading down to the leaf containing the word in the tree. The paper demonstrates that this method provides a great speedup : $O(\log_2(N_t))$ instead of $O(N_t)$, but at the cost of a small loss in perplexity.

The second approach uses a log-bilinear model (Mnih and Hinton (2007)) combined with an iterative clustering algorithm. The latter consists in training a hierarchical log-bilinear (HLBL) model, and then applying an EM algorithm to the resulting word embeddings to perform a top-down (root to leaf) clustering of words. The resulting binary hierarchy is used to initialize a new HLBL for training, and so forth and so on. The first HLBL tree is constructed randomly. They demonstrate a significant reduction in perplexity over the the previous implementations of hierarchical softmax.

The third approach is similar to the first and second, but differs in how words are clustered. As opposed to an LBL, they do not use a standard NNLM. They begin by training a NNLM using a parameterized softmax output layer truncated to consider only a small subset of the vocabulary. They reduce the dimensionality of the resulting input word embeddings, and use these to perform a recursive k-means clustering to generate a hierarchy of words. The recursive clustering divides a word class (or sub- class) only if the number of words in this class is above an empirical threshold.

The final approach is completely different in that it only affects training time, which is also the case for (Collobert et al. (2011)). It approximates the denominator of the softmax by sampling it from a unigram distribution. It only leads to a speedup of about 10 on CPU as compared to the pure softmax solution. This is mainly due to the cost of sampling from a multinomial distribution. Furthermore, there is yet to be an open-source GPU implementation. However, the perplexity does seem to suffer very little from this approximation, making it a suitable alternative to the previous approaches.

Our approach, which we name the *SoftMaxTree* is very similar to the first three approaches in that it uses a hierarchical representation of words to accelerate the process. It also has in common with the third approach the use of a non-binary tree. However, unlike any of these solutions, we make no use of embeddings, and thus do not require training an LBL model or a NNLM to obtain these. We instead use a clustering method that uses the relations between words, which will be detailed in the next section. We chose this kind of approach over embedding-based clustering as we already had code for it.

3.3.2 Similarity-Graph Partitioning

Various techniques can be used to perform a hierarchical clustering of words. The most commonly known is Brown clustering (Brown et al. (1992)) [1]. As outlined in the previous section, some techniques use the word embeddings obtained from training an NNLM.

Our algorithm is divided into the following phases:

1. Optimize words for set-based queries using b-tree indexes ;
2. Represent each word by a weighted set of related words ;
3. Create a directed graph of word nodes and similarity-weighted arrows ;
4. Cluster words into k_L equally sized partitions ; and
5. Recursively cluster partitions into k_ℓ equally sized partitions for $\ell = 1 \rightarrow L-1$.

Step 1

For step 1, the entire corpus is parsed, tokenized and loaded into a relational database management system (RDBMS) like PostgreSQL [2]. In order to save mem-

1. https://github.com/percyliang/brown-cluster
2. https://github.com/nicholas-leonard/equanimity/tree/0.7/nlp

ory, we use an SQL table mapping all words to integers. The entire corpus needs to be parsed twice. Once for mapping each unique words to its own integer keys, as well as counting the occurrences of each word, and consolidating words with less then 3 occurrences into the special "<UNK>" word [3]. The second pass stores each sentence as a PostgreSQL Array of word keys (integers) [4]. The advantage of using an RDBMS is that it provides set-based queries and optimization facilities. In particular, SQL tables columns can be indexed using b-trees in order to optimize random access. As we will see below, this will prove invaluable for optimizing our partitioning algorithm. The first optimization consists in expanding the word key array of each sentence into a less-memory efficient table indexed by the position of each word and its key. [5] This new storage representation allows the next step to be performed with greater processing efficiency — it is otherwise intractable.

Step 2

For step 2, we first measure the inverse document frequency (Jones (1972)) of each word, where each document is a sentence.

$$\text{idf}(t, D) = \log(1 + \frac{N}{d \in D : t \in d}) \tag{3.7}$$

where t is a word, d is a sentence, N is the number of sentences, and D is the corpus of all sentences. We do this in order to weigh the relationship of each word to a sentence using the ubiquitous *tf-idf* scheme (Salton (1991)):

$$\text{tfidf}(t, d, c, D) = \text{tf}(t, c) \times \text{idf}(t, D) \tag{3.8}$$

where c is the union of all contexts of a word, $\text{tf}(t, c)$ is the number of occurrences of word t in the word contexts c. In our case, we define a context as the 10 words preceding a word in the same sentence. We use this definition of a context in order to match it as closely as possible to the future use of language modeling. The main advantages of *tf-idf* are its simplicity and its ability to downscale the importance of frequent words. Once this is accomplished, we use the table of word positions

3. https://github.com/nicholas-leonard/equanimity/blob/0.7/nlp/parse1.lua
4. https://github.com/nicholas-leonard/equanimity/blob/0.7/nlp/parse2.lua
5. https://github.com/nicholas-leonard/equanimity/blob/0.7/nlp/billion-word-benchmark.sql

(see step 1), and the inverse document frequencies to create a table mapping each unique word to the set of 700 words with the highest $\text{tfidf}(t, d, c, D)$. In other words, each word t will be represented by a set A of up to 700 words t, each weighted by our measure of *tf-idf*.[6]

Step 3

For step 3, we measure the similarity between all possible combinations of S_t. With a vocabulary of 800 000 words, we have an upper bound of 640 000 000 000 similarity measures, which is intractable in terms of persistent storage and memory. So we limit our graph of similarity to 700 arrows per word, which results in a much more tractable upper bound of 560 000 000 arrows, or approximately 23 GB of memory. For our measure of similarity, we use cosine similarity:

$$\text{sim}(A, B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum\limits_{t \in A \cup B} A_t \times B_t}{\sqrt{\sum\limits_{t \in A} (A_t)^2} \times \sqrt{\sum\limits_{t \in B} (B_t)^2}} \tag{3.9}$$

where A_t and B_t measure $\text{tfidf}(t, d, c_A, D)$ and $\text{tfidf}(t, d, c_B, D)$ respectively. The graph is constructed concurrently using a simple script which can parallelize SQL queries.[7] Using 8x2.6 GHz Xeon-CPU-cores and index-optimized tables the process terminates in approximately 20 hours. The resulting table represents a graph of similarity arrows linking nodes representing unique words. We are now ready to perform the first level of clustering.

Step 4

We begin step 4 by assigning each of the n_K partitions to a set k of approximately $n_k = 10$ words. We thus choose $n_K = \frac{N_t}{n_k}$. The clustering process will continue to constrain partitions to have approximately the same size n_k. This is intentional as we will require such a balanced partitioning to make efficient use of the GPU for predictions. The final step of the initialization is to calculate $\text{aff}(k, t)$

6. https://github.com/nicholas-leonard/equanimity/blob/0.7/nlp/word-clustering.sql
7. https://github.com/nicholas-leonard/equanimity/blob/0.7/nlp/parallel_sql.py

measuring the affinity of each word-node t to partition k:

$$\mathrm{aff}(k,t) = \sum_{t' \in k : t \neq t'} \mathrm{sim}(t,t') \tag{3.10}$$

Now initialized, the partitioning process can begin. It is very similar in essence to MajorClust (Stein and Niggemann (1999)) in that it clusters the nodes using a graph of similarity edges. However, our algorithm differs in that it results in equally sized groups, which is the discriminating factor between clustering versus partitioning algorithms. The algorithm begins by randomly selecting a word-node t from the graph. It then ranks all of its connecting arrows $\mathrm{sim}(t,t')$ by its $\mathrm{pull}(t,k)$ measuring the similarity pull (or attraction) of word-node t to partition k:

$$\mathrm{pull}(t,k) = \sum_{t' \in k} \mathrm{sim}(t,t') \times \mathrm{aff}(k,t') \tag{3.11}$$

Before venturing on, we coarsely define $\mathrm{den}(k)$ measuring the density of the subgraph connecting nodes in partition k:

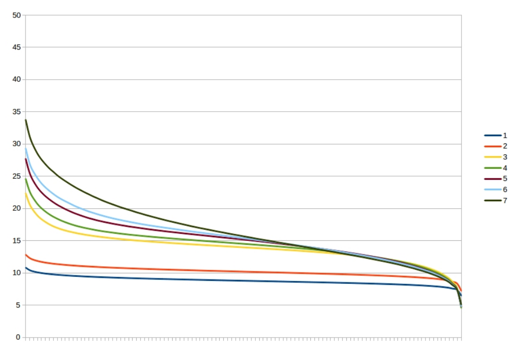
$$\mathrm{den}(k) = \sum_{t \in k} \mathrm{aff}(k,t) \tag{3.12}$$

This measure is very quick to compute for all clusters using SQL.

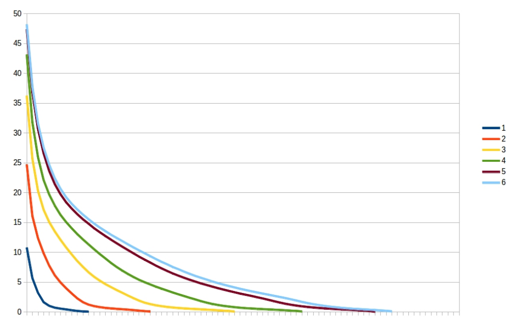
Initially, sampled word-node t would be assigned to the partition k with the highest $\mathrm{pull}(t,k)$, subject to the constraint :

$$\min_{t'} \left[\mathrm{aff}(k,t')\right] \frac{n_k + 1}{n_k} > \sum_{t' \in k} \mathrm{sim}(t,t') \quad \text{for } t' \in k \tag{3.13}$$

But this would result in an unbalanced distribution. By this we mean a distribution with few partitions having very high $\mathrm{den}(k)$, while the majority of partitions would have very poor density. To understand this, we can compare this behavior to small startups (poor partitions) having their best employees (word-nodes) hired by big corporations (rich partitions) due to the attractiveness of higher salaries (greater $\mathrm{pull}(t,k)$). A vicious circle naturally develops where the poorest partitions cannot retain their word-nodes, nor attract better ones, as these are attracted to the richest (highest $\mathrm{den}(k)$). Two classes of partitions come to exist: the rich and the poor with no middle ground.



(a) Equal distribution



(b) Unequal distribution

Figure 3.1: Bar graph of den(k) ranked from richest to poorest. Figure a depicts the algorithm favoring an equal distribution, while b depicts it favoring an unequal distribution. Each line is identified by its order in the sequence of snapshots. The inequality of the partition densities grows as a function of time. For this first level ($\ell = 1$) of the hierarchy, each partitioning process takes approximately 48 hours to complete.

Our solution was to assign sampled word-node t to the partition k with the **lowest** pull(t, k), subject to the constraint 3.13. This would slow down the inevitable growth of inequality. As demonstrated in fig. 3.1, the inequality grows, but only a small proportion of partitions see their density reduced over time. This method thus provides us with a balanced partitioning, such that each partition is guaranteed a minimum utility in terms of density, or conversely, guaranteeing a partition of similar peers to all word-nodes. Furthermore, where Total density of the partitioning is simply defined as:

$$\text{Total density} = \sum_k \text{den}(k) \tag{3.14}$$

. The unequal scheme complete with a total density of 700 after 48 hours, while the equal scheme does so with a total density of 1500. This demonstrates the effectiveness of the partitioning approach that favors equality.

Finally, we established a simple scheme to keep the size of partitions constant. When a sampled word-node is to be transfered to another partition, we continue the process by attempting to transfer the worst word-nodes from that new partition to another, and so forth and so on. When no further transfers can be performed, we send a random word-node from this last partition to the first, thereby closing the exchange loop and keeping all clusters balanced. The task is parallelized by having different processes perform independent exchange loops, thereby accelerating the process. [8].

Step 5

Step 5 is a recursion in that steps 2 through 4 are repeated using the previous level of partitions as nodes instead of words. Like word-nodes, each partition-node k_ℓ of level ℓ is represented by a weighted set of related words. The set of words is the union of all the word-nodes related to those nodes contained in its partition — we found that this worked much better then aggregating similarity arrows (for e.g. by summing their similarity measures) of adjacent partitions to create new similarity arrows between partitions. The *tf-idf* is thenceforth measured as if the partitions were nodes, and so forth and so on. The process is restarted until $n_{K_\ell} \approx n_k$.

The result of the process is a balanced hierarchy of N_ℓ levels, where each level contains n_{K_ℓ} partitions of size n_k having approximately equal den(k), or utility. For our task we found that an $n_k \approx 10$ worked best – although with hindsight, we would have chosen to use $n_k \approx 30$ subject to constraint $n_k \leq 32$ for obtaining greater efficiency from the GPU. When large n_k was used we found that the last levels would result in poorer results. See table 3.1 for example word partitions. You can see that it does a relatively good job of grouping related words together. Yet in many sets, one word seems to be out of place. This is due to the item exchange loops which must terminate by sending a random word to the first partition in the loop. Near the end of training, these loops are very short, making such out of place words very common.

8. https://github.com/nicholas-leonard/equanimity/blob/0.7/nlp/cluster.py

Macvicar,Jackson-Lee,Rainger,MacVicar,Aidoo,Marikar,Feren,Stainback
Tonioli,Tertrais,Sekoli,Gollnisch,Pischiutta,Schiemsky,Saelzer,Metsu
19.44,20.12,13.36,08.57,13.45,15.29,12.01,19.56
Rimasse,Escarra,McGlinsey,Swithenbank,Divoll,Podberesky,Saporta,Gunvalson
Dechy,Kosciusko-Morizet,Pechalat,Dezeure,Djurberg,Tadena,Saugeon
Hixon-Denton,Dreiser,Roethke,Boutrous,DeReese,Simburudali,Karasik,Bikel
6pm,4pm,10am,10pm,5pm,2pm,7pm,3pm,9am
Bardem,Arruabarrena,Saviola,Aguirre,Camunas,Aguirresarobe,Vazquez,Culson
Bourque,Hoksbergen,John-Baptiste,Preval,Umlauft,Rougeau,Syler,Meulensteen
Xiaoling,Xichun,Renbao,Shu-chen,Baoquan,Xinbo,Xianguo,Pengyong
Scaroni,Ugge,Boffetta,Timoni,Nespoli,Micheli,Bandini,Zeppilli
Pouw,Laub,Brulliard,Karlekar,Yurko-Mauro,Immergut,Huttary,Thurig
Monoceros,Scorpius,Cepheus,Serpens,Piscis,Cetus,Puppis,Vulpecula
overdue,drawn-out,haul,stretches,waits,queues,distances,johns
Doumato,Rigby,Smeal,Bimla,Holmes-Norton,Squillari,Bron,Skelhorne
Kospi,Yonhap,Unification,KOSPI,Chosun,mass-circulation,KJ,Kopsi
Venuto,Memmo,Resta,Centa,Matteo,Maggio,Rollo,Sieno
Earnhardt,Meyerrose,Petroskey,Sveum,Steyn,Bumpers,Kildee,Begg-Smith
Harkett,Kirkness,Croucher,Kingscott,Glenton,Drane,Fullarton,Roney
Korolev,Ustyugov,Artyukhin,Drattsev,Artyukin,Dadonov,Rogaev,Chigishev
al-Awlaki,al-Aulaqi,Sadat,el-Sadat,Al-Awlaki,al-Awlaqi,El-Ibrahimi,Gargash
Veasley,Stargell,McCovey,Pelote,Mullins-trained,Galick,McGinest,Limond
Fedotenko,Salei,Pukhov,Chagaev,Amerkhanov,Boidakovs,Ponomariov,Kishmaria
Zak-Cohen,Ostrosky,DeFalco,Osterhout,Fouhy,Hulfish,Halaas,Haroules
Twitterton,DeGuerin,King-Smith,Enberg,Ebersol,Costolo,Cavatt,Cavett
fifth,sixth,seventh,second,third,fourth,eighth,back-to-back

Table 3.1: Level 1 word partitions ranked 10000 in terms of descending den(k)

3.3.3 SoftMaxTree

Once the word-partitioning process is complete, we can justify its cost by using it to increase the throughput of our NNLM. The first step in doing so is to implement a hierarchical softmax module that can be reused throughout our experiments. The hierarchical softmax, hereby referred to as *SoftMaxTree*, is defined as a product over the probability of all nodes in the path to the target word:

$$P(Y = t|X) = P(Y = t|X, \forall c \ni t) \prod_{c \ni t} P(c|X, \forall c' \ni c) \tag{3.15}$$

where t is the target word, X and Y are respectively variable input and outputs, c is a partition-node. This optimization reduces the algorithmic complexity of the last layer from $O(n_o N_t)$ to $O(n_o \log_{n_k}(N_t))$, where n_o is the size of the output embedding space, i.e. the size of the input to the parameterized softmax output layer.

This efficiency gain is however only theoretical. In practice, the gain isn't as pronounced due to the difficulty of parallelizing this on GPU. We were however able to build a very fast CUDA [9] and C [10] implementation, using the Torch7 Collobert et al. (2011) machine learning library. Both implementations are wrapped by the same Lua code [11], thereby abstracting away the differences between C and CUDA for the user. The **SoftMaxTree** Module is open-source (BSD license) and available for use with Torch7 in the nnx [12] package.

A benchmark was prepared using the vocabulary hierarchy with $N_t \approx 800000$, $n_k \approx 10$, or its inverse $N_\ell = 6$, and $n_o = 50$. Two NNLM are compared, one using the SoftMaxTree, the other a standard parameterized softmax output layer. Both neural networks have a lookup table using a context size of $n_x = 5$ and an input embedding space of size $n_i = 50$. The NNLM has a parameterized hidden layer transforming the input context to the output embedding space. The speedup of the SoftMaxTree C implementation is approximately 475 compared to the vanilla NNLM. And the CUDA SoftMaxTree implementation is approximately 1.5 times the speed of its C counterpart.

9. https://github.com/nicholas-leonard/cunnx/blob/master/SoftMaxTree.cu
10. https://github.com/clementfarabet/lua—nnx/blob/master/generic/SoftMaxTree.c
11. https://github.com/clementfarabet/lua—nnx/blob/master/SoftMaxTree.lua
12. https://github.com/clementfarabet/lua—nnx

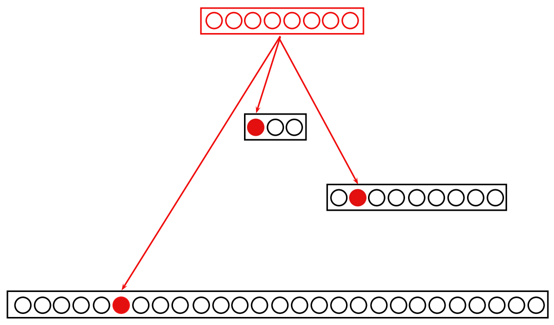


Figure 3.2: Example SoftMaxTree Activation Pattern (in red) for a 3 level tree. The output embedding (and input to this layer) is on top, the conditional word probabilities $P(Y|X, C_1, C_2)$ are on the bottom. All block (empty) leaf neurons need not be computed, thus the efficiency gain.

With this speedup we are able to quickly train a NNLM on the Billion Words dataset using nothing but a single consumer-grade GPU[13]. We demonstrate this by training two NNLMs identical in every respect except for the output layer. We allocate to each the same amount of time on a GPU. The baseline will use an NVIDIA Titan Black, while the SoftMaxTree will be handicapped by an NVidia Quadro 4000, a much slower card 2 generations behind the Titan. The hyper-parameters are $n_x = 5$, $n_i = n_o = 128$, using the same hierarchy and vocabulary as above. This time, our model includes two parameterized hidden layers, joined by $n_h = 512$ hidden neurons. We use a learning rate of 0.1 for both. The results are presented in figure 3.3. Within the alloted time, the baseline can only reach a minimum test perplexity of 511, while the SoftMaxTree reaches 292. Another way of interpreting these results is that it takes a little less than 3 days for the SoftMaxTree to obtain a lower perplexity than that obtained by the baseline after 16 days.

To understand the potential perplexity that can be reached by an NNLM using the SoftMaxTree, we train a much larger NNLM having a context of $n_x = 9$, embeddings of $n_i = n_o = 384$ and $n_h = 1024$ hidden neurons. The model is trained for 132 epochs of 10 million iterations each (where each iteration predicts a word), on an NVIDIA Titan Black GPU over the course of 2 weeks. The model was able to reach a PPL of 188 on both the validation and test set, and a PPL of 190 on the training set through cross-validation and early-stopping. This better than one of

13. Note that the original paper uses thousands of CPUs to perform its benchmark.

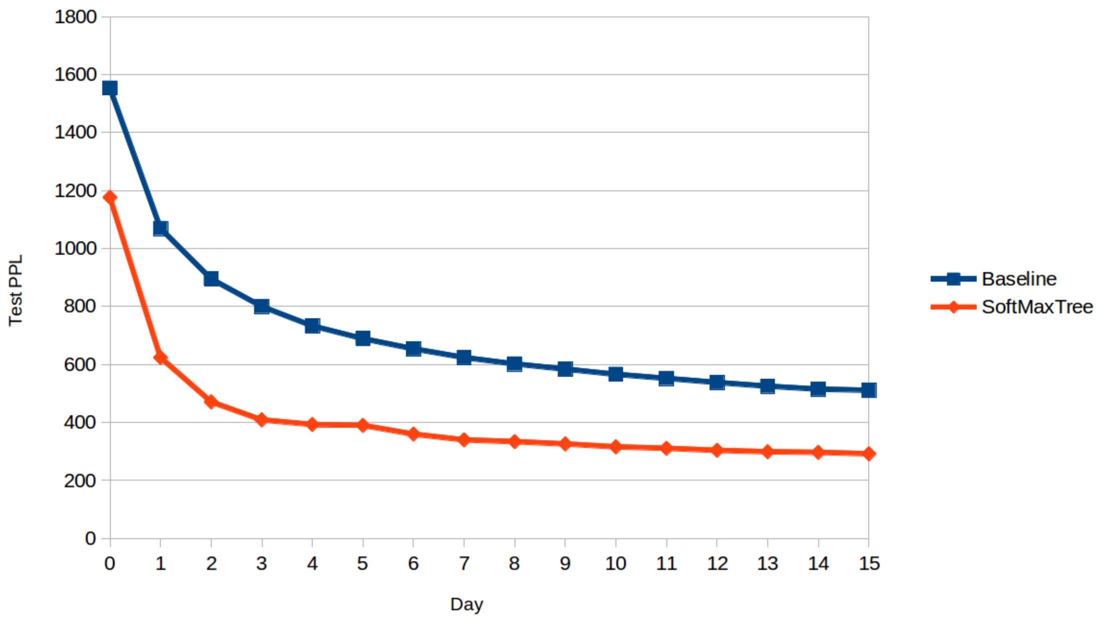


Figure 3.3: Learning curves comparing an NNLM one using SoftMaxTrees to a baseline NNLM using a vanilla softmax output layer. Each point is the minimum test perplexity btained through cross-validation at the end of each day. The SoftMaxTree was able to complete 82 epochs on a much slower card in the time it took the baseline to complete 16.

the **Interpolated KN 5-gram, 15M n-grams** baseline used in the original paper which reached a PPL of 243 after 300 CPU-hours of training. This however does not beat *Katz 5-gram, 15M n-grams* which reached a PPL of 128 after only 300 CPU-hours of training (Chelba et al. (2013)). Since the network wasn't over-fitting, adding more parameters could yield better results.

It is possible to argue that the SoftMaxTree, which is a kind of hierarchical softmax (with N_ℓ levels), is an application of distributed conditional computation. But this is mostly true for training (because it requires the targets). In the case of measuring perplexity, it happens that it also works for evaluation purposes. But in the application of predicting, say, the n most likely words following a context x, the best we can hope for is a kind of tree traversal algorithm (dynamic programming) to find these optimally.

3.4 Block-Sparse Mixtures of Experts

The Block-Sparse Mixture of Experts is a model developed for demonstrating conditional computation in the hidden layers of a neural network. Each hidden representation is divided into K segments of n neurons. Using the mixture of experts terminology, each such segment is an expert. Each hidden representation thus has its own set of experts S_ℓ, where ℓ also indexes the preceding parameterized layer M_ℓ. Segmenting hidden representations into experts implies dividing the parameter space into blocks. Parameterized layer M_ℓ would be divided into $N_\ell = K_{\ell-1} \times K_\ell$ blocks. If l and m respectively index the input and output representations of parameterized layer M_ℓ, then each block has weight matrix W_{lm} (each depicted as an arrow in figure 3.4) and bias vector b_m.

As depicted in figure 3.4, this neural network structure forms the experts of a block-sparse mixture of experts. On the other hand, the gaters, as depicted in figure 3.5, look like a standard MLP with an output space (a set of output neurons) for each expert set S_ℓ. Actually, each output space O_ℓ has a K_ℓ units, one for each expert (or segment) in S_ℓ. Both the gater network (figure 3.5) and expert network (figure 3.4) receive input from the same input space, which we can think of as consisting of just one expert. In the case of language models, the input space is the concatenation of context word embeddings. The outputs of both networks are however totally different.

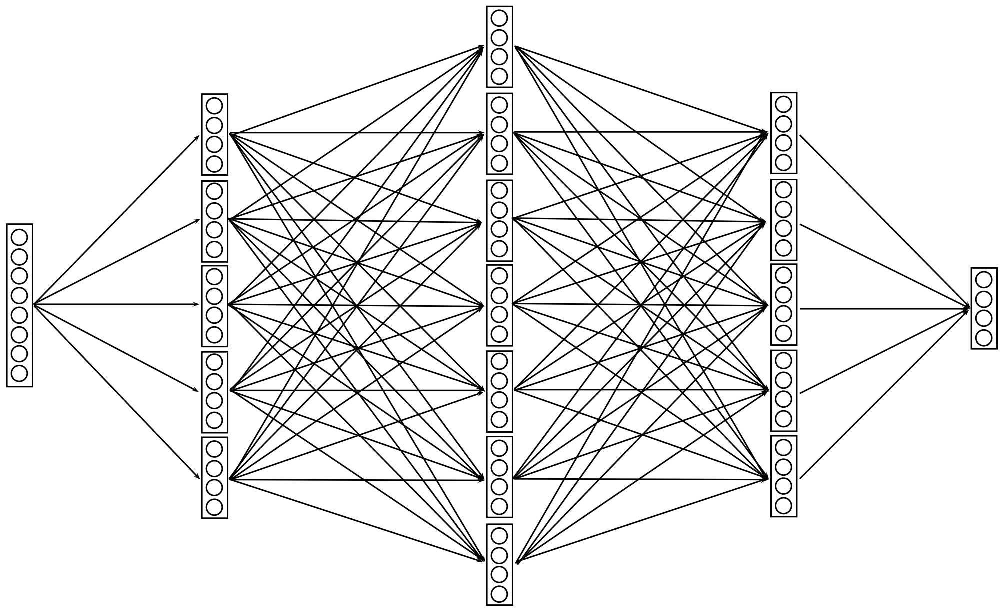


Figure 3.4: Block-Sparse Experts architecture. The parameterized layers are sparse by block. The representations are sparse by segment. Each such segment is depicted here as a rectangular box of neurons (circles). The input and output representation of the network are dense (they only have one segment).

3.4.1 BlockSparse Gater

In the previous chapter's NDT and in (Eigen et al. (2013)) the gaters received input directly from different layers of representations in the neural network. Here we consolidate all gaters into a single MLP and share only the lookup table between gater and experts. This approach, other than being used for implementing standard mixture of experts models, was chosen for two reasons: speed and consistency.

As outlined in section 3.4.2, the block-sparse operation isn't as efficient as a matrix-matrix multiply operation having the same amount of multiplication-additions, such that any efficiency gain requires very sparse operations. If we were to attach each gaters directly to its preceding sparse hidden representations (the expert segments), it would utilize the block-sparse operation and therefore be bound by the same efficiency constraints. The sparsity of the block-sparse operation used by the gater would only occur on the inputs of the gater, as opposed to both input and output (doubly sparse) where significant efficiency gains can be expected. By combining gaters into their own MLP without any block-sparse operations, the whole network can be more efficient (or faster).

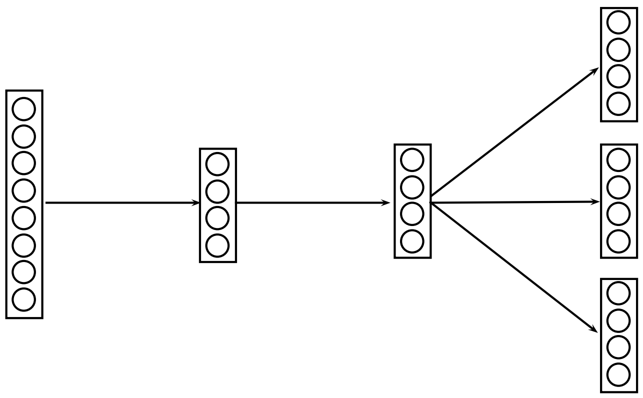


Figure 3.5: The Gater used with the Block-Sparse Experts. There are 3 output spaces, one for each of the Block-Sparse Expert hidden representation layers. When turned on, the gater output neuron will activate the commensurate segment in the expert hidden presentation.

As for the matter of consistency, consolidating gaters into a singular MLP means that they can more easily coordinate their choices — which is determined by the last hidden representation. Furthermore, the variable nature of the Block-Sparse experts Module — where a different set of experts is chosen for each example —, may not so easily bleed into the gaters themselves as these do not receive inputs or gradients from any block-sparse operations.

The gater outputs a sparse configuration of experts. By sparse we mean that approximately 90% of experts are turned off. We implement this by a combination of two methods: NoisyReLU and LazyKBest, which are detailed below. Each is implemented as its own Torch7 Module in the open-source cunnx[14] package. They are applied to the output of gater to obtain fixed-length sparse representations.

NoisyReLU

In (Bengio et al. (2013)), a noisy rectified linear unit (NoisyReLU) is used to produce sparse outputs :

$$y = \max(0, h + z) \tag{3.16}$$

where h is the tensor output an affine transform (equation 1.5) and z is a tensor of the same size. During training, z elements are sampled from a normal distribution with mean zero and a standard deviation of σ, a hyper-parameter. During evaluation, z is set to zero. This helps in reviving ReLUs that never have positive

14. https://github.com/nicholas-leonard/cunnx

h (dead units). While the original paper impose the sparsity constraint by using an L2-norm constraint that adapts its weight dynamically to maintain a specified sparsity factor, we change the NoisyReLU equation to more directly control the sparsity:

$$y = \max(0, h + z - c) \tag{3.17}$$

where c is a tensor of thresholds. During training, these are updated such that the desired level of sparsity u (the per-unit proportion of non-zero activations over D_t) is maintained. This is accomplished by keeping an exponential moving average $\hat{u}$ of the actual proportion of non-zero activations:

$$\hat{u}_{t+1} = \lambda \hat{u}_t + (1 - \lambda)\mathrm{I}_{[y>0]} \tag{3.18}$$

where y is the output of the NoisyReLU, I_e is the indicator function that is 1 when expression e is true, 0 otherwise, t indexes the iteration, and λ is the weight of the past in the moving average. The resulting $\hat{u}_{t+1}$ is thus a number between 0 and 1. During training, the threshold c is updated as follows:

$$c = \alpha(\hat{u} - u) \tag{3.19}$$

where α is a hyper-parameter. During evaluation, the current value of c is used, but not updated. Typically, $u \approx 0.1$. A C/CUDA NoisyReLU implementation as a Torch7 Module is available online[15]. In our experiments, we found that a $\sigma = 1$ and $\alpha = 1$ worked well and that some noise was often better than no noise. We also found that large values of $\lambda = 0.99$ were necessary to keep the distribution of examples to experts balanced. As for u, we set it to $u = \frac{k}{K}$, where k is the number of chosen expert-segments in a sparse representation made up of a total of K such segments.

LazyKBest

While the NoisyReLU implements a sparsity constraint on the gater outputs, it cannot guarantee an exact level of sparsity for all examples. For reasons of efficiency and ease of implementation, the Block-Sparse experts interface expects a fixed number of experts chosen per example, as well as their indices. Therefore, we

15. https://github.com/nicholas-leonard/cunnx/blob/master/NoisyReLU.lua

follow the NoisyReLU with a LazyKBest Module[16], which is a lazy — yet very fast — implementation for selecting the k highest output activations of the NoisyReLU – or any input for that matter. It divides the vector of activations into k segments, and selects the index and value of highest activation of each such segment, thereby returning k activations and their indices. Each index corresponds to an expert segment of the Block-Sparse Module. The LazyKBest module thus guarantees an exact proportion of active units. The inactive units receive no gradient as if their activation were zero.

3.4.2 BlockSparse Experts

In (Bengio et al. (2013)), all conditional computation experiments are performed by masking the unused experts for each example. In effect, no computational savings were demonstrated as the output of each expert was computed for every example; the efficiency gains were still only theoretical. For the BlockSparse experiments, all operations were implemented on the GPU to measure the potential efficiency gains that can be obtained through DCC. To do this, we make use of the BlockSparse Module which implements a matrix-matrix multiplication which is sparse by block.

Excluding parameters, the operation requires the following arguments for each example:

- **Input activation tensor** x of size $n_i \times k_l$ where n_i is the size of each input expert-segment and k_l is the number of active input expert-segments (section 3.4.1);
- **Input index vector** u of size k_l (one index per active input expert-segment);
- **Output index vector** v of size k_m (one index per active output expert-segment);
- **Output scale vector** g of size k_m, the ordered output of the gater's LazyKBest Module;

For each output expert-segment y_m, where m indexes the output of the commensurate gater output, the block-sparse operation takes the following form:

$$y_m = \sigma(\sum_{l}^{k_l} W_{v_m u_l} x_l + b_{v_m}) \tag{3.20}$$

16. https://github.com/nicholas-leonard/cunnx/blob/master/LazyKBest.cu

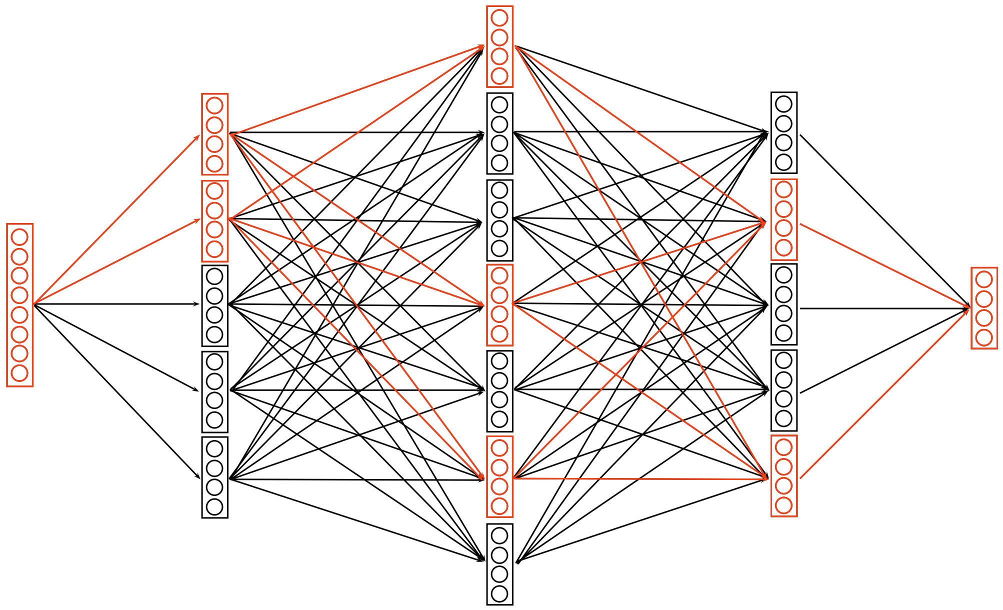


Figure 3.6: Example Block-Sparse Mixture of Experts activation pattern (in red). This kind of path would be activated by the gater, which selects which hidden segments to use.

where u_l and v_m are respectively the l-th and m-th index of the input and output index vectors, $W_{v_m u_l}$ is a weight matrix of size $n_m \times n_l$, and σ is an activation function (section 1.4.1). In our experiments, we use the tanh activation function. The block-sparse operation is thus potentially *doubly-sparse* — sparse input and output representations —, as only a fraction of input and output expert-segments need to be considered. When both input and output are sparse, the potential efficiency gain is greater. For e.g. choosing $k_l = k_m = 10$ from a total of $K_l = K_m = 100$ expert-segments on each end, each example utilizes only $\frac{k_l k_m}{K_l K_m} = \frac{100}{10000} = 1\%$ of all expert-blocks.

However, not all block-sparse operations are doubly-sparse. As can be seen in figure 3.6, the first and last layer of block-sparse operations aren't doubly-sparse, and therefore cannot benefit from as large an efficiency gain as the hidden block-sparse layers. The input and output of the multi-layer Block-Sparse experts are both dense. All layers use the same underlying implementation of the block-sparse operation, where dense inputs or outputs are considered to have $k = K = 1$.

BlockSparse

The code for the block-sparse operation is implemented as a Torch7 Module called BlockSparse which is open-source and available on-line[17]. The Module was implemented for GPU-only[18] using two different approaches.

The first makes use of the CUBLAS **cublasSgemmBatched** sub-program[19], which is optimized to perform a batch of small matrix-matrix multiplication operations on GPU. In our case, the input to each such operation is actually the vector x_{u_l} (or the gradient of the loss w.r.t. to y_m) which is multiplied with a single block weight matrix. This approach is well suited for small weight matrices, typically with less than 200×200 elements.

When the weight matrices have more element than this, the Module utilizes a different approach involving CUDA streams. These can be used to launch CUDA kernels asynchronously and concurrently. Each stream is abstracted like a queue of kernels to execute. Each stream's kernels are executed sequentially, but the GPU can process kernels from different streams concurrently (at the same time using different CUDA-cores). In this case, we use the **cublasSgemv** operation to multiply an expert-segment input with an expert-block weight matrix. This operation is particularly useful for the first block-sparse layer as the dense input vector, which is the concatenation of the context of word embeddings, is very large.

Both approaches require the use of an additional hand-written kernel for performing reductions like summing the outputs of each expert-block resulting from different input-segments and adding the biases (forward pass only). The BlockSparse Module automatically chooses the best approach given the dimensions of the expert-block weight matrices. Furthermore, the Module can translate dense inputs or outputs to their equivalent sparse forms for use with the underlying block-sparse operation.

Benchmarks

A benchmark was performed to compare the block-sparse operation to two baselines: *partial* and *full dense*. The first involves a standard matrix-matrix multiplication using the same amount of multiplication-additions as the block-sparse

17. https://github.com/nicholas-leonard/cunnx/blob/master/BlockSparse.lua
18. https://github.com/nicholas-leonard/cunnx/blob/master/BlockSparse.cu
19. http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemmbatched

does. The second is also a matrix-matrix multiplication, but uses the same amount of multiplication-additions as the BlockSparse has parameters. All benchmarks include the forward, backward and parameter update operations. Both the partial and full dense baselines are performed using the Torch7 Linear Module[20]. The baseline is available in the cunnx package's unit testing script[21].

The **partial dense** operation is expected to be faster than the BlockSparse as it can reuse the memory more efficiently by using the GPU's shared and registered memory caches. This is because the weight matrix is the same for all examples (rows in the input matrix), and can thus be cached once and reused for all examples. On the other hand, the block-sparse operation has a different set of weight matrices for each example, and thus cannot make such an effective use of memory caching.

As for the **full dense** operation, we expect it to be slower than the block-sparse operation *given enough sparsity.* Since block-sparse only computes transformations between active input and output expert-segments while the full-dense operation essentially computes the transformation between all expert-segments, we should expect a larger proportion of non-active versus active experts to result in a greater speedup. Of course, the full-dense baseline uses a dense matrix-matrix multiplication which means that it benefits from the same caching advantage as the partial dense baseline.

Description	**FD Speedup**	**PD Slowdown**
bs:8 e:384(8)x32-384(8)x32	34.67	4.54
bs:128 e:384(8)x32-384(8)x32	31.47	14.28
bs:512 e:384(8)x32-384(8)x32	24.64	24.21
bs:128 e:192(4)x64-192(4)x64	37.34	11.56
bs:128 e:96(2)x128-96(2)x128	37.52	11.32
bs:128 e:1(1)2048-384(8)x32	20.30	21.54
bs:128 e:768(16)x16-768(16)x16	1.05	21.83
bs:128 e:768(16)x16-768(16)x16	1.17	12.60

Table 3.2: BlockSparse benchmarks of speedup over full dense baseline (FD Speedup column) and slowdown over partial dense baseline (PD Slowdown column).

For our benchmark, we choose $K_l = K_m = 384$ total expert-segments, $k_l = k_m = 8$ expert-segments per example, $n_m = n_l = 32$ units per expert-segment, and a

20. https://github.com/torch/nn/blob/master/doc/simple.md#nn.Linear
21. https://github.com/nicholas-leonard/cunnx/blob/master/test/test.lua#L239

batch size of 128 examples. This is represented as **bs:128 e:384(8)x32-364(8)x32** in table 3.2 and all further benchmarks use the same nomenclature. We performed the benchmark on an NVIDIA Titan Black. The block-sparse gets a speedup of 31.47 over the full-dense baseline, while it is 14.28 times slower than the partial dense baseline. To be clear, the full dense performs a matrix-matrix multiplication of an input of size 128×12288 by a weight matrix of size 12288×12288. The partial dense performs this for an input of size 128×256 by a weight matrix of size 256×256.

To confirm that the baselines get an advantage by making effective use of the cache, we change the batch size to 8. The speedup of the block-sparse over full dense is now 34.67, while it is now only 4.54 times slower than the partial dense baseline. If we use a batch size of 512, the speedup and slowdown are 24.64 and 24.21 respectively.

Again with a batch size of 128, if we instead use larger expert-blocks $n_m = n_l = 64$ but keep the same amount of parameters and multiplication-additions by reducing the number of chosen and total expert-segments: $k_l = k_m = 4$ and $K_l = K_m = 192$, we can have a better idea of the efficiency of each individual expert-block transformation. The speedup and slowdown are now 37.34 and 11.56 respectively. If we continue in this direction by using $n_m = n_l = 128$, $k_l = k_m = 2$ and $K_l = K_m = 96$, the speedup and slowdown are now 37.52 and 11.32 respectively. This indicates that while a speedup can be obtained by increasing the size of experts while keeping the number of parameters and multiplication-additions equal, these gains become less significant as the size of experts increase. If instead, we reduce the size of experts and keep the same amount of parameters and multiplication-additions by using $n_m = n_l = 16$, $k_l = k_m = 16$ and $K_l = K_m = 768$, the speedup and slowdown are 20.30 and 21.54 respectively. This is to be expected as the GPU makes better uses of the resources available when using few large-matrices instead of a many small ones. Our algorithm would also prefer the latter as the output spaces of the gater have K_m and K_l units, which would be an additional strain on the capacity of the gater (more experts to discriminate).

Finally, as the input and output BlockSparse layers have a dense input, we will consider some benchmarks that do not involve a doubly-sparse matrix. First, we use a $n_l = 2048$, $n_m = 32$, $K_l = k_l = 1$, $k_m = 8$ and $K_m = 384$ which simulates the input layer of a context of size 8 with embeddings of size 256. The speedup and slowdown are 1.05 and 21.83 respectively. As expected, the speedup is much less pronounced while the slowdown is only a little less pronounced as compared to our initial benchmark. If we perform the same experiment to simulate an output layer using $n_l = 32$, $n_m = 256$, $K_m = k_m = 1$, $k_l = 8$ and $K_l = 384$, we obtain a speedup and a slowdown of 1.17 and 12.60 respectively. These two benchmarks imply that a significant speedup will only be obtained in the hidden block-sparse layers, not in the input and output layers. This is bad news as NNLMs are known to be most effective with shallow networks (section 3.2). This is also the reason why the gater receives its input directly from the dense input (the output of the lookup table), as opposed to receiving it from a sparse representation layer – the gater wouldn't be doubly-sparse.

3.4.3 BlockSparse Mixture

The BlockMixture Module[22] is a composite object encapsulating the gater and the different BlockSparse (expert) layers. It abstracts away many of the intricacies of forward and backward propagating between BlockSparse experts and the gater. It expects a dense input and has a dense output. A benchmark was performed using 4 BlockSparse Modules (or 3 hidden sparse representations). The 3 sparse representations respectively have $k_1 = 4$, $k_2 = 8$ and $k_3 = 4$, $K_1 = 128$, $K_2 = 256$ and $K_3 = 128$, with $n_1 = n_2 = n_3 = 64$. The input to the BlockMixture is of size $n_0 = 1024$ and the output is of size $n_4 = 256$. The batch size is 128. The gater has one hidden representation of size 256. Since there are 3 sparse representations, the gater has 3 output spaces of size K_1, K_2 and K_3. We compare the model to the usual partial and full dense baselines. In this case, the baselines are 4-layer MLPs using 4 Linear Modules. The baselines do not integrate the capacity of the gater. The benchmark evaluates the aggregate time of forward, backward and parameter updates. On an NVIDIA Titan Black GPU, the speedup and slowdown are respectively 6.01 and 22.21, where the gater uses up 15% of the BlockMixture computation time. On a lesser NVIDIA Quadro 4000 GPU, the speedup and slowdown are respectively 13.68 and 16.25 where the gater uses up 7.3% of the BlockMixture computation time. This is evidence that high-performance cards are more optimized towards dense matrix-matrix operations, making it even harder for

22. https://github.com/nicholas-leonard/cunnx/blob/master/BlockMixture.lua

us to obtain a speedup. Nevertheless, the BlockMixture implementation seems to offer a 6-fold speedup over its dense counterparts.

3.4.4 Results

In this section we present the results of training block-sparse mixture of experts (BSME) on the Google One Billion Words dataset (section 3.1). Due to the large size of the dataset, we began experimenting with different hyper-parameter configurations of the BlockMixture Module on a small subset (1/30th) of the dataset. We divided these preliminary experiments into two phases of increasing training time before moving onto the final phase utilizing the entire dataset with large epochs and longer training runs.

Phase 1

We used a baseline BlockMixture model with two sparse hidden representations which we later varied one or two hyper-parameters at a time to get an idea of the ideal configuration. The 2 sparse representations respectively have $k_1 = k_2 = 8$, $K_1 = K_3 = 224$, with $n_1 = n_2 = 32$. The NNLM uses a lookup table for the first layer with an embedding size of $n_i = 100$ and a context size of 5 such that the BlockMixture receives and input of size $n_0 = 500$. For the output we use a SoftMaxTree Module (section 3.3.3) with an output embedding size of 100 such that the output of the BlockMixture is also of size $n_3 = 100$. We used a hard constraint on the maximum norm of incoming weights of 2 and a batch size of 256 examples. The gater used 2 hidden representations of size 128 such that it was a 3 parameterized layer MLP. By training it for 100 epochs of 200 thousand iterations (examples) each with a learning rate of 0.1, we obtained a minimum PPL of 1130 on the training set. Note that the test and validation PPL were highly similar, and no over-fitting was noticed during any of our experiments on the Google One Billion Words dataset. Each experiment took approximately 6 hours to complete.

We then performed some experiments by keeping either the gater or experts from learning (a learning rate of 0) and obtained a PPL of 1450 and 2410, respectively. This confirmed that both the experts and gater were indeed working together to optimize the PPL, and that most of the learning capacity stemmed from the experts.

By modifying the baseline BSME to constrain the norm of incoming weights to be 10 instead of 2, we were able to obtain a PPL of 1030. We found larger maximum norms to have the same effect. In any case, this constraint is necessary for the SoftMaxTree module as it tends to result in NaN errors otherwise.

We also modified the baseline BSME gater to use a SoftMax Module, or a SoftMax followed by an EquanimityConstraint (algorithm 2), instead of the NoisyReLU (section 3.4.1). Even though all cases were followed by a LazyKBest Module (section 3.4.1), we found that NoisyReLU worked better than the SoftMax in terms of keeping the distribution of examples to experts balanced, and better than the SoftMax followed by an EquanimityConstraint in terms of PPL. In the former case, the NNLM tended to form a monopoly by utilizing the same set of 8 expert-segments for both sparse representations. The NoisyReLU on the other hand utilized all experts to varying degrees, were the 5 most active expert-segments were used on average 5.43% of the time, and the 5 least active were used 1.56% of the time, which isn't too far from the target activation frequency of $\frac{k}{K} = \frac{8}{224} = \frac{1}{28} = 3.57\%$. In any case, a little unbalance is to be expected.

We further modified the baseline to use $k_1 = k_2 = 4$, $K_1 = K_2 = 112$ and $n_1 = n_2 = 64$, which we know from the benchmarks presented in section 3.4.2 to have a greater speedup over the baseline. The result was a PPL of 1090, which isn't a significant improvement, but combined with the speedup was enough to justify this configuration for later experiments. It may be that fewer decisions to be made by the gater (or less expert-segments to discriminate from) may prove to be an easier task than more experts with less capacity.

Again, starting from the baseline, we increased the number of expert-segments to $K_1 = K_3 = 448$ to evaluate the model's ability to utilize this extra capacity and were disappointed by a PPL of 1110. This points to the possibility that the BSME was unable to make effective use of extra capacity given a fixed budget of multiplication-additions. Of course, this may have only mean that further training is required to obtain further reductions in perplexity.

For comparison, we evaluated 2 NNLM using the same lookup table and SoftMaxTree dimensions but without the BSME. The first model had a dense hidden representation (2 parameterized hidden layers) between the input and output Modules, while the second used two dense hidden representations. Hidden representation sizes were chosen such that the speed (examples per second) of all three models was approximately equal. The sizes of representations are 4400 for the first model, and 1350 for the second. After 100 epochs of the same size, we obtained PPLs of 1050 and 1400 for the first and second respectively. The NNLM having the same depth as our baseline BSME model would seem to take much more time in reaching low PPL while the more shallow NNLM would do so a little faster.

Concerning the hyper-parameters affecting the NoisyReLU we found that a standard deviation of 1 worked well, and that we needed to increase the weight of the past in the exponential moving average for smaller batch sizes in order to keep the distribution of examples to experts balanced. For a batch size of 256, a weight of the past of 0.99 worked well.

Finally, we also found that using a sparse initialization (section 1.4.2) and a learning rate of 0.5 worked better. Changing the relative learning rate of experts versus gater was shown to have little effect other than slowing down the learning in both cases. Similarly, changing the depth or breadth of the gater had little effect, resulting in similar PPLs to the baseline model. Therefore we concluded that smaller capacity gaters could be used, thereby increasing the efficiency of the model.

Phase 2

During phase 2 of our experimentation, we increased the number of iterations per epoch to 400 thousand to see how the model would fare with double the amount of training time — again with 100 epochs. This time our baseline model used the best hyper-parameter configurations obtained from the previous phase. We used 2 sparse representations with $k_1 = k_2 = 4$, $K_1 = K_2 = 112$, with $n_1 = n_2 = 64$. The NNLM uses the same configuration of lookup table and SoftMaxTree for the input and output layer as that used in the previous phase. We used a hard constraint on the maximum norm of incoming weights of 10 and a batch size of 256 examples. The gater utilized a single hidden representation of size 128. By training it for 100 epochs of 400 thousand iterations each with a learning rate of 0.5, we obtained a minimum PPL of 900.

To explore the effect of additional depth on the BMSE model, the number of sparse representations was increased to 3. We found that with a configuration of $k_1 = k_2 = k_3 = 8$, $K_1 = 224$, $K_2 = 448$ and $K_3 = 224$, with $n_1 = n_2 = n_3 = 32$ and every other hyper-parameter in the phase 2 baseline being equal, the model was able to reach a PPL of 780. This would seem to indicate that the BMSE model can cope well with added depth.

By using a learning rate of 0.2 instead of 0.5 for the baseline of this phase, we obtained a PPL of 770, which may indicate that a learning of 0.5 is only useful to accelerate the first iterations of training.

We also experimented with different batch sizes and found that smaller batch sizes tended to make the distribution of examples to experts more skewed while larger ones kept it more balanced.

Final Phase

For our final experiments, we attempted to train different configurations of BSME NNLMs which were small variations of the best model discovered in the previous phase. Training was performed using epochs of 10 million iterations each on the full dataset. Using a context size of 5 and input and output embedding sizes of $n_i = n_o = 128$, 3 sparse hidden representations with $k_1 = k_2 = k_3 = 8$, $K_1 = 224$, $K_2 = 448$ and $K_3 = 224$, with $n_1 = n_2 = n_3 = 32$, a batch size of 256, a learning rate of 0.4 which was decayed to 0.2 at epoch 10, 0.1 at epoch 25, 0.05 at epoch 50, and gater with one hidden representation of size 128, we found that training would plateau at a PPL of approximately 630 on all 3 sets after 80-90 epochs.

For a similar model that differed only in $K_1 = 112$, $K_2 = 224$ and $K_3 = 112$ and in using a gater with 2 hidden representations of size 64, we found that the training would plateau at a PPL of approximately 820 after 80-90 epochs. In this model, the distribution of examples to experts was somewhat skewed in the first sparse representation were the 5 most active expert-segments received an average of approximately 15% of all examples.

Using the first model presented in this last phase, differing only in its use of a large input and output embedding size of $n_i = n_o = 384$ and a large context of size of 10, the BSME would plateau after 50-60 epochs at a PPL of 990.

We also trained two baseline to compare the BSME to a NNLM of the same depth. Using 3 dense hidden representations of size 2000, which had approximately the same speed as the BSME model using an input and output embedding size of $n_i = n_o = 384$ and a large context of size of 10, we were able to obtain a PPL of 480 after 70 epochs. Again the PPL for train, validation and test sets were approximately the same.

The second baseline was performed using $n_i = n_o = 128$, a context size of 5 and 3 dense hidden representations of size 2000. It obtained a PPL of 400 after 70 epochs. Both baselines used the same learning rate schedule as the BSME models.

All experiments in this final phase took 2 weeks to run on different cards. The BSME and baseline NNLM with large input and output embedding spaces and context sizes were run on NVIDIA Titan Black GPUs while the remainder were run on a GTX 480 or Quadro 4000 GPU.

3.4.5 Conclusion

We demonstrated that the potential efficiency gains for a block-sparse mixture of experts could be obtained as compared to their full-dense alternative. However, the implementation is always slower than its partial dense alternative due to its more effective use of caching on the GPU. The speedup of the full BlockMixture model as compared to the full dense alternative is still only a 6 to 15-fold increase, depending on the GPU card. While memory caching may be to blame, there is a possibility that the parallelism of the GPU may not be fully utilized in the cublasSgemmBatched sub-program, although this is still the best option available for computing matrix-matrix multiplications for batches of small weight matrices.

Any potentially significant speedup can only be obtained from the hidden block-sparse operations as they are doubly-sparse. To alleviate this, we may chose to consider the context of word embedding as a sparse representation where words in the context can be turned off and on using a gater. Not all previous words are equally useful in predicting the next word and a gater may be able to discriminate these such that the first block-sparse layer can also be doubly-sparse.

While an efficiency gain was demonstrated, the training potential of the BSME model was not. Lack of proper hyper-optimization and time may be the main cause, as training a single model on such large dataset requires many weeks of training. We have also constrained ourselves to use small batch sizes in order to stay in a training regime where the BSME is more efficient and can therefore be compared to weaker baselines. This may prove to be too difficult a task for DCC as the BSME seems to be easier to train — in terms of keeping the distribution of examples to experts balanced — when using larger batch sizes like 512 or 1024. Instead, we should endeavor to make the evaluation as fast as possible as it is more likely to be used in an on-line production environment — which would naturally be more suited for small batch sizes.

The BSME has a problem with convergence which may be attributed to a variety of factors which only many experiments would be able to confirm or deny. The fact of the matter is that its behavior after 500 million iterations is different than after only 10 million. To explore this regime, we should hyper-optimize a model up to 50 million iterations and begin training different variations of it from that point.

Nevertheless, the NoisyReLU and LazyKBest Modules seem to work well together for controlling the sparsity and uniformity (or equanimity) of the distribution of experts to examples and vice-versa. The LazyKBest is also an excellent solution to controlling the exact sparsity of any distribution as it is implemented as a hard constraint.

3.5 Closing Remarks

In this thesis, we explored 2 variants of DCC. The first utilized a kind of Reinforcement Learning (RL) where experts would compete against each other to win the right to learn examples by backpropagation (or specialize) and to increase the gater probability of sampling the expert given the example. The second utilized a layer-wise block-sparse mixture of experts approach, where experts would still compete against each other, but less explicitly, as this was more of a side-effect of the BP learning. While the NDT model was formulated as a tree of experts, the BSME formed a more general digraph of experts. The digraph offered more possible combinations of active experts, but wasn't as successful as its simpler tree alternative — although they weren't compared on the same dataset or training algorithm.

In any case, I believe that simple application of DL/BP approaches to DCC won't be enough to make it a successful venture. Conditional models are competitive systems not so different than market economies. They must be kept balanced, consistent and under control in order to converge to a valid solution. This solution implies a division of labor, where each expert is specialized in transforming a representation into another, such that it forms an important component of a computation graph of such expert transformations. In the context of RL, each example is a state, and each combination of experts is an action where the gater is an agent which must choose the correct actions given the state in order to maximize a reward (or minimize a loss). A more formal application of RL to the problem of DCC may prove very successful.

The problem of DCC may also be formulated for evolutionary algorithms where each expert and gater is a gene and each model is composed of a configuration of such genes. A population of models could be trained in parallel sharing different parameterized genes where the best models would survive, reproduce and mutate to form variations on their genes, including using new freshly initialized genes or other existing genes. This approach may help to find and train better experts as we have noticed the importance of using a good random initialization for training conditional models.

One of the most difficult aspects of conditional models is keeping the distribution of examples to experts (and experts to examples) balanced such that most experts are useful and don't get left behind in training. We explore different solutions to this including an equanimity constraint, reinforcing a fixed amount of examples per expert, and controlling the activity of a NoisyReLU using an adaptive threshold. These approaches seem to work well in keeping the experts balanced, although they may also have a negative effect on training when they are too strong. In any case, without them, the best experts have a tendency to monopolize examples. This is a vicious circle in a competitive system as any initial advantage translates to increased learning (a resource) which translates to a competitive advantage and so forth and so on.

We also explored different ways of imposing a desired level of sparsity on the output of the gater. In the ESSRL-NDT model, experts were sampled without replacement from a multinomial probability distribution output by the gaters. This approach allowed for controlling the exact sparsity during both training and evaluation. In the BSME model, a simple LazyKBest Module was used to find the k most active gater outputs, each mapping to an expert.

Bibliography

Bell, A. J. and T. J. Sejnowski (1997). The âĂIJindependent componentsâĂİ of natural scenes are edge filters. *Vision research 37*(23), 3327–3338.

Bengio, Y. (2009a). Learning deep architectures for ai. *Foundations and Trends® in Machine Learning 2*(1), 1–127.

Bengio, Y. (2009b, January). Learning deep architectures for ai. *Found. Trends Mach. Learn. 2*(1), 1–127.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pp. 437–478. Springer.

Bengio, Y., A. Courville, and P. Vincent (2013). Representation learning: A review and new perspectives. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 35*(8), 1798–1828.

Bengio, Y. and U. de MontrÑĽeal (2002). New distributed probabilistic language models. *Dept. IRO, Université de Montréal, Montréal, QC, Canada, Tech. Rep 1215.*

Bengio, Y., P. Lamblin, D. Popovici, and H. Larochelle (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems 19*, 153.

Bengio, Y., N. Léonard, and A. Courville (2013, August). Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *ArXiv e-prints*.

Bengio, Y., H. Schwenk, J.-S. Senécal, F. Morin, and J.-L. Gauvain (2006). Neural probabilistic language models. In *Innovations in Machine Learning*, pp. 137–186. Springer.

Bengio, Y., J.-S. Senécal, et al. (2003). Quick training of probabilistic neural nets by importance sampling. In *AISTATS Conference*.

Braden, R. (1989, 10). Requirements for internet hosts – communication layers. RFC 1122.

Brown, P. F., P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai (1992). Class-based n-gram models of natural language. *Computational linguistics 18*(4), 467–479.

Chelba, C., T. Mikolov, M. Schuster, Q. Ge, T. Brants, and P. Koehn (2013). One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*.

Coates, A., A. Y. Ng, and H. Lee (2011). An analysis of single-layer networks in unsupervised feature learning. In *International Conference on Artificial Intelligence and Statistics*, pp. 215–223.

Collobert, R., Y. Bengio, and S. Bengio (2003). Scaling large learning problems with hard parallel mixtures. *International Journal of pattern recognition and artificial intelligence 17*(03), 349–365.

Collobert, R., K. Kavukcuoglu, and C. Farabet (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, Number EPFL-CONF-192376.

Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research 12*, 2493–2537.

Dauphin, Y. N. and Y. Bengio (2013). Big neural networks waste capacity. *arXiv preprint arXiv:1301.3583*.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1–38.

Eigen, D., M. Ranzato, and I. Sutskever (2013). Learning factored representations in a deep mixture of experts. *arXiv preprint arXiv:1312.4314*.

Erhan, D., P.-A. Manzagol, Y. Bengio, S. Bengio, and P. Vincent (2009). The difficulty of training deep architectures and the effect of unsupervised pre-training. In *International Conference on Artificial Intelligence and Statistics*, pp. 153–160.

Fellbaum, C. (1998). *WordNet*. Wiley Online Library.

Field, D. J. (1987). Relations between the statistics of natural images and the response properties of cortical cells. *JOSA A 4*(12), 2379–2394.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics 36*(4), 193–202.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks. In *International Conference on Artificial Intelligence and Statistics*, pp. 249–256.

Glorot, X., A. Bordes, and Y. Bengio (2011). Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, Volume 15, pp. 315–323.

Goodfellow, I. J., D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio (2013). Maxout networks. In *ICML*.

Hinton, G., S. Osindero, and Y.-W. Teh (2006). A fast learning algorithm for deep belief nets. *Neural computation 18*(7), 1527–1554.

Hinton, G. E. (1977). Relaxation and its role in vision.

Hinton, G. E., N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Jacobs, R. A., M. I. Jordan, S. J. Nowlan, and G. E. Hinton (1991). Adaptive mixtures of local experts. *Neural computation 3*(1), 79–87.

Jarrett, K., K. Kavukcuoglu, M. Ranzato, and Y. LeCun (2009). What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2146–2153. IEEE.

Jones, K. S. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation 28*(1), 11–21.

Jordan, M. I. and R. A. Jacobs (1994). Hierarchical mixtures of experts and the em algorithm. *Neural computation 6*(2), 181–214.

Krizhevsky, A., I. Sutskever, and G. E. Hinton (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105.

Le, H.-S., I. Oparin, A. Allauzen, J. Gauvain, and F. Yvon (2011). Structured output layer neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pp. 5524–5527. IEEE.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*(11), 2278–2324.

Marčelja, S. (1980). Mathematical description of the responses of simple cortical cells*. *JOSA 70*(11), 1297–1300.

Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 735–742.

Mnih, A. and G. Hinton (2007). Three new graphical models for statistical language modelling. In *Proceedings of the 24th international conference on Machine learning*, pp. 641–648. ACM.

Mnih, A. and G. E. Hinton (2009). A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pp. 1081–1088.

Mnih, A. and Y. W. Teh (2012). A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*.

Morin, F. and Y. Bengio (2005). Hierarchical probabilistic neural network language model. In *AISTATS*, Volume 5, pp. 246–252. Citeseer.

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, Volume 27, pp. 372–376.

Nowlan, S. J. and G. E. Hinton (1991). Evaluation of adaptive mixtures of competing experts. *Advances in neural information processing systems 3*, 774–780.

Ornstein, L. (1996, December 31). Unsupervised neural network classification with back propagation. US Patent 5,590,218.

Orr, G. B. and K.-R. Müller (1998). *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*. Springer-Verlag.

Pinto, N., D. D. Cox, and J. J. DiCarlo (2008). Why is real-world visual object recognition hard? *PLoS computational biology 4*(1), e27.

Rifai, S., P. Vincent, X. Muller, X. Glorot, and Y. Bengio (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 833–840.

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review 65*(6), 386.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (2002). Learning representations by back-propagating errors. *Cognitive modeling 1*, 213.

Salton, G. (1991). Developments in automatic text retrieval. *Science 253*(5023), 974–980.

Schwenk, H. and J.-L. Gauvain (2005). Training neural network language models on very large corpora. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pp. 201–208. Association for Computational Linguistics.

Stein, B. and O. Niggemann (1999). On the nature of structure and its identification. In *Graph-Theoretic Concepts in Computer Science*, pp. 122–134. Springer.

Sutskever, I., J. Martens, G. Dahl, and G. Hinton (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pp. 1139–1147.

Vincent, P., H. Larochelle, Y. Bengio, and P.-A. Manzagol (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pp. 1096–1103. ACM.

Xu, L. and S.-i. Amari (2009). Combining classifiers and learning mixture-of-experts. *Encyclopedia of artificial intelligence*, 318–326.

Yoshua Bengio, R. D. and P. Vincent (2001). A neural probabilistic language model. *Advances in Neural Information Processing Systems 13*.

Zeiler, M. D. and R. Fergus (2013). Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*.