Université de Montréal

Sequential Modeling, Generative Recurrent Neural Networks, and Their Applications to Audio

par
Soroush Mehri

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

Décembre, 2016

Résumé

L'apprentissage profond s'est imposé comme étant le cadre de concrétisation d'une intelligence artificielle spécialisée ; le chemin rêvé de beaucoup vers un futur où l'IA est omniprésente ou ce qu'on appellerait une intelligence artificielle générale. Durant ce projet, notre motivation a été l'envie de dompter cette puissante approche d'apprentissage afin de réaliser une avancée considérable vers la création d'une “Machine Parlante”.

Cette thèse décrit un modèle statistique paramétrique pour la génération inconditionnelle et de bout en bout de séquences audio dont la parole, des onomatopées et de la musique. Contrairement aux travaux réalisés dans ce sens dans le domaine du traitement du signal, les modèles qu'on propose se basent uniquement sur les échantillons audio bruts sans aucune manipulation ou extraction préalable de caractéristiques. La dimension générale de notre approche lui permet d'être appliquée à tout autre domaine - à savoir le traitement naturel du langage - dont les données requièrent une représentation séquentielle des données.

Les chapitres 1 et 2 sont consacrés aux principes de bases de l'apprentissage automatique et de l'apprentissage profond. Les chapitres suivants détaillent l'approche adoptée afin d'atteindre notre but.

Mots clés: intelligence artificielle, apprentissage automatique, réseaux de neurones profonds, apprentissage de représentations, modélisation séquentielle, modèles génératifs, génération audio

Summary

By far Deep Learning showed to be the most promising venue of achieving applied Artificial Intelligence which has been the dream of many as the path toward AI-powered future and eventually the Artificial General Intelligence. In this work we are interested in harnessing this powerful method to make bigger strides in the direction of creating a “Talking Machine”.

This thesis is dedicated to presenting a parametric statistical model for generating unconditional audio sequences including speech, onomatopoeia, and music in an end-to-end manner. Proposed model does not benefit from any handcrafted features that are developed over the course of many years in the field of signal processing rather operates on raw sample audio. As a general framework it can also potentially be applied in other domains that require modeling sequential data; e.g. Natural Language Processing.

Chapter 1 and 2 give a brief overview of the background topics including machine learning and basic building blocks of deep learning algorithms. Following chapters of this thesis present our endeavor toward the aforementioned goal.

Keywords: artificial intelligence, machine learning, deep neural networks, representation learning, sequential modeling, generative models, audio generation

Contents

List of Figures

List of Tables

List of Abbreviations

AGI	Artificial General Intelligence
AI	Artificial Intelligence
ANN	Artificial Neural Network
BiRNN	Bidirectional Recurrent Neural Network
BP	Backpropagation
BPTT	Backpropagation Through Time
CE	Cross Entropy
DAG	Directed Acyclic Graph
DL	Deep Learning
DNN	Deep Neural Network
FNN	Feedforward Neural Network
GD	Gradient Descent
GMM	Gaussian Mixture Model
HMM	Hidden Markov Model
GRU	Gated Recurrent Unit
i.i.d	Independent and Identically Distributed
LSTM	Long Short-Term Memory
MDN	Mixture Density Networks
ML	Machine Learning
MLE	Maximum Likelihood Estimation
MLP	Multi-Layer Perceptron
MSE	Mean Squired Error
NaN	Not a Number
NLL	Negative Log-Likelihood
NLP	Natural Language Processing
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
SL	Supervised Learning
SVM	Support Vector Machine
TBPTT	Truncated Backpropagation Through Time
UL	Unsupervised Learning

Acknowledgments

First and foremost, I would like to express my deepest appreciation to my family, friends, colleagues, and advisors. This has been an amazing experience and without you this thesis would not be possible.

To my mother, father, and brother, you have always supported me throughout my life by all the means you could. You are amazing.

Dr. Yoshua Bengio, Dr. Aaron Courville, and Dr. Roland Memisevic, I am truly grateful for your help and guidance. You have always been patient, willing to assist me, and helping me succeed, even when it was not easy. Specially, I should thank you for providing such a vibrant academic environment in MILA. Without any exaggeration, there was not even a single day that I did not learn something new and valuable.

Thanks to all of my friends and colleagues (in alphabetical order): Adriana Romero, Akram Erraqabi, Alexandre de Brébisson, Amjad Almahairi, Anirudh Goyal, Arnaud Bergeron, Bart van Merriënboer, Benjamin Scellier, Çaglar Gulçehre, César Laurent, Chiheb Trabelsi, David Scott Krueger, David Warde-Farley, Dmitriy Serdyuk, Dzmitry Bahdanau, Faruk Ahmed, Félix Gingras Harvey, Francesco Visin, Frédéric Bastien, Guillaume Alain, Harm De Vries, Ishaan Gulrajani, Iulian Vlad Serban, Joao Felipe Santos, Jonathan Lucuix-André, Jose Sotelo, Junyoung Chung, Kelvin Xu, Kratarth Goel, Kundan Kumar, Kyle Kastner, Kyunghyun Cho, Laurent Dinh, Linda Peinthière, Li Yao, Mathieu Germain, Mehdi Mirza, Mohamed Ishmael Belghazi, Mohammad Pezeshki, Myriam Côté, Negar Rostamzadeh, Orhan Firat, Pascal Lamblin, Philemon Brakel, Pierre-Luc Carrier, Rithesh Kumar, Saizheng Zhang, Samira Shabanian, Shubham Jain, Sina Honari, Vincent Dumoulin, Vincent Michalski, Ying Zhang, and Zhouhan Lin. Best of luck to you.

Special thanks to MILA staff members, Speech Synthesis team, Nicolas Chapados for this template, and Akram Erraqabi for translation of the French summary. Thanks to the rest of my colleagues in MILA, my friends, and those who I might have forgotten.

I also acknowledge the support of the following agencies for research funding and computing support that made this work feasible: NSERC, Calcul Québec, Compute Canada, the Canada Research Chairs, CIFAR, and Ubisoft.

1 Fundamentals of Deep Learning

In this chapter we are scratching the surface of what Machine Learning (ML) and Deep learning (DL) are and principles of machine learning will be formalized. Reader's familiarity with basic calculus, numerical computation, linear algebra, probability, and information theory is required.

1.1 Machine Learning

Machine learning as a subset of Artificial Intelligence is to algorithmically learn from data where data is assumed to be observations from a complex probability distribution or an environment that an agent would interact with. This is, however, different from other approaches in AI that harness domain-experts to explicitly build a computer program for the given task. These learning algorithms ideally would "learn" from many examples or experience in the environment which is quite similar to how humans learn. Instead of explicitly creating programs for solving a problem, one can think of machine learning as a tool to create these programs.

Continuing on the drawn example from learning in humans, it should be noted that there is an important *generalization* aspect to these learning algorithms. This ensures us that a learned algorithm or more specifically its learned features, is a fit hypothesis not only for the current examples but also for the unseen future data.

Generally in machine learning a family of functions or class of hypothesis $\mathcal{F}$ is considered to solve a particular task by the means of finding generalizable patterns in the dataset $\mathcal{D}$. In this context $\mathcal{D}$ is assumed to be a collection of data points $x \in \mathbb{R}^N$ independently drawn from an unknown identical distribution of interest, i.e. this is a collection of independent and identically distributed $(i.i.d)$ random variables. Besides from i.i.d assumption, other priors usually are assumed about $\mathcal{D}$ or $\mathcal{F}$. A learning algorithm provides steps in order to find a function or model

$f \in \mathcal{F}$ parameterized by θ that describes the data distribution the best or performs the task (quantified by a performance measures) optimally.

1.1.1 Machine Learning Tasks

Following is a limited list of some of the most common tasks that machine learning showed to be a promising solution.

Classification task is to specify which category among k labels an input belongs. A model $f : \mathbb{R}^N \rightarrow \{1 \dots k\}$ is the result of learning algorithm that performs this task where k is usually much smaller than the N. The primary goal of this task is to accurately predict the label and not truly discovering the underlying factors. Examples of this tasks are spam detection, image recognition, face recognition, speaker identification, and fraud detection.

Regression task expect the system to predict a numeric value for an input which takes the form of $f : \mathbb{R}^N \rightarrow \mathbb{R}$. Regression involves estimating or predicting a response where as classification is identifying group membership. Predicting the market value of a house given its specifications is an example of a regression task.

Machine Translation deals with translating a sequence from one language to another language. In a more broader view this could be seen as a sequence to sequence mapping where language can be interpreted as complex rules on how to put tokens in specific orders. This is, however extensively used in natural language translation, e.g. English to Chinese.

Density Estimation task is again to find a function $f : \mathbb{R}^N \rightarrow \mathbb{R}$ where $f(x) = p_{model}(x)$ is the probability mass function of the space of x. The task is slightly more complex here as the model should learn the hidden structure of data just by seeing a limited set of examples. Having an explicit model of the true probability distribution ideally would allow us to perform other relevant tasks including denoising, imputing the missing values, compression, and synthesis. Current models are not designed to performed well in all of the sub-tasks as they usually require computing an intractable denominator in p known as partition function.

1.1.2 Learning Styles

A learning algorithm also can be categorized by the style of its interaction with data or environment. Organizing these algorithms based on their style is useful to define a task and properly construct and use the model and dataset.

Supervised Learning. In this form of learning, each data point in the dataset is a pair of (input, desired output/label/target) and learning algorithm instruct the model to learn the mapping function. Classification and regression are examples of supervised learning tasks.

Unsupervised Learning. There is no explicit target available in the dataset hence no instruction could be provided. Learning algorithm should find the structure or prominent features by only accessing to the input space of the dataset. Clustering, dimensionality reduction, and density estimation are categorized as unsupervised tasks.

Reinforcement Learning. The machine is trained to make decisions to maximize overall reward that could be immediate or delayed from interacting with environment. This style of learning is mostly based on trial and error. Provided response from agent's environment in the form of reward, it will then change its decision making policy without any form of supervision. An agent can learn by being set loose in its environment, without the need for specific training data to be generated and then used to teach the agent. Game playing and control problems could be addressed by this style of learning.

1.1.3 Function Approximation

So far machine learning was introduced in relation to finding an appropriate function for a task. Having access to only a finite set of examples from the true data distribution p_{data} (known as *dataset*), one approach would be to learn an approximate function f defined by its parameters θ. To formalize this discussion, in case of supervised learning we can assume $\tilde{y} = f_\theta(x)$ is the approximate function for the true $x \to y$ generator.

The goal is to find parameters that for all instances of (x, y) from data distribution, predicted target $\tilde{y}$ be as close as possible to the true target y measured by distance metric d. This can be defined as cost function C as:

$$C(\theta) = \mathbb{E}_{(x,y)\sim p_{data}(x,y)}[d(\tilde{y}, y)] = \int\limits_x \int\limits_y d(f_\theta(x), y) p_{data}(x, y)\, dx\, dy \tag{1.1}$$

This is known as expected risk or loss (Vapnik, 2013) which is by itself intractable.

Using Monte Carlo integration expected loss could be approximated as empirical loss:

$$\tilde{C}(\theta) = \frac{1}{n}\sum_{i=1}^{n} d(\tilde{y}^i, y^i) \tag{1.2}$$

where n is the number of data points in dataset, $\tilde{y}^i = f_\theta(x^i)$ is the approximation of y^i, target variable for i-th data point (x^i, y^i) in a given dataset.

In following sections we will have a discussion of how to find the best set of parameters $\tilde{\theta} = \arg\min_\theta \tilde{C}(\theta)$. As expected loss C is intractable, from this point when referring to cost or C, the tractable empirical loss $\tilde{C}$ is what we are taking into account.

1.2 Deep Learning and Artificial Neural Networks

Deep Learning (DL) is a specific class of tools for addressing machine learning tasks that utilizes Artificial Neural Networks (ANN or NN), inspired by human brain – literally the most known powerful computing machine. Neural networks are loosely formed around processing units found in neuro-science literature in order to learn abstract hierarchy of representations and features.

1.2.1 Modeling Neurons

Human brain is constructed from single neurons in a complex inter-connected web known as biological neural networks. A closer look at this network, as in Fig. 1.1 [1], shows each neuron is connected to a much smaller subset of all neurons by connection between their axon and dendrite terminals. These cells communicate through a synapse (gap between axon of neuron and dendrite of another) if overall input signals are passed a cell-specific threshold. This state of a neuron is called activation which causes further transmission of signal to following cells. Biological neural units first formalized by McCulloch and Pitts (1943) with linear threshold

1. Source: videolectures.net/deeplearning2015_bottou_neural_networks

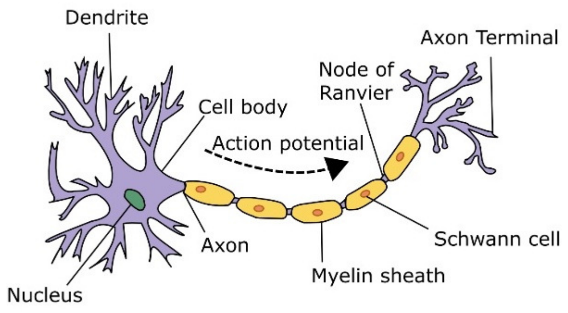


Figure 1.1 – Schematic of a biological neuron

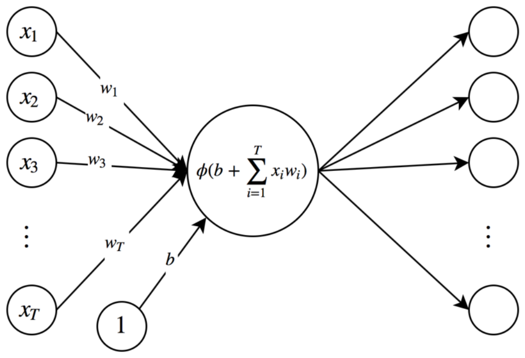


Figure 1.2 – Simplified neuron model

units. Fig. 1.2 is a simplified model of a biological neuron which is summarized in Eq. 1.3. ϕ signifies an activation function (or non-linearity) and accordingly, its input is named pre-activation.

$$\phi(b + \sum_{i=1}^{T} x_i w_i) \tag{1.3}$$

In other word, each neuron performs a dot product with the weights and inputs, adds the bias and applies the non-linearity. A compact vectorized reformulation of Eq. 1.3 presented in Eq. 1.4. In this reformulation, we are assuming that inputs $\mathbf{x} = \{x_1, \dots, x_T\}$ are passing information to their S fully-connected immediate neighboring neurons. Calculating pre-activation for all the S neighbors can be summarized as a vector-matrix multiplication between W and $\mathbf{x}$. Note that $\mathbf{b}$ is a vector of biases b and $+$ operator in this equation is a vector-vector addition.

$$\phi(\mathbf{b} + W'\mathbf{x}) \tag{1.4}$$

Some of the choices for activation function ϕ that operates on pre-activation z (either element-wise or on a tensor) are as follow:

Step function. This was originally used in Perceptron models that exactly simulated the threshold of a cell.

$$Step(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases} \tag{1.5}$$

Logistic functions. It is a family of functions in the following general form in "S" shape that includes sigmoid function (Eq. 1.7), hyperbolic tangent (Eq. 1.8), and hard-sigmoid (Eq. 1.9; an ultra-fast piece-wise linear approximate). This unit is derived from probabilistic interpretation of machine learning algorithms like logistic regression.

$$logistic(z) = \frac{a}{1 + e^{-b(z-z_0)}} \tag{1.6}$$

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

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{1.8}$$

$$hard{-}\sigma(z) = \max(0, \min(1, \frac{z+1}{2})) \tag{1.9}$$

Rectifier. Also known as ramp function, Rectifier Linear Unit (ReLU; Eq. 1.10) was first introduced by Hahnloser et al. (2000). Previously training deep neural networks (more on them in future sections) considered to be difficult (see Glorot and Bengio (2010)), despite them being powerful models. Glorot et al. (2011) first showed that they facilitate training deep neural networks, making this activation function and its variants including noisy-ReLU (Nair and Hinton (2010); Eq. 1.11) and leaky-ReLU (Maas et al. (2013); Eq. 1.12) the new standard for deep learning algorithms. α in leaky-ReLU is usually set to small values from 0.01 to 0.2 or in general case of parametric ReLU (He et al., 2015) could be a parameter like any other parameters in θ that could be learned. Softplus (Dugas et al., 2001) is a smooth approximate of ReLU with the interesting property that its derivative is the sigmoid function (Eq. 1.7).

$$ReLU(z) = \max(0, z) \tag{1.10}$$

$$Noisy{-}ReLU(z) = \max(0, z + \epsilon); \epsilon \sim \mathcal{N}(0, \sigma(z)) \tag{1.11}$$

$$Leaky{-}ReLU(z) = \begin{cases} z & z \geq 0 \\ \alpha z & z < 0 \end{cases} \tag{1.12}$$

$$softplus(z) = \ln(1 + e^z) \tag{1.13}$$

One-sided slope of ReLU brings the benefit of mathematical justifications (Montufar et al., 2014) alongside with biological plausibility (Hahnloser et al., 2003). Also having a hard-zero as one linear piece enforces sparse activation throughout the neural network but with the cost of having "dead" units. Among other benefits are efficient computation (in compare to logistic functions due to lack of exponentiation e) and helping the learning procedure that will be discussed in section 1.3.

Softmax. Unlike previous activation functions, softmax function (normalized exponential function) introduced by Bridle (1990) is operating on a set of pre-activation values. Considering logistic functions map a value to a bounded value (e.g. 0 to 1), softmax is generalizing the same idea by mapping S pre-activation values to S values each bounded between 0 and 1 while keeping their sum equal to 1 (Eq. 1.14).

$$Softmax(z_1, \ldots, z_S)_j = \frac{e^{z_j}}{\sum\limits_{s=1}^{S} e^{z_s}} \tag{1.14}$$

Armed with model of a neuron we can perform simple classification tasks. One way of doing this is by *binary SVM classifier* in which a max-margin hinge loss could be added to the output of the neuron. *Logistic regression* or *binary softmax classifier* is another way which requires interpreting Eq. 1.3 as probability of model favoring one of the classes. To make sure that the output is a valid probability distribution we can use $\phi = \sigma$ in Eq. 1.7. Probability of an input $\mathbf{x}$ belonging to class C_0 is defined by $p(y = C_0|\mathbf{x}; \theta = \{w_1, \ldots, w_T, b\})$. Probability of other class C_1 can be computed by $p(y = C_1|\mathbf{x}; \theta) = 1 - p(y = C_0|\mathbf{x}; \theta)$. Setting a threshold of 0.5 can define a decision boundary over the input space for each class.

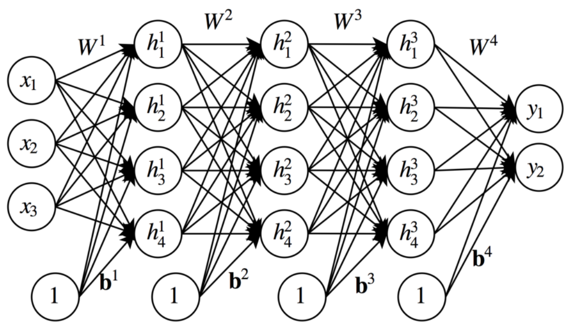


Figure 1.3 – An example of an MLP with three intermediate/hidden layers, each with dimensionality of 4, mapping $\mathbf{x} = \{x_1, x_2, x_3\}$ to $y = \{y_1, y_2\}$ with $\theta = \{W^1, W^2, W^3, W^4, \mathbf{b}^1, \mathbf{b}^2, \mathbf{b}^3, \mathbf{b}^4\}$ as parameters of this particular network.

1.2.2 Feed-Forward Neural Networks

Brain is the best example of a computing machine that efficiently combines its simple but numerous processing units, i.e. neurons, to solve complex tasks like reading, decision making and language understanding. Achieving this is possible by breaking the complex problems into high-level and abstract sub-tasks or representation of an input or stimuli.

An approximate of a function using a neural networks is usually done by directed acyclic graphs (DAGs) that transforms input to consecutive intermediate representations and finally to the output space. Hence, a function of interest $y = f(\mathbf{x}; \theta)$ is represented by $f(\mathbf{x}; \theta) = f^3(f^2(f^1(\mathbf{x}; \theta^1); \theta^2); \theta^3)$ with each f^i being an arbitrary and differentiable function. These chain structures are extensively used in this field.

Feed-forward neural networks, also called multilayer perceptrons (MLPs; see Fig. 1.3) are the first model of neural architecture that can simulate layers of representation in brain. By choosing f^i functions to be similar to Eq. 1.4 (linear transformation by multiplication by a weight matrix W, followed by a translation by bias $\mathbf{b}$, and finally a non-linearity of ϕ) we can have a simple form of an MLP.

So far we have talked about function approximation as a way of finding or mimicking the true function in machine learning. It should be pointed out here that investigating the representational power of MLPs is important to make this approach successful. Cybenko (1989) and Hornik et al. (1989) showed that family of MLPs, even with single hidden layer, are theoretically powerful. Multilayer feed-forward networks are capable of approximating any function to any desired degree of accuracy regardless of the activation function, dimensionality of input space, and the input space environment. It is noteworthy that flexibility of these models, huge amount of data, availability of high power computing machines, priors that can defeat the curse of dimensionality, and efficient inference are the main ingredients in current rise of deep learning.

Although there are no theoretical constraints for success of MLPs, in practice these models are not capable of solving all the tasks due to inadequate learning, insufficient number of hidden units or the lack of a deterministic relationship between input and target. In upcoming sections other architectures will be introduced that are more suited.

1.3 Learning as optimization

To complete the definition of a learning machine, we should define and discuss different cost functions and how to procedurally "learn" the parameters in order to decrease the cost function efficiently.

1.3.1 Cost Functions

Defining an appropriate cost function for a deep learning model plays an important role in success of the final system. Output of a cost function (also known as loss function or objective function) is to summarize in a single number how far the model is from the true generating function. There are some basic cost functions that are widely used. Modifying or some times combining these cost functions make the learning easier by encoding the prior knowledge of the problem in the model itself.

We are usually interested in a conditional distribution $p(y|x;\theta)$ in which case Maximum Likelihood Estimation is a prominent approach. This is measured by cross-entropy between model distribution and the data distribution:

$$C(\theta) = -\mathbb{E}_{x,y\sim p_{data}} \log p_{model}(y|x) \tag{1.15}$$

Otherwise we could be interested in a conditional statistic of the distribution. For example, we could train a predictor $f(x;\theta)$ to predict the mean of y. This is possible by finding a function such as below:

$$f^*(x) = \mathbb{E}_{y \sim p_{data}(y|x)}[y] \tag{1.16}$$

$$f^* = \arg\min_f \mathbb{E}_{x,y \sim p_{data}} ||y - f(x)||^2 \tag{1.17}$$

See Goodfellow et al. (2016) for more examples and detailed explanation. In this section basic loss functions will be introduced with the goal of minimizing the dissimilarity between y (the desired target) and $\tilde{y}$ (its approximation).

Square. It is used in regression problems to encourage the output of the model which is a real-valued number to be close to the true value.

$$\frac{1}{2}(\tilde{y} - y)^2 \tag{1.18}$$

Gaussian. When predicting the mean of a Gaussian distribution maximum likelihood is equivalent to minimizing the mean square error.

$$p(y|x) = \mathcal{N}(y; \tilde{y}, I) \tag{1.19}$$

Generalization of this cost could be found in Mixture Density Networks (Bishop (1994); MDNs) that assumes each data point has some probability to be associated to a mixture of Gaussians (Eq. 1.20). Each component (with its own set of parameters) has a contribution (α_i) in the final distribution. The network then produces the *parameters* (including the contribution weights for each individual component) for this distribution.

$$p(y|x) = \sum_{i=1}^{k} \alpha_i(x) \mathcal{N}_i(y; \mu_i(x), \Sigma_i(x)); \quad s.t. \sum_{i=1}^{k} \alpha_i = 1 \tag{1.20}$$

Cross-entropy. For categorical output a Softmax unit can produce the probabilities for each category. This is true for Multinoulli output distributions. Here we are trying to maximize:

$$\log p(y = C_j; z_1, \ldots, z_S) = \log softmax(z_1, \ldots, z_S)_j \tag{1.21}$$

$$= z_j - \log \sum_{s=1}^{S} e^{z_s} \tag{1.22}$$

In the special case of Bernoulli as the output distributions which only needs one number to be defined, sigmoid activation function could be used to predict $p(y = 1|x)$.

1.3.2 Gradient-based Iterative Optimization

The cost function quantifies the quality of a set of parameters θ for the task at hand. The goal of optimization is then to find parameters that minimizes the loss function. As this is a search problem, most of algorithms in classic AI can be applied in this context.

For example one naive strategy would be to try random search. Although simple, this idea will not work in practice at all. Next idea could be to refine initial parameters so that they can get better over time in the hope of finding the optimal solution. This is inherently similar to local search algorithms, namely stochastic hill climbing, where at each point one single step would be taken in a random direction if the cost function decreases. This is better than random search in parameter space but still computationally wasteful.

As it turns out in this situation where we only have access to local information of the cost surface, following the gradient for an infinitesimal step at each point is the best direction downward. We are also going to ignore the global searching algorithms that harness Bayesian optimization.

Gradient Descent (GD) is overshadowing all other local iterative optimization algorithms in deep learning. As the name suggests, this algorithm is entirely dependent on calculating the gradient with regard to every single parameter in θ given an input x.

Considering a complex computation of a function in a DAG form that transform the input to the output, it would be ideal to compute the gradient of cost using the chain rule. The purpose of *backpropagation* (BP) or *backward propagation of errors* is to compute the gradient for all the parameters. In this computation graph for all the modules one should be able to compute the gradient of the *module's output* with regard to *its input* and *its parameters*, if any.

Backpropagation is possible with repeated two phases: forward-pass (propagation/inference) and backward-pass. In the forward-pass when the network is exposed to an input, the (pre-)activations are computed layer by layer (or in the

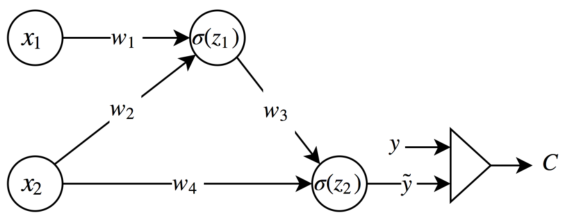


Figure 1.4 – A toy example of a DAG, representing a computation graph that transforms $\mathbf{x} = \{x_1, x_2\}$ to $\tilde{y}$, with sigmoid activation function and square loss.

case of a DAG, module by module) until it reaches the output layer where the cost function is computed. Computing the gradients for the output layer neurons are straightforward as the outputs explicitly appear in the cost function. However, recursively applying the chain rule and re-using the pre-computed activation values from forward-pass is used to compute the gradients for intermediate layers. Following section is an example of backpropagation applied to an arbitrary DAG in Fig. 1.4.

Forward-pass to compute the cost (empirical loss C) given x_1 and x_2 where value of the activation functions would be stored for further usage:

$$z_1 = w_1 x_1 + w_2 x_2 \tag{1.23}$$

$$z_2 = w_3 \sigma(z_1) + w_4 x_2 \tag{1.24}$$

$$\tilde{y} = \sigma(z_2) \tag{1.25}$$

$$C = \frac{1}{2}(\tilde{y} - y)^2 \tag{1.26}$$

Corresponding gradients for each intermediate module:

$$\frac{\partial z_1}{\partial w_1} = x_1 \quad and \quad \frac{\partial z_1}{\partial w_2} = x_2 \tag{1.27}$$

$$\frac{\partial z_2}{\partial w_3} = \sigma(z_1) \quad and \quad \frac{\partial z_2}{\partial w_4} = x_2 \quad and \quad \frac{\partial z_2}{\partial z_1} = w_3 \sigma(z_1)(1 - \sigma(z_1)) \tag{1.28}$$

$$\frac{\partial \tilde{y}}{\partial z_2} = \sigma(z_2)(1 - \sigma(z_2)) \tag{1.29}$$

$$\frac{\partial C}{\partial \tilde{y}} = \tilde{y} - y \tag{1.30}$$

This leads us to computing $\nabla C(\theta)$ with chain rule in backward-pass where

$\theta = \{w_1, w_2, w_3, w_4\}$:

$$\frac{\partial C}{\partial z_2} = \frac{\partial C}{\partial \tilde{y}} \frac{\partial \tilde{y}}{\partial z_2} = (\tilde{y} - y)\sigma(z_2)(1 - \sigma(z_2)) \tag{1.31}$$
$$\frac{\partial C}{\partial w_3} = \frac{\partial C}{\partial z_2} \frac{\partial z_2}{\partial w_3} = \frac{\partial C}{\partial z_2}\sigma(z_1) \tag{1.32}$$
$$\frac{\partial C}{\partial w_4} = \frac{\partial C}{\partial z_2} \frac{\partial z_2}{\partial w_4} = \frac{\partial C}{\partial z_2} x_2 \tag{1.33}$$

$$\frac{\partial C}{\partial z_1} = \frac{\partial C}{\partial z_2} \frac{\partial z_2}{\partial z_1} = \frac{\partial C}{\partial z_2} w_3 \sigma(z_1)(1 - \sigma(z_1)) \tag{1.34}$$
$$\frac{\partial C}{\partial w_2} = \frac{\partial C}{\partial z_1} \frac{\partial z_1}{w_2} = \frac{\partial C}{\partial z_1} x_2 \tag{1.35}$$
$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial z_1} \frac{\partial z_1}{w_1} = \frac{\partial C}{\partial z_1} x_1 \tag{1.36}$$

Notice that in the computation of gradients we are re-using a lot of computation from forward-pass and gradients from higher modules. This is more efficient than fully expanding the loss expression and getting the gradients which will be more apparent when dealing with complex architectures. In addition, the whole process can be applied to all tensor types and modules as long as the derivation is defined.

The gradient of a function ∇C is a vector in the direction of greatest rate of increase in the value of function and the magnitude measures this rate. Assuming we have access to the gradient of the cost function, for example from backpropogation algorithm, the optimization step to update the parameters would be:

$$\theta_{new} \leftarrow \theta_{old} - \eta \nabla C(\theta_{old}) \tag{1.37}$$

where η is the learning rate that specifies the step size in each update. As a clarification, it should be pointed out that backpropagation algorithm is simply a step-by-step procedure to compute the exact symbolic expression for the gradients in a function and a computational graph in general. It should not be confused by algorithms that use these gradients, e.g. optimization methods.

Eq. 1.37 is the update rule known as full-batch gradient descent which in practice works quite well. However, one problem with this method is that it needs to compute the cost and the gradient for all the data points in the dataset (see Eq. 1.2).

To solve this issue, it is more common to use the stochastic approximation of Gradient Decent algorithm known as Stochastic Gradient Descent (SGD; Robbins and Monro (1951)) – its noisy estimate given a smaller random sample of data points that are present in the whole dataset. This requires many iterations over different *mini-batches* instead of full batch and updating the parameters accordingly which is more economical.

From section 1.1.3, we approximated the expected loss with empirical loss as a way of measuring the quality of a function approximator. Empirical loss was defined for a dataset with n data points as the average distance between predicted target $\tilde{y}^i$ and true target y^i measured by d (superscript i is the index of a data point in the dataset):

$$\frac{1}{n}\sum_{i=1}^{n} d(\tilde{y}^i, y^i) \tag{1.38}$$

In SGD this loss is approximated once more for mini-batch M, a much smaller subset of dataset:

$$C_M(\theta) = \frac{1}{|M|}\sum_{m\in M} d(\tilde{y}^m, y^m) \tag{1.39}$$

where $|M|$ is the size of mini-batch and m is the index of data points inside mini-batch. It should be noted that the expected value of C_M and the expected loss are the same. Now we can re-write the update rule for a given mini-batch M:

$$\theta_{new} \leftarrow \theta_{old} - \eta \nabla C_M(\theta_{old}) \tag{1.40}$$

where

$$\nabla C_M(\theta) = \frac{1}{|M|}\sum_{m\in M} \nabla d(\tilde{y}^m, y^m) \tag{1.41}$$

is an unbiased estimator of the gradient and its variance is proportional to the size of mini-batch, $|M|$.

Fig. 1.5 [2] shows the sensitivity of this algorithm to the choice of learning rate for a hypothetical optimization scenario. Finding an optimal learning rate can make

2. Source: http://www.benfrederickson.com/numerical-optimization/

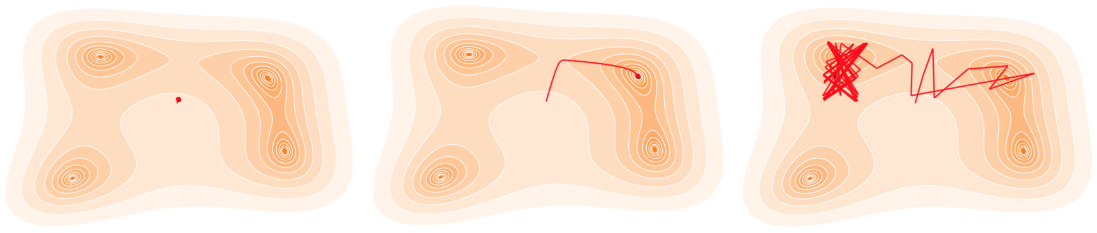

(a) Low learning rate. **(b)** Moderate learning rate. **(c)** High learning rate.

Figure 1.5 – Gradient descent algorithm applied to $f(x_1, x_2) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$ for different learning rates. With learning rate set too low, the convergence time is increasing and there is a lot of wasted computation. On the other hand, increasing the learning rate might cause oscillation if not causing unstable computation due to NaN.

a difference between a well-trained and an underfitted model. There are more advanced techniques to mitigate this problem by scheduling the learning rate while some of these techniques use an approximate of second-order information to adjust to the curvature of the loss surface.

So far we have discussed the methods that manipulate a global learning rate for all the parameters. As a next step to improve upon the previous methods, we are considering optimization methods that adapt the learning rate per-parameter.

AdaGrad. Duchi et al. (2011) originally proposed it as an adaptive learning rate method. In a nutshell, parameters that get less updates will be updated with larger values while those parameters that are updated more frequently are getting smaller updates. This is a good choice when dealing with sparse data.

$$g_{t,i} = \nabla_\theta C(\theta_i) \tag{1.42}$$

$$G_{t+1,ii} \leftarrow G_{t,ii} + g_{t,i}^2 \tag{1.43}$$

$$\theta_{t+1,i} \leftarrow \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i} \tag{1.44}$$

ϵ is the smoothing term to avoid division by zero and G is accumulating squares of the gradients with regard to θ_i on the diagonal.

RMSprop. AdaGrad's main weakness is that diagonal of G keeps growing throughout the training as a result of added positive numbers. This can aggressively reduce the learning rate and make the learning stop prematurely. RMSprop, as shown below, instead uses a moving average of the squared gradients.

$$G_{t+1,ii} \leftarrow \gamma G_{t,ii} + (1-\gamma) g_{t,i}^2 \tag{1.45}$$

$$\theta_{t+1,i} \leftarrow \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i} \tag{1.46}$$

Here, γ is decay rate, another hyper-parameter with typical value of 0.99.

Adam. Proposed by Kingma and Ba (2014) is similar to RMSprop but with added momentum. Besides from keeping track of a exponentially decaying average of past squared gradients (v_t in Eq. 1.47), Adam will keep an exponentially decaying average of past gradients m_t similar to momentum as well. Notice that in this equation we are using the vectorized notation.

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{1.47}$$

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{1.48}$$

In the original work, authors observed that when initializing v_t and m_t with zero vectors, they are biased toward zero specially when β_1 and β_2 are close to 1. Following equations is how they have solved this problem:

$$\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t} \tag{1.49}$$

$$\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t} \tag{1.50}$$

Subsequently, the update rule, with suggested default value of $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e{-}8$, is:

$$\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{1.51}$$

Orthogonal to the usage of a better optimization methods, one can specifically alter the approximating function, i.e. neural network, or its parameterization in order to facilitate training procedure. One that we are interested in is Weight Normalization (Salimans and Kingma, 2016). The general idea here is to reparameterize every weight matrix W in the network to decouple their norm g and the direction $\frac{v}{||v||}$.

$$W = g\frac{v}{||v||} \tag{1.52}$$

$$\nabla_g C = \frac{\nabla_W C.v}{||v||} \tag{1.53}$$

$$\nabla_v C = \frac{g}{||v||}\nabla_W C - \frac{g\nabla_g C}{||v||^2}v \tag{1.54}$$

This is a simple but effective idea to obtain faster convergence.

2 Sequential Modeling

Data comes in different modalities. Modeling sequential and variable length data is an interesting and significant task. When modeling sequences we are interested in mapping an input sequence into an output sequence. Fig. 2.1[1] is broadly categorizing the tasks based on the input and target of the neural network. In this figure, rectangles represent vectors and arrows represents mapping functions between vectors.

One to one. There are no sequential variable involved and a stationary function processes fixed-sized input to fixed-sized output. Handwritten digit recognition is an example of this category.

One to many. The output is a sequence, but not the input; for example image captioning.

Many to one. Activity recognition in video is an instance of a task that requires a single output given a sequence. Another example as stated in Karpathy (2015) could be sentiment analysis, i.e. classifying a given sentence.

Many to many. This is possible by swiping the information from one sequence to a summary vector and generating the new sequence. Neural translation systems are categorized under this task.

Synced many to many. A one to one correspondence is considered between each term in the input and the target of the same timestep. Named-entity recognition along with per-frame labeling of a video clip are applications of this category.

For other applications where there is no specific target sequence, one can try to predict the next token in the sequence given previous ones. Hence, the target sequence becomes input sequence but shifted. As we are dealing with temporal sequences there is a natural order in the data which makes this method a viable solution. In general, joint distribution for a sequence is formulated by product rule

1. Source: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`

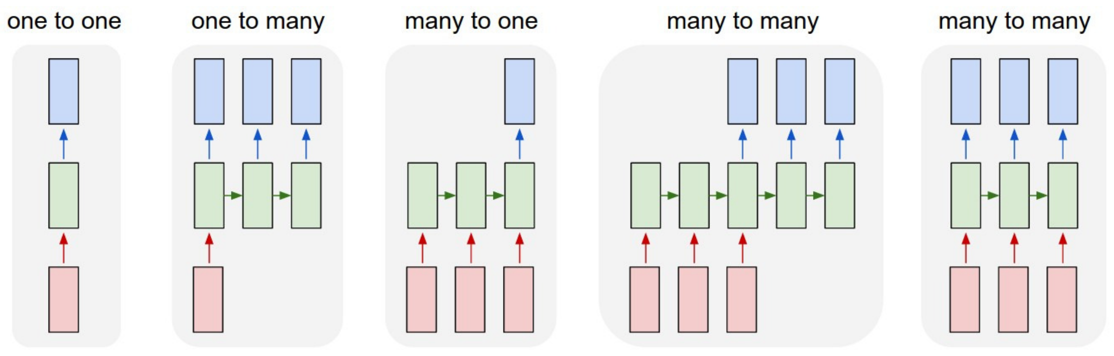


Figure 2.1 – Sequential modeling tasks based on the type of input and output.

as follow where $x = \{x_1, \ldots, x_T\}$ are terms in the sequence:

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p(x_t | x_1, \ldots, x_{t-1}) \tag{2.1}$$

Nevertheless, exploiting teaching signals from predicting next term in a sequence while making use of supervised learning methods, make the distinction between supervised and unsupervised learning less clear.

2.1 Memoryless Models

To start out with, we will consider models that predict next term given a fixed context window from past using delay taps. Simplest form of these models are called linear autoregressive models in which the output variable depends linearly on the past values. See Fig. 2.2 for a sketch of these models.

A generalized variation of this model is using hidden layers to change the dependency of the output variable from linear to more representative and complex form of non-linear with regard to input. Simply put, next term would be a function of few previous terms parameterized by a feed-forward neural network or an autoregressive MLP.

These models cannot cope with existing long-term dependencies in a sequence as they ignore the information residing outside of their context window.

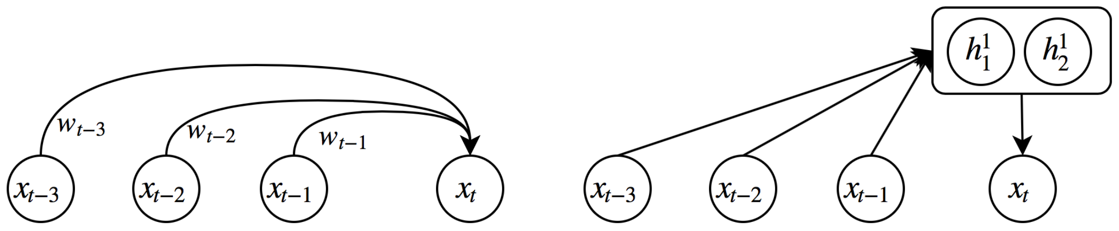


Figure 2.2 – Linear autoregressive model on the left and autoregressive MLP on the right.

2.2 Hidden States as Memory

If we can incorporate proper internal dynamics to the hidden states, we can expect the model to *remember* the relevant information for a long time implicitly in the hidden states.

2.2.1 Hidden Markov Models

Prerequisite to computing the left-hand side of the Eq. 2.1 is to compute the conditional probability of $p(x_t|x_1, \dots, x_{t-1})$. As an initial attempt we can assume each observation is independent of all previous ones except for the last one, which corresponds to first-order Markov chain:

$$p(x_t|x_1, \dots, x_{t-1}) = p(x_t|x_{t-1}) \tag{2.2}$$

With such a model estimation of parameters by maximum likelihood is relatively easy and this translates to having a dataset with each pair of output as a data point.

While this model has close ties to bigrams, dependencies inside of a sequence can easily go beyond just previous term. One way to overcome this is to formulate it as higher-order Markov models, e.g. second-order Markov model or trigram model. It will soon become apparent that the dataset will become sparse quite fast with the increase of the order as many counts would be zero in training data.

As an alternative we can assume a generative process where the observations are the result of stochasticity of some hidden states z. z is now a hidden state variable that can be a first-order Markov chain by itself that generates a state sequence. This is the core definition of Hidden Markov Models (HMMs; Fig. 2.3).

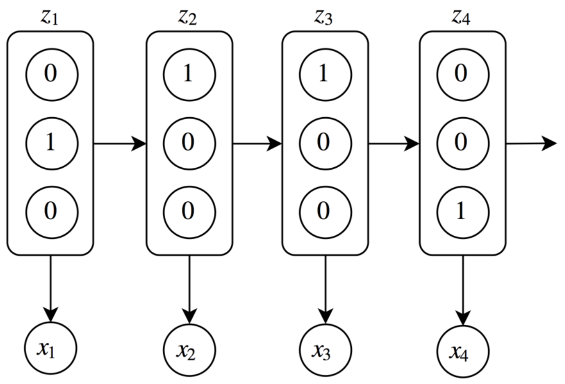


Figure 2.3 – Generative procedure in Hidden Markov Model

HMMs have discrete hidden states as a one-hot vector, i.e. in each timestep only one state is active from all H states. The stochasticity of transitions between states are encoded in a transition matrix A. In the next step of the generative process, state sequence is converted to observable symbols by the means of emission probabilities $p(x_t|z_t)$. In speech recognition in which HMMs are a dominant solution, the emissions (observation) are modeled using Gausian Mixture Models. GMM-HMMs are trained to maximize the likelihood of generating the observed features.

In short, the goal is to capture the joint probability of hidden states and observations:

$$p(x_1, \ldots, x_T, z_1, \ldots, z_t) = \prod_{t=1}^{T} p(z_t|z_{t-1})p(x_t|z_t) \tag{2.3}$$

$$p(z_t = \alpha|z_{t-1} = \beta) = A_{\alpha\beta} \tag{2.4}$$

With HMMs computing probability of observed sequence (forward-backward algorithm), inferring most likely hidden state path (Viterbi algorithm), and learning (Baum-Welch EM algorithm) are possible with efficient algorithms which made them popular for over two decades.

2.2.2 Recurrent Neural Networks

Although HMMs are being fully studied and optimized, these models suffer from some fundamental limitations. Having a one-of-H representation for the choice of

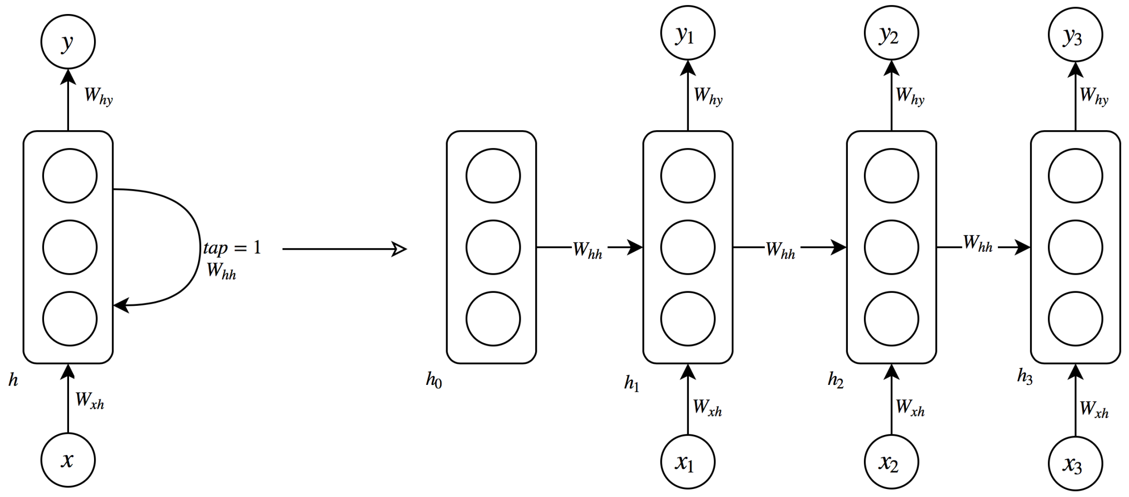


Figure 2.4 – “Unrolling” a recurrent neural network in time.

hidden state results in remembering only $log(H)$ bits of information from past. This restricts the model to keep track of more information which is a must-have for real-world complex applications. A simple example of this limitation is when generating a sentence where the model has to be consistent and coherent throughout the sequence. Remembering basic indicators, e.g. context, references, syntax, and semantic to name a few, requires flexible and powerful representation.

Recurrent Neural Networks (RNNs) have become the de facto tools for modeling complex sequential data by harnessing *distributed hidden states*. This allows these model to have exponential power over HMMs to store information in an efficient manner.

RNNs are tied closely to the concepts in deep learning. These models have the same building blocks as feed-forward neural networks (introduced in previous chapter) but with different structure. We concretely formalized the computation of an approximate function in a DAG which did not have any recurrent (feedback) connections. As shown in Fig. 2.4 we can add these connections to process variable length sequences.

Each recurrent connection is defined by its delay tap d and a node can have multiple of them. When unrolling (unfolding) the graph in time delay tap indicates that current node at time t should be connected to the same node at time $t + d$. A simple RNN has one recurrent connection with delay tap equal one. An RNN is called to have *skip-connection through time* (Bengio, 1991) when has a delay tap greater than one. Not to be confused by Clockwork RNNs (Koutnik et al., 2014) which partition the hidden states that operate in different time scales.

More formally, a standard recurrent neural network models temporal dynamics of an input sequence $x = \{x_1, x_2, \ldots, x_T\}$, by computing the hidden vector sequence $h = \{h_1, h_2, \ldots, h_T\}$ and output vector sequence $y = \{y_1, y_2, \ldots, y_T\}$ following this formula for $t = 1$ to T:

$$h_t = R(W_{xh}x_t + W_{hh}h_{t-1}) \tag{2.5}$$

$$\tilde{y}_t = Softmax(W_{hy}h_t) \tag{2.6}$$

where R could be an element-wise operator ($tanh$ for example) or any other function. For the simplicity of notation we pushed the biases b_h (in Eq. 2.5) and b_y (in Eq. 2.6) inside the weight matrices and added a unit with value 1 to the units in h vector.

Fig. 2.4 also shows *parameter sharing* in RNNs; another recurring term in deep learning literature, seen in Jordan (1986) and Elman (1990). As the name indicates, parameters are shared in calculation of different nodes of the computation graph. In this specific example weight matrices W_{xh} (from x to h), W_{hh} (from h to h), and W_{hx} (from h to x) are shared over different timesteps which gives this ability to the network structure to match to length of an arbitrary sequence and extract the same type of features from inputs in different timesteps.

It is worth noting that although the transformations between x, h, and y are parameterized by weight matrices, it is easily possible to generalize them to any differentiable function which is the case usually to use MLPs for input-to-hidden and hidden-to-output connections and proper activation function (Pascanu et al., 2013).

In addition, presence of input or output at each timestep is not necessary. Design choices or task constraints could potentially lead us toward one of the aforementioned architectures in Fig. 2.1.

These models are theoretically shown to be Turing-Complete; they are able to approximate any algorithm given enough time and neurons. Originally Siegelmann (1999) explicitly simulated a pushdown automaton with two stacks (computationally equivalent to Turing machine) by a RNN. This has an interesting consequence that there exist certain questions about the behavior of RNNs that are undecidable.

2.3 Training Recurrent Neural Networks

When dealing with an output sequence y it is common to define a cost function per timestep and minimize the sum over all timesteps in Eq. 1.39. For example in the special case of discrete output and the NLL objective function on a Multinoulli variable for each timestep given the state of the RNN we can write:

$$d_t(\tilde{y}_t, y_t) = -y_t \log \tilde{y}_t \tag{2.7}$$

$$d(\tilde{y}, y) = \frac{1}{T}\sum_{t=1}^{T} d_t(\tilde{y}_t, y_t) = -\frac{1}{T}\sum_{t=1}^{T} y_t \log \tilde{y}_t \tag{2.8}$$

In this case, think of Fig. 2.4 where each y_t is an English word from a fixed vocabulary or an specific bin from quantized amplitude of audio sample at each timestep (section 3.4.2).

By the sum rule of differentiation:

$$\theta_i \in \theta : \qquad \frac{\partial d}{\partial \theta_i} = \frac{1}{T}\sum_{t=1}^{T} \frac{\partial d_t}{\partial \theta_i} \tag{2.9}$$

Backpropagation Through Time (BPTT) is the name given to this algorithm which in essence is the same as backpropagation simply applied to an unrolled computation graph. By the increase in the length of a training sequence, we need to backpropagate through more layers. This makes the training harder and this is a common practice to split the long sequences and apply *Truncated* BPTT.

2.4 Variants of Recurrent Neural Networks

In practice, RNNs are hard to train, specially with gradient descent to model long-range dependencies (Bengio et al., 1994; Hochreiter et al., 2001; Pascanu et al., 2013). This is attributed to two main problems: *exploding* and *vanishing* gradients.

The first is referred to as large increase in the norm of the gradient during training which prevents us from training the model by causing numerical instability and contaminating the whole graph with NaN/Inf. The latter exhibits this opposite behavior; norm of the gradient exponentially fast decreases to zero which prevents the model to attend to the long-range dependencies in the sequence. Interestingly, both of which are caused by recursive use of W_{hh} and previous state h_{t-1} when computing the gradients.

Exploding gradient can be remedied mostly by gradient norm clipping (Pascanu et al., 2013) and proper initialization. On the other hand, mitigating vanishing gradient problem takes more effort. Again, initialization can help alongside with regularization or using ReLU instead of *tanh*. The most effective and popular option is to use gated variants of standard RNN.

2.4.1 Long Short-Term Memory

First proposed by Hochreiter and Schmidhuber (1997) is a popular architecture to equip RNNs with memory module. An important point to bear in mind is that these variants are just another choice for R in Eq. 2.5 to combat vanishing gradient problem by the means of gating mechanism and the dynamic of the rest of the network will stay untouched. In the following $\circ$ means element-wise multiplication.

$$i = \sigma(W^i_{xh}x_t + W^i_{hh}h_{t-1}) \tag{2.10}$$
$$o = \sigma(W^o_{xh}x_t + W^o_{hh}h_{t-1}) \tag{2.11}$$
$$f = \sigma(W^f_{xh}x_t + W^f_{hh}h_{t-1}) \tag{2.12}$$
$$g = tanh(W^g_{xh}x_t + W^g_{hh}h_{t-1}) \tag{2.13}$$
$$c_t = c_{t-1} \circ f + g \circ i \tag{2.14}$$
$$h_t = tanh(c_t) \circ o \tag{2.15}$$

Again we are hiding the biases (b^i, b^o, b^f, and b^g) for simplicity. i, o, and f are called input, output, and forget gates respectively. g is a candidate hidden state which is exactly what we have used in the standard RNN so far but with renaming the parameters. c_t is the internal memory of the architecture or memory cells that is not accessible from outside.

The intuition behind the complicated design of this architecture is that instead of using multiplication to get the new state it is less prone to vanishing gradient if we use addition. New state can be computed by adding the information on top of previous hidden states. But not everything that has been added to the state is useful or relevant to the task. Forget gate f with value between 0 and 1 is responsible to erase the memory. $W_{xh}x_t$ and $W_{hy}h_t$ in Eq. 2.5 are also regulated by input and output gates in order to let the network (by learning) decides when and how let the information should go through.

2.4.2 Gated Recurrent Units

Gated Recurrent Units (GRUs) are proposed by Chung et al. (2014) as a simpler alternative to LSTM but with the same goal. Having less parameters due to lack of output gate, GRU shows competitive results and there is no need to tune for sensitive initial bias of the forget gate (Greff et al., 2015). Following shows the calculation of R in GRU:

$$z = \sigma(W^z_{xh}x_t + W^z_{hh}h_{t-1}) \tag{2.16}$$

$$r = \sigma(W^z_{xh}x_t + W^r_{hh}h_{t-1}) \tag{2.17}$$

$$s = tanh(W^s_{xh}x_t + W^s_{hh}(h_{t-1} \circ r)) \tag{2.18}$$

$$h_t = (1 - z) \circ s + z \circ h_{t-1} \tag{2.19}$$

In GRU gates are reduced from 3 to 2, namely r reset and z update gates. There is no internal memory and when computing the output there is no second activation function.

2.4.3 Stacked Recurrent Neural Neural Networks

Considering the RNN models that has been introduced so far, a slice of the model that converts x_t to y_t is not deep. Stacked RNNs (Graves et al., 2013) or deep RNNs are simply adding layers of RNNs or their variants on top of each other so that input of top layer is the output of previous one. To prevent the vanishing gradient as a result of increase in hierarchical depth one can add skip-connections from input to the hidden units and from hidden units to output in order to help the flow of the gradient (Graves, 2013). In NLP and speech recognition tasks stacked RNNs are the dominant architecture of choice.

2.4.4 Bidirectional Recurrent Neural Networks

RNNs are considered "causal" structures that swipe the information from past to the present (Goodfellow et al., 2016). Bidirectional RNNs (BiRNNs) first introduced by Schuster and Paliwal (1997) and shown to be successful by Graves (2012). These models allow for access to look-ahead features (from future) that the prediction of targets could be dependent on them. This is basically implemented by running one RNN in forward direction and one backward (or equivalently, feeding the data in reverse) and computing the output based on the past (from forward RNN) and future (from backward RNN) information.

$$\overrightarrow{h}_t = R(\overrightarrow{W}_{xh}x_t + \overrightarrow{W}_{hh}\overrightarrow{h}_{t-1}) \tag{2.20}$$

$$\overleftarrow{h}_t = R(\overleftarrow{W}_{xh}x_t + \overleftarrow{W}_{hh}\overleftarrow{h}_{t-1}) \tag{2.21}$$

$$\tilde{y}_t = Softmax(W_{hy}h_t) = Softmax(W_{hy}[\overrightarrow{h}_t; \overleftarrow{h}_t]) \tag{2.22}$$

Each of the RNN models presented here also could be assumed to be a deep RNN. BiRNNs are however not useful for generative models as there is no future information when generating solely based on past.

3 SampleRNN

3.1 Prologue

SampleRNN: An Unconditional End-to-End Neural Audio Generation Model. Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, José Sotelo, Aaron Courville, Yoshua Bengio, 5th International Conference on Learning Representations (ICLR 2017), submitted and under review. [1]

Contribution. Inspired by PixelRNN (van den Oord et al., 2016) I worked on a similar model but with real-valued data when it has been decided to borrow the idea of output quantization for Audio Generation task. Ishaan Gulrajani greatly contributed to the implementation of first prototype of the idea and multi-scale hierarchy. I re-implemented the same idea independently as a second proof of concept, for large scale experiments, and extended use cases where it was needed for large hyper-parameter search and examining or comparing the models properly. I also designed different experiments to deconstruct the model and its components, showed the importance of hierarchy in our model (as was expected in theory), and conducted human evaluation (qualitative measure) alongside with some of experiments for quantitative measures. Kundan Kumar was responsible for WaveNet (Oord et al., 2016) re-implementation (a recent competing model for the same task) and the rest of the experiments. Kundan Kumar, Rithesh Kumar, Shubham Jain, and José Sotelo also brought significant insight by working on variant/hybrid models. My supervisors, Yoshua Bengio and Aaron Courville, were the leads of the whole project with invaluable support and guidance. First three authors were mostly involved in preparation of the first draft of the paper while the review and revision process was an extensive collaborative effort.

1. At the time of writing reviews have been favorable with ratings 9, 8, and 8 out of 10.

3.2 Abstract

In this paper we propose a novel model for unconditional audio generation based on generating one audio sample at a time. We show that our model, which profits from combining memory-less modules, namely autoregressive multilayer perceptrons, and stateful recurrent neural networks in a hierarchical structure is able to capture underlying sources of variations in the temporal sequences over very long time spans, on three datasets of different nature. Human evaluation on the generated samples indicate that our model is preferred over competing models. We also show how each component of the model contributes to the exhibited performance.

3.3 Introduction

Audio generation is a challenging task at the core of many problems of interest, such as text-to-speech synthesis, music synthesis and voice conversion. The particular difficulty of audio generation is that there is often a very large discrepancy between the dimensionality of the the raw audio signal and that of the effective semantic-level signal. Consider the task of speech synthesis, where we are typically interested in generating utterances corresponding to full sentences. Even at a relatively low sample rate of 16kHz, on average we will have 6,000 samples per word generated.[2]

Traditionally, the high-dimensionality of raw audio signal is dealt with by first compressing it into spectral or hand-engineered features and defining the generative model over these features. However, when the generated signal is eventually decompressed into audio waveforms, the sample quality is often degraded and requires extensive domain-expert corrective measures. This results in complicated signal processing pipelines that are to adapt to new tasks or domains. Here we propose a step in the direction of replacing these handcrafted systems.

In this work, we investigate the use of recurrent neural networks (RNNs) to model the dependencies in audio data. We believe RNNs are well suited as they

2. Statistics based on the average speaking rate of a set of TED talk speakers sixminutes.dlugan.com/speaking-rate

have been designed and are suited solutions for these tasks (see Graves (2013), Karpathy (2015), and Siegelmann (1999)). However, in practice it is a known problem of these models to not scale well at such a high temporal resolution as is found when generating acoustic signals one sample at a time, e.g., 16000 times per second. This is one of the reasons that Oord et al. (2016) profits from other neural modules such as one presented by Yu and Koltun (2015) to show extremely good performance.

In this paper, an end-to-end unconditional audio synthesis model for raw waveforms is presented while keeping all the computations tractable.[3] Since our model has different modules operating at different clock-rates (which is in contrast to WaveNet), we have the flexibility in allocating the amount of computational resources in modeling different levels of abstraction. In particular, we can potentially allocate very limited resource to the module responsible for sample level alignments operating at the clock-rate equivalent to sample-rate of the audio, while allocating more resources in modeling dependencies which vary very slowly in audio, for example identity of phoneme being spoken. This advantage makes our model arbitrarily flexible in handling sequential dependencies at multiple levels of abstraction.

Hence, our contribution is threefold:

1. We present a novel method that utilizes RNNs at different scales to model longer term dependencies in audio waveforms while training on short sequences which results in memory efficiency during training.
2. We extensively explore and compare variants of models achieving the above effect.
3. We study and empirically evaluate the impact of different components of our model on three audio datasets. Human evaluation also has been conducted to test these generative models.

3.4 SampleRNN Model

In this paper we propose SampleRNN (shown in Fig. 3.1), a density model for audio waveforms. SampleRNN models the probability of a sequence of waveform

3. Code github.com/soroushmehr/sampleRNN_ICLR2017 and samples soundcloud.com/samplernn/sets

samples $X = \{x_1, x_2, \ldots, x_T\}$ (a random variable over input data sequences) as the product of the probabilities of each sample conditioned on all previous samples:

$$p(X) = \prod_{i=0}^{T-1} p(x_{i+1}|x_1, \ldots, x_i) \tag{3.1}$$

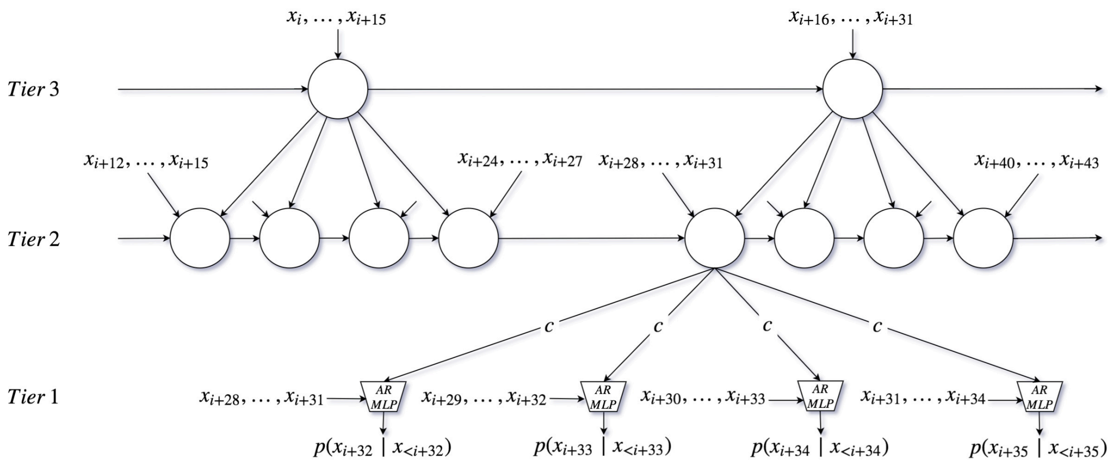
RNNs are commonly used to model sequential data which can be formulated as:

$$h_t = RNNCell(h_{t-1}, x_{i=t}) \tag{3.2}$$

$$p(x_{i+1}|x_1, \ldots, x_i) = Softmax(MLP(h_t)) \tag{3.3}$$

with $RNNCell$ being one of the known memory cells (Section 3.5). However, raw audio signals are challenging to model because they contain structure at very different scales: correlations exist between neighboring samples as well as between ones thousands of samples apart.

SampleRNN helps to address this challenge by using a hierarchy of modules, each operating at a different temporal resolution. The lowest module processes individual samples, and each higher module operates on an increasingly longer timescale and a lower temporal resolution. Each module conditions the module below it, with the lowest module outputting sample-level predictions. The entire hierarchy is trained jointly end-to-end by backpropagation.



$x_i, \ldots, x_{i+15}$
$x_{i+16}, \ldots, x_{i+31}$
Tier 3
$x_{i+12}, \ldots, x_{i+15}$
$x_{i+24}, \ldots, x_{i+27}$
$x_{i+28}, \ldots, x_{i+31}$
$x_{i+40}, \ldots, x_{i+43}$
Tier 2
c
c
c
c
Tier 1
$x_{i+28}, \ldots, x_{i+31}$
AR MLP
$x_{i+29}, \ldots, x_{i+32}$
AR MLP
$x_{i+30}, \ldots, x_{i+33}$
AR MLP
$x_{i+31}, \ldots, x_{i+34}$
AR MLP
$p(x_{i+32} \mid x_{<i+32})$
$p(x_{i+33} \mid x_{<i+33})$
$p(x_{i+34} \mid x_{<i+34})$
$p(x_{i+35} \mid x_{<i+35})$

Figure 3.1 – Snapshot of the unrolled model at timestep i with $K = 3$ tiers. As a simplification only one RNN and up-sampling ratio $r = 4$ is used for all tiers.

3.4.1 Frame-level Modules

Rather than operating on individual samples, the higher-level modules in SampleRNN operate on *non-overlapping frames* of $FS^{(k)}$ ("Frame Size") samples at the k^{th} level up in the hierarchy at a time (frames denoted by $f^{(k)}$). Each frame-level module is a deep RNN which summarizes the history of its inputs into a conditioning vector for the next module downward.

The variable number of frames we condition upon up to timestep $t-1$ is expressed by a fixed length hidden state or memory $h_t^{(k)}$ where t is related to clock rate at that tier. The RNN makes a memory update at timestep t as a function of the previous memory $h_{t-1}^{(k)}$ and an input. This input for top tier $k = K$ is simply the input frame. For intermediate tiers ($1 < k < K$) this input is a linear combination of conditioning vector from higher tier and current input frame. See Eqs. 3.4–3.5.

Because different modules operate at different temporal resolutions, we need to upsample each vector c at the output of a module into a series of $r^{(k)}$ vectors (where $r^{(k)}$ is the ratio between the temporal resolutions of the modules) before feeding it into the input of the next module downward (Eq. 3.6). We do this with a set of $r^{(k)}$ separate linear projections.

Here we are formalizing the frame-level module in tier k. Note that following equations are exclusive to tier k and timestep t for that specific tier. To increase the readability, unless necessary superscript (k) is not shown for t, $inp^{(k)}$, $W_x^{(k)}$, $h^{(k)}$, $RNNCell^{(k)}$, $W_j^{(k)}$, and $r^{(k)}$.

$$inp_t = \begin{cases} W_x f_t^{(k)} + c_t^{(k+1)}; & 1 < k < K \\ f_t^{(k=K)}; & k = K \end{cases} \tag{3.4}$$

$$h_t = RNNCell(h_{t-1}, inp_t) \tag{3.5}$$

$$c_{(t-1)*r+j}^{(k)} = W_j h_t; \qquad 1 \leq j \leq r \tag{3.6}$$

Our approach of upsampling with $r^{(k)}$ linear projections is exactly equivalent to upsampling by adding zeros and then applying a linear convolution. This is sometimes called "perforated" upsampling in the context of convolutional neural networks (CNNs). It was first demonstrated to work well in Dosovitskiy et al. (2016) and is a fairly common upsampling technique.

3.4.2 Sample-level Module

The lowest module (tier $k = 1$; Eqs. 3.7–3.9) in the SampleRNN hierarchy outputs a distribution over a sample x_{i+1}, conditioned on the $FS^{(1)}$ preceding samples as well as a vector $c_i^{(k=2)}$ from the next higher module which encodes information about the sequence prior to that frame. As $FS^{(1)}$ is usually a small value and correlation in nearby samples are easy to model by a simple memoryless module, we implement it with a multilayer perceptron (MLP) rather than RNN which slightly speeds up the training. Assuming e_i represents x_i after passing through embedding layer (section 3.4.2), conditional distribution can be achieved by:

$$f_{i-1}^{(1)} = flatten([e_{i-FS^{(1)}}, \ldots, e_{i-1}]) \tag{3.7}$$

$$f_i^{(1)} = flatten([e_{i-FS^{(1)}+1}, \ldots, e_i])$$

$$inp_i^{(1)} = W_x^{(1)} f_i^{(1)} + c_i^{(2)} \tag{3.8}$$

$$p(x_{i+1}|x_1, \ldots, x_i) = Softmax(MLP(inp_i^{(1)})) \tag{3.9}$$

We use a Softmax because we found that better results were obtained by discretizing the audio signals and outputting a Multinoulli distribution rather than using a Gaussian or Gaussian mixture to represent the conditional density of the original real-valued signal. When processing an audio sequence, the MLP is convolved over the sequence, processing each window of $FS^{(1)}$ samples and predicting the next sample. At generation time, the MLP is run repeatedly to generate one sample at a time. Table 3.1 shows a considerable gap between the baseline model RNN and this model, suggesting that the proposed hierarchically structured architecture of SampleRNN makes a big difference.

Output Quantization

Following van den Oord et al. (2016), the sample-level module models its output as a q-way discrete distribution over possible quantized values of x_i (that is, the output layer of the MLP is a q-way Softmax).

To demonstrate the importance of a discrete output distribution, we apply the same architecture on real-valued data by replacing the q-way Softmax with a Gaussian Mixture Models (GMM) output distribution. Table 3.2 shows that our model outperforms an RNN baseline even when both models use real-valued outputs. However, samples from the real-valued model are almost indistinguishable from random noise.

In this work we use linear quantization with $q = 256$, corresponding to a per-sample bit depth of 8. Unintuitively, we realized that even linearly decreasing the bit depth (resolution of each audio sample) from 16 to 8 can ease the optimization procedure while generated samples still have reasonable quality and are artifact-free.

In addition, early on we noticed that the model can achieve better performance and generation quality when we *embed the quantized input values* before passing them through the sample-level MLP (see Table 3.4). The embedding steps maps each of the q discrete values to a real-valued vector embedding. However, real-valued raw samples are still used as input to the higher modules.

Conditionally Independent Sample Outputs

To demonstrate the importance of a sample-level autoregressive module, we try replacing it with "Multi-Softmax" (see Table 3.4), where the prediction of each sample x_i depends only on the conditioning vector c from Eq. 3.9. In this configuration, the model outputs an entire *frame* of $FS^{(1)}$ samples at a time, modeling all samples in a frame as conditionally independent of each other. We find that this Multi-Softmax model (which lacks a sample-level autoregressive module) scores significantly worse in terms of log-likelihood and fails to generate convincing samples. This suggests that modeling the joint distribution of the acoustic samples inside each frame is very important in order to obtain good acoustic generation. We found this to be true even when the frame size is reduced, with best results always with a frame size of 1, i.e., generating only one acoustic sample at a time.

3.4.3 Truncated BPTT

Training recurrent neural networks on long sequences can be very computationally expensive. Oord et al. (2016) avoid this problem by using a stack of dilated convolutions instead of any recurrent connections. However, when they can be trained efficiently, recurrent networks have been shown to be very powerful and expressive sequence models. We enable efficient training of our recurrent model using *truncated backpropagation through time*, splitting each sequence into short subsequences and propagating gradients only to the beginning of each subsequence. We experiment with different subsequence lengths and demonstrate that we are able to train our networks, which model very long-term dependencies, despite backpropagating through relatively short subsequences.

Table 3.3 shows that by increasing the subsequence length, performance substantially increases alongside with train-time memory usage and convergence time. Yet it is noteworthy that our best models have been trained on subsequences of length 512, which corresponds to 32 milliseconds, a small fraction of the length of a single a phoneme of human speech while generated samples exhibit longer word-like structures.

Despite the aforementioned fact, this generative model can mimic the existing long-term structure of the data which results in more natural and coherent samples that is preferred by human listeners. (More on this in Sections 3.5.2–3.5.3.) This is due to the fast updates from TBPTT and specialized frame-level modules (Section 3.4.1) with top tiers designed to model a lower resolution of signal while leaving the process of filling the details to lower tiers.

3.5 Experiments and Results

In this section we are introducing three datasets which have been chosen to evaluate the proposed architecture for modeling raw acoustic sequences. The description of each dataset and their preprocessing is as follows:

Blizzard which is a dataset presented by Prahallad et al. (2013) for speech synthesis task, contains 315 hours of a single female voice actor in English; however, for our experiments we are using only 20.5 hours. The training/-validation/test split is 86%-7%-7%.

Onomatopoeia[4], a relatively small dataset with 6,738 sequences adding up to 3.5 hours, is human vocal sounds like grunting, screaming, panting, heavy breathing, and coughing. Diversity of sound type and the fact that these sounds were recorded from 51 actors and many categories makes it a

4. Courtesy of Ubisoft

challenging task. To add to that, this data is extremely unbalanced. The training/validation/test split is 92%-4%-4%.

Music dataset is the collection of all 32 Beethoven's piano sonatas publicly available on `https://archive.org/` amounting to 10 hours of non-vocal audio. The training/validation/test split is 88%-6%-6%.

See Fig. 3.2 for a visual demonstration of examples from datasets and generated samples. For all the datasets we are using a 16 kHz sample rate and 16 bit depth. For the Blizzard and Music datasets, preprocessing simply amounts to chunking the long audio files into 8 seconds long sequences on which we will perform truncated backpropagation through time. Each sequence in the Onomatopoeia dataset is few seconds long, ranging from 1 to 11 seconds. To train the models on this dataset, zero-padding has been applied to make all the sequences in a mini-batch have the same length and corresponding cost values (for the predictions over the added 0s) would be ignored when computing the gradients.

We particularly explored two gated variants of RNNs—Gated Recurrent Units (GRUs) (Chung et al., 2014) and Long Short Term Memory Units (LSTMs) (Hochreiter and Schmidhuber, 1997). For the case of LSTMs, the forget gate bias is initialized with a large positive value of 3, as recommended by Zaremba (2015) and Gers (2001), which has been shown to be beneficial for learning long-term dependencies.

As for models that take real-valued input, e.g. the RNN-GMM and SampleRNN-GMM (with 4 components), normalization is applied per audio sample with the global mean and standard deviation obtained from the train split. For most of our experiments where the model demands discrete input, binning was applied per audio sample.

All the models have been trained with teacher forcing and stochastic gradient decent (mini-batch size 128) to minimize the Negative Log-Likelihood (NLL) in bits per dimension (per audio sample). Gradients were hard-clipped to remain in [-1, 1] range. Update rules from the Adam optimizer (Kingma and Ba, 2014) ($\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 1e{-}8$) with an initial learning rate of 0.001 was used to adjust the parameters. For training each model, random search over hyper-parameter values (Bergstra and Bengio, 2012) was conducted. The initial RNN state of all the RNN-based models was always learnable. Weight Normalization (Salimans and Kingma, 2016) has been used for all the linear layers in the model (except for the embedding layer) to accelerate the training procedure. Size of the embedding layer

Table 3.1 – Test NLL in bits for three presented datasets.

Model	Blizzard	Onomatopoeia	Music
RNN (Eq. 3.2)	1.434	2.034	1.410
WaveNet (re-impl.)	1.480	2.285	1.464
SampleRNN (2-tier)	1.392	2.026	**1.076**
SampleRNN (3-tier)	**1.387**	**1.990**	1.159

Table 3.2 – Average NLL on Blizzard test set for real-valued models.

Model	Average Test NLL
RNN-GMM	-2.415
SampleRNN-GMM (2-tier)	**-2.782**

was 256 and initialized by standard normal distribution. Orthogonal weight matrices used for hidden-to-hidden connections and other weight matrices initialized similar to He et al. (2015). In final model, we found GRU to work best (slightly better than LSTM). 1024 was the the number of hidden units for all GRUs (1 layer per tier for 3-tier and 3 layer for 2-tier model) and MLPs (3 fully connected layers with ReLU activation with output dimension being 1024 for first two layers and 256 for the final layer before softmax).

3.5.1 WaveNet Re-implementation

We implemented the WaveNet architecture as described in Oord et al. (2016). Ideally, we would have liked to replicate their model exactly but owing to missing details of architecture and hyperparameters, as well as limited compute power at our disposal, we made our own design choices so that the model would fit on a single GPU while having a receptive field of around 250 milliseconds, while having a reasonable number of updates per unit time. Although our model is very similar

Table 3.3 – Effect of subsequence length on NLL (bits per audio sample) computed on the Blizzard validation set.

Subsequence Length	32	64	128	256	512
NLL Validation	1.575	1.468	1.412	1.391	1.364

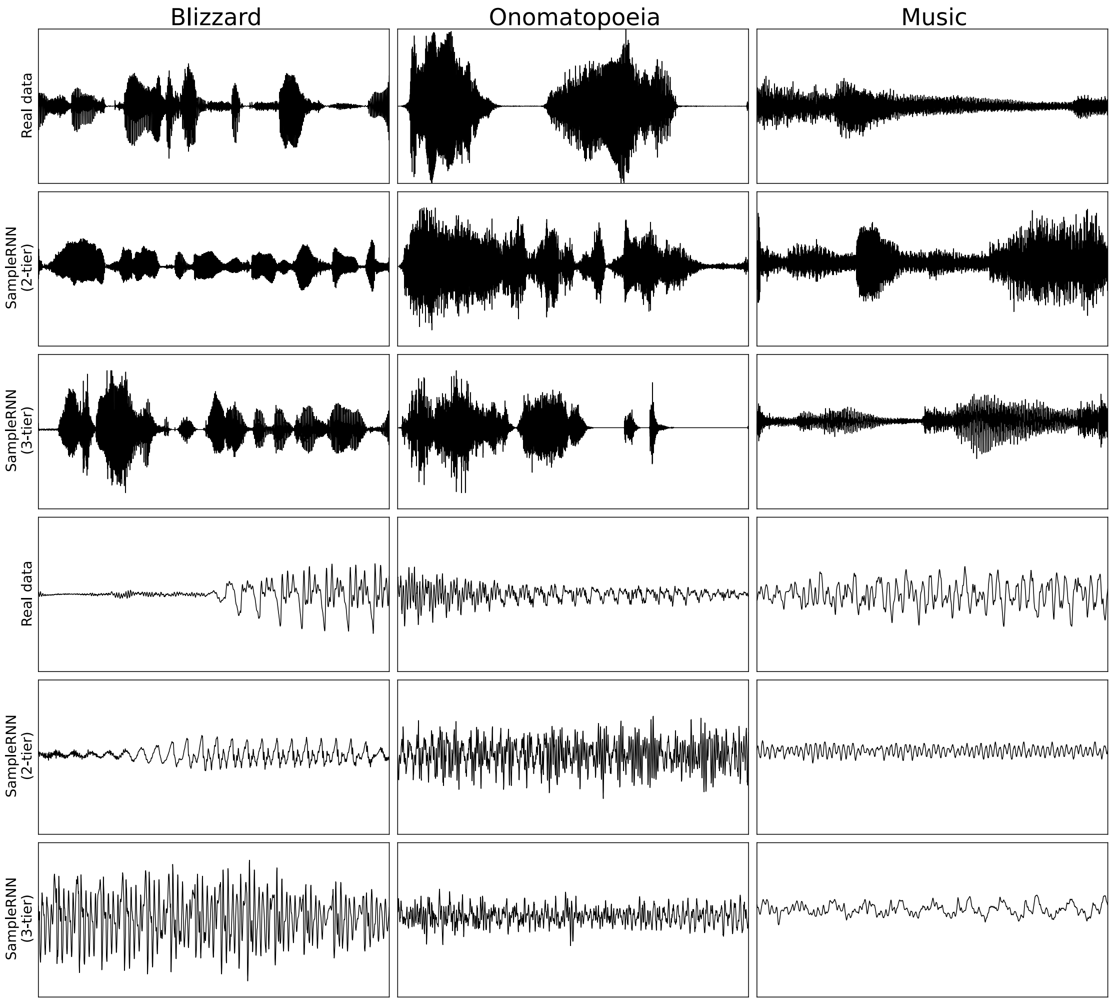


Figure 3.2 – Examples from the datasets compared to samples from our models. In the first 3 rows, 2 seconds of audio are shown. In the bottom 3 rows, 100 milliseconds of audio are shown. Rows 1 and 4 are ground truth from which one can see how the datasets look different and have complex structure in low resolution which the frame-level component of the SampleRNN is designed to capture. Samples also to some extent mimic the same global structure. At the same time, zoomed-in samples of our model shows that it can perfectly resemble the high resolution structure present in the data as well.

Table 3.4 – Test (validation) set NLL (bits per audio sample) for Blizzard. Variants of SampleRNN are provided to compare the contribution of each component in performance.

Model	**NLL Test (Validation)**
SampleRNN (2-tier)	1.392 (1.369)
Without Embedding	1.566 (1.539)
Multi-Softmax	1.685 (1.656)

to WaveNet, the design choices, e.g. number of convolution filters in each dilated convolution layer, length of target sequence to train on simultaneously (one can train with a single target with all samples in the receptive field as input or with target sequence length of size T with input of size receptive field + T - 1), batch-size, etc. might make our implementation different from what the authors have done in the original WaveNet model. Hence, we note here that although we did our best at exactly reproducing their results, there would very likely be different choice of hyper-parameters between our implementation and the one of the authors.

For our WaveNet implementation, we have used 4 dilated convolution blocks each having 10 dilated convolution layers with dilation 1, 2, 4, 8 up to 512. Hence, our network has a receptive field of 4092 acoustic samples i.e. the parameters of multinomial distribution of sample at time step t, $p(x_i) = f_\theta(x_{i-1}, x_{i-2}, \ldots x_{i-4092})$ where θ is model parameters. We train on target sequence length of 1600 and use batch size of 8. Each dilated convolution filter has size 2 and the number of output channels is 64 for each dilated convolutional layer (128 filters in total due to gated non-linearity). We trained this model using Adam optimizer with a fixed global learning rate of 0.001 for Blizzard dataset and 0.0001 for Onomatopoeia and Music datasets. We trained these models for about one week on a GeForce GTX TITAN X. We dropped the learning rate in the Blizzard experiment to 0.0001 after around 3 days of training.

3.5.2 Human Evaluation

Apart from reporting NLL, we conducted AB preference tests for random samples from four models trained on the Blizzard dataset. For unconditional generation of speech which at best sounds like mumbling, this type of test is the one which is more suited. Competing models were the RNN, SampleRNN (2-tier), SampleRNN (3-tier), and our implementation of WaveNet. The rest of the models were excluded as the quality of samples were definitely lower and also to keep the number of pair comparison tests manageable. We will release the samples that have been used in this test too.

All the samples were set to have the same volume. Every user is then shown a set of twenty pairs of samples with one random pair at a time. Each pair had samples from two different models. The human evaluator is asked to listen to the

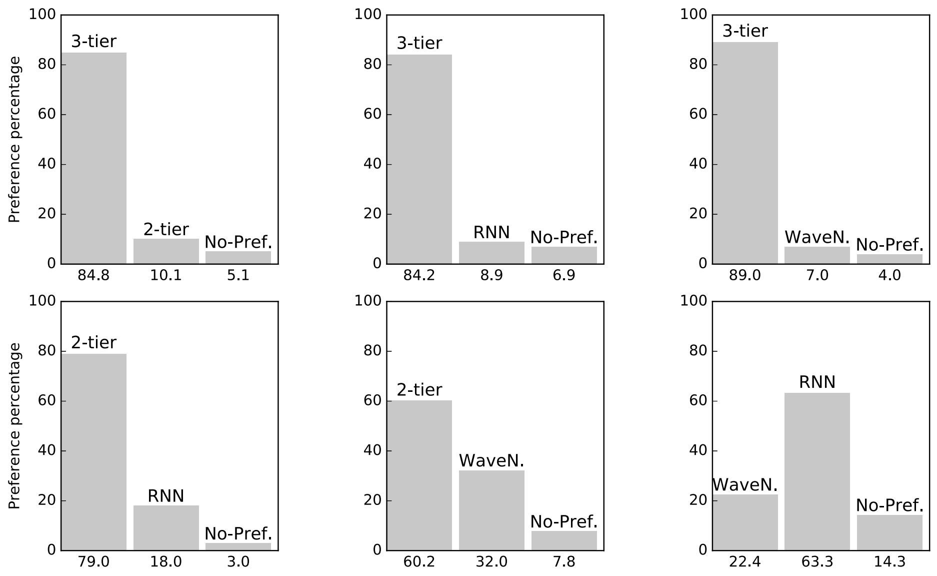


Figure 3.3 – Pairwise comparison of 4 best models based on the votes from listeners conducted on samples generated from models trained on Blizzard dataset.

samples and had the option of choosing between the two model or choosing not to prefer any of them. Hence, we have a quantification of preference between every pair of models. We used the online tool made publicly available by Jillings et al. (2015).

Results in Fig. 3.3 clearly points out that SampleRNN (3-tier) is a winner by a huge margin in terms of preference by human raters, then SampleRNN (2-tier) and afterward two other models, which matches with the performance comparison in Table 3.1.

The same evaluation was conducted for Music dataset except for an additional filtering process of samples. Specific to only this dataset, we observed that a batch of generated samples from competing models (this time restricted to RNN, SampleRNN (2-tier), and SampleRNN (3-tier)) were either music-like or random noise. For all these models we only considered random samples that were not random noise. Fig. 3.4 is dedicated to result of human evaluation on Music dataset.

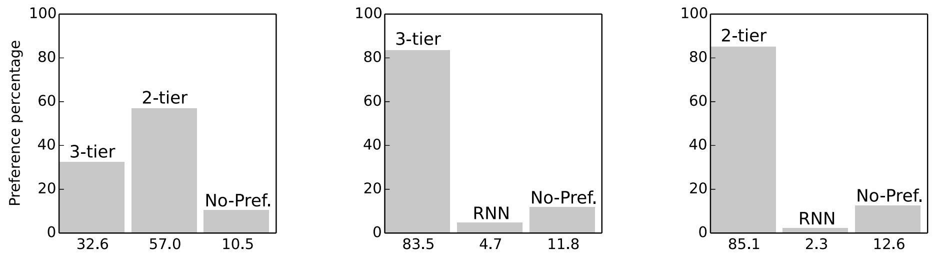


Figure 3.4 – Pairwise comparison of 3 best models based on the votes from listeners conducted on samples generated from models trained on Music dataset.

3.5.3 Quantifying Information Retention

For the last experiment we are interested in measuring the memory span of the model. We trained our model, SampleRNN (3-tier), with best hyper-parameters on a dataset of 2 speakers reading audio books, one male and one female, respectively, with mean fundamental frequency of 125.3 and 201.8Hz. Each speaker has roughly 10 hours of audio in the dataset that has been preprocessed similar to Blizzard. We observed that it learned to stay consistent generating samples from the same speaker without having any knowledge about the the speaker ID or any other conditioning information. This effect is more apparent here in comparison to the unbalanced Onomatopoeia that sometimes mixes two different categories of sounds.

Another experiment was conducted to test the effect of memory and study the effective memory horizon. We inject 1 second of silence in the middle of sampling procedure in order to see if it will remember to generate from the same speaker or not. Initially when sampling we let the model generate 2 seconds of audio as it normally do. From 2 to 3 seconds instead of feeding back the generated sample at that timestep a silent token (zero amplitude) would be fed. From 3 to 5 seconds again we sample normally; feeding back the generated token.

We did classification based on mean fundamental frequency of speakers for the first and last 2 seconds. In 83% of samples SampleRNN generated from the same person in two separate segments. This is in contrast to a model with fixed past window like WaveNet where injecting 16000 silent tokens (3.3 times the receptive field size) is equivalent to generating from scratch which has 50% chance (assuming each 2-second segment is coherent and not a mixed sound of two speakers).

3.6 Related Work

Our work is related to earlier work on auto-regressive multi-layer neural networks, starting with Bengio and Bengio (1999), then NADE (Larochelle and Murray, 2011) and more recently PixelRNN (van den Oord et al., 2016). Similar to how they tractably model joint distribution over units of the data (e.g. words in sentences, pixels in images, etc.) through an auto-regressive decomposition, we transform the joint distribution of acoustic samples using Eq. 3.1.

The idea of having part of the model running at different clock rates is related to multi-scale RNNs (Schmidhuber, 1992; El Hihi and Bengio, 1995; Koutnik et al., 2014; Sordoni et al., 2015; Serban et al., 2016).

Chung et al. (2015) also attempt to model raw audio waveforms which is in contrast to traditional approaches which use spectral features as in Tokuda et al. (2013), Bertrand et al. (2008), and Lee et al. (2009).

Our work is closely related to WaveNet (Oord et al., 2016), which is why we have made the above comparisons, and makes it interesting to compare the effect of adding higher-level RNN stages working at a low resolution. Similar to this work, our models generate one acoustic sample at a time conditioned on all previously generated samples. We also share the preprocessing step of quantizing the acoustics into bins. Unlike this model, we have different modules in our models running at different clock-rates. In contrast to WaveNets, we mitigate the problem of long-term dependency with hierarchical structure and using stateful RNNs, i.e. we will always propagate hidden states to the next training sequence although the gradient of the loss will not take into account the samples in previous training sequence.

4 Conclusion

In this thesis we propose a novel model that can address unconditional audio generation in the raw acoustic domain, which typically has been done until recently with hand-crafted features. We are able to show that a hierarchy of time scales and frequent updates will help to overcome the problem of modeling extremely high-resolution temporal data. That allows us, for this particular application, to learn the data manifold directly from audio samples. We show that this model can generalize well and generate samples on three datasets that are different in nature. We also show that the samples generated by this model are preferred by human raters.

Success in this application, with a general-purpose solution as proposed here, opens up room for more improvement when specific domain knowledge is applied. This method, however, proposed with audio generation application in mind, can easily be adapted to other tasks that require learning the representation of sequential data with high temporal resolution and long-range complex structure.

A A model variant

SampleRNN-WaveNet hybrid model has two modules operating at two different clock-rate. The slower clock-rate module (frame-level module) sees one frame (each of which has size *FS*) at a time while the faster clock-rate component(sample-level component) sees one acoustic sample at a time i.e. the ratio of clock-rates for these two modules would be the size of a single frame. Number of sequential steps for frame-level component would be *FS* times lower. We repeat the output of each step of frame-level component *FS* times so that number of time-steps for output of both the components match. The output of both these modules are concatenated for every time-step which is further operated by non-linearities for every time-step independently before generating the final output.

In our experiments, we kept size of a single frame (*FS*) to be 128. We tried two variants of this model: 1. fully convolutional WaveNet and 2. RNN-WaveNet. In fully convolutional WaveNet, both modules described above are implemented using dilated convolutions as described in original WaveNet model. In RNN-WaveNet, we use high capacity RNN in the frame-level module to model the dependency between frames. The sample-level WaveNet in RNN-WaveNet has receptive field of size 509 samples from the past.

Although these models are designed with the intention of combining the two models to harness their best features, preliminary experiments show that this variant is not meeting our expectations at the moment which directs us to a possible future work.

Bibliography

Bengio, Y. (1991). Artificial neural networks and their application to sequence recognition.

Bengio, Y. and S. Bengio (1999). Modeling high-dimensional discrete data with multi-layer neural networks. In *NIPS*, Volume 99, pp. 400–406.

Bengio, Y., P. Simard, and P. Frasconi (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks 5*(2), 157–166.

Bergstra, J. and Y. Bengio (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*(Feb), 281–305.

Bertrand, A., K. Demuynck, V. Stouten, et al. (2008). Unsupervised learning of auditory filter banks using non-negative matrix factorisation. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 4713–4716. IEEE.

Bishop, C. M. (1994). Mixture density networks.

Bridle, J. S. (1990). Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. pp. 211–217. Morgan Kaufmann.

Chung, J., C. Gulcehre, K. Cho, and Y. Bengio (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Chung, J., K. Kastner, L. Dinh, K. Goel, A. C. Courville, and Y. Bengio (2015). A recurrent latent variable model for sequential data. In *Advances in neural information processing systems*, pp. 2980–2988.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*(4), 303–314.

Dosovitskiy, A., J. Springenberg, M. Tatarchenko, and T. Brox (2016). Learning to generate chairs, tables and cars with convolutional networks.

Duchi, J., E. Hazan, and Y. Singer (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*(Jul), 2121–2159.

Dugas, C., Y. Bengio, F. Bélisle, C. Nadeau, and R. Garcia (2001). Incorporating second-order functional knowledge for better option pricing. *Advances in neural information processing systems*, 472–478.

El Hihi, S. and Y. Bengio (1995). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS*, Volume 400, pp. 409. Citeseer.

Elman, J. L. (1990). Finding structure in time. *Cognitive science 14*(2), 179–211.

Gers, F. (2001). *Long short-term memory in recurrent neural networks.* Ph. D. thesis, Universität Hannover.

Glorot, X. and Y. Bengio (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, Volume 9, pp. 249–256.

Glorot, X., A. Bordes, and Y. Bengio (2011). Deep sparse rectifier neural networks. In *Aistats*, Volume 15, pp. 275.

Goodfellow, I., Y. Bengio, and A. Courville (2016). Deep learning. Book in preparation for MIT Press.

Graves, A. (2012). Neural networks. In *Supervised Sequence Labelling with Recurrent Neural Networks*, pp. 15–35. Springer.

Graves, A. (2013). Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*.

Graves, A., A.-r. Mohamed, and G. Hinton (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pp. 6645–6649. IEEE.

Greff, K., R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber (2015). Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*.

Hahnloser, R. H., R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung (2000). Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature 405*(6789), 947–951.

Hahnloser, R. H., H. S. Seung, and J.-J. Slotine (2003). Permitted and forbidden sets in symmetric threshold-linear networks. *Neural computation 15*(3), 621–638.

He, K., X. Zhang, S. Ren, and J. Sun (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1026–1034.

Hochreiter, S., Y. Bengio, P. Frasconi, and J. Schmidhuber (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural computation 9*(8), 1735–1780.

Hornik, K., M. Stinchcombe, and H. White (1989). Multilayer feedforward networks are universal approximators. *Neural networks 2*(5), 359–366.

Jillings, N., D. Moffat, B. De Man, and J. D. Reiss (2015, July). Web Audio Evaluation Tool: A browser-based listening test environment. In *12th Sound and Music Computing Conference*.

Jordan, M. (1986). Serial order: a parallel distributed processing approach. technical report, june 1985-march 1986. Technical report, California Univ., San Diego, La Jolla (USA). Inst. for Cognitive Science.

Karpathy, A. (2015). The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*.

Kingma, D. and J. Ba (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koutnik, J., K. Greff, F. Gomez, and J. Schmidhuber (2014). A clockwork rnn. *arXiv preprint arXiv:1402.3511*.

Larochelle, H. and I. Murray (2011). The neural autoregressive distribution estimator. In *AISTATS*, Volume 1, pp. 2.

Lee, H., P. Pham, Y. Largman, and A. Y. Ng (2009). Unsupervised feature learning for audio classification using convolutional deep belief networks. In *Advances in neural information processing systems*, pp. 1096–1104.

Maas, A. L., A. Y. Hannun, and A. Y. Ng (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, Volume 30.

McCulloch, W. S. and W. Pitts (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics 5*(4), 115–133.

Montufar, G. F., R. Pascanu, K. Cho, and Y. Bengio (2014). On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pp. 2924–2932.

Nair, V. and G. E. Hinton (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814.

Oord, A. v. d., S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Pascanu, R., C. Gulcehre, K. Cho, and Y. Bengio (2013). How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*.

Pascanu, R., T. Mikolov, and Y. Bengio (2013). On the difficulty of training recurrent neural networks. *ICML (3) 28*, 1310–1318.

Prahallad, K., A. Vadapalli, N. Elluru, G. Mantena, B. Pulugundla, P. Bhaskararao, H. Murthy, S. King, V. Karaiskos, and A. Black (2013). The blizzard challenge 2013–indian language task. In *Blizzard Challenge Workshop 2013*.

Robbins, H. and S. Monro (1951). A stochastic approximation method. *The annals of mathematical statistics*, 400–407.

Salimans, T. and D. P. Kingma (2016). Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *arXiv preprint arXiv:1602.07868*.

Schmidhuber, J. (1992). Learning complex, extended sequences using the principle of history compression. *Neural Computation 4*(2), 234–242.

Schuster, M. and K. K. Paliwal (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing 45*(11), 2673–2681.

Serban, I. V., A. Sordoni, Y. Bengio, A. Courville, and J. Pineau (2016). Building end-to-end dialogue systems using generative hierarchical neural network models. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI-16).*

Siegelmann, H. T. (1999). Computation beyond the turing limit. In *Neural Networks and Analog Computation*, pp. 153–164. Springer.

Sordoni, A., Y. Bengio, H. Vahabi, C. Lioma, J. Grue Simonsen, and J.-Y. Nie (2015). A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, pp. 553–562. ACM.

Tokuda, K., Y. Nankaku, T. Toda, H. Zen, J. Yamagishi, and K. Oura (2013). Speech synthesis based on hidden markov models. *Proceedings of the IEEE 101*(5), 1234–1252.

van den Oord, A., N. Kalchbrenner, and K. Kavukcuoglu (2016). Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*.

Vapnik, V. (2013). *The nature of statistical learning theory.* Springer Science & Business Media.

Yu, F. and V. Koltun (2015). Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*.

Zaremba, W. (2015). An empirical exploration of recurrent network architectures.