Conditional Computation in Deep and Recurrent Neural Networks

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Andrew Scott Davis

August 2016

Acknowledgements

First and foremost, I would like to thank my lovely fiancée Michelle. Without her incomparable support and tremendous ability to keep things in focus, the work here simply would not have happened. Great thanks go to my parents Jim and Patsy, who on top of all of their incredible parental support, also listened to me go on and on about my work, allowing me to hone my abilities to communicate with people who haven't dedicated their lives to studying machine learning. I would also like to thank my advisor Dr. Itamar Arel for his guidance throughout the years, as well as the amazing opportunities he gave me to apply machine learning to practical problems very early on. Thanks to all the MIL members past and present (both Bens, Stephen, Derek, Bobby, Tom, Aaron) – we had a lot of good conversations and talked about a lot of really neat ideas. Another thanks to my data science buddies (Mike, Sharon, John, Jay, Mahdi) and chief data science buddy Matt – I truly appreciate you all for giving me the freedom to complete this work.

Abstract

Recently, deep learning models such as convolutional and recurrent neural networks have displaced state-of-the-art techniques in a variety of application domains. While the computationally heavy process of training is usually conducted on powerful graphics processing units (GPUs) distributed in large computing clusters, the resulting models can still be somewhat heavy, making deployment in resource-constrained environments potentially problematic. This work is concerned with the idea of conditional computation, where the model is given the capability to learn how to avoid computing parts of the graph. This allows for models where the number of parameters (and in a sense, the model's capacity to learn) can grow at a faster rate than the computation that is required to propagate information through the graph. Two cases of conditional computation are explored – in the feed forward case, a technique is developed that trades off accuracy for potential computational benefits, and in the recurrent case, techniques that yield practical speed benefits on a language modeling task are demonstrated. Given the rapidly expanding domain of problems where deep learning proves useful, the work presented here can help enable the future scalability requirements of deploying trained models.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, a resurgence of interest in neural networks has led to the redefinition of state-of-the-art in many fields, such as computer vision (Krizhevsky et al., 2012) (Russakovsky et al., 2014), language modeling (Mikolov et al., 2011), and speech recognition (Mohamed et al., 2011)(Graves et al., 2013). While these neural networks are based on the same kinds of neural networks pioneered in the 1980s and 1990s, there are several reasons why interest has been rekindled. First, the collection and storage of massive datasets has become commonplace. While kernelized and distance-based methods such as support vector machines and k-nearest neighbors tend to have a complexity that grows quadratically with respect to the number of samples in the training set, the computational complexity and memory requirements of neural networks tend to grow linearly. While there are methods for alleviating the quadratic growth in computational complexity (Kumar et al., 2009), neural networks do not require such approximation methods to scale to very large datasets.

Second, computational power has increased substantially in recent years, especially with the increasing utility of using GPUs (graphics processing units) as massively parallel processing units. The computationally heavy building blocks of modern neural networks such as matrix-matrix multiplications and convolutions map very well to GPU hardware, often achieving a considerable percentage of the GPU's

peak computational capacity (Nath et al., 2010) and an order of magnitude decrease in runtime over code run on a CPU for some workloads. Advances in distributed computing have allowed for the training of massive neural networks on very large datasets (Le et al., 2012)(Coates et al., 2013), which would have been all but impossible on smaller computer clusters.

Third, theoretical and empirical understanding of training neural networks, specifically deep and recurrent networks, has expanded greatly. Prior to (Hinton et al., 2006) and (Bengio et al., 2007), training fully-connected neural networks deeper than two hidden layers was widely viewed as impractical and unnecessary, and other models with better theoretical guarantees and training algorithms such as random forests and support vector machines were generally preferred. However, advances in unsupervised pre-training (e.g., stacked autoencoders and restricted Boltzmann machines) led to techniques that allowed modestly deep neural networks to be trained. The investigation of non-saturating nonlinearities (rectified linear, maxout, ℓ_p, etc.) have led to models that converge faster. Advances in optimization (e.g., Nesterov's accelerated momentum, parameter-wise setting of learning rates, gradient clipping) and a better understanding of weight initialization have allowed for faster convergence and better search for local optima, allowing the training of complex models with solution spaces that are difficult to optimize over.

As high-end GPUs and distributed training algorithms and infrastructure become faster and more efficient, the trend tends to favor larger models with more hidden units and more hidden layers. However, these models are in some cases deployed on resource constrained environments such as low-power CPUs that are not becoming faster at as great of a pace. For these applications, it is necessary to develop methods that can strike a balance between speed and accuracy. One such method, known as conditional computation (Bengio, 2013), draws inspiration from algorithms such as decision trees. Decision trees have the notable attribute that they do not traverse the entire set of parameters in order to compute the function's output. In contrast, in neural networks, the output is a function of every parameter in the model. By reformulating neural

networks in ways that don't require every parameter in order to compute the output, the ratio between the capacity of the model and the computation required to compute it increases, allowing for models with more capacity at less computational cost. In this work, the problem of conditional computation is addressed in both feed-forward neural networks as well as recurrent neural networks.

In Chapter 2, the relevant literature on feed-forward as well as recurrent neural networks is reviewed, including the optimization and architectural advances required to efficiently train them. Next, conditional computation along with the prior art is reviewed. The literature review concludes with some additional relevant information about BLAS to introduce vocabulary and practical considerations for implementing conditional computation. In Chapter 3, an effort of implementing conditional computation in feed-forward neural networks is described, using a low-rank approximation of the weight matrices of the neural network as a way to predict which activations of a rectified linear unit will be positive or zero, allowing the model to estimate in advance which hidden activations should be calculated. In Chapter 4, it is demonstrated how conditional computation can be applied to recurrent neural networks – in particular, by inducing sparsity in the output gating of a gated recurrent unit (GRU), the computation of elements of the hidden state update can be skipped and simply pass through the previous value. This approach demonstrates practical speedups with the deployment of the model on a CPU. Chapter 5 concludes the dissertation with a summary of contributions, as well as future research directions of conditional computation.

Chapter 2

Background and Literature Review

2.1 Machine Learning

Machine learning is a field of study concerned with the development of algorithms that can in some sense learn from data (Russell et al., 1995). Machine learning is generally separated into three subfields - supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, an algorithm is given some input data x and desired output data y, and is tasked with learning a function $f(x)$ that best fits the output data y. A good solution will not only fit the training data, but it will also possess predictive power, being able to generalize to some unseen data. Examples of common supervised models are regression models, random forests, support vector machines, and neural networks. In the unsupervised setting, an algorithm is only given x, and it must learn to summarize or explain the data's unseen structure in some way. Examples of unsupervised models are clustering models such as k-means and mixture models, principal components analysis (PCA), and matrix factorization techniques such as singular value decomposition (SVD) and non-negative matrix factorization. In the reinforcement learning setting, the objective is to teach an agent to traverse some environment in order to accomplish a goal. The agent learns with the assistance of a feedback mechanism which provides positive feedback to

the agent when it takes an action that is beneficial to accomplishing the goal, and negative feedback when it takes a detrimental action. Common reinforcement learning algorithms include Q-learning, actor-critic models, and policy gradient approaches.

2.2 Foundational Neural Models - Biological Inspiration

Neural networks are a class of models that draw inspiration from biological neural networks. (McCulloch and Pitts, 1943) and (Hebb, 1949) is widely regarded as being the pioneering work. (McCulloch and Pitts, 1943) introduced the McCulloch-Pitts model of the neuron (described in Eq. 2.1), which is the basis of most neural networks in the literature today, expressed as a dot product between some model parameters w and an input x, followed by a nonlinearity:

$$f(x) = \sigma\left(\sum_{i}^{n} w_i x_i\right) \tag{2.1}$$

(Hebb, 1949) introduced a theory describing how neurons might adapt in the learning process, described in Eq. 2.2. In this unsupervised setting, the weights between neurons x_i and x_j are updated to reflect the correlation between the neurons.

$$w_{i,j} \leftarrow w_{i,j} + \nu x_i x_j \tag{2.2}$$

(Rosenblatt, 1958) introduced a supervised learning model, allowing the simple models of neurons from (McCulloch and Pitts, 1943) to adapt their weights in order to learn simple discriminatory functions of the form

$$f(x) = u\left(w^t x + b\right) \tag{2.3}$$

where $w \in \mathbb{R}^n$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}$, and $u(\cdot)$ is the heaviside step function. (Minsky and Papert, 1969) studied the limitations of the perceptron model, showing that Eq. 2.3 is only capable of separating linearly separable data, famously using the XOR function as an example that the perceptron could not estimate.

2.3 Modern Neural Networks

In the perceptron model, the relationship between the input and the target variable is linear (or affine, if a bias is included), which can be limiting if the relationship between the input and the target variable is nonlinear. Modern neural networks introduce nonlinearity into the model by way of hidden layers, which represent the input in a latent space. Hidden layers may have nonlinear interactions between layers, which gives the model a way to represent highly nonlinear functions for classification or regression.

2.3.1 Calculating the Feedforward Pass for Fully-Connected Models

For a neural network with one hidden layer, the network's function $f(x) : \mathbb{R}^d \to \mathbb{R}^n$ can be defined as:

$$f(W_1, W_2, b_1, b_2, x) = \phi_2(W_2\phi_1(W_1x + b_1) + b_2) \tag{2.4}$$

where $\phi_l(\cdot)$ is the activation function for the l^{th} layer, $W_l \in \mathbb{R}^{d_{l+1} \times d_l}$ is a mapping from layer l to $l+1$, $b_l \in \mathbb{R}^{d_l}$ is a bias term, and $x \in \mathbb{R}^{d_1}$ is an input sample. More generally, a neural network with any number of hidden layers may be calculated by recursively applying

$$a_{l+1} \leftarrow \phi_{l+1}(W_l a_l + b_l) \tag{2.5}$$

where $a_l \in \mathbb{R}^{d_l}$ and setting $a_1 = x$, $f(x) = a_n$, where n is the number of layers in the network.

2.3.2 Cost Functions

When a neural network is used for regression tasks, the following loss function may be used:

$$J(x, y) = \frac{1}{N} \sum_{i=1}^{N} \|y_i - f(x_i)\|_2^2 \tag{2.6}$$

where x_i is the i^{th} of N input samples, y_i is the i^{th} target, and $\|\cdot\|_2^2$ is the square of the ℓ_2 norm. This loss function is commonly known as MSE, or Mean Squared Error. When a neural network is used for classification tasks, the following loss function is generally favored over MSE:

$$J(x, y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{d} \log\left(f(x_{i,j})\right) y_{i,j} \tag{2.7}$$

This loss function is known as cross entropy (CE), and is a much more natural choice for the categorical targets seen in classification. While CE and MSE are certainly the first choice for their respective tasks, there are many more kinds of costs. For example, a neural network can use a hinge loss on binary classification problems

$$J(x, y) = \frac{1}{N} \sum_{i=1}^{N} \max\left(0, 1 - f(x_i)\, y_i\right) \tag{2.8}$$

to obtain a max-margin boundary similar to an SVM (Cortes and Vapnik, 1995), or a neural network could directly optimize the BLEU score (Papineni et al., 2002) to obtain better results on machine translation tasks. As long as the cost function is differentiable, it can be used with a neural network.

2.3.3 Backpropagation

In order to search for a set of weights that are optimized for the task at hand, a neural network needs a cost function, i.e., a real-valued measure of how well the neural network is accomplishing its task, and a way to calculate the derivative of the weights with respect to this cost function. The derivative of the weights with respect to the cost function may be used in gradient descent, an iterative procedure to obtain some local minimum for the cost function by moving the weights in the direction of steepest descent of the cost.

Obtaining these gradients is accomplished through repeated applications of the chain rule in a process more generally known as automatic differentiation, and has been discovered by multiple sources in multiple contexts (Kelley, 1960), (Linnainmaa, 1970), (Rumelhart et al., 1988), (Werbos, 1990). The overall goal is to obtain the gradient of each weight $w_{i,j}$ with respect to the loss function $J\left(f\left(x\right),t\right)$ where $f\left(x\right)=y$ is the output of the neural network and t is the desired target for input x:

$$\frac{\delta J\left(y,t\right)}{\delta w_{i,j}} \tag{2.9}$$

As an example, the following shows backpropagation through a feed-forward neural network with one hidden layer h, linear output units y, and a mean squared error cost function for a single input example x:

$$\frac{\delta J}{\delta y}=\frac{\delta}{\delta y}\left(y-t\right)^{2}=2\left(y-t\right) \tag{2.10}$$

$$\frac{\delta J}{\delta h}=\frac{\delta J}{\delta y}\frac{\delta y}{\delta h}=W_{h}^{t}\frac{\delta J}{\delta y} \tag{2.11}$$

$$\frac{\delta J}{\delta W_{h}}=\left(\frac{\delta J}{\delta y}\right)^{t}h \tag{2.12}$$

$$\frac{\delta J}{\delta W_{x}}=\left(\frac{\delta J}{\delta h}\right)^{t}x \tag{2.13}$$

with the input-to-hidden weight matrix W_x and hidden-to-output weight matrix W_h.

Backpropagating through more complicated neural network structures such as recurrent and convolutional neural networks quickly becomes a tedious task to compute by hand. Fortunately, an approach known as automatic differentiation greatly simplifies the process of implementing new kinds of models. There are several software libraries such as (Bergstra et al., 2010), (Collobert et al., 2011), and (Abadi et al., 2015) that efficiently calculate gradients over many types of neural network architectures.

2.3.4 Batches and Minibatches

In most cases, it is not necessary to feed the entire training set through the neural network before obtaining the weight gradients. Instead, a smaller randomly selected subset of the training set, also known as a "minibatch" may be used to estimate the gradient at the current point in the parameter space, resulting in a noisy (but generally a good enough estimation) gradient for significantly less computation. This method is known as "stochastic gradient descent" (SGD). While it is more difficult to analyze theoretically (Bertsekas, 1999), SGD has many practical benefits over gradient descent, such as more rapid gradient updates and the ability to tune minibatch sizes to run faster on a particular target architecture.

2.4 Deep Neural Networks

Generally speaking, a "Deep Neural Network" is any neural network with more than one hidden layer. While the proofs of (Hornik et al., 1989) and (Cybenko, 1989) raise the question of the necessity of using many hidden layers, strong empirical results of deep neural networks outperforming shallow methods serve as evidence that deep yet narrow neural networks are somehow easier to optimize than shallow and wide neural networks. On the theoretical front, (Bengio, 2009) offers an argument in support

of the efficacy of deep and narrow neural networks by providing a comparison to representing boolean operations.

2.4.1 Greedy Layer-Wise Pre-Training

With the work of (Hinton et al., 2006) and (Bengio et al., 2007), training neural networks with many hidden layers became practical through the principle of unsupervised pre-training, that is, the notion of initializing the weights of a neural network so that models with several layers can be trained. By stacking autoencoders (a neural network that is trained to replicate its input after encoding it in some different space via the hidden layer) or restricted Boltzmann machines (a stochastic neural network that estimates the probability distribution of its input), the weights could be efficiently initialized. After these unsupervised neural networks are trained, the weights of the stacked autoencoders or restricted Boltzmann machines are copied to the neural network to be initialized, and a "fine-tuning" process completes the training procedure by further refining the weights connecting the final hidden layer and a logistic layer for supervised classification. Both methods were successful in advancing the state-of-the-art on several relevant computer vision benchmarks and increased interest in the viability of deep neural networks. These approaches continue to provide inspiration on the frontiers of semi-supervised and unsupervised learning tasks.

2.4.2 Activation Functions

In order to obtain nonlinear behavior from a neural network, it is necessary to apply a nonlinearity to the model. Such a nonlinearity is called an activation function, and an activation function is applied to the result of the linear transformation between layers. Historically, so-called "saturating nonlinearities" such as sigmoidal and hyperbolic tangent have been used, but such nonlinearities have somewhat fallen out of use in feed-forward neural networks due to their exacerbation of the "vanishing gradients"

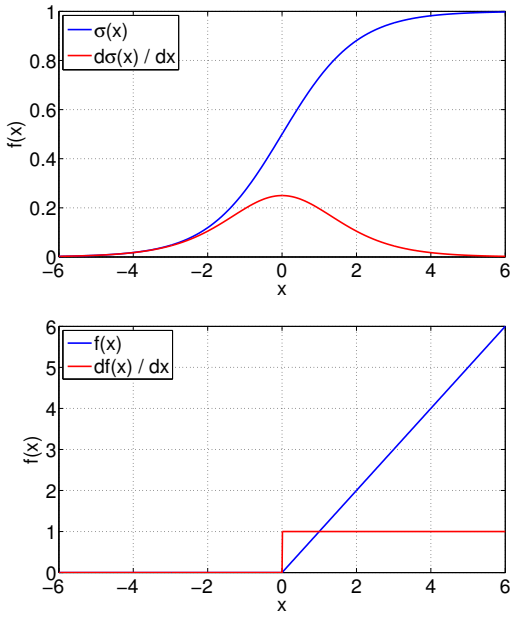

Figure 2.1: An illustration comparing the gradient of sigmoidal and rectified linear functions. As long as x is positive, $relu\,(x)$ will produce a strong gradient. $\sigma\,(x)$, however, has a fairly narrow range over its domain where a strong gradient is produced.

problem, where the gradient becomes increasingly weak as the error is backpropagated through the network. Other activation functions, such as "rectified linear" (ReLU) (Nair and Hinton, 2010) and its many extensions, "maxout" (Goodfellow et al., 2014) and "ℓ_p" (Gulcehre et al., 2014) have very large regions of the function where the gradient is far from zero, which leads to a stronger gradient during backpropagation. In addition, these activation functions tend to produce sparse gradients, which are hypothesized to improve the conditioning on the Hessian during optimization (Bengio, 2013). Such properties allow first-order optimization methods to more rapidly reach a local minimum (Bertsekas, 1999). Figure 2.1 illustrates the differences in the derivatives between sigmoidal and rectified-linear functions.

Because the rectified linear unit is simple to implement and apparently eases the optimization problem substantially, many extensions to the rectified linear unit have been proposed and are summarized in Figure 2.2. Leaky ReLUs (Maas et al., 2013) address the "dead neuron" problem where a number of hidden units with the ReLU

ReLU	(Krizhevsky et al., 2012)	$f(x) = \begin{cases} x & x > 0 \\ 0 & \text{else} \end{cases}$
Leaky ReLU	(Maas et al., 2013)	$f(x) = \begin{cases} x & x > 0 \\ \alpha x & \text{else} \end{cases}$
PReLU	(He et al., 2015)	$f(x_i) = \begin{cases} x_i & x_i > 0 \\ \alpha_i x & \text{else} \end{cases}$
ELU	(Clevert et al., 2015)	$f(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & \text{else} \end{cases}$

Figure 2.2: A listing of piecewise activation functions building off of ReLU

nonlinearity may be zero or become zero because the parameters bias the hidden units to be negative prior to applying the activation. In this case, there can be no gradient information propagated back through the parameters associated with the dead neuron, and the neuron will likely remain dead throughout the course of training. In order to discourage dead neurons from wasting capacity, leaky ReLUs multiply negative values of x by a small positive value instead of by zero. This allows for similar behavior to ReLUs, but less careful initialization (such as initializing biases to a large positive value) is necessary, as mostly-dead neurons still can have slight gradients backpropagating through them. Parametric ReLUs (PReLUs) (He et al., 2015) generalize the leaky ReLU, attaching an additional parameter α_i to each hidden unit, allowing the model to adaptively control the degree to which negative activations are scaled. Exponential-linear units (ELUs) (Clevert et al., 2015) are motivated by the observation that the mean of a ReLU activation over the samples in a batch can never be zero unless all of the activations are zero. The more widely used weight initialization techniques (more detail in 2.4.3) assume the input to a hidden layer is zero-mean, unit variance, which a ReLU activation would violate. While the statistics induced by the ELU aren't strictly zero-mean unit-variance, they are likely closer to the assumption than ReLU statistics, so dependence on proper weight initialization is less important.

2.4.3 Advanced Weight Initialization Techniques

While greedy layer-wise pretraining is effective at finding weight initializations that supervised training can more rapidly optimize, there now exist more direct modifications to the neural network architectures or training algorithms that do not require pretraining as an initial step. Less naive weight initialization strategies (compared to drawing weights i.i.d. from some simple probability distribution) have been long known to increase the speed of convergence for shallow neural networks (Nguyen and Widrow, 1990) (Yam and Chow, 2000), and strategies using sparse weight initializations were introduced in (Martens, 2010), allowing for much faster convergence in deep and recurrent neural networks. In such a sparse weight initialization, the weights $W_{i,j}$ are initially chosen as $W_{i,j} \sim \mathcal{N}(0, \sigma^2)$, and then a large portion of the indices i, j, are set to zero in order to prevent the nonlinearities from saturating, which can slow learning substantially.

Some weight initialization strategies such as (Glorot and Bengio, 2010) and (LeCun et al., 2012) start by assuming the inputs x will be sampled from a simple distribution, such as $x \sim \mathcal{N}(0, 1)$, and then derive ways to scale the weights such that the hidden activations h_i of $h_i = w_i^T x$ will also be close to $y_i \sim \mathcal{N}(0, 1)$. Similar to the justification for the sparse weight initialization strategy of (Martens, 2010), scaling w_i by $\frac{1}{\sqrt{d}}$ where $x \in \mathbb{R}^d$ and $y \in \mathbb{R}^h$ has the effect of initializing $\sigma(y_i)$ to be in a non-saturated state. (Glorot and Bengio, 2010) derives a scaling factor $\frac{1}{\sqrt{d+h}}$ which strikes a compromise between the hidden activations y_i and the gradients of the hidden activations with respect to the loss $\frac{\delta y_i}{\delta J}$ being close to $\mathcal{N}(0, 1)$.

(Saxe et al., 2014) addresses the weight initialization problem from a different perspective by studying the training dynamics of deep, yet linear, models. While such a model can be trivially condensed into a single layer network by multiplying the weight matrices together to combine into a single weight matrix, (Saxe et al., 2014) nevertheless shows that the backpropagation of information through such networks has similar difficulties to the backpropagation of information through nonlinear

networks. By initializing the weight matrices to be orthogonal, information flows forward and backwards through the network much more easily, due to the fact that the singular values are equal to one, so successive multiplications (corresponding to the feed-forward) or transposed multiplications (corresponding to backpropagation) do not make the vector norms of the inputs (or gradients) increase or decrease as information is initially propagated through the network. (Saxe et al., 2014) shows that orthogonal weight initialization works very well in deep nonlinear models, as long as a scaling factor dependent on the activation function is applied to the weight initialization as well.

2.4.4 Advanced Optimization Techniques

Optimizing the weights of a neural network is a difficult nonconvex optimization problem generally solved by first-order methods. However, the loss surface of a particular neural network with respect to its training data has been hypothesized to have degenerate structure (e.g., large portions of the parameter space that have close to zero gradient, troublesome saddle points, areas where the Hessian is poorly conditioned), making it difficult for simpler optimization methods to efficiently solve (Sutskever et al., 2013), (Pascanu et al., 2013), (Dauphin et al., 2014). Second-order methods can better deal with the hypothesized loss surfaces, but are difficult to use in practice due to the quadratic scaling of the number of parameters (which can be in the millions or billions for large neural networks). Approximate second-order methods (Liu and Nocedal, 1989) can be used, but tend to require larger batch sizes than SGD.

(Sutskever et al., 2013) re-framed Nesterov's accelerated gradient method (Nesterov, 1983) in the context of momentum, allowing for much more rapid convergence with very little computational overhead. (Martens, 2010) introduced a different method to speed up the convergence rate by using "Hessian-Free" optimization, an efficient quasi-Newton method. Some methods such as (Schaul et al., 2012) and (Zeiler, 2012) address concerns with setting the learning rate and defining tedious

learning rate schedules by giving every hidden activation, or even every weight, an independent and adaptive learning rate.

Efficient methods for parallelizing SGD (Recht et al., 2011) (Dean et al., 2012) (Huang et al., 2014) (Zhang et al., 2015) have enabled the scaling of training large neural networks across many computing clusters. In such schemes, parallelization is achieved through data parallelism, where a set of worker nodes in a computing cluster are assigned different partitions of some training data. The node calculates the weight gradient with respect to its local training data and broadcasts these gradients back to a weight parameter node, which then sends a new set of weights to the worker. In such schemes, the workers communicate weight gradients asynchronously, that is, every worker is calculating gradients with some weights that are potentially out of sync with the weights on the weight node. Nevertheless, such asynchronous distributed methods work very well in practice, sometimes delivering near-linear speedups as worker nodes are added.

2.4.5 Normalizing Activation Values

Recent techniques have focused on finding ways to mitigate vanishing gradients in backpropagation by ensuring that the hidden activations of the model don't become too saturated. Batch normalization (Ioffe and Szegedy, 2015) deals with vanishing gradients by constraining pre-activation values to be zero-mean unit-variance by applying the following transform:

$$BN\left(x\right) = \gamma \frac{x - \mu_x}{\sqrt{\sigma_x^2 + \epsilon}} + \beta \tag{2.14}$$

where μ_x is the sample mean, σ_x^2 is the sample variance, γ and β are trainable parameters, and ϵ is a small constant intended to provide numerical stability in the case that σ_x^2 is close to zero for the statistics of a hidden activation. This transform reduces the impact of so-called "covariate shift", allowing the model to train more

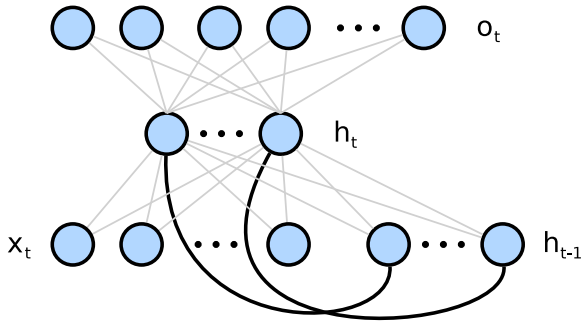

Figure 2.3: An Elman recurrent neural network. o_t represents the output units, h_t represents the hidden units for the current timestep, x_t represents the input, and h_{t-1} represents the hidden units for the previous timestep. The bolded connection from h_t to h_{t-1} represents the delayed recurrence.

quickly by keeping activation statistics across layers and hidden activations more uniform throughout training.

2.5 Recurrent Neural Networks

Contrasted with the connection structure of a feed-forward neural network, in which connectivity is restricted to a hidden unit $a_l^{(i)}$ to another hidden unit in a higher layer $a_{l+n}^{(j)}$, $n > 0$, recurrent neural networks can have self-connecting and recurrently connected units. Such a configuration endows the network with a notion of memory, enabling recurrent models to deal with sequences such as time-series data. Given the large number of connections a node in the neural network may have with another, it is useful to restrict the study of recurrent neural networks to simpler connection topologies. A simple example of a recurrent topology is to delay the hidden layer by one timestep and feed the delayed hidden layer back into the input. This configuration is known as an Elman network (Elman, 1993), and is depicted in Figure 2.3.

2.5.1 Backpropagation Through Time

In order to train an RNN to capture the dynamics of some time-varying sequence, simple backpropagation is usually not suitable, as it only considers the immediate

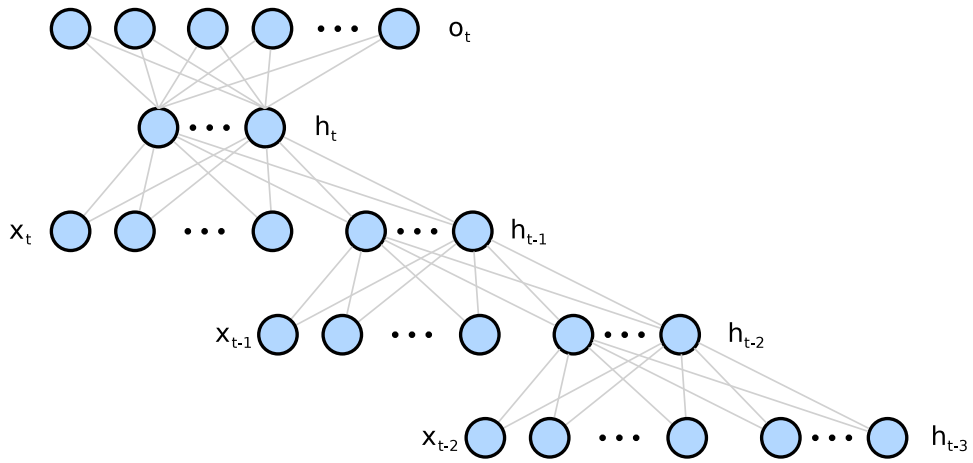

Figure 2.4: A recurrent neural network unfolded twice for backpropagation through time. The previous inputs x_{t-1}, x_{t-2}, ..., and the previous hidden activations h_{t-1}, h_{t-2}, ... must be retained so that backpropagation can take place.

input and previous hidden state. Because only the context from the previous timestep is considered in simple backpropagation, long-term dependencies in the input sequence are not captured. In order to capture these dependencies, backpropagation through time (Werbos, 1990) is typically used, in which the network structure must be "unrolled" into a deeper recurrent neural network so that previous input and hidden activations can be considered during backpropagation, as depicted in Figure 2.4. The unrolled input-to-hidden weight gradients are averaged together with equal weighting during the weight update.

2.5.2 Difficulty of Training

RNNs have been regarded as difficult to train when compared to simpler time-series models (Bengio et al., 2013a). The nonlinear transformations between timesteps can lead to chaotic behavior, and the model is difficult to formalize analytically. In addition, gradient-based training of RNNs tends to suffer from the so-called "vanishing" and "exploding" gradients problems (Bengio et al., 1994). In the "vanishing gradients" situation, the gradients becomes increasingly small as the error

is backpropagated through time, leading to a less emphasis on the capturing of longer-term dependencies that BPTT is designed to address. In the "exploding gradients" problem, the opposite occurs – the gradients may increase suddenly and without bound, which adds very large values to the weights, which pushes the model very far away from the local minimum it was approaching. To address the exploding gradients behavior, strategies such as clipped weights (Bengio et al., 2013a) are employed whereby weight gradients are truncated if they exceed some predefined threshold.

2.5.3 Addressing Vanishing Gradients by Architectural Choices

To alleviate the vanishing gradients behavior, one can carefully architect a hidden unit that utilizes a series of gates in order to retain information over time, such as long short-term memory units (LSTMs) (Hochreiter and Schmidhuber, 1997) or gated recurrent units (GRUs) (Chung et al., 2014). An LSTM unit extends the Elman network by adding a series of multiplicative gates to the update calculation of the hidden state:

$$i_t = \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) \tag{2.15}$$

$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \tag{2.16}$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \sigma\left(W_c x_t + U_c h_{t-1} + b_c\right) \tag{2.17}$$

$$o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) \tag{2.18}$$

$$h_t = o_t \cdot \sigma\left(c\right) \tag{2.19}$$

i_t is the "input gate", f_t is the "forget gate", c_t is the "memory cell", and o_t is the output gate. $W_{(.)}$ and $U_{(.)}$ are trainable parameters, and operations such as $f_t \cdot c_{t-1}$ are element-wise multiplication. The parameters associated with the input gate give the model the ability to control how much new information can come into the memory cell c_t. The parameters associated with the forget gate give the model the ability to

control how much of the memory cell of the previous timestep c_{t-1} to consider in the state update. Similarly, the parameters of the output gate give the model the ability to control how much of the memory cell gets propagated to the next layer. All of these elements together give the LSTM the ability to dynamically control how much information enters and leaves the memory cell, resulting in a model that can learn how to remember or forget certain aspects of the input sequence. The LSTM unit has many application-dependent extensions (Gers et al., 2003) (Graves and Schmidhuber, 2005) (Kalchbrenner et al., 2015).

GRUs are similarly motivated by giving the model the ability to learn how to remember and forget, while requiring fewer parameters than the LSTM. The GRU only has two gates, the forget gate r_t and the output gate z_t. The forget gate gives the model the ability to block or pass information from the previous hidden state h_{t-1} through an element-wise multiplication with a value that is bound between $(0, 1)$. The output gate z_t is a convex combination of a proposal state $\hat{h}_t$ and the previous hidden state. These gates give the model the ability to explicitly save information over many timesteps, but in a simplified way compared to the LSTM. Empirically, GRUs show competitive performance with LSTMs when controlling for model size (Chung et al., 2014)

2.5.4 Addressing Vanishing Gradients by Weight Initialization

Noting that the vanishing and exploding gradients problem primarily comes from the singular values of the weight matrices straying far from values of 1, some approaches consider the vanishing gradients problem by trying to keep the weights in this region. In (Mikolov et al., 2014), the authors make a modification to the simple Elman network by constraining some of the hidden states to change more slowly, while imposing no such constraint on the other hidden states, demonstrating comparable performance to LSTMs on a language modeling task.

Similarly, in (Le et al., 2015), the recurrent weights U of the Elman network are simply initialized to the identity matrix, without any additional guidance. Such an initialization would allow for gradients associated with the hidden units to flow quite freely in the beginning of training. The authors use ReLU as the hidden activation, which is quite notable given the unboundedness of the non-zero portion of the function, potentially leading to greater risk of the norm of the activations increasing without bound, which would lead to exploding gradients as well as exploding activation values. Nevertheless, the authors demonstrate comparable performance relative to LSTMs on language modeling, speech recognition, as well as a "sequential MNIST" task where the objective is to classify digits given a long sequence of individual pixel values from the MNIST dataset.

(Arjovsky et al., 2015) proposes to directly parameterize the weight matrices of a recurrent neural network as unitary matrices, which imposes a constraint that the singular values will always be equal to 1. This modification, while not trivial to implement and involving dealing with activations as complex numbers, greatly improves a simple RNN's ability to store information over very long timesteps without adding the additional complexity of specialized gates.

2.5.5 Model Regularization

As with all other models used for regression and classification, neural networks are susceptible to overfitting on the training data. Many of the same regularization techniques used for logistic and linear regression also apply to neural networks. Some examples of regularization techniques for neural networks include:

- ℓ_1 regularization on the weights or activations:

$$R(\lambda, \theta) = \lambda \|\theta\|_1 \tag{2.20}$$

- ℓ_2 regularization on the weights:

$$R(\lambda, \theta) = \lambda \|\theta\|_2 \tag{2.21}$$

- Kullback-Leibler penalty on the activations:

$$R(\lambda, \rho, \hat{\rho}) = \lambda \sum_{i=1}^{n_h} KL(\rho \| \hat{\rho}_i) \tag{2.22}$$

- Dropout regularization:

$$a_{l+1} = (\phi(W_l a_l) + b_l) \odot S \tag{2.23}$$

These penalties are applied by adding the penalty to the loss function. The gradient of the loss with respect to the weights is usually a function of the weights themselves, rather than the activations. Consequently, the calculation of many of these regularization techniques is simple and straightforward, with the exception of the Kullback-Leibler penalty, which requires extra information to be backpropagated in order to update weights in the correct direction.

ℓ_1 penalties (Tibshirani, 1996) on the weights versus the activations have different roles in terms of how they regularize. An ℓ_1 penalty on the weights pushes the gradient in a direction that favors sparse weights, which may be desirable in some situations (e.g., feature selection in logistic regression models, weight pruning in neural networks) and is somewhat application dependent. The ℓ_1 penalty is not typically used to regularize the weights of a neural network. An ℓ_1 penalty on the activations, however, has the straightforward interpretation of pushing the gradient in a direction that favors sparse activations. ℓ_2 regularization, also known as Tikhonov regularization (Tikhonov and Arsenin, 1977), has the tendency to shrink the weights of the neural network, which favors solutions that are less likely to overfit by limiting the dynamic capacity of the model (Bishop et al., 1995).

The Kullback-Leibler (KL) penalty is an alternative way of encouraging sparse activations (Ngiam et al., 2011), and allows for greater control over how many hidden units are sparse in a given layer. By setting a target sparsity ρ, the Kullback-Leibler divergence is calculated between the target sparsity and some measure of average sparsity in a hidden layer. While KL penalties were commonly used to impose sparsity constraints on the hidden activations of sparse autoencoders, they are not generally used to regularize neural networks used in a supervised setting.

Dropout regularization (Hinton et al., 2012) works slightly differently, as there is no additional term added to the loss function. Instead, hidden units are randomly omitted with probability p on a per-sample basis, as indicated by the element-wise multiplication $\odot$ with the masking matrix S. The function of dropout regularization has many potential interpretations. The most common interpretation is to view dropout as a model averaging technique, where each randomly sampled dropout mask S selects one of 2^H models (where H is the number of hidden units in the model). In this interpretation, an input sample will be trained with respect to one of very many submodels, but these submodels practice extreme weight sharing. When obtaining outputs from the model when validating or testing, one can repeatedly sample from the stochastic model in order to obtain a Monte-Carlo average of the output. However, it is more common to ignore the dropout step and instead divide the weight matrices by $1 - p$ to compensate for the fact that the model's activations are more active without the stochastic elimination of hidden units. While this step gives exactly the expected value of the stochastic model for single-layer models such as linear or logistic regression, it becomes less exact for deep models, but is a good enough approximation in practice. (Goodfellow et al., 2013) addresses this potential deficiency by engineering an activation function called "maxout" that closer matches the stochastic behavior of a dropout model when applying the simple ensemble averaging trick.

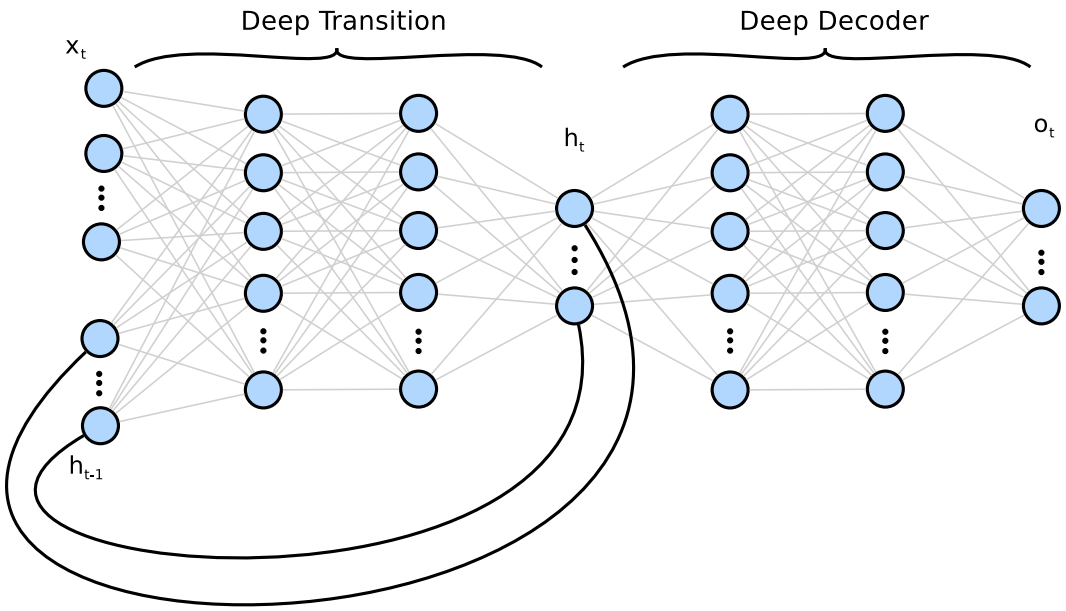

Figure 2.5: A DOT-RNN. The deep transition allows for an increase in the nonlinearity expressible in the input-to-recurrent state mapping. The deep decoder allows for an increase in the nonlinearity between the recurrent state and the desired output.

2.5.6 Deep Recurrent Neural Networks

RNNs can be made deep in a variety of ways. In one such configuration, one can "stack" RNNs on top of each other, allowing for the model to capture temporal dependencies at different time scales (El Hihi and Bengio, 1995). Alternatively, one can seek to make the transition function $f(W_1, U, x, h) = \phi(W_1 x + Uh)$ (i.e., the function that calculates the recurrent state) deeper through the addition of several nonlinearities between the input and the recurrent state. One can also make the decoding function $f(W_2, h) = \phi(W_2 h)$ deeper by adding several layers of nonlinearities between the recurrent state and the neural network output. Adding depth to the transition and decoder functions is known as a "Deep Output and Transition" (DOT) RNN (Pascanu et al., 2014), and is illustrated in Figure 2.5.

2.6 Conditional Computation

Conditional computation, originally proposed in (Bengio, 2013), is the notion of skipping the computation of some nodes given the values of other nodes in the network. Some training regularization or sparsification techniques such as (Hinton et al., 2012) and (Makhzani and Frey, 2014) tend to skip the required computation in a random or completely fixed manner. (Bengio, 2013), on the other hand, proposes to "drop them [the calculation of hidden units] in a learned and optimized way." Formally, the objective is to define a gating function:

$$f : \mathbb{R}^n \rightarrow \{0, 1\}^m \tag{2.24}$$

that makes a binary decision of whether to gate activation a^i_{l+1} based on the current state of the network.

The crucial motivation for conditional computation is that of substantially increasing model capacity (e.g., the amount of information a neural network can store) while reducing the growth of computation required to scale to larger models. Presently, the most sophisticated neural network computer vision or natural language processing models contain hundreds of millions to billions of parameters (Sermanet et al., 2014) (Jozefowicz et al., 2016). Such models tend to take several days to several weeks to train. In order to scale to more difficult classification and inference problems such as language modeling and machine translation, many more parameters may be needed in the model, implying that several weeks to several months of training may be required, unless the model is given more computational resources (e.g., faster hardware as a result of Moore's law, more nodes on a computing cluster, more memory, etc.) A useful conditional computation model would allow for greatly scalable neural networks.

2.6.1 Mixtures of Experts

The hard mixtures of experts model (MoE) (Collobert et al., 2003) is an early example of increasing the number of parameters relative to the amount of required computation. This model is a variation of the MoE model (Jacobs et al., 1991). In the MoE ensemble, several classifiers are trained to specialize on specific subsets of the input space. When obtaining an output label, each classifier comes with an associated weighting (or confidence), which is averaged to obtain a better estimation of the class label. In the hard MoE model, however, a single classifiers output label is chosen (based on some stochastic or deterministic function of the classifiers' confidences). While the hard MoE offers no computational benefits during training because all of the gaters are computed in order to determine the optimal assignment of examples to gaters, the output label can be obtained with $\frac{1}{k}$ fewer computations, assuming k classifiers. The MoE model was extended to a deeper model in (Eigen et al., 2014), albeit with no immediate computational benefits.

2.6.2 Prior Art

In a technical note, (Cho and Bengio, 2014) expands on the ideas initially proposed in (Bengio, 2013), detailing a parameterization of a model that has an exponentially increasing ratio of capacity to required computation. (Cho and Bengio, 2014) proposes obtaining a k-bit vector from the activations of a binary gating function $g(x) \in \mathbb{R}^k$. $g(x)$ could be a simple element-wise thresholding $g(x, \tau) = \max(x, \tau)$, or it could be sampled from a multinomial distribution with probabilities given by $g(x) = \sigma(Ux)$. If each bit vector corresponds to a different weight matrix selection, there would be 2^k different weight matrices to choose given an input example x, yielding a rapidly increasing ratio of parameters to required computation as k increases. Alternatively, the bit vector can be interpreted as a series of directions to descend in a binary search tree (with "0" interpreted as a move to the left node, and a "1" interpreted as a move to the right node).

In (Léonard, 2015), various types of conditional computation are applied to feed-forward neural networks on a handwritten digit classification task (LeCun and Cortes, 2010) and a large language modeling task (Chelba et al., 2013). Similar to (Cho and Bengio, 2014), (Léonard, 2015) uses tree-structured weight matrices on the MNIST dataset, but has difficulties with the stability of the proposed ESSRL algorithm. However, (Léonard, 2015) obtains impressive wall-time speedups in training feed-forward neural language models by formulating the sparsity as block-sparse – that is, instead of each hidden unit being free to be sparse or non-sparse, large contiguous blocks of hidden units take on the same sparsity. In (Bacon et al., 2015) and (Bengio et al., 2015), the authors use a reinforcement learning approach to learn how to gate the hidden activations in a feed-forward neural network. While there are modest speed gains to this approach, it appears they are only applicable when comparing the specialized block-sparse implementation with a single-core implementation of a matrix multiplication.

2.7 Other Methods of Accelerating Neural Networks

Aside from conditional computation, there are many other classes of methods that attempt to accelerate neural networks, generally with applications towards deploying trained neural networks. Reducing bit depth of activations and weights (Lin et al., 2015) (Kim and Smaragdis, 2016) are potential avenues of both speeding up models and reducing memory requirements, especially on custom hardware implementations. In such implementations, the activations and/or parameters of the neural networks are quantized from hardware-native 32-bit floating point (Kahan and Palmer, 1979) to a far fewer number of bits, in some cases as few as 1. This has immediate benefits in terms of memory requirements, reducing the amount of storage required by a factor as great as 32x. In the case that the weights and activations are quantized to one bit,

addition and multiplication can be re-cast as bit-wise OR and AND operations, which are much cheaper than floating point addition and multiplication. In the case that weights or activations can not be so aggressively quantized, there are still caching benefits, where larger proportions of the model weights can be stored in L1 or L2 cache, providing faster accesses to model parameters when computing activations.

Other approaches use low-rank or tensor decompositions of the parameters (Denton et al., 2014) (Jaderberg et al., 2014) (Lebedev et al., 2014) to reduce the number of parameters, and thus, the computation time required to feed-forward a sample or batch of samples through the model.

2.8 A Brief BLAS Primer

BLAS (Basic Linear Algebra System) is a standardized library of routines that implement commonly used linear algebra operations. Element-wise addition, dot-products, matrix-vector multiplication, and matrix-matrix multiplication are provided, along with type-specific (i.e., single-precision versus double-precision floating point) subroutines. Some BLAS implementations, such as OpenBLAS and Intel's Math Kernel Library, are highly optimized with performance in mind, and strive to perform close to the peak performance of the target architecture when possible.

BLAS Level 1 operations are the simplest, and provide operations on vectors such as dot products and norms. Level 2 operations cover matrix-vector operations such as matrix-vector multiplication. Level 3 operations are concerned with matrix-matrix operations, such as matrix-matrix multiplication. Level 1 operations exploit the peak floating point capabilities of many architectures, as the computations are slowed down by the lack of reuse by the cache. Such operations are referred to as "memory bound", as the operation spends more time waiting to fetch data from memory than actually doing the computation. Level 2 operations are also memory bound, but the cache can be used to accelerate computation to some degree. Level 3 operations (Dongarra et al., 1990) tend to optimize very well, with some implementations

reaching 90% of the target platform's computational speed (as measured in FLOP/s). This behavior is attributable to the "surface-to-volume" effect (Dongarra et al., 1989), where the ratio of computation to input data is fairly high. In the case of matrix-matrix multiplication, there are $O(n^3)$ floating point operations, but only $O(n^2)$ data elements, allowing for more opportunities to re-use data, and thus, take advantage of the cache. For matrix-vector multiplication, however, there are $O(n^2)$ floating point operations and $O(n^2)$ data elements, which makes caching less useful.

Chapter 3

Conditional Computation in Feed-Forward Neural Networks

3.1 The Activation Estimation Approach

3.1.1 Activation Estimation-Based Models

An activation estimator is an auxiliary set of hidden units that computes the gating function Eq. 2.24 by way of an intermediate linear bottleneck layer that is substantially smaller than the input or output dimensionality of Eq. 2.24. This implies the addition of two sets of weight matrices, U, the matrix connecting the input to the intermediate layer, and V, the matrix connecting the intermediate layer to the output, illustrated in 3.1. In (Bengio et al., 2013b), the hidden units in the intermediate layer can take on any nonlinearity, and the output of the activation estimator is sigmoidal. The gating decisions are made by sampling a binomial distribution with probability ρ_i, where $\rho_i = a^i_{ae}$, effectively treating the output as a probability of gating the i^{th} hidden unit. The weights U and V are learned by backpropagation.

In (Davis and Arel, 2014), the intermediate layer of the activation estimator is required to be linear, and the output layer is the $sign\,(\cdot)$. The activation estimator

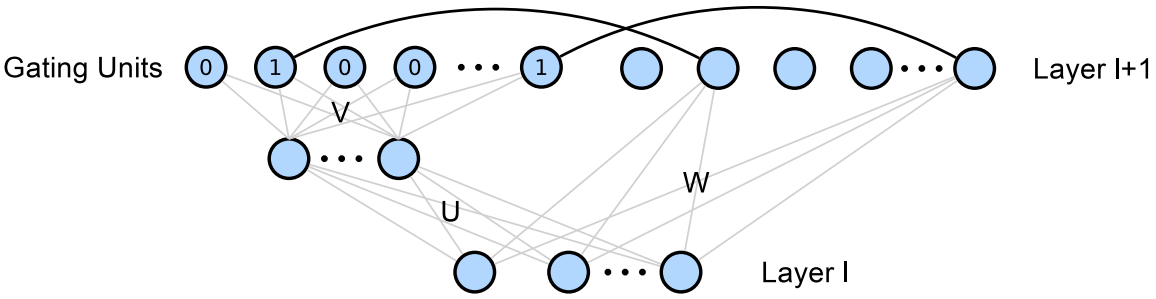

Figure 3.1: An illustration of an activation estimator layer gating the hidden activations on layer $l+1$ based on the activations on layer l. Gated (i.e., hidden units that do not propagate past the gate) units do not need to be calculated.

weights are determined by an occasional recalculation of the SVD, setting $U = \hat{U}_k \hat{\Sigma}_k$ and $V = \hat{V}_k^t$, where $\hat{U}_k$, $\hat{\Sigma}_k$, and $\hat{V}_k^t$ are the submatrices that satisfy the conditions of making the product UV a $rank\,(k)$ approximation. The result of UVa_l, where a_l is an input sample, is such that $sign\,(UVa_l) \approx sign\,(Wa_l) = sign\,(a_{l+1})$

3.1.2 Redundancy in Parameterization

Several authors (Denil et al., 2013) (Denton et al., 2014) have noted the redundancy in the parameters in deep neural networks. In (Denil et al., 2013), the redundancy is exploited in the context of distributed computing, whereby the filters of a convolutional neural network are shown to have high spatial correlation, allowing the reduction of communication between workers by sending only a subset of filter weights and reliably inferring the other weights. In some cases, the authors were able to reduce the number of sent parameters by 95%, significantly reducing inter-node communication when training a large model across several machines. In (Denton et al., 2014), the authors note a similar redundancy, but exploit it instead by factorizing the filters into low-rank approximations, allowing for the filter responses to be calculated more quickly. In this case, the authors could obtain a 2-3x reduction in the work required for the feed-forward operation with only a slight degradation in classifier performance.

From a different perspective, the redundancy can be seen in both the activations as well as the weights. If a layer of a neural net has a weight matrix W_l such that W_l can be closely approximated with a rank-k matrix $\hat{W}_l$, then the resulting matrix multiplication $a_{l+1} = \hat{W}_l a_l$ is at most rank-k as well, by the inequality $rank\,(AB) \leq \min\,(rank\,(A)\,, rank\,(B))$.

3.1.3 Estimating the Activation Sign

In many neural networks, the rectified linear activation function, $relu\,(x) = \max\,(0, x)$, is used for its fast convergence properties and ease of implementation. It is important to emphasize that this activation function is zero for all negative values, and positive for all positive values. For this particular activation function, the fact that $sign\,(a_{l+1}) \approx sign\left(\hat{W}_l a_l\right)$ can be exploited to begin building an activation estimator that can predict which output activations are likely to be non-zero, and thus need to be calculated. Section 3.1.1 introduced the notion of the activation estimator, and this subsection will expand on this idea in greater detail.

Given the activation a_l of layer l of a neural network, the activation a_{l+1} of layer $l+1$ is given by:

$$a_{l+1} = \phi(W_l a_l) \tag{3.1}$$

where $\phi(\cdot)$ denotes the function defining the hidden unit's nonlinearity, $a_l \in \mathbb{R}^{h_l \times n}$, $a_{l+1} \in \mathbb{R}^{h_{l+1} \times n}$, $W_l \in \mathbb{R}^{h_{l+1} \times h_l}$. If the weight matrix is highly redundant, as in (Denil et al., 2013), it can be well-approximated using a low-rank representation and we may rewrite (3.1) as

$$a_{l+1} \approx \phi(U_l V_l a_l) \tag{3.2}$$

where $U_l V_l$ is the low-rank approximation of W_l, $U_l \in \mathbb{R}^{k \times h_l}$, $V_l \in \mathbb{R}^{h_{l+1} \times k}$, $k \ll \min(h_l, h_{l+1})$. So long as $k < \frac{h_l h_{l+1}}{h_l + h_{l+1}}$, the low-rank multiplication $U_l V_l a_l$ requires fewer arithmetic operations than the full-rank multiplication $W_l a_l$, assuming the multiplication by U_l occurs first. When $\phi(\cdot)$ is the rectified-linear function, such

that all negative elements of the linear transform $W_l a_l$ become zero, one only needs to estimate the sign of the elements of the linear transform in order to predict the zero-valued elements. Assuming the weights in a deep neural network can be well-approximated using a low-rank estimation, the small error in the low-rank estimation is of marginal relevance in the context of recovering the sign of the operation.

Given a low-rank approximation $W_l \approx U_l V_l = \hat{W}_l$, the estimated sign of a_{l+1} is given by

$$sign(a_{l+1}) \approx sign(\hat{W}_l a_l) \tag{3.3}$$

Each element $(a_{l+1})_{i,j}$ is given by a dot product between the row vector $W_l^{(i)}$ and the column vector $a_l^{(j)}$. If $sign(\hat{W}_l^{(j)} a_l) = -1$, then the true activation $(a_{l+1})_{i,j}$ is likely negative, and will likely become zero after the rectified-linear function is applied. Considerable reductions in computation are possible if we skip those dot products based on the prediction; such gains are especially substantial when the network is very sparse. The overall activation for a hidden layer l augmented by the activation estimator is given by $\phi(W_l a_l) \odot S_l$, where $\odot$ denotes the element-wise product and S_l denotes a matrix of zeros and ones, where

$$(S_l)_{i,j=} \begin{cases} 0, & sign\left((U_l V_l a_l)_{i,j}\right) = -1 \\ 1, & sign\left((U_l V_l a_l)_{i,j}\right) = +1 \end{cases} \tag{3.4}$$

The Singular Value Decomposition (SVD) is a common matrix decomposition technique that factorizes a matrix $A \in \mathbb{R}^{m \times n}$ into $A = U\Sigma V^T$, $U \in \mathbb{R}^{m \times m}, \Sigma \in \mathbb{R}^{m \times n}, V \in \mathbb{R}^{n \times n}$. By (Eckart and Young, 1936), the matrix A can be approximated using a low rank matrix $\hat{A}_r$ corresponding to the solution of the constrained optimization of

$$\min_{\hat{A}_r} \|A - \hat{A}_r\|_F \tag{3.5}$$

where $\|\cdot\|_F$ is the Frobenius norm, and $\hat{A}_r$ is constrained to be of rank $r < rank(A)$. The minimizer $\hat{A}_r$ is given by taking the first r columns of U, the first r diagonal

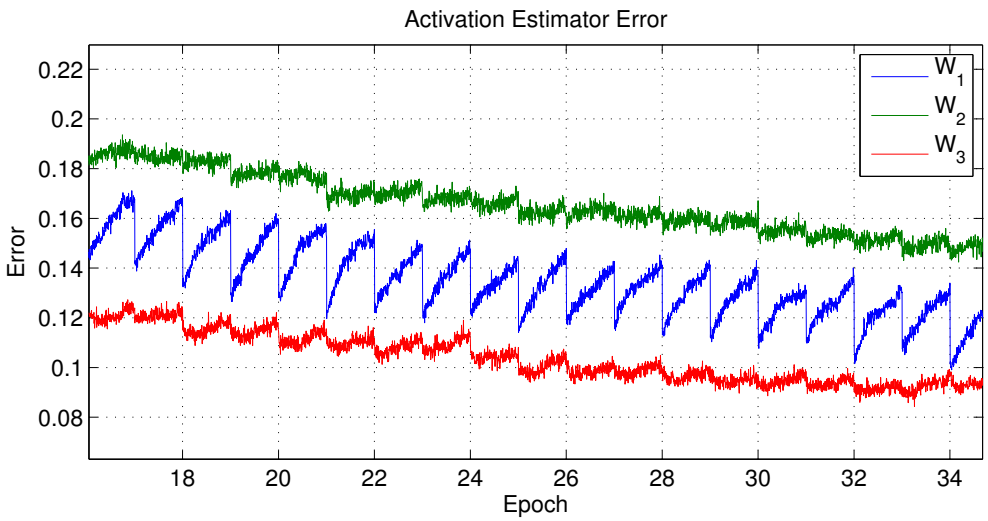

Figure 3.2: An illustration of the error of the activation estimator (as measured by the percentage of correct gating decisions over a minibatch) as the current weights deviate from the weights used in the low-rank estimation.

entries of Σ, and the first r columns of V. The resulting matrices U_r, Σ_r, and V_r are multiplied, yielding $\hat{A}_r = U_r \Sigma_r V_r^T$. The low-rank approximation $\hat{W} = UV$ is then defined such that $\hat{W} = U_r(\Sigma_r V_r^T)$, where $U = U_r$ and $V = \Sigma_r V_r^T$.

Unfortunately, calculating the SVD is an expensive operation, on the order of $O(mn^2)$, so recalculating the SVD upon the completion of every minibatch adds significant overhead to the training procedure. Given that we are uniquely interested in estimating in the sign of $a_{l+1} = W_l a_l$, we can opt to calculate the SVD less frequently than once per minibatch, assuming that the weights W_l do not change significantly over the course of a single epoch so as to corrupt the sign estimation. Figure 3.2 shows an example of the error of the activation estimator oscillating as the SVD is recalculated in the beginning of each training epoch.

3.1.4 Theoretical Upper Limits of Speed Gains

For every input example, a standard neural network computes $\phi(Wa)$, where $a \in \mathbb{R}^{d \times N}$ and $W \in \mathbb{R}^{h \times d}$, where N is the number of input examples. Assuming additions and multiplications are constant-time operations, the matrix multiplication requires

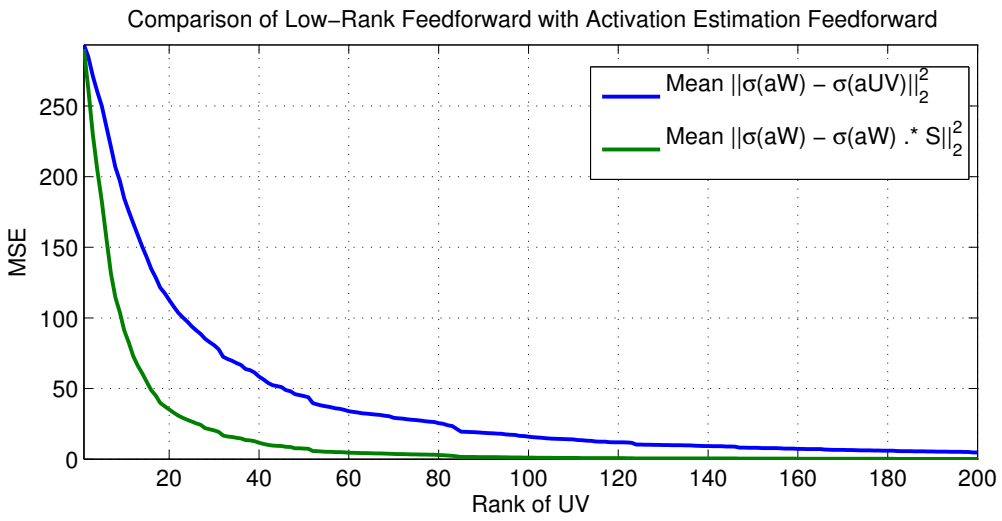

Figure 3.3: A summary of the errors introduced by the low-rank approximation. The blue line indicates the error between the actual activation and the activations obtained through a low-rank approximation of the weight matrix. The green line indicates the error if a full feedforward with the original weight matrix is combined with the mask of the activation estimator. The activation estimator can reliably determine the sign of the output activation with a fairly low rank.

$N(2d-1)h$ floating point operations (we need to compute Nh dot products, where each dot product consists of d multiplications and $d-1$ additions), and the activation function requires Nh floating point operations, yielding $N(2d-1)h+Nh$ operations. The activation estimator $sign(UVa)$, $U \in \mathbb{R}^{h\times k}$, $V \in \mathbb{R}^{k\times d}$ requires $N(2d-1)k + N(2k-1)h$ floating point operations for the low-rank multiplication followed by Nh operations for the $sign(\cdot)$ function, yielding $N(2d-1)k + N(2k-1)h + Nh$. However, given a sparsity coefficient $\alpha \in [0,1]$ (where $\alpha = 0$ implies no hidden units are active, and $\alpha = 1$ implies all hidden units are active), a conditional matrix multiplication would require $\alpha N(2d-1)h + \alpha Nh$ operations.

Altogether, the number of floating point operations for calculating the feedforward in a layer in a standard neural network is

$$F_{nn} = N(2d-1)h + Nh \tag{3.6}$$

and the number of floating point operations for the activation estimation network with conditional computation is

$$F_{ae} = N\left(2d-1\right)k + N\left(2k-1\right)h + Nh + \alpha h\left(N\left(2d-1\right)h + Nh\right) \tag{3.7}$$

The relative reduction of floating point operations for a layer can be represented as $\frac{F_{nn}}{F_{ae}}$, and is simplified as

$$\gamma_l = \frac{2dh}{k\left(2d+2h-1\right)+2\alpha dh} \tag{3.8}$$

For a neural network with many layers, the relative speedup is given by

$$\gamma_{NN} = \frac{\sum_{i=1}^{L} F_{nn}^{(l)}}{\sum_{i=1}^{L} F_{ae}^{(l)}} \tag{3.9}$$

where $F_{nn}^{(l)}$ is the number of floating point operations for the l^{th} layer of the full network, and $F_{ae}^{(l)}$ is the number of floating point operations for the l^{th} layer of the network augmented by the activation estimation network. The overall speedup is greatly dependent on the sparsity of the network and the overhead of the activation estimator.

3.2 Experiments

All hidden units are rectified-linear, and the output units are softmax trained with a negative log-likelihood loss function. The weights, w, are initialized by (Glorot and Bengio, 2010) and the biases are initialized to zero. In all experiments, the dropout probability p is fixed to 0.5 for the hidden layers. The learning rate γ is scheduled such that every 50 epochs, the learning rate is multiplied by 0.9.

Table 3.1: Hyperparameters for SVHN and MNIST experiments.

	SVHN	**MNIST**
Architecture	1024-1500-700-400-200-10	784-1000-600-400-10
Init Learning Rate	0.0001	0.0001
Learning Rate Scaling	0.9	0.9
Input Dropout	0.2	0.3
Dropout	0.5	0.5
Optimization	ADAM	ADAM

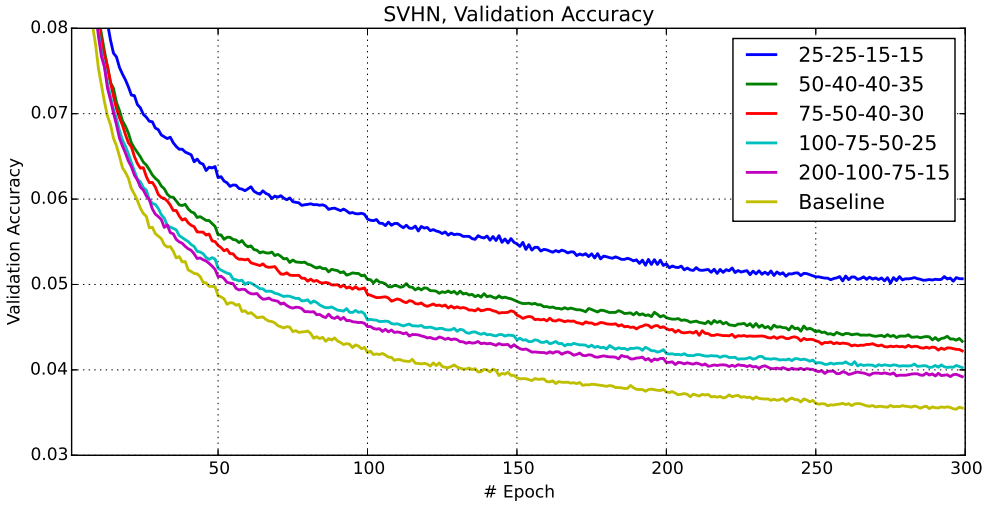

Figure 3.4: Classification error of the validation set for SVHN on seven configurations of the activation estimator for each hidden layer. The 'control' network has no activation estimator and is used as a baseline of comparison for the other networks. The legend is sorted by final validation set error (highest to lowest).

To simplify prototyping, the feed-forward is calculated for a layer, and the activation estimator is immediately applied before the next layer activations are used. This is equivalent to bypassing the calculations for activations that are likely to produce zeros. In practice, re-calculating the SVD once per epoch for the activation estimator seems to be a decent tradeoff between activation estimation accuracy and computational efficiency, but this may not necessarily be true for other datasets.

3.2.1 Experimental Results - SVHN

Street View House Numbers (SVHN) (Netzer et al., 2011) is a large image dataset containing over 600,000 labelled samples of digits taken from street signs. Each sample is an RGB 32×32 (3072-dimensional) image. The dataset is normalized for the neural network by subtracting out the mean and dividing by the standard deviation for each of the 3072 input variables. 15% of the training set is held out for validation. The architecture was held fixed while the hyperparameters were chosen randomly over 30 runs using a network with no activation estimation. The hyperparameters of the neural network with the lowest resulting validation error were then used for all experiments.

To evaluate the sensitivity of the model's performance as the rank of the activation estimator is varied, several parameterizations for the activation estimator are evaluated. Each network is trained with the hyperparameters in Table 3.1, and the results of six parameterizations are shown in Figure 3.4. Each parameterization is described by the rank of each approximation, e.g., '200-100-75-15' describes a network with an activation estimator using a 200-rank approximation for W_1, a 100-rank approximation for W_2, a 75-rank approximation for W_3, and a 15-rank approximation for W_4. Note that a low-rank approximation is not necessary for W_5 (the weights connecting the last hidden layer to the output layer), as we do not want to approximate the activations for the output layer.

Table 3.2 summarizes the test set error for the control and activation estimation networks. W_1 appears to be most sensitive, increasing the test set error from 7.9483% $\pm$ 0.1105% to 8.3405% $\pm$ 0.1425% when the rank of $\hat{W}_1$ is lowered from 100 to 75. The rank of $\hat{W}_4$ appears to be the least sensitive, as the 200-100-75-15 model performs much better than the 25-25-15-15, indicating that information loss has more of an impact in the lower layers.

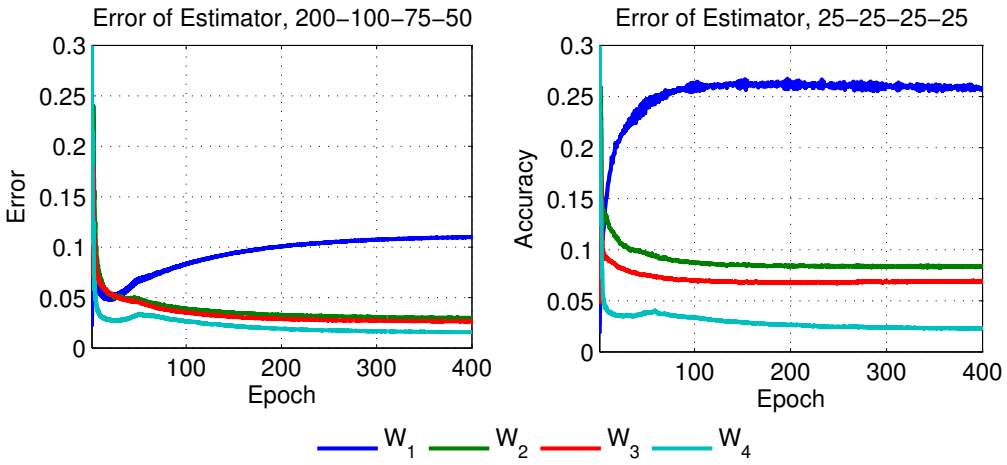

Figure 3.5: A comparison of a low-rank activation estimator and a higher-rank activation estimator. In this instance, a 25-25-25-25 activation estimator is too coarse to adequately capture the structure of the weight matrices.

Table 3.2: SVHN test set error averaged over ten runs, ($\pm$) indicates one standard deviation.. Note that the test set is drawn from a smaller (and much more difficult to classify) set of samples, so validation error is much less than the test error.

Network	**Error** (%)
Control	7.0079 $\pm$ 0.0572
200-100-75-15	7.7866 $\pm$ 0.0981
100-75-50-25	7.9483 $\pm$ 0.1105
75-50-40-30	8.3405 $\pm$ 0.1425
50-40-40-35	8.5084 $\pm$ 0.1307
25-25-15-15	9.7726 $\pm$ 0.1422

3.2.2 Experimental Results - MNIST

MNIST is a well-known dataset of hand-written digits containing 70,000 28×28 labelled images, and is generally split into 60,000 training and 10,000 testing examples. To normalize the data, the grayscale values $[0, 255]$ are divided by 128 and then subtracted by 1, resulting in a floating point representation in $[-1, 1]$. To select the hyperparameters, the training data is split into 50,000 samples for the training set and 10,000 samples for the validation set. The architecture is held fixed while the other hyperparameters were chosen randomly over 30 runs using a network with no activation estimation. The hyperparameters of the neural network with the lowest resulting validation error were then used for all experiments. Several parameterizations for the activation estimator are evaluated for a neural network trained with the hyperparameters listed in Table 3.1 using the same approach as the SVHN experiment above. The results for the validation set plotted against the epoch number are shown in Figure 3.6, and the final test set accuracy is reported in Table 3.3.

A neural network with a very low-rank weight matrix in the activation estimation can train well on MNIST. Lowering the rank from 784-600-400 to 50-35-25 impacts performance negligibly. Ranks as low as 25-25-25 does not lessen performance too greatly, and ranks as low as 15-10-5 yield a classifier capable of 1.525% error. Interestingly, the 10-10-5 run exhibits an initial decrease in classification error, followed by a gradual increase in classification error as training progresses. In the initial epochs, the hidden layer activations are perhaps more predictable, making the activation estimation a much simpler task for the initial epochs. Such a case is illustrated in Figure 3.5. However, as the pattern of the activation signs diversifies as the network continues to train, the lower-rank approximations begin to fail.

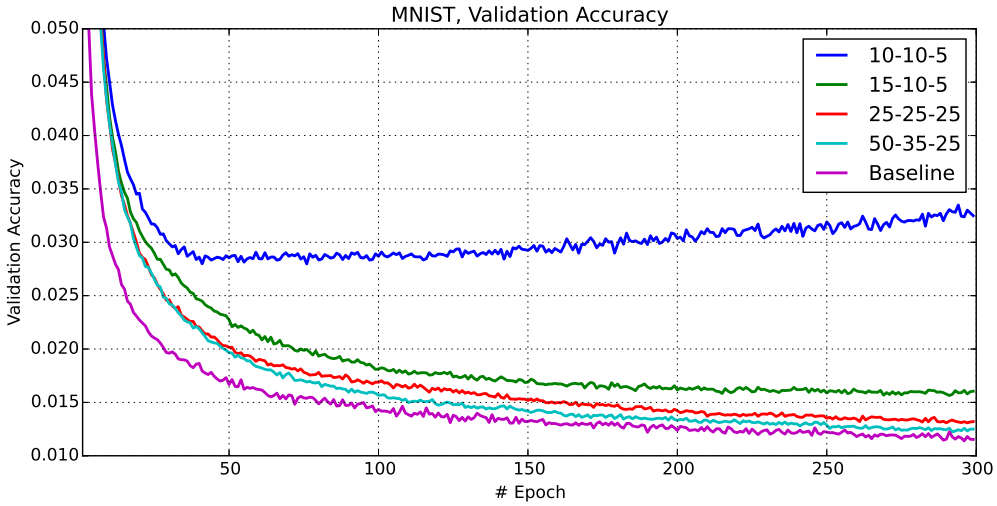

Figure 3.6: Classification error of the validation set for MNIST on five configurations of the activation estimator for each hidden layer. The legend is sorted by final validation set error (highest to lowest).

Table 3.3: MNIST test set error (percentage) averaged over 10 runs. ($\pm$) indicates one standard deviation.

Network	**Error** (%)
Control	1.1240 $\pm$ 0.0737
50-35-25	1.1500 $\pm$ 0.0671
25-25-25	1.2335 $\pm$ 0.0502
15-10-5	1.5245 $\pm$ 0.0698
10-10-5	3.2730 $\pm$ 0.2556

3.2.3 Conclusions

Low-rank estimations of weight matrices of a neural network obtained via once-per-epoch SVD work very well as efficient estimators of the sign of the activation for the next hidden layer. In the context of rectified-linear hidden units, computation time can be reduced greatly if this estimation is reliable and the hidden activations are sufficiently sparse. This approach is applicable to any hard-thresholding activation function, such as the functions investigated in Goroshin and LeCun (2013), and can be easily extended to be used with convolutional neural networks.

While the activation estimation error does not tend to deviate too greatly between minibatches over an epoch, as illustrated in Figure 3.2, this is not guaranteed. An online approach to the low-rank approximation would therefore be preferable to a once-per-epoch calculation. In addition, while the low-rank approximation given by SVD minimizes the objective function $\|A - \hat{A}_r\|_F$, this is not necessarily the best objective function for an activation estimator, where we seek to minimize $\|\sigma(aW) - \sigma(aW \cdot S)\|$, which is a much more difficult and non-convex objective function. Also, setting the hyperparameters for the activation estimator can be a tedious process involving expensive cross-validation when an adaptive algorithm could instead choose the rank based on the spectrum of the singular values. Therefore, developing a more suitable low-rank approximation algorithm could provide a promising future direction of research.

In Ba and Frey (2013), the authors propose a method called "adaptive dropout" by which the dropout probabilities are chosen by a function optimized by gradient descent instead of fixed to some value. This approach bears some resemblance to this paper, but with the key difference that the approach in Ba and Frey (2013) is motivated by improved regularization and this paper's method is motivated by computational efficiency. However, the authors introduce a biasing term that allows for greater sparsity that could be introduced into this paper's methodology. By modifying the conditional computation unit to compute $sgn(aUV - b)$, where b is

some bias, we can introduce a parameter that can tune the sparsity of the network, allowing for a more powerful trade-off between accuracy and computational efficiency.

Chapter 4

Conditional Computation in Recurrent Neural Networks

4.1 Gated Recurrent Unit

The gated recurrent unit (GRU) (Chung et al., 2014) is similar to the LSTM (Hochreiter and Schmidhuber, 1997), but it contains some simplifications to the recurrent structure of the unit. While the LSTM contains three gates (input gate, forget gate, and output gate), the GRU has two – a forget gate r_t, and an output gate z_t that is used to update the next hidden state h_t as a convex combination of the previous state h_{t-1} and a proposal state $\hat{h}_t$:

$$z_t = \sigma_z \left(W_z x_t + U_z h_{t-1} + b_z \right) \tag{4.1}$$

$$r_t = \sigma_r \left(W_r x_t + U_r h_{t-1} + b_r \right) \tag{4.2}$$

$$\hat{h}_t = h \left(W_h x_t + U_h \left(r_t \cdot h_{t-1} \right) + b_h \right) \tag{4.3}$$

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \hat{h}_t \tag{4.4}$$

where $\sigma_z(\cdot)$ and $\sigma_r(\cdot)$ are element-wise nonlinearities with a range of $(0, 1)$, $h(\cdot)$ is any elementwise nonlinearity, and $W_{(\cdot)}$, $U_{(\cdot)}$, and $b_{(\cdot)}$ are trainable parameters. The states are vectors $z_t, r_t, \hat{h}_t, h_t \in R^h$, the input vector $x_t \in R^d$, non-recurrent weights $W_{(\cdot)} \in R^{h \times d}$, and recurrent weights $U_{(\cdot)} \in R^{h \times h}$. A common selection for the gating functions $\sigma_z(\cdot)$ and $\sigma_r(\cdot)$ is the logistic sigmoid function $\frac{1}{1+e^{-x}}$, and a common selection for the hidden state proposal $h(\cdot)$ is the hyperbolic tangent.

4.2 Accelerating the Gated Recurrent Unit

Existing models of conditional computation rely on predicting sparsity in the activations to determine which portions of the neural network must be calculated. Sparsity in feed forward networks is typically encouraged by adding an ℓ_1 or a Kullback-Leibler penalty to the activations, which can encourage the rate of sparsity necessary for conditional computation, or can be encouraged much more mildly by dropout regularization. However, recent literature shows that dropout, and thus sparsity, must be carefully applied to recurrent units such as LSTMs. If dropout is not carefully applied, then the model suffers due to the corruption of information flow over many timesteps.

Instead of relying on the sparsity of the hidden activations to reduce computational burden, the model can be modified in order to rely on the sparsity of the gating activation z_t. In Eq. 4.1, when z_t approaches zero, the influence of the proposal state $\hat{h}_t$ diminishes, and the previous state h_{t-1} is passed forward to the next timestep. If z_t is exactly zero, then no computation is required for an element i of h_t^i, as the value of h_{t-1}^i can simply be copied forward.

In its current configuration, calculating z_t accounts for approximately $^1/_3$ of the floating point operations. If z_t had all zero values, the best case reduction in floating point operations would be only $^1/_3$, allowing for a 3x speed increase. In order to obtain greater acceleration, it is necessary to reduce the number of floating point operations required to compute z_t. Here, we introduce two methods, both of which

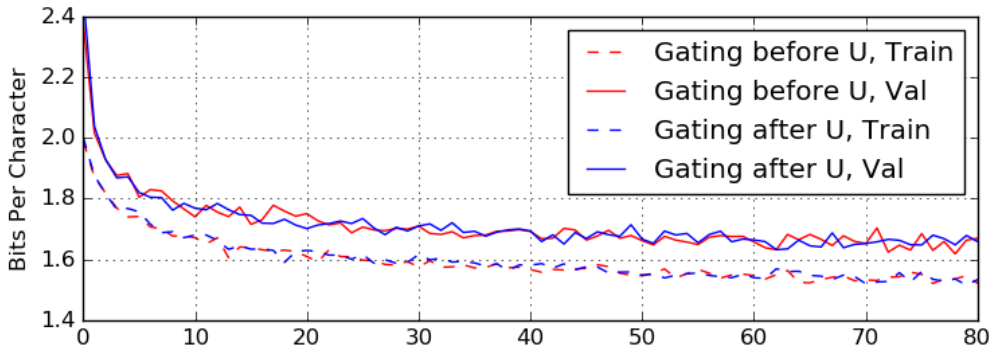

Figure 4.1: Demonstration of the lack of effect of moving the forget gate from $U\left(h_{t-1} \cdot r_t\right)$ to $r_t \cdot U\left(h_{t-1}\right)$.

can be seen as low-rank constraints on W_z and U_z, which reduce the computational requirements of matrix-vector or matrix-matrix operations.

In addition to imposing a low-rank constraint on W_z and U_z, a change to Eq. 4.3 must be made. Because the objective is to bypass the calculation of individual entries h_t^i of h_t, all elements associated with the computation h_t^i must be able to bypass as well. In the computation of the next state proposal $\hat{h}_t$, r_t is gates h_{t-1} prior to the linear transformation through U_h. Therefore, all elements of r_t must be computed in order to compute $\hat{h}_t$. To decrease the number of floating point operations further, a simple modification of 4.3 moves the forget gate r_t to the outside of the linear transformation:

$$\hat{h}_t = h\left(W_h x_t + r_t \cdot U_h h_{t-1} + b_h\right) \tag{4.5}$$

In this sense, the forget gate now gates the linear transformation $U_h h_{t-1}$ of the previous state h_{t-1} rather than the previous state itself, which should have no effect on the ability of the model to limit the transmission of information from previous timesteps, as demonstrated in Figure 4.1. With these modifications, the potential efficiency gains become primarily a function of the sparsity of the output gating z_t.

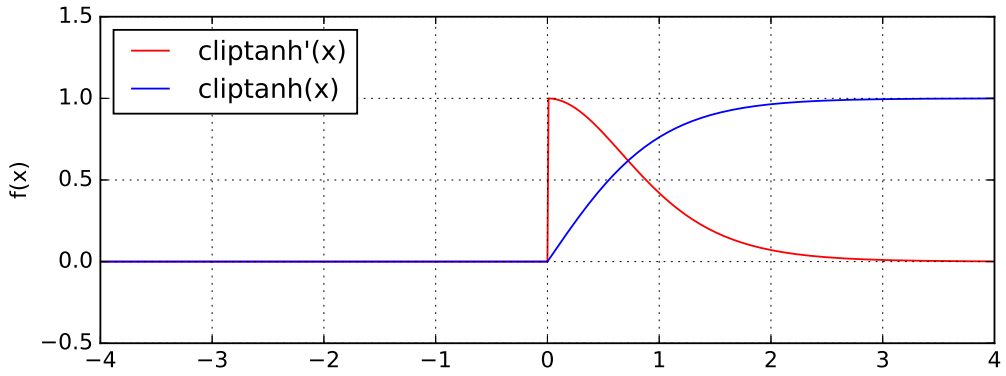

Figure 4.2: An illustration of the clipped tanh function and its derivative.

4.3 Constraining the Sparsity of z_t

Because the potential computational benefits are now mostly dependent on z_t, it is important to introduce a mechanism to control the sparsity of z_t. First, we propose an activation function in the range of $[0, 1]$ that produces values that reach zero, instead of merely approaching it in a limit. To this end, we propose replacing the activation function of z_t from a sigmoidal activation to a clipped hyperbolic tangent:

$$f(x) = \begin{cases} \tanh(x) & x > 0 \\ 0 & \text{else} \end{cases} \tag{4.6}$$

In the positive range where $x > 0$, the hyperbolic tangent has similar properties to the sigmoidal activation, in that it gradually saturates to a value of 1. On the negative end, however, it behaves like a rectifier, blocking any preactivation with a negative value from propagating forward (or backward, during backpropagation). This property allows for the simple induction of sparsity in z_t.

In order to control the level of sparsity in z_t, we propose a modification of batch normalization:

$$BN(z_t, s) = \frac{z_t - \mu}{\sigma^2} - s \tag{4.7}$$

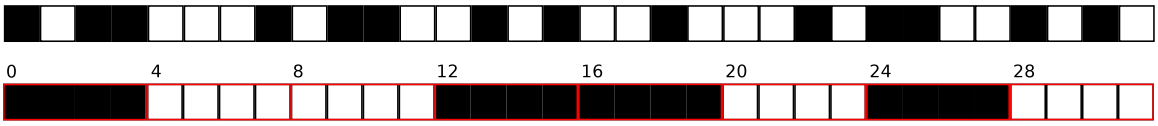

Figure 4.3: An illustration contrasting unstructured sparsity (above) with block-sparsity (below). In this case, the block-sparse representation is constrained to be sparse in contiguous chunks of length 4, and the sparsity pattern must align with the red outlines.

μ and σ^2 correspond to the mean and variance minibatch statistics. Instead of allowing for trainable vectors β and γ in the affine transformation, we have a hyperparameter s that allows for more direct control of the sparsity of z_t.

4.4 Block-Sparse Gating versus Unstructured Gating

In the implementation of conditional computation for the purposes of training models that will be faster at test time, the structure of the sparsity is a significant consideration. To this end, there are two types of sparsity that may be employed - that of unstructured sparsity and block sparsity. In the unstructured setting, there are no constraints imposed on the sparsity pattern of the z_t gating. In the block sparse setting, however, the sparsity pattern is constrained in the sense that contiguous sets of activations are active or inactive with respect to each other. Figure 4.3 illustrates the difference between the two types of sparsity. Given the difference between the nature of the computations, an unstructured gating is well suited to processing one sample at a time (e.g., a mobile phone processing a single voice stream), and block-sparse gating is well suited to processing several examples at a time (e.g., a server batch processing several examples at once.) Because BLAS libraries are very well optimized, we choose to implement the block-sparse and the unstructured gating with BLAS primitives.

4.4.1 Unstructured Gating

The unstructured gating is formulated with a targeted use case of single example processing, that is, instead of sending several examples in a minibatch through the GRU, only one sample is sent. Potential use cases involve real-time applications where there is only a single example that can be processed. In such cases, there is a significantly lower degree of parallelism that a CPU or GPU can exploit, so obtaining speed benefits simply by skipping the dot products between the input vector and the weight vectors corresponding to sparsified activations is relatively straightforward, and can be accomplished simply by copying the non-sparse weight vectors to some temporary storage, calling GEMV from the BLAS library, and writing the result back to the appropriate output elements, seen in the code listing in Figure A.1. In some cases, the copy operation implied by W[:,IDXS] may cause too much overhead. A simple C implementation of GEMV may run faster, especially when compiled with the Intel C compiler (ICC) with auto-vectorization and auto-threading enabled.

If the parameterization of z_t is left as-is, the greatest speed increase we could obtain is around 3x, as the calculation of z_t is roughly one-third of the operations in the GRU state update equation. If z_t outputs a zero vector, then the other two-thirds of the required computations may be skipped, resulting in the 3x speed increase. In order to raise this upper bound, we reparameterize the z_t update with a bottleneck layer. This is different from the approach outlined in Chapter 3, where the weights were parameterized as low-rank. In this case, we will project x_t and h_{t-1} to a lower-dimensional space g, apply a nonlinearity such as ReLU, and then expand this representation to the space corresponding to the hidden state dimensionality h:

$$z_t^{lr} = f\left(W_z^{lr} x_t + U_z^{lr} h_{t-1} + b_z^{lr}\right) \tag{4.8}$$

$$z_t = \sigma\left(BN\left(W_z z_t^{lr} + b_z\right)\right) \tag{4.9}$$

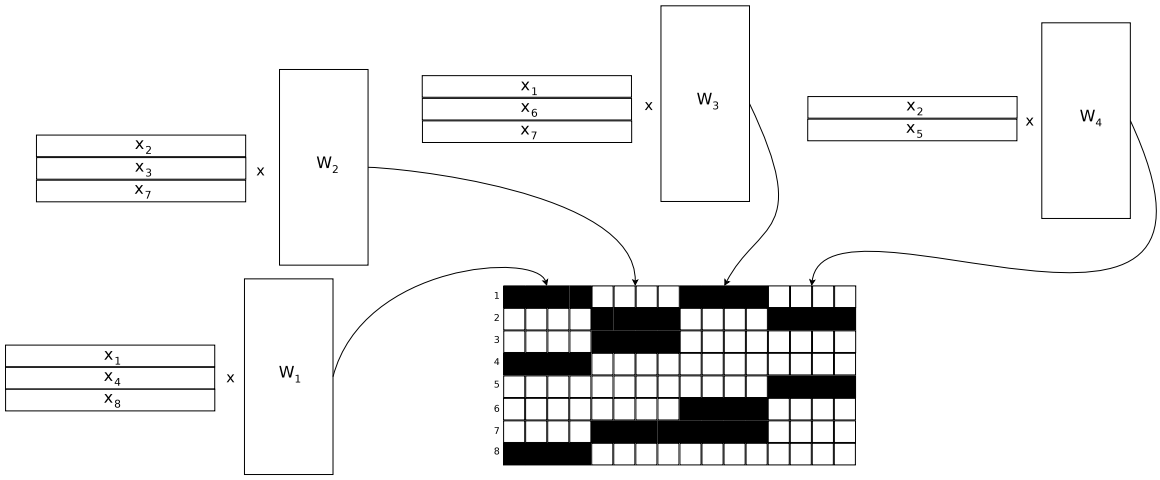

Figure 4.4: An illustration of the block-sparse multiplication recast as several matrix-matrix multiplications, given a sparsity mask.

where $BN(\cdot)$ may be either of the two batch normalization approaches introduced above. In reparameterizing z_t in such a way, we are increasing the upper bound of the speedup this approach can yield at the expense of the capacity or expressiveness of the gating function.

4.4.2 Block-Sparse Gating

The block-sparse gating is formulated with a targeted use case of batch processing. Unlike the unstructured gating, the block-sparse gating has the potential to be implemented with multiple matrix-matrix multiplications, as illustrated in Figure 4.4. The block sparse approach is also used in (Léonard, 2015) (Bengio et al., 2015) in order to exploit sparsity for speedup in feed forward networks. In such a formulation, the parameterization of z_t can be expressed exactly as in Equation 4.8 and Equation 4.9, except W_z and b_z are fixed to non-trainable values:

$$W_z = \begin{bmatrix} \mathbf{1}^T & \mathbf{0}^T & \cdots & \mathbf{0}^T \\ \mathbf{0}^T & \mathbf{1}^T & \cdots & \mathbf{0}^T \\ \vdots & \vdots & & \vdots \\ \mathbf{0}^T & \mathbf{0}^T & \cdots & \mathbf{1}^T \end{bmatrix}, b_z = 0 \tag{4.10}$$

where $\mathbf{0}^T, \mathbf{1}^T \in \mathbb{R}^{h/g}$ represent a h/g-dimensional vector of zeros and ones, respectively. In this fixed parameterization, the individual elements i of state $\left(z_t^{lr}\right)_i$ will be copied to a block-sparse vector:

$$z_t = \begin{bmatrix} \left(\mathbf{z_t^{lr}}\right)_\mathbf{1} & \left(\mathbf{z_t^{lr}}\right)_\mathbf{2} & \cdots & \left(\mathbf{z_t^{lr}}\right)_\mathbf{g} \end{bmatrix} \tag{4.11}$$

where $\left(\mathbf{z_t^{lr}}\right)_\mathbf{i}$ is defined as a h/g-dimensional vector with all entries equal to $\left(z_t^{lr}\right)_i$. Such a transformation can be implemented efficiently with a copy rather than a matrix multiplication, resulting in an operation that requires significantly fewer FLOPs than the fully trainable low-rank bottleneck approach.

4.5 Experiments

4.5.1 Character-Level Language Modeling and text8

In order to study the effects and speed benefits of the alternative GRU parameterizations, we evaluate the models as applied to language modeling on the TEXT8 dataset. In language modeling, the goal is to train a probabilistic model $p\left(x_t \mid \theta, x_{t-1}, \ldots, x_{t-n}\right)$ that estimates the probability of a token x_t occurring given some history of tokens $x_{t-1}, \ldots, x_{t-n}$. Tokens may be specific words in the case of word-level language modeling, or they may be individual characters in the case of character-level language modeling. There are benefits and drawbacks to either approach: handling massive vocabularies and out-of-vocabulary tokens in word-level modeling can pose challenges, but the extra parameters required to memorize particular words and the extra modeling effort required to handle sequences spanning longer-term dependencies make character level modeling less desirable in some applications.

The TEXT8 dataset consists of 10^8 bytes from an abbreviated and cleaned English Wikipedia dump. The dataset is stripped of all non-alphabetical characters such as XML markup, punctuation, and so forth. In the character-level modeling task, there are 27 tokens - lower-case a-z, as well as a 'space' token to provide separation between

Table 4.1: Hyperparameters for the block-sparse gated language models.

Architecture	27-1024-1024-27	27-1024-1024-27	27-1024-1024-27
Block Size	16	32	64
Gating Dim	64	32	16
Learning Rate	0.001		
Optimizer	ADAM		
z Gating Nonlinearity	CLIPTANH $(\cdot)$		
r Gating Nonlinearity	$\sigma(\cdot)$		
State Nonlinearity	$\tanh(\cdot)$		
$W_{(\cdot)}$ *Initialization*	Glorot		
$U_{(\cdot)}$ *Initialization*	Orthogonal		

individual words. Given the size of the corpus and the diversity of the content, TEXT8 is a common dataset to evaluate language modeling techniques. The first 95% of the dataset is used as training data, and the remaining 5% is used as validation data. The measure of performance on TEXT8 is given by the bits-per-character (BPC) metric.

Both the block sparse and the unstructured models are trained with truncated BPTT, backpropagating 50 timesteps per update while retaining the hidden state between sequences. The hidden state is reset every 1000 updates. Both models use a minibatch size of 64. The tokens are represented as one-hot vectors, making the input and output dimensionalities a size of 27. Both models have a softmax output and are trained with categorical cross entropy.

4.5.2 Conditional Models - Block Sparse

In order to evaluate the block-sparse approach and to understand how varying the blocks sizes and batch normalization biases impact the overall speed and accuracy of the models, we train twelve networks: the product of choices between the block size $bs = [16, 32, 64]$ and the biases $s = [0.00, -0.25, -0.50, -0.75]$. The training curves of the block-sparse models are given in Figures A.4, A.5, A.6, and A.7. The acceleration factors over a densely calculated baseline are given in Table 4.2.

Table 4.2: Block-sparse model acceleration factor over a fully-dense model. All entries in the table are averaged over 10 trials of 1000 feed-forwards.

Block Size / Gating Dim.	16/64	32/32	64/16
$s = 0.00$	0.73x	1.06x	1.20x
$s = -0.25$	1.88x	1.72x	1.68x
$s = -0.50$	1.72x	1.62x	1.68x
$s = -0.75$	1.57x	1.62x	1.95x

In general, as the s term becomes more negative, greater speedups are achievable. However, this comes at the cost of less accuracy in the models – as the s term is lowered from 0.00 to -0.25, the validation BPC raises from around 1.80 to around 1.90 for the 64/16 model. Lowering s to -0.5 reduces the BPC for the 16/64 and 32/32 models only slightly, but increases the BPC for the 64/16 model to approximately 1.95. Lowering s to -0.75 introduces instabilities into the validation accuracy and degrades the accuracy of the models significantly.

While the greatest achieved speedup is with the 64/16 model with $s = -0.75$, the most practical model is the 16/64 model with $s = -0.25$. This model realizes a good tradeoff between speed and accuracy, only marginally increasing the validation BPC while resulting in a model that runs around 1.88x faster than the baseline.

4.5.3 Conditional Models - Unstructured

Similar to the block sparse experiments, we train twelve networks on the product of choices between rank sizes $r = [16, 32, 64]$ and biases $s = [0.00, -0.25, -0.50, -0.75]$. The training curves of the unstructured models are given in Figures A.8, A.9, A.10, and A.11. The acceleration factors over a densely calculated baseline are given in Table 4.4.

In general, the unstructured models fit the data better, likely due to the less significant limitations placed on the gating units. In the unstructured model, all entries of z_t are free to change independently, whereas in the block-sparse model, all entries in z_t of a particular block are constrained to have the same value. As

Table 4.3: Hyperparameters for the unstructured sparsity gated language models.

Architecture	27-1024-1024-27	27-1024-1024-27	27-1024-1024-27
z Gating Rank	16	32	64
z Biases	$[0, -0.25, -0.50, -0.75]$		
Learning Rate	0.001		
Optimizer	ADAM		
z Gating Nonlinearity	CLIPTANH $(\cdot)$		
r Gating Nonlinearity	$\sigma(\cdot)$		
State Nonlinearity	$\tanh(\cdot)$		
$W_{(\cdot)}$ *Initialization*	Glorot		
$U_{(\cdot)}$ *Initialization*	Orthogonal		

Table 4.4: Unstructured sparsity model acceleration factor over a fully-dense model. All entries in the table are averaged over 10 trials of 1000 feed-forwards.

Rank Dimensionality	16	32	64
$s = 0.00$	1.55x	1.38x	1.43x
$s = -0.25$	1.87x	1.67x	1.63x
$s = -0.50$	1.86x	1.66x	1.99x
$s = -0.75$	2.55x	2.18x	2.29x

with the block-sparse results, we observe that as s decreases, the potential speedups increase. However, the unstructured parameterization appears to result in models that significantly and unstably overfit when $s \geq -0.50$.

4.5.4 Conclusions

With the block-sparse as well as the unstructured parameterizations, speedups of around 1.8x are possible, but require trading off accuracy compared to slower models. As z_t becomes more sparse as s becomes more negative, the hidden states are forced to pass through their previous activations instead of being allowed to produce new ones. This results in a model that can not react to rapid changes as well as a model with less sparse z_t gatings. Therefore, care must be taken when setting the s hyperparameter, as it is likely highly dependent on the target dataset. This problem is especially pronounced with the block-sparse model, where the z_t gatings are required to take on the same value for each particular block and cannot gate individual h_{t-1} activations.

In order to be more reduce the strains of this limitation, the block-sparse model could be modified to allow for individual gatings for the elements of z_t that are non-zero. Such a solution would allow for the block-sparsity that enables batch-wise conditional computation to be accelerated, while adding the individual gating behavior that gives unstructured sparsity an edge in BPC performance.

Chapter 5

Conclusions and Future Work

5.1 Summary of Contributions

The work presented here explored the application of conditional computation to feed-forward as well as recurrent neural networks. First, it was demonstrated that through low-rank decompositions, a gating mechanism can potentially decrease the number of required floating point operations to send a sample through a feed-forward neural network. Building on these principals, it was then shown that similar gating structures can be elegantly learned by backpropagation in recurrent neural networks. In the recurrent case, significant speedups were measured in two scenarios applied to a language modeling task: first, when the model processes only one example at a time, and second, when the model processes multiple samples at once in parallel.

5.2 Future Work

While this work is of relevance to scenarios where one must train fast models for deployment on resource-constrained environments, or environments where low latency in real-time conditions is a hard requirement, the investigation into models of conditional computation that practically accelerate backpropagation are an important

research direction. Initial results applying the block-sparse GRU model in feed-forward operation on GPUs were not promising, requiring impractically high sparsity and impractically large models in order to reach a break-even point between the conditional models and their dense counterparts. (Léonard, 2015) reports similar findings, noting that significant speed improvements were only possible in the sparse-to-sparse connections. Because the GRU states are not sparse, such a sparse-to-sparse approach would not be directly applicable.

Both the unstructured sparsity as well as the block-sparse approaches are both hindered by the surface-to-volume effect (Dongarra et al., 1989) where the ratio of computation to the number of elements in conditional computation is $\alpha\frac{n^3}{n^2}$ compared to the ratio in dense matrix-matrix multiplication of $\frac{n^3}{n^2}$, where α is the sparsity induced by conditional computation. As α approaches the minimum sparsity $\frac{1}{n}$, the ratio begins to resemble that of matrix-vector operations, rather than matrix-matrix operations, a task where GPUs reach a substantial percentage of peak FLOPs. Even in the block-sparse case where the sparsity is structured around allowing matrix-matrix operations, the operations are much smaller in terms of the number of samples as well as the output dimensionality, and therefore don't fully take advantage of the parallel capabilities of the GPU.

One potential way to accelerate recurrent neural networks with conditional computation on GPUs would lie in an alternative formulation where the surface-to-volume effect is less pronounced, and large matrix-matrix operations can be performed. While this approach could meet these criteria by using very large block and batch sizes, such block and batch sizes far exceed the current memory limitations even of high-end GPUs, and training such large models would be sure to overfit without aggressive regularization and training datasets far larger than are presently available.

5.3 Publications

Davis, A. and Arel, I. (2016). Faster Gated Recurrent Units via Conditional Computation. *International Conference on Machine Learning and Applications.* Submitted for publication, pending review.

Davis, A. and Arel, I. (2014). Low-rank approximations for conditional computation in deep neural networks. *International Conference on Learning Representations, Workshop Track.*

S. Young, I. Arel, A. Davis, A. Mishtal (2014). Hierarchical Spatiotemporal Feature Extraction using Recurrent Online Clustering. *Pattern Recognition Letters*

A. Davis, I. Arel (2010). On the Episode Duration Distribution Spanning Arbitrary States in Fixed-Policy Markov Decision Processes. *Proc. of the 23rd Florida Artificial Intelligence Research Society Conference*

S. Anuradha Bulusu, I. Arel, B. Arazi, A. Davis, G. Bitar (2010). A Data Security Protocol for the Trusted Truck System *Proc. 6th Annual Cyber Security and Information Intelligence Research Workshop*

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. 9

Arjovsky, M., Shah, A., and Bengio, Y. (2015). Unitary evolution recurrent neural networks. *arXiv preprint arXiv:1511.06464*. 20

Ba, J. and Frey, B. (2013). Adaptive dropout for training deep neural networks. In *Advances in Neural Information Processing Systems*, pages 3084–3092. 41

Bacon, P.-L., Bengio, E., Pineau, J., and Precup, D. (2015). Conditional computation in neural networks using a decision-theoretic approach. 26

Bengio, E., Bacon, P.-L., Pineau, J., and Precup, D. (2015). Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*. 26, 49

Bengio, Y. (2009). Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127. 9

Bengio, Y. (2013). Deep learning of representations: Looking forward. In Dediu, A.-H., Martn-Vide, C., Mitkov, R., and Truthe, B., editors, *Statistical Language*

and Speech Processing, volume 7978 of *Lecture Notes in Computer Science*, pages 1–37. Springer Berlin Heidelberg. 2, 11, 24, 25

Bengio, Y., Boulanger-Lewandowski, N., and Pascanu, R. (2013a). Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE. 17, 18

Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H., et al. (2007). Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153. 2, 10

Bengio, Y., Léonard, N., and Courville, A. C. (2013b). Estimating or propagating gradients through stochastic neurons for conditional computation. *CoRR*, abs/1308.3432. 29

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166. 17

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX. 9

Bertsekas, D. P. (1999). Nonlinear programming. 9, 11

Bishop, C. M. et al. (1995). Neural networks for pattern recognition. 21

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. (2013). One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*. 26

Cho, K. and Bengio, Y. (2014). Exponentially increasing the capacity-to-computation ratio for conditional computation in deep learning. *CoRR*, abs/1406.7362. 25, 26

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*. 18, 19, 43

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*. 12

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1337–1345. 2

Collobert, R., Bengio, Y., and Bengio, S. (2003). Scaling large learning problems with hard parallel mixtures. *International Journal of pattern recognition and artificial intelligence*, 17(03):349–365. 25

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376. 9

Cortes, C. and Vapnik, V. (1995). Support vector machine. *Machine learning*, 20(3):273–297. 7

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314. 9

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941. 14

Davis, A. and Arel, I. (2014). Low-rank approximations for conditional computation in deep neural networks. *International Conference on Learning Representations, Workshop Track*. 29

Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Senior, A., Tucker, P., Yang, K., Le, Q. V., et al. (2012). Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231. 15

Denil, M., Shakibi, B., Dinh, L., de Freitas, N., et al. (2013). Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156. 30, 31

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. (2014). Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277. 27, 30

Dongarra, J. J., Du Croz, J., Hammarling, S., and Duff, I. S. (1990). A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17. 27

Dongarra, J. J., Sorensen, D. C., and Hammarling, S. J. (1989). Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1):215–227. 28, 56

Eckart, C. and Young, G. (1936). The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218. 32

Eigen, D., Razanto, M., and Sutskever, I. (2014). Learning factored representations in a deep mixture of experts. *International Conference on Learning Representations*. 25

El Hihi, S. and Bengio, Y. (1995). Hierarchical recurrent neural networks for long-term dependencies. In *NIPS*, pages 493–499. Citeseer. 23

Elman, J. L. (1993). Learning and development in neural networks: The importance of starting small. *Cognition*, 48(1):71–99. 16

Gers, F. A., Schraudolph, N. N., and Schmidhuber, J. (2003). Learning precise timing with lstm recurrent networks. *The Journal of Machine Learning Research*, 3:115–143. 19

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256. 13, 35

Goodfellow, I. J., Bulatov, Y., Ibarz, J., Arnoud, S., and Shet, V. (2014). Multi-digit number recognition from street view imagery using deep convolutional neural networks. *International Conference on Learning Representations*. 11

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. (2013). Maxout networks. *arXiv preprint arXiv:1302.4389*. 22

Goroshin, R. and LeCun, Y. (2013). Saturating auto-encoders. *arXiv preprint arXiv:1301.3577*. 41

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE. 1

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm networks. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 4, pages 2047–2052. IEEE. 19

Gulcehre, C., Cho, K., Pascanu, R., and Bengio, Y. (2014). Learned-norm pooling for deep feedforward and recurrent neural networks. In *Machine Learning and Knowledge Discovery in Databases*, pages 530–546. Springer. 11

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1026–1034. 12

Hebb, D. O. (1949). *The organization of behavior: A neuropsychological approach*. John Wiley & Sons. 5

Hinton, G., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554. 2, 10

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*. 22, 24

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780. 18, 43

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366. 9

Huang, T., Lin, Z., Hailin, J., Yang, J., and Paine, T. (2014). Gpu asynchronous stochastic gradient descent to speed up neural network training. *International Conference on Learning Representations*. 15

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 448–456. 15

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. (1991). Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87. 25

Jaderberg, M., Vedaldi, A., and Zisserman, A. (2014). Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*. 27

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*. 24

Kahan, W. and Palmer, J. (1979). On a proposed floating-point standard. *ACM SIGNum Newsletter*, 14(si-2):13–21. 26

Kalchbrenner, N., Danihelka, I., and Graves, A. (2015). Grid long short-term memory. *arXiv preprint arXiv:1507.01526*. 19

Kelley, H. J. (1960). Gradient theory of optimal flight paths. *Ars Journal*, 30(10):947–954. 8

Kim, M. and Smaragdis, P. (2016). Bitwise neural networks. *arXiv preprint arXiv:1601.06071*. 26

Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114. 1, 12

Kumar, S., Mohri, M., and Talwalkar, A. (2009). Sampling techniques for the nystrom method. In *International Conference on Artificial Intelligence and Statistics*, pages 304–311. 1

Le, Q., Ranzato, M., Monga, R., Devin, M., Chen, K., Corrado, G., Dean, J., and Ng, A. (2012). Building high-level features using large scale unsupervised learning. In *International Conference in Machine Learning*. 2

Le, Q. V., Jaitly, N., and Hinton, G. E. (2015). A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*. 20

Lebedev, V., Ganin, Y., Rakhuba, M., Oseledets, I., and Lempitsky, V. (2014). Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*. 27

LeCun, Y. and Cortes, C. (2010). Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*. 26

LeCun, Y. A., Bottou, L., Orr, G. B., and Müller, K.-R. (2012). Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer. 13

Léonard, N. (2015). Distributed conditional computation. 26, 49, 56

Lin, Z., Courbariaux, M., Memisevic, R., and Bengio, Y. (2015). Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*. 26

Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7. 8

Liu, D. C. and Nocedal, J. (1989). On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528. 14

Maas, A. L., Hannun, A. Y., and Ng, A. Y. (2013). Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, page 1. 11, 12

Makhzani, A. and Frey, B. (2014). k-sparse autoencoders. *International Conference on Learning Representations*. 24

Martens, J. (2010). Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742. 13, 14

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

Mikolov, T., Deoras, A., Povey, D., Burget, L., and Cernocky, J. (2011). Strategies for training large scale neural network language models. In *Automatic Speech Recognition and Understanding (ASRU), 2011 IEEE Workshop on*, pages 196–201. IEEE. 1

Mikolov, T., Joulin, A., Chopra, S., Mathieu, M., and Ranzato, M. (2014). Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*. 19

Minsky, M. and Papert, S. (1969). Perceptrons. 6

Mohamed, A., Sainath, T. N., Dahl, G., Ramabhadran, B., Hinton, G. E., and Picheny, M. A. (2011). Deep belief networks using discriminative features for phone recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5060–5063. IEEE. 1

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 807–814. 11

Nath, R., Tomov, S., and Dongarra, J. (2010). An improved magma gemm for fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515. 2

Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376. 14

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 4. 37

Ngiam, J., Coates, A., Lahiri, A., Prochnow, B., Le, Q. V., and Ng, A. Y. (2011). On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 265–272. 22

Nguyen, D. and Widrow, B. (1990). Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. In *Neural Networks, 1990., 1990 IJCNN International Joint Conference on*, pages 21–26. IEEE. 13

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics. 7

Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2014). How to construct deep recurrent networks. *International Conference on Learning Representations*. 23

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1310–1318. 14

Recht, B., Re, C., Wright, S., and Niu, F. (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701. 15

Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386. 5

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*. 8

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2014). Imagenet large scale visual recognition challenge. 1

Russell, S., Norvig, P., and Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25:27. 4

Saxe, A. M., McClelland, J. L., and Ganguli, S. (2014). Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *International Conference on Learning Representations*. 13, 14

Schaul, T., Zhang, S., and LeCun, Y. (2012). No more pesky learning rates. *Journal of Machine Learning Research*, 28:343–351. 14

Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2014). Overfeat: Integrated recognition, localization and detection using convolutional networks. *International Conference on Learning Representations*. 24

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147. 14

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288. 21

Tikhonov, A. N. and Arsenin, V. Y. (1977). Solutions of ill-posed problems. 21

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560. 8, 17

Yam, J. Y. and Chow, T. W. (2000). A weight initialization method for improving training speed in feedforward neural network. *Neurocomputing*, 30(1):219–232. 13

Zeiler, M. D. (2012). Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*. 14

Zhang, S., Choromanska, A. E., and LeCun, Y. (2015). Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693. 15

Appendix

```
def blocksparse_gemv(W, x, z):
  # initialize output storage
  y = np.zeros(1, W.shape[1])

  # get indices of weight vectors
  idxs = z > 0

  # do matrix multiplication and indexing
  y[0,idxs] = x * W[:,idxs]

  return y
```

Figure A.1: An example of the unstructured gating implemented with a matrix-vector products and indexing operations in numpy syntax.

```
void cond_gemv(float *a, float *x, int *idx,
 float *y, int m, int n, int s, int clear_y)
{
  // clear y if requested
  if( clear_y == 1 )
  {
    memset(y, 0, sizeof(float) * n);
  }

  #pragma omp parallel
  for( int i=0; i < s; i++ )
  {
    int bias = idx[i] * n;

    float ytmp = 0;

    for( int j=0; j < n; j++ )
      ytmp += a[bias+j] * x[j];

    y[idx[i]] = ytmp;
  }
}
```

Figure A.2: A simple GEMV implemented in C. When compiled with ICC, performance is competitive with MKL's GEMV when applied to conditional computation.

```
def blocksparse_gemm(W, x, z, ngates):
  # initialize output storage
  y = np.zeros(x.shape[0], W.shape[1])

  # get gate dim to calculate indexing
  gatedim = W.shape[0] / ngates

  # iterate over gates
  for g in range(ngates):
    # calculate ranges for indexing y and W
    r = (g * gatedim, (g+1) * gatedim)

    # get samples to send through this gate
    idxs = z[:,g] > 0

    # do matrix multiplication and indexing
    y[idxs, r[0]:r[1]] = x[idxs,:] * W[:,r[0]:r[1]]

  return y
```

Figure A.3: An example of the block-sparse gating implemented with a matrix-matrix products and indexing operations in numpy syntax.

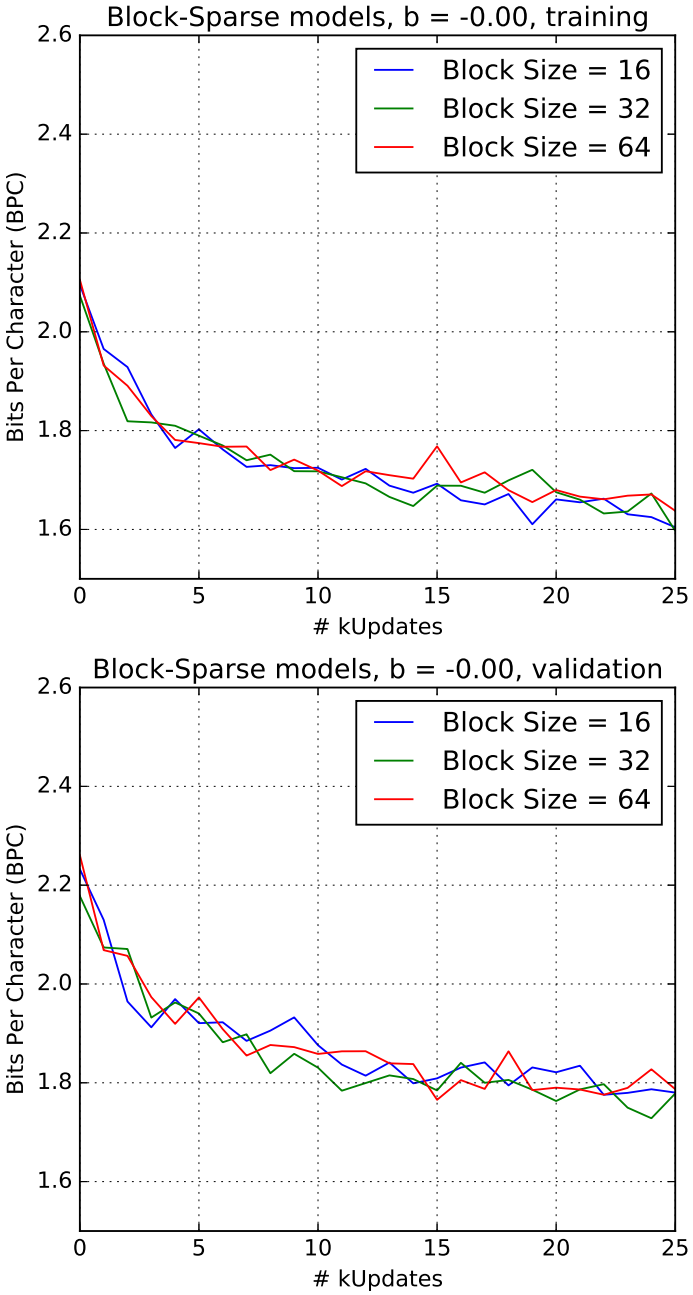

Figure A.4: A plot of the block-sparse model training and validation performances as measured in BPC as the models train. β for these models is 0.00.

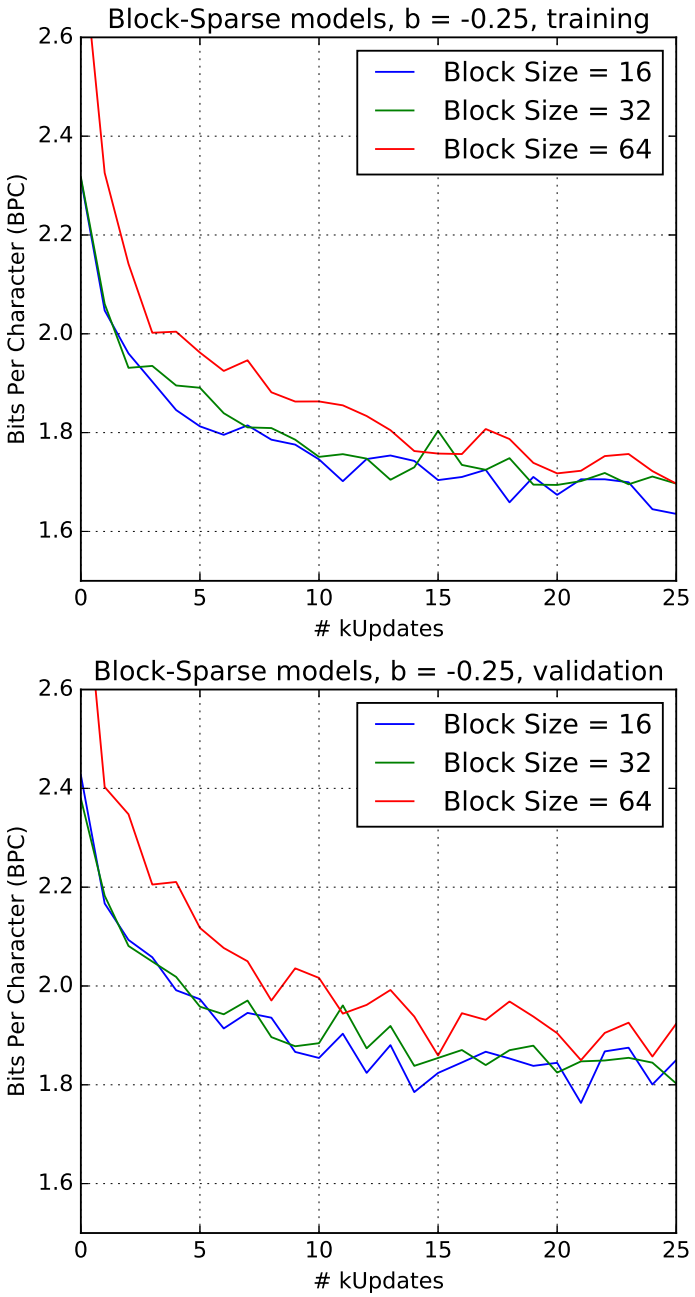

Figure A.5: A plot of the block-sparse model training and validation performances as measured in BPC as the models train. β for these models is -0.25.

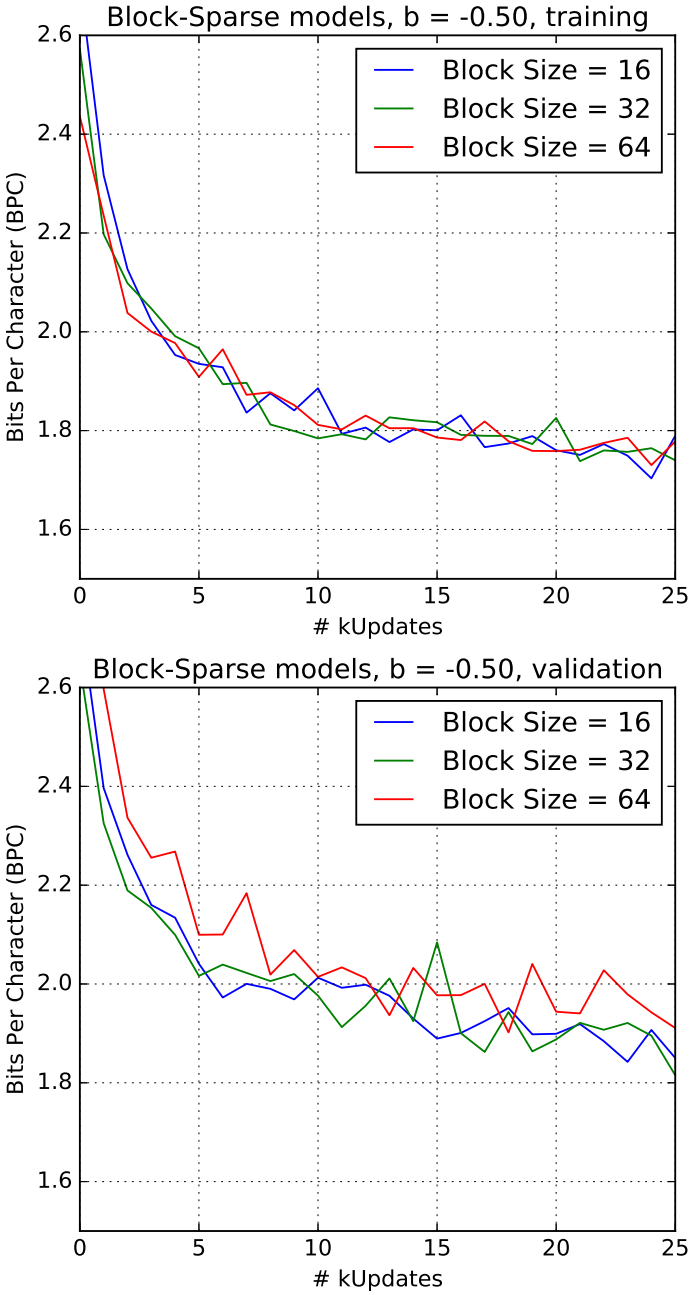

Figure A.6: A plot of the block-sparse model training and validation performances as measured in BPC as the models train. β for these models is -0.50.

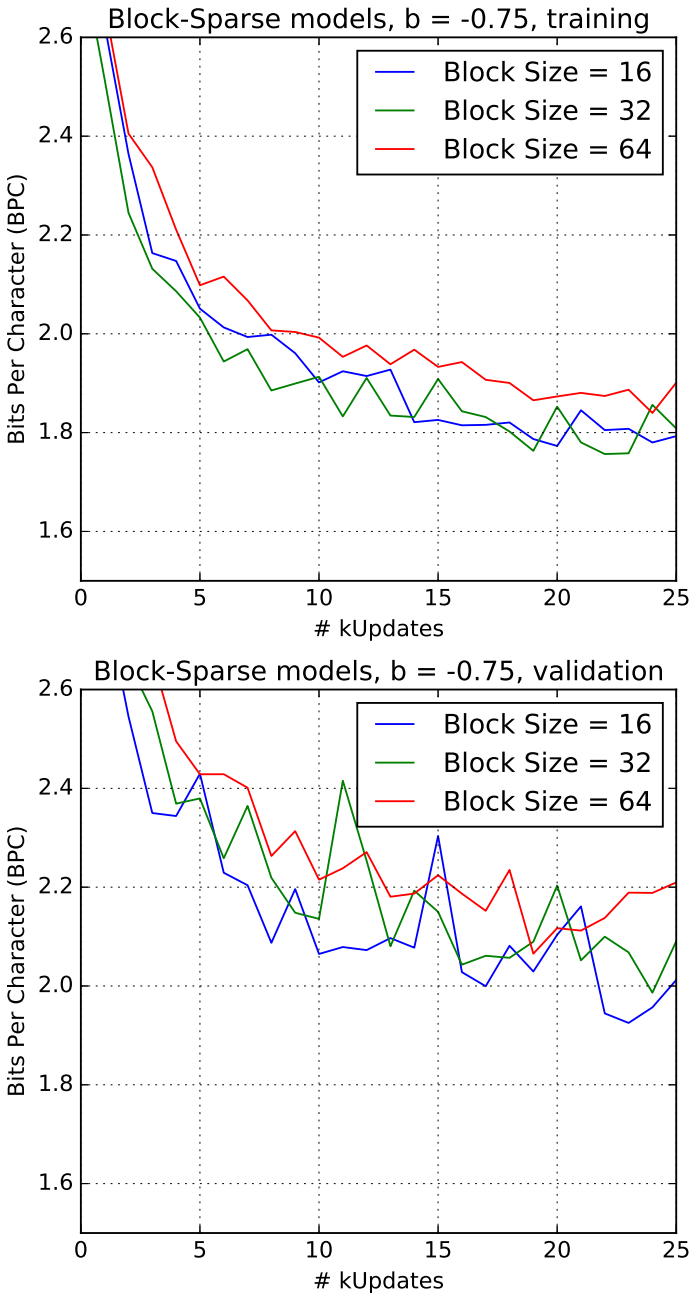

Figure A.7: A plot of the block-sparse model training and validation performances as measured in BPC as the models train. β for these models is -0.75.

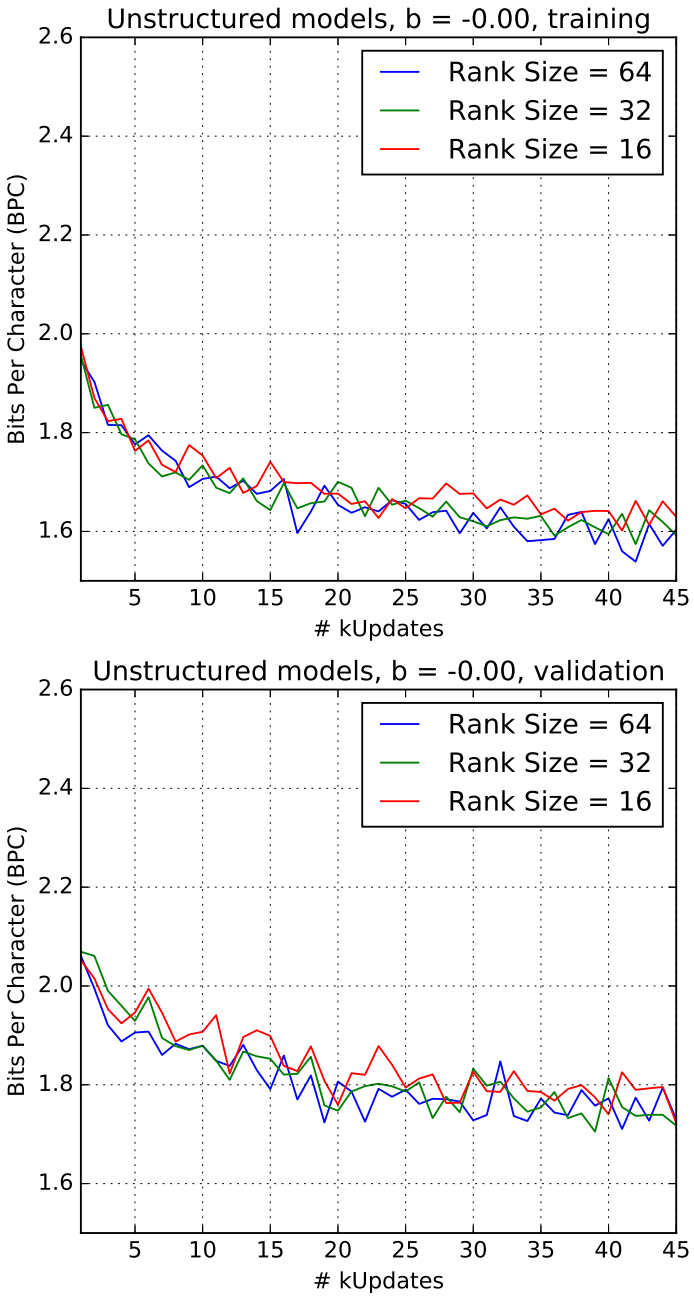

Figure A.8: A plot of the unstructured sparse model training and validation performances as measured in BPC as the models train. β for these models is 0.00.

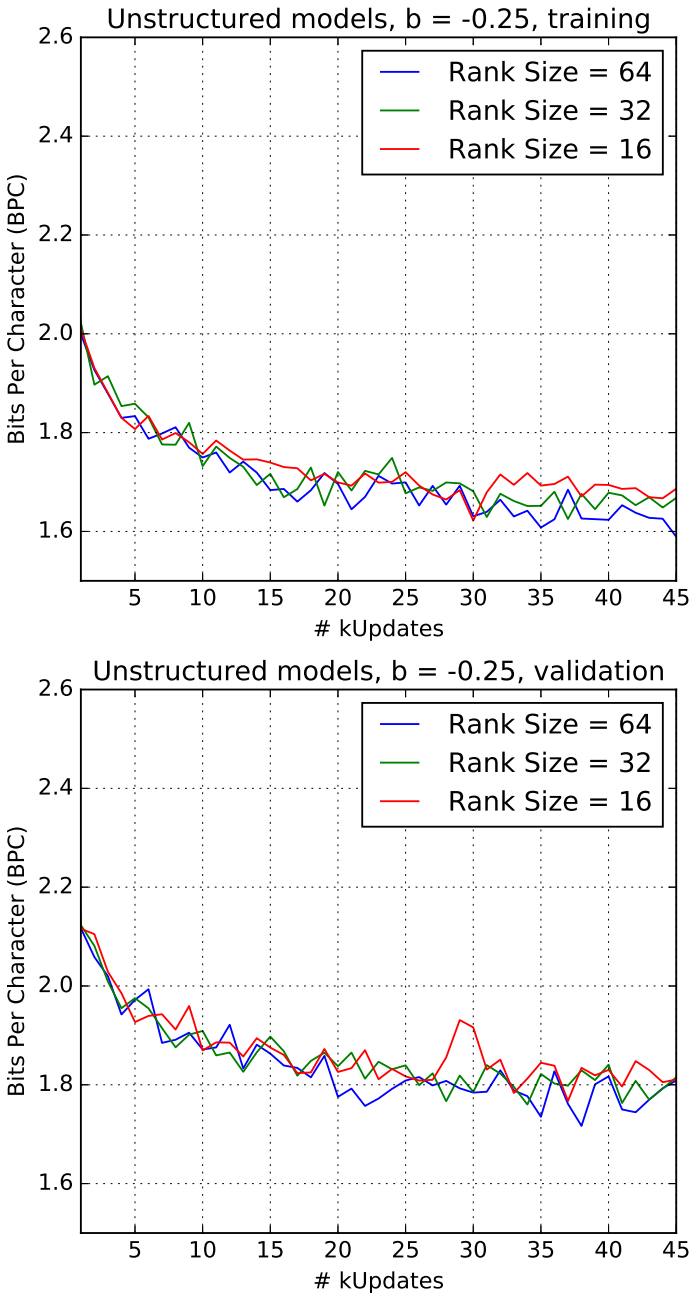

Figure A.9: A plot of the unstructured sparse model training and validation performances as measured in BPC as the models train. β for these models is -0.25.

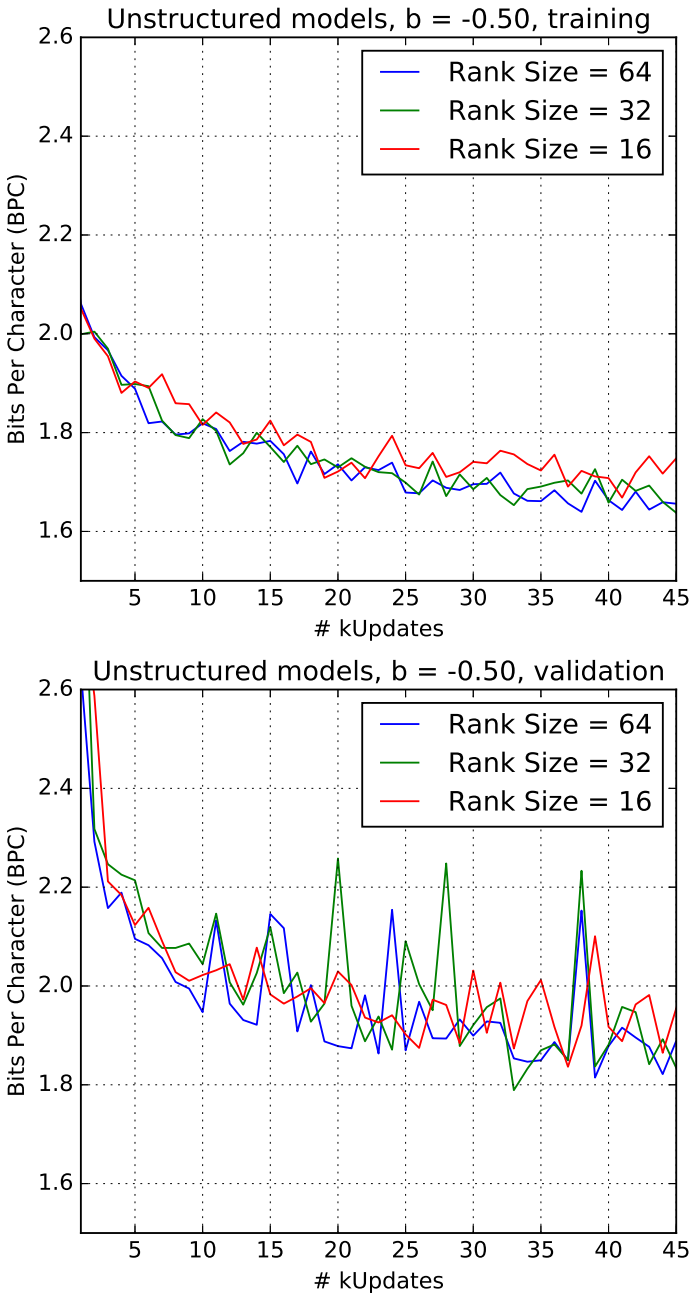

Figure A.10: A plot of the unstructured sparse model training and validation performances as measured in BPC as the models train. β for these models is -0.50.

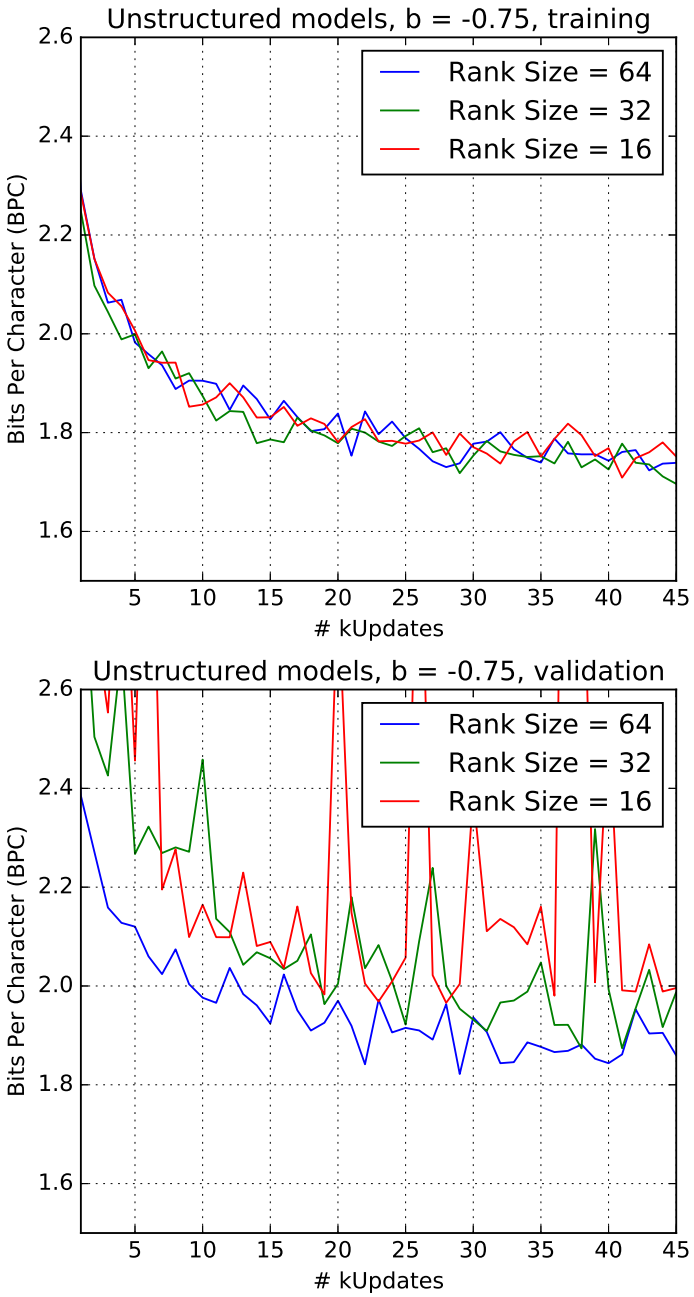

Figure A.11: A plot of the unstructured sparse model training and validation performances as measured in BPC as the models train. β for these models is -0.75.

Vita

Andrew Scott Davis was born on September 15 1988 to Jim and Patsy Davis of Brentwood, Tennessee. In 2006, he enrolled in the University of Tennessee as a computer engineering major. In the spring of 2009, he began working as a research assistant in Dr. Itamar Arel's Machine Intelligence Lab, sparking an immediate shift of interest from signal processing to machine learning. In 2010, Andrew graduated from the University of Tennessee with a BS in Computer Engineering, and decided to continue his graduate studies at the Machine Intelligence Lab. After spending several summers working for a machine learning startup, Andrew found himself in a summer internship at the Oak Ridge National Laboratory in 2013. After a chance encounter with a data scientist at a fast-growing cybersecurity startup, Andrew took an internship position in the summer of 2014, which led to a full-time position later that summer. After two years of working full-time and continuing work on his dissertation, Andrew completed his PhD in the summer of 2016.