A Critical Review of Recurrent Neural Networks for Sequence Learning

Zachary C. Lipton
University of California, San Diego
zlipton@cs.ucsd.edu

June 5th, 2015

Abstract

Countless learning tasks require dealing with sequential data. Image captioning, speech synthesis, music generation, and video game playing all require that a model generate sequences of outputs. In other domains, such as time series prediction, video analysis, and music information retrieval, a model must learn from sequences of inputs. Significantly more interactive tasks, such as natural language translation, engaging in dialogue, and robotic control, often demand both.

Recurrent neural networks (RNNs) are a powerful family of connectionist models that capture time dynamics via cycles in the graph. Unlike feedforward neural networks, recurrent networks can process examples one at a time, retaining a state, or *memory*, that reflects an arbitrarily long context window. While these networks have long been difficult to train and often contain millions of parameters, recent advances in network architectures, optimization techniques, and parallel computation have enabled large-scale learning with recurrent nets.

Over the past few years, systems based on state of the art long short-term memory (LSTM) and bidirectional recurrent neural network (BRNN) architectures have demonstrated record-setting performance on tasks as varied as image captioning, language translation, and handwriting recognition. In this review of the literature we synthesize the body of research that over the past three decades has yielded and reduced to practice these powerful models. When appropriate, we reconcile conflicting notation and nomenclature. Our goal is to provide a mostly self-contained explication of state of the art systems, together with a historical perspective and ample references to the primary research.

1 Introduction

Recurrent neural networks (RNNs) are a superset of feedforward neural networks, augmented with the ability to pass information across time steps. They

are a rich family of models capable of nearly arbitrary computation. A well-known result by Siegelman and Sontag from 1991 demonstrated that a finite sized recurrent neural network with sigmoidal activation functions can simulate a universal Turing machine [52]. In practice, the ability to model temporal dependencies makes recurrent neural networks especially suited to tasks where input and/or output consist of *sequences* of points that are not independent.

1.1 Why Recurrent Nets?

In this section, we address the fundamental reasons why recurrent neural networks warrant serious study for modeling sequential input and output. To be clear, we are motivated by a desire to achieve empirical results. This warrants clarification because recurrent nets have roots in both cognitive modeling and supervised machine learning, and owing to this difference of perspectives, many of these papers have different aims and priorities. In the foundational papers, generally published in cognitive science and computational neuroscience journals ([32], [33], [17]), biologically plausible mechanisms are emphasized. In other papers ([51], [55], [34]), biological inspiration is downplayed in favor of achieving empirical results on important tasks and datasets. Given the empirical aim, we now address three significant questions one might reasonably want answered before reading further.

Why Model Time Explicitly?

In light of the practical success and economic value of time-agnostic models, this is a fair question. Support vector machines, logistic regression, and feedforward networks have proved immensely useful without explicitly modeling time. Arguably, it is precisely the assumption of independence that has led to much recent progress in machine learning. Further, many models implicitly capture time by concatenating each input with some number of its immediate predecessors and successors, presenting the machine learning model with a sliding window of context about each point of interest. This approach has been used with deep belief nets for speech modeling in [42].

Unfortunately, despite the usefulness of the independence assumption, it precludes modeling long-range time-dependencies. For example, a model trained using a finite-length context window of length 5 could never be trained to answer the simple question, *"what was the data point seen* 10 *time steps ago?"* For a practical application such as call center automation, such a limited system might learn to route calls, but could never participate in an extended dialogue. Since the earliest conception of artificial intelligence, we have sought to build systems that interact with humans in time. In Alan Turing's groundbreaking paper *Computing Machinery and Intelligence*, he proposes an "Imitation Game" which judges a machine's intelligence by its ability to convincingly engage in dialogue [63]. Besides dialogue systems, modern interactive systems of economic importance include self-driving cars and robotic surgery, among others. Ignoring

an explicit model of time, it seems unlikely that any combination of classifiers or regressors can be cobbled together to provide this functionality.

Why Neural Networks and Not Markov Models?

Recurrent neural networks are not the first models to capture time dependencies. Markov chains, which model transitions between observed sequences of states ($s^{(1)}$, $s^{(2)}$, ... , $s^{(T)}$), were first described by mathematician Andrey Markov in 1906. Hidden Markov models (HMMs), which model observed data $(o^{(1)}, o^{(2)}, ..., o^{(T)})$ as probabilistically dependent upon unobserved states, were described in the 1950s and have been widely studied since the 1960s. However, traditional Markov model approaches are limited because their states must be drawn from a modestly sized discrete state space $s_j \in S$. The Viterbi algorithm, which is used to perform efficient inference on hidden Markov models, scales in time $O(|S|^2)$. Further, the transition table capturing the probability of moving between any two adjacent states is of size $|S|^2$. Thus, standard operations are infeasible with an HMM when the set of possible hidden states is larger than roughly 10^6 states. Further, each hidden state $s^{(t)}$ can depend only on the previous state $s^{(t-1)}$. While it is possible to extend any Markov model to account for a larger context window by creating a new state-space equal to the cross product of the possible states at each time in the window, this procedure grows the state space exponentially with the size of the window, rendering Markov models computationally impractical for modeling long-range dependencies.

Given the limitations of Markov models, we ought to explain why it is sensible that connectionist models, i.e., artificial neural networks, should fare better. First, recurrent neural networks can capture long-range time dependencies, overcoming the chief limitation of Markov models. This point requires a careful explanation. As in Markov models, any state in a traditional RNN depends only on the current input as well as the state of the network at the previous time step [1]. However, the hidden state at any time step can contain information from an arbitrarily long context window. This is possible because the number of distinct states that can be represented in a hidden layer of nodes grows exponentially with the number of nodes in the layer. Even if each node took only binary values, the network could represent 2^N states where N is the number of nodes in the hidden layer. Given real-valued outputs, even assuming the limited precision of 64 bit numbers, a single hidden layer of nodes can represent 2^{64^N} distinct states. While the potential expressive power grows exponentially with the number of nodes in the hidden representation, the complexity of both inference and training grows only quadratically.

Second, it is generally desirable to extend neural networks to tackle any supervised learning problem because they are powerful learning models that achieve state of the art results in a wide range of supervised learning tasks.

[1] Bidirectional recurrent neural networks (BRNNs) [51] extend RNNs to model dependence on both past states and future states. Traditional RNNs only model the dependence of each event on the past. This extension is especially useful for sequence to sequence learning with fixed length examples.

Over the past several years, storage has become more affordable, datasets have grown far larger, and the field of parallel computing has advanced considerably. In the setting of such large high-dimensional datasets, simple linear models under-fit and often underutilize computing resources. Deep learning methods, in particular those based on *deep belief networks* (DNNs), which are greedily built by stacking restricted Boltzmann machines, and convolutional neural networks, which exploit the local dependency of visual information, have demonstrated record-setting results on many important applications. Neural networks are especially well-suited for machine perception tasks, where the raw underlying features are not individually informative. This success is attributed to their ability to learn hierarchical representations, unlike traditional algorithms, which rely upon hand-engineered features. However, despite their power, feedforward neural nets have limitations. Most notably, they rely on the assumption of independence among the data points. Additionally, these networks generally rely on input consisting of fixed length vectors. Thus it is sensible to extend these powerful learning tools to model data with temporal structure, especially in the many domains where neural nets are already the state of the art.

Are RNNs Too Expressive?

As described earlier, finite-sized RNNs with sigmoidal activations are Turing complete. The capability of RNNs to run arbitrary computation clearly demonstrates their expressive power, but one could argue that the C programming language is equally capable of expressing arbitrary programs. And yet there are no papers claiming that the invention of C represents a panacea for machine learning. Out of the box, C offers no simple way of efficiently exploring the space of programs. There is no straightforward way to calculate the gradient of an arbitrary C program to minimize a chosen loss function. Further, the biggest problem with treating the set of programs expressible in C as a family of machine learning models is that this set is far too large. Given any finite sized dataset, there exist countless programs which can overfit the data, generating desired output but failing to generalize to test data.

Why then should RNN's not suffer from similar problems? First, given any fixed architecture (set of nodes, edges, and activation functions), the recurrent neural networks described in this paper are fully differentiable end to end. The derivative of the loss function can always be calculated with respect to each of the parameters (weights) in the model. Second, while the Turing-completeness of finite-length RNNs is an impressive property, given any fixed-size RNN and a specific architecture, it is not actually possible to generate any arbitrary program. Further, unlike an arbitrary program composed in C, a recurrent neural network can be regularized via standard techniques such as weight decay, dropout, and limiting the degrees of freedom.

1.2 Comparison to Prior Work

The literature on recurrent neural networks can seem impenetrable to the uninitiated. Shorter papers assume familiarity with a large body of background literature. Diagrams are frequently underspecified, failing to indicate which edges span time steps and which don't. Worse, jargon abounds while notation is frequently inconsistent across papers or overloaded within papers. Readers are frequently in the unenviable position of having to synthesize conflicting information across many papers in order to understand but one. For example, in many papers subscripts index both nodes and time steps. In others, h simultaneously stands for link functions and a layer of hidden nodes. The variable t simultaneously stands for both time indices and targets, sometimes in the same equation. Many terrific breakthrough papers have appeared recently, but clear reviews of recurrent neural network literature are rare.

Among the most useful resources are Alex Graves' 2012 book on supervised sequence labelling with recurrent neural networks [23] and Felix Gers' doctoral thesis [18]. More recently, [14] covers recurrent neural nets for language modeling. Other resources focus on a specific technical aspect such as [48], which surveys gradient calculations in recurrent neural networks. In this review paper, we aim to provide a readable, intuitive, and consistently notated review on recurrent neural networks for sequence learning. We emphasize models, algorithms, and results, but focus equally on distilling the intuitions that have guided this largely heuristic and empirical field. In addition to concrete modeling details, we offer qualitative arguments, a historical perspective, and comparisons to alternative methodology where appropriate.

2 Background

Here we introduce formal notation and provide a brief background on neural networks.

2.1 Time

To be clear, RNNs are not limited to sequences which index time. They have been used successfully on non-temporal sequence data, including genetic data [3]. However, computation proceeds in time, and many important applications have an explicit or implicit temporal aspect. While we refer to time throughout this paper, the methods described here are applicable to wider family of tasks.

In this paper, by time, we refer to data points $\boldsymbol{x}^{(t)}$ that arrive and desired outputs $\boldsymbol{y}^{(t)}$ that are generated in a discrete *sequence* of *time steps* indexed by t. We use superscripts with parentheses and not subscripts to obviate confusion between time steps and neurons. Our sequences may be of finite length or countably infinite. When they are finite, we call the maximum time index of the sequence T. Thus a sequence of consecutive inputs can be denoted $(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(T)})$ and outputs can be notated $(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)}, ..., \boldsymbol{y}^{(T)})$ These time

steps may be equally spaced samples from a continuous real-world process. Examples would include the still images that comprise the frames of videos or the discrete amplitudes sampled at fixed intervals to comprise audio recordings. The time steps may also be ordinal, with no exact correspondence to durations. In fact, these techniques can be extended to domains including genetic sequences, where the sequence has a defined order but no real correspondence to time. This is the case with natural language. In the word sequence "John Coltrane plays the saxophone", $\boldsymbol{x}^{(1)}$ = John, $\boldsymbol{x}^{(2)}$ = Coltrane, etc.

2.2 Neural Networks

Neural networks are biologically inspired models of computation. Generally, a neural network consists of a set of *artificial neurons*, commonly referred to as *nodes* or *units*, and a set of directed edges between them, which intuitively represent the *synapses* in a biological neural network. Associated with each neuron j is an activation function l_j, which is sometimes called a link function. We use the notation "l_j" and not "h_j" (unlike some other papers) to distinguish the activation function l_j from the values of the hidden nodes in a network, which is commonly notated $\boldsymbol{h}$ in the RNN literature.

Associated with each edge from node j' to j is a weight $w_{jj'}$. Following the convention adopted in several foundational recurrent net papers ([31], [18], [20], [61]), we index neurons with j and j', and by $w_{jj'}$, we denote the weight corresponding to the directed edge from node j' to node j. It is important to note that in many papers, textbooks, and lecture notes, the indices are flipped and $w_{j'j} \neq w_{jj'}$ denotes the weight on the directed edge from the node j' to the node j as in [16] and on Wikipedia [67].

The value v_j of each neuron j is calculated by applying its activation function to a weighted sum of its inputs (Figure 1):

$$v_j = l_j \left(\sum_{j'} w_{jj'} \cdot v_{j'} \right).$$

For convenience, we term the weighted sum inside the parenthesis the *incoming activation* and notate it as a_j. We represent this entire process in figures by depicting neurons as circles and edges as arrows connecting them. When appropriate, we mark the exact activation function with a symbol, e.g., σ for sigmoid.

Common choices for the activation function include the sigmoid $\sigma(z) = 1/(1 + e^{-z})$ and the *tanh* function $\phi(z) = (e^z - e^{-z})/(e^z + e^{-z})$ which has become common in feedforward neural nets and was applied to recurrent nets in [61]. Another activation which has become the state of the art in deep learning research is the rectified linear unit (ReLU) $l_j(z) = \max(0, z)$. These units have been demonstrated to improve the performance of many deep neural networks ([42], [46], [71]) on tasks as varied as speech processing and object recognition, and have been used in recurrent neural networks by [6].

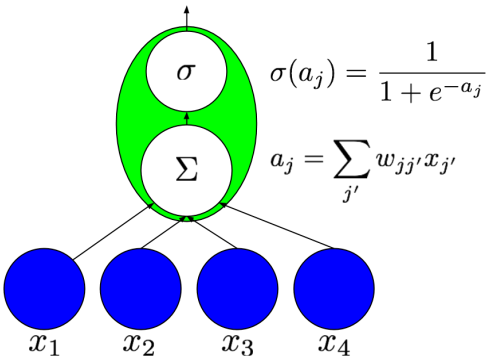

Figure 1: An artificial neuron computes a nonlinear function of a weighted sum of its inputs.

The activation function at the output nodes depends upon the task. For multi-class classification, we apply a softmax nonlinearity to the layer. The softmax function calculates

$$\hat{y}_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}$$

where K is the total number of outputs. The denominator is normalization consisting of a sum of exponentials over all output nodes, ensuring that the output sums to 1. For multilabel classification the activation function is simply a point-wise sigmoid. For regression we may have linear output. Due to the overwhelming number of current applications involving multi-class classification, especially for recurrent nets, in this paper, unless otherwise specified, we assume that softmax is applied at the output.

2.3 Feedforward Neural Networks

With a neural model of computation, one must determine the order in which computation should proceed. Should nodes be sampled one at a time and updated, or should the value of all nodes be calculated at once and then all updates applied simultaneously? Feedforward neural networks (Figure 2) are a restricted class of neural networks which deal with this problem by forbidding cycles in the graph. Thus all nodes can be arranged into layers. The outputs in each layer can be calculated given the outputs from the lower layers.

The input $\boldsymbol{x}$ to a feedforward network is presented by setting the values of the lowest layer. Each higher layer is then successively computed until output is generated at the topmost layer $\hat{\boldsymbol{y}}$. These networks are frequently used for supervised learning tasks such as classification and regression. Learning is accomplished by iteratively updating each of the weights to minimize a loss

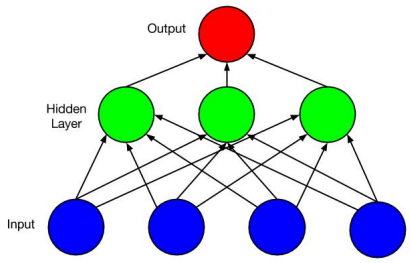

Figure 2: A feedforward neural network. An example is presented to the network by setting the values of the blue nodes. Each layer is calculated successively as a function of the prior layers until output is produced at the topmost layer.

function, $\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y})$, which penalizes the distance between the output $\hat{\boldsymbol{y}}$ and the target $\boldsymbol{y}$. Backpropagation, an algorithm introduced to neural networks in [50], computes the gradient of the loss with respect to each parameter using the chain rule. While the optimization surfaces for neural networks are highly non-convex, and exact optimization is known to be an NP-Hard problem, a large body of work on heuristic pre-training and optimization techniques has led to impressive empirical success on many supervised learning tasks. Convolutional neural networks, developed by Yann LeCun, [37] are a variant of feedforward neural network that, since 2012, hold records in many computer vision tasks, such as object detection [35].

Feedforward networks, however, are limited. After each example is processed, the entire state of the network is lost. If each data point is independently sampled, this presents no problem. But if data points are related in time, this is unacceptable. Frames from video, snippets of audio, and words pulled from sentences, represent settings where the independence assumption fails.

2.4 Training Neural Networks via Backpropagation

The most successful algorithm for training neural networks is backpropagation, introduced to neural networks by Rumelhart et al. in 1985 [50]. Backpropagation uses the chain rule to calculate the derivative of a loss function $\mathcal{L}$ with respect to each parameter in the network. The weights are then adjusted by gradient descent. Because the loss surface is non-convex, there is no assurance that backpropagation will reach a global minimum. However, in practice, networks trained with backpropagation and gradient following techniques have been

remarkably successful. In practice, most networks are trained with stochastic gradient descent (SGD) using mini-batches. Here, w.l.o.g. we discuss only the case with batch size equal to 1. The stochastic gradient update equation is given by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} F_i$$

where η is the learning rate and $\nabla_{\boldsymbol{w}} F_i$ is the gradient of the objective function with respect to the parameters $\boldsymbol{w}$ as calculated on a single example (x_i, y_i).

In practice, many variants of SGD are used to accelerate learning. Some popular heuristics, such as AdaGrad [15], AdaDelta [70], and RMSprop [1], adaptively tune the learning rate for each feature. AdaGrad, arguably the most popular, adapts the learning rate by caching the sum of squared gradients with respect to each parameter at each time step. The step size for each feature is scaled to the inverse of this cache. This leads to fast convergence on convex error surfaces, but because the cached sum is monotonically increasing, AdaGrad has a monotonically decreasing learning rate. This may be undesirable on highly non-convex loss surfaces. RMSprop modifies AdaGrad by introducing a decay factor on the cache, transforming the monotonic growing cache into a moving average. Momentum methods are another common SGD variant used to train neural networks. These methods add to each update a decaying sum of the previous updates. When the momentum parameter is well-tuned and the network is initialized well, momentum methods can train deep nets and recurrent nets competitively with more computationally expensive methods like the Hessian Free optimizer [60].

To calculate the gradient in a feedforward neural network, backpropagation proceeds as follows. First, an example is forward propagated through the network to produce a value v_j at each node and outputs $\hat{\boldsymbol{y}}$ at the topmost layer. Then, a loss function $\mathcal{L}(\hat{y}_k, y_k)$ is assessed at each output node k. Subsequently, for each output node k, we can calculate

$$\delta_k = \frac{\partial \mathcal{L}(\hat{y}_k, y_k)}{\partial \hat{y}_k} \cdot l'_k(a_k).$$

Given these values δ_k, for each node in the level prior we can calculate

$$\delta_j = l'(a_j) \sum_k \delta_k \cdot w_{kj}.$$

This calculation is performed successively for each lower layer to calculate δ_j for every node j given the δ values for each node connected by an outgoing edge. Each value δ_j represents the derivative $\partial \mathcal{L}/a_j$ of the total loss function w.r.t. that node's *incoming activation*. Given the values v_j calculated on the forward pass, and the values δ_j calculated on the backward pass, the derivative of the loss $\mathcal{L}$ with respect a given parameter $w_{jj'}$ is given by

$$\frac{\partial \mathcal{L}}{\partial w_{jj'}} = \delta_j v_{j'}.$$

Large scale feedforward neural networks trained via backpropagation have set many large-scale machine learning records, most notably on the computer vision task of object detection ([38], [35]).

Several other methods have been explored for learning the weights in a neural network. Early work, including Hopfield nets [32], learned via a Hebbian principle but did not produce networks useful for discriminative tasks. A number of papers from the 1990s ([5], [26]) championed the idea of learning neural networks with genetic algorithms with some even claiming that achieving success on real-world problems by applying many small changes to a network's weights was impossible. Despite the subsequent success of backpropagation, interest in genetic algorithms persists. Several recent papers explore genetic algorithms for neural networks, especially as means of learning the architecture of neural networks, a problem not addressed by backpropagation ([4], [27]). By *the architecture* we mean the number of layers, the number of nodes in each, the connectivity pattern among the layers, the choice of activation functions, etc.

One open question in neural network research is how to exploit sparsity in training. In a neural network with sigmoidal or tanh activation functions, the nodes in each layer never take value exactly 0. Thus, even if the inputs are sparse, the nodes at each hidden layer are not. However, a rectified linear units (ReLUs) introduce sparsity to hidden layers [21]. In this setting, a promising path may be to store the sparsity pattern when computing each layer's values and use it to speed up computation of the next layer in the network. A growing body of recent work ([11], [36], [53], [39]), thus read shows that given sparse inputs to a linear model with any standard regularizer, sparsity can be fully exploited even if the gradient is not sparse (owing to regularization). Given sparse hidden layers, these approaches can be extended to the layer-wise computations in neural networks.

3 Recurrent Neural Networks

Recurrent neural networks are a strict superset of feedforward neural networks, augmented by the inclusion of recurrent edges that span adjacent time steps, introducing a notion of time to the model. While RNNs may not contain cycles among the conventional edges, recurrent edges may form cycles, including self-connections. At time t, nodes receiving input along recurrent edges receive *input activation* from the current example $\boldsymbol{x}^{(t)}$ and also from hidden nodes $\boldsymbol{h}^{(t-1)}$ in the network's previous state. The output $\hat{\boldsymbol{y}}^{(t)}$ is calculated given the hidden state $\boldsymbol{h}^{(t)}$ at that time step. Thus, input $\boldsymbol{x}^{(t-1)}$ at time $t-1$ can influence the output $\hat{\boldsymbol{y}}^{(t)}$ at time t by way of these recurrent connections.

We can show in two equations, all calculations necessary for computation at each time step on the forward pass in a simple recurrent neural network:

$$\boldsymbol{h}^{(t)} = \sigma(W_{hx}\boldsymbol{x} + W_{hh}\boldsymbol{h}^{(\boldsymbol{t-1})} + \boldsymbol{b}_h)$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(W_{yh}\boldsymbol{h}^{(t)} + \boldsymbol{b}_y)$$

Here W_{hx} is the matrix of weights between the input and hidden layers and W_{hh} is the matrix of recurrent weights between the hidden layers at adjacent time steps. The vectors $\boldsymbol{b}_h$ and $\boldsymbol{b}_y$ are biases which allow each node to learn an offset.

Most of the models discussed in this paper consist of networks with recurrent hidden layers. However, some proposed models, such as Jordan Networks, allow for connections between the outputs in one state and the hidden layer in the next. Others, such as Sutskever et al.'s model for sequence to sequence learning [62], compute the output at each time step and pass a representation of this information as the input at the following time step.

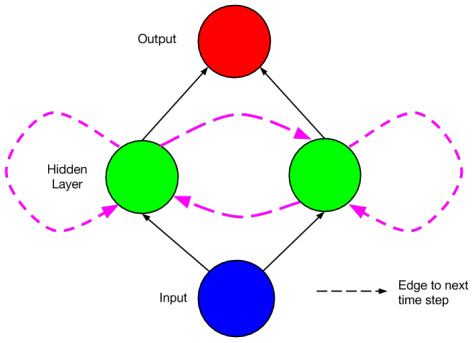

Figure 3: A simple recurrent network. At each time step t, activation is passed along solid edges as in a feedforward network. Dashed edges connect the source node j' at time t, i.e., $j'^{(t)}$ to the target node at the following time step $j^{(t+1)}$.

A simple recurrent network is depicted in Figure 3. The dynamics of this network across time steps can be visualized by *unfolding* the network (Figure 4). Given this picture, the model can be interpreted not as cyclic, but rather as a deep network with one layer per time step and shared weights across time steps. It's then clear that the unfolded network can be trained across many time steps using backpropagation. This algorithm is called *backpropagation through time* (BPTT), and was introduced in 1990 [66].

3.1 Past Approaches

Early foundational work on recurrent networks took in the 1980s. In 1982, Hopfield introduced a family of recurrent neural networks [32]. We'll avoid a prolonged discussion of these networks as they are not currently used for sequence learning. Hopfield's nets have some pattern recognition capabilities but offer no clear method for supervised training. These networks are defined

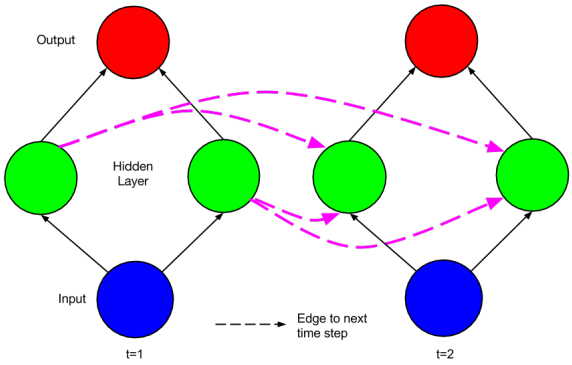

Figure 4: Visualizing the network unfolded across time steps.

by the values of the weights between nodes and the link functions are simple thresholding at 0. In these nets, a pattern is placed in the network by setting the values of the nodes. The network then runs for some time according to its update rules, and eventually a pattern is read out. The networks are useful for recovering a stored pattern from a corrupted version and are the forerunners of Boltzmann machines and auto-encoders.

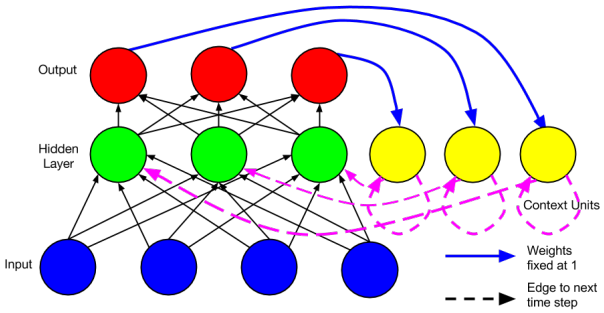

Figure 5: A recurrent neural network as proposed by Jordan (1986).

Jordan networks (Figure 5), introduced by Michael Jordan in 1986, present an early architecture for supervised learning on sequences ([33]). A Jordan

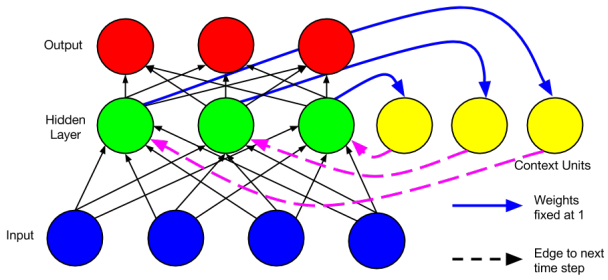

Figure 6: An Elman network as described in *Finding Structure in Time* (1990) [17]. Hidden units are connected 1-to-1 to context units across time steps, which in turn feed back into the corresponding hidden units.

network resembles a feedforward network with a single hidden layer, but is extended with *context units*. [2] Outputs are fed into the context units, which then feed in to the hidden nodes at the following time step. Additionally, context units have self-connected edges. Intuitively, these self-connected edges gives Jordan networks a means to send information across multiple time steps without perturbing the output at each intermediary time step. In Jordan's setup, the network was used to plan actions and thus these context units allow the network to *remember* actions taken (outputs) at the previous time steps. Several modern architectures utilize some form of direct feedback from output nodes. [62] translates sentences between natural languages. When generating text sequences, the word chosen at each time step is fed into the network at the following time step.

Elman networks, introduced in [17], simplify the structure in the Jordan network. Associated with each unit in the hidden layer is a single context unit. Each context unit j' takes as input the state of the corresponding hidden node j at the previous time step, along an edge of unit weight $w_{j'j} = 1$. Effectively, this entire setup is equivalent to a simple RNN in which each hidden node has a single self-connected recurrent edge. Some of Elman's ideas, including fixed-weight edges and the importance of self-connected recurrent edges in hidden nodes survive in Hochreiter and Schmidhuber's subsequent work on Long Short-Term Memory [31].

In the paper, Elman trains the network using backpropagation and demonstrates that the network can learn long-range time dependencies. The paper features two sets of experiments. The first extends the logic operation *exclusive*

[2] In Jordan's paper, he calls these "state units". Elman calls a corresponding structure "context units". In this paper we simplify terminology by using only "context units".

OR (XOR) to the time domain by concatenating sequences of three tokens. For each three-token segment, e.g. "011", the first two tokens ("01") are chosen randomly and the third ("1") is chosen by performing XOR on them. A completely random guess should achieve accuracy of 50%. A perfect system should perform the same as random for the first two tokens, but guess the third token perfectly, achieving accuracy of 66.6...%. Elman's simple network does in fact do get close to this maximum achievable score.

3.2 Training Recurrent Networks

Learning with recurrent neural networks has long been considered to be difficult. As with all neural networks, the optimization is NP-Hard. But learning on recurrent networks can be especially hard due to the difficulty of learning long-range dependencies as described by Bengio et al in 1994 [8] and expanded upon in [30]. The well known problems of *vanishing* and *exploding* gradients occur when propagating errors across many time steps. As a trivial example, consider a network with a single input node, a single output node, and a single recurrent hidden node (Figure 7). Now consider an input passed to the network at time τ and an error calculated at time t, assuming input of 0 in the intervening time steps. Owing to the weight tying across time steps (the recurrent edge at hidden node j always has the same weight), the impact of the input at time τ on the output at time t will either explode exponentially or rapidly approach zero as $t - \tau$ grows large, depending on whether the weight $|w_{jj}| > 1$ or $|w_{jj}| < 1$ and also upon the activation function in the hidden node (Figure 8). Given activation function $l_j = \sigma$, the vanishing gradient problem is more pressing, but with a rectified linear unit $max(0, x)$, it's easier to imagine the exploding gradient, even with this trivial example. In [47], Pascanu et al. give a thorough mathematical treatment of the vanishing and exploding gradient problems, characterizing exact conditions under which these problems may occur. Given these conditions under which the gradient may vanish or explode, they suggest an approach to training via a regularization term, which forces the weights to values where the gradient neither vanishes nor explodes.

Truncated backpropagation through time (TBPTT) is one solution to this problem for continuously running networks [68]. With TBPTT, some maximum number of time steps is set along which error can be propagated. While TBPTT with a small cutoff can be used to alleviate the exploding gradient problem, it requires that one sacrifice the ability to learn long-range dependencies.

The optimization problem represents a more fundamental obstacle that cannot as easily be dealt with by modifying network architecture. It has been known since at least 1993 that optimizing even a 3-layer neural network is NP-Complete [10]. However, recent empirical and theoretical studies suggest that the problem may not be as hard in practice as once thought. [13] shows that while many critical points exist on the error surfaces of large neural networks, the ratio of saddle points to true local minima increases exponentially with the size of the network

Fast implementations and improved gradient following heuristics have ren-

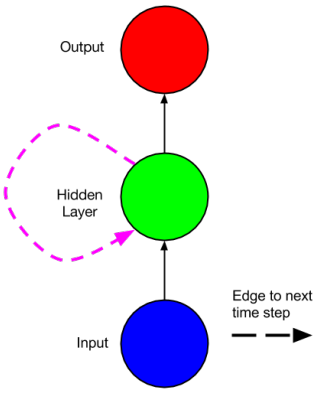

Figure 7: A simple recurrent net with one input unit, one output unit, and one recurrent hidden unit.

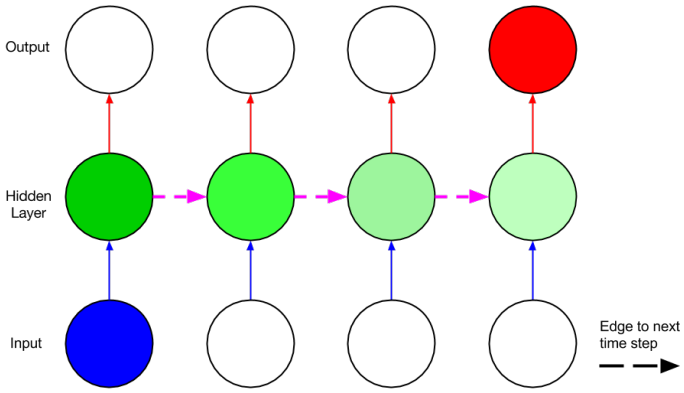

Figure 8: A visualization of the vanishing gradient problem, using the architecture depicted in Figure 7. If the weight along the purple edge is less than one, the effect of the input at the first time step on the output at the final time step will rapidly diminish as a function of the size of the interval in between. An illustration like this appears in [23]

.

dered RNN training feasible. For example, implementations of forward and backward propagation using GPUs, such as Theano ([9]) and Torch ([12]), have made it easy to implement fast training algorithms. In 1996, prior to the introduction of the LSTM, attempts to train recurrent nets to bridge long time gaps were shown to perform no better than random guessing [29]. However, successfully trained RNNs are now relatively common. Sutskever and Martens reported success training recurrent neural networks with a Hessian-Free, i.e., truncated Newton approach [44] and applied it to a network which learns to generate text one character at a time in [61]. In the paper that described the abundance of saddle points on the error surfaces of neural networks ([13]), the authors present a saddle-free version of Newton's method. Unlike Newton's method, which is attracted to critical points, including saddle points, this variant is specially designed to escape from them. Experimental results include a demonstration of improved performance on recurrent networks. Newton's method requires computing the Hessian, which is prohibitively expensive for large networks, scaling quadratically with the number of parameters. While their algorithm only approximates the Hessian, it is still computationally expensive compared to SGD. Thus the authors describe a hybrid approach whereby the saddle-free Newton method is applied in places where SGD appears to be *stuck*.

3.3 Modern RNNs

The most successful RNN architectures for sequence learning date to two papers from 1997. The first, *Long Short-Term Memory*, by Hochreiter and Schmidhuber, introduces the memory cell, a unit of computation that replaces traditional artificial neurons in the hidden layer of a network. With these memory cells, networks are able to overcome some difficulties with training encountered in earlier recurrent nets. The second, *Bidirectional Recurrent Neural Networks*, by Schuster and Paliwal, introduces the BRNN architecture in which information from both the future and the past are used to determine the output at any time t. This is in contrast to previous systems, in which only past input can affect the output, and has been used successfully for sequence labeling tasks in natural language processing, among others. Fortunately, the two innovations are not mutually exclusive, and have been successfully combined by Graves et al. for phoneme classification [25] and handwriting recognition [24].

3.3.1 Long Short-Term Memory (LSTM)

In 1997, to overcome the problem of vanishing gradients, Hochreiter and Schmidhuber introduced the LSTM model. This model resembles a standard neural network with a recurrent hidden layer, only each ordinary node (Figure 1) in the hidden layer is replaced with a memory cell (Figure 9). The memory cell contains a node with a self-connected recurrent edge of weight 1, ensuring that the gradient can pass across many time steps without vanishing or exploding. To distinguish that we are referencing a memory cell and not an ordinary node, we use the index c.

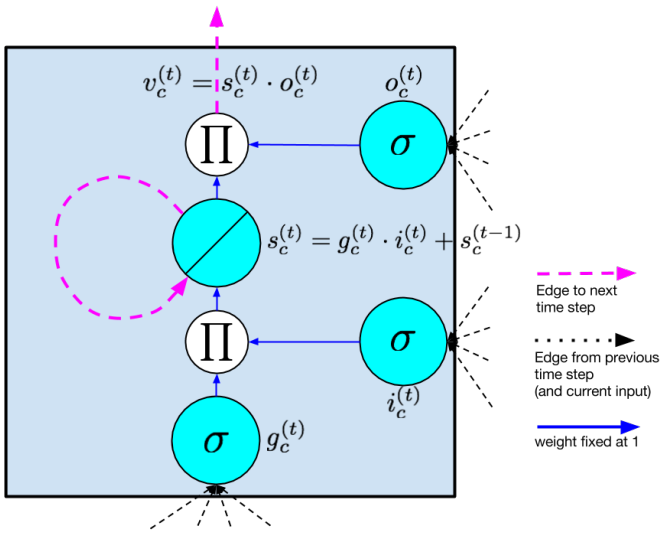

Figure 9: LSTM memory cell as initially described in Hochreiter [31]. The self-connected node is the internal state s. The diagonal line indicates that it is linear, i.e. no link function is applied there. Nodes marked "Π" output the product of their inputs. Dashed lines indicate recurrent edges and pink edges have fixed weight of 1.

The term "Long Short-Term Memory" comes from the following intuition. Simpler recurrent neural networks have *long term memory* in the form of weights. The weights change very slowly over time encoding general knowledge about the data. They also have *short term memory* in the form of ephemeral activations, which pass from each node's output to successive nodes. The LSTM model introduces an intermediary sort of memory via the memory cell. A memory cell is a composite of simpler units with the novel addition of multiplicative nodes, represented in diagrams with Π. All elements of the LSTM cell are enumerated and described below.

- *Internal State:* At the heart of each memory cell is a node s with linear activation, which is referred to in the original paper as the "internal state" of the cell. We index cells with c and thus the internal state of a cell c is s_c.
- *Constant error carousel:* The internal state s_c has a self-connected (recurrent) edge with weight 1. This edge, called the *constant error carousel*, spans adjacent time steps with constant weight, assuring that error can flow across time steps without vanishing.
- *Input Node:* This node behaves as an ordinary neuron, taking input from the rest of the network (at the previous time step) as well as from the input. In the original paper and most subsequent work the input node is labeled g. We adhere to this convention but note that it may be confusing as g does not stand for *gate*. In the original paper, the gates are called y_{in} and y_{out} but this is confusing because y generally stands for output in the machine learning literature. Seeking comprehensibility, we break with this convention and use i, f, and o to refer to input, forget and output gates respectively as in [62]. When we use vector notation we are referring to the values of the nodes in an entire layer of cells. For example, $\boldsymbol{g}$ is a vector containing the value of g at each memory cell in a layer. When the subscript c is used, it is to refer to an individual memory cell.
- *Multiplicative Gating:* Multiplicative gates are distinctive features of the LSTM model. Here a sigmoidal unit called a *gate* is learned given the input and the incoming recurrent connections from the previous time step. Some value of interest is then multiplied by this output. If the gate outputs 0, flow through the gate is cut off. If the gate outputs 1, all activation is passed through the gate.
  - *Input gate:* The original LSTM contains two gates. The first is an *input gate* i_c, which is multiplied by the *input* node g_c.
  - *Output gate:* The second gate is termed the *output gate*, which we notate as o_c. This gate is multiplied by the value of the internal state s_c to produce the value of v_c output by the memory cell . This then feeds into the LSTM hidden layer at the next time step $\boldsymbol{h}^{(t+1)}$ as well as the output nodes $\hat{y}^{(t)}$ at the current time step.

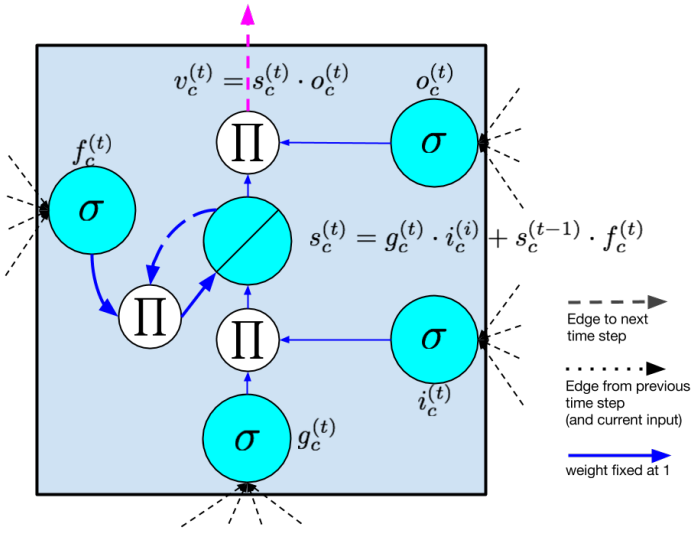

Figure 10: LSTM memory cell with forget gate as described by Felix Gers et al. in [20].

Since the original LSTM was introduced, several variations have been proposed. *Forget gates*, proposed in 2000 by Gers and Schmidhuber [20], add a gate similar to input and output gates to allow the network to flush information from the constant error carousel. Also in 2000, Gers and Schmidhuber proposed peephole connections [19], which pass from the carousel directly to the input and output gates of that same node without first having to pass through the output gate. They report that these connections improve performance on timing tasks where the network must learn to measure precise intervals between events. While peephole connections are not common in modern papers, forget gates have become a mainstay of LSTM work.

Put formally, computation in the LSTM model proceeds according to the following calculations which must be evaluated at each time step. This gives the full algorithm for a modern LSTM with forget gates.

$$\boldsymbol{g}^{(t)} = \phi(W_{gx}\boldsymbol{x}^{(t)} + W_{ih}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_g)$$

$$\boldsymbol{i}^{(t)} = \sigma(W_{ix}\boldsymbol{x}^{(t)} + W_{ih}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i)$$

$$\boldsymbol{f}^{(t)} = \sigma(W_{fx}\boldsymbol{x}^{(t)} + W_{fh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f)$$

$$\boldsymbol{o}^{(t)} = \sigma(W_{ox}\boldsymbol{x}^{(t)} + W_{oh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o)$$

$$\boldsymbol{s}^{(t)} = \boldsymbol{g}^{(t)} \odot \boldsymbol{i}^{(i)} + \boldsymbol{s}^{(t-1)} \odot \boldsymbol{f}^{(t)}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{s}^{(t)} \odot \boldsymbol{o}^{(t)}$$

where $\odot$ stands for element-wise multiplication. The calculations for the simpler LSTM without forget gates is given by setting $\boldsymbol{f}^{(t)} = 1$ for all t. We use the tanh function ϕ for the input node g following the latest state of the art setup of Zaremba and Sutskever in [69]. However, in the original LSTM paper [31], the activation function for g is the sigmoid σ. Again, $\boldsymbol{h}^{(t-1)}$ is a vector containing the values v_c output by each memory cell c in the hidden layer at the previous time step.

Intuitively, in terms of the forward pass, the LSTM can learn when to let activation into the internal state. So long as the input gate takes value 0, no activation can get in. Similarly, the output gate learns when to let the value out. When both gates are *closed*, the activation is trapped in the LSTM, neither growing nor shrinking, nor affecting the output at the intermediary time steps. In terms of the backwards pass, the constant error carousel enables the gradient to propagate back across many time steps, neither exploding nor vanishing. In this sense, the gates are learning when to let *error* in, and when to let it out. In practice, the LSTM has shown a superior ability to learn long-range dependencies as compared to simple RNNs. Consequently, the majority of state of the art application papers covered in this review use the LSTM model.

3.4 Bidirectional Recurrent Neural Networks (BRNNs)

Along with the LSTM, one of the most used RNN setups is the bidirectional recurrent neural network (BRNN) (Figure 11) first described in [51]. In this architecture, there are two layers of hidden nodes. Both hidden layers are connected to input and output. The two hidden layers are differentiated in that the first has recurrent connections from the past time steps while in the second the direction of recurrent of connections is flipped, passing activation backwards in time. Given a fixed length sequence, the BRNN can be learned with ordinary backpropagation. The following three equations describe a BRNN:

$$\boldsymbol{h}_f^{(t)} = \sigma(W_{h_f x}\boldsymbol{x} + W_{h_f h_f}\boldsymbol{h}_f^{(t-1)} + \boldsymbol{b}_{h_f})$$

$$\boldsymbol{h}_b^{(t)} = \sigma(W_{h_b x}\boldsymbol{x} + W_{h_b h_b}\boldsymbol{h_b}^{(t+1)} + \boldsymbol{b}_{h_b})$$

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}(W_{y h_f}\boldsymbol{h_f}^{(t)} + W_{y h_b}\boldsymbol{h_b}^{(t)} + \boldsymbol{b}_y)$$

Where $\boldsymbol{h}_f^{(t)}$ and $\boldsymbol{h}_b^{(t)}$ correspond to the hidden layers in the forwards and backwards directions respectively.

One limitation of the BRNN is that cannot run continuously, as it requires a fixed endpoint in both the future and in the past. Further, it is not an appropriate machine learning algorithm for the online setting, as it is implausible to receive information from the future, i.e., sequence elements that have not been observed. But for sequence prediction over a sequence of fixed length, it is often sensible to account for both past and future data. Consider the natural language task of *part of speech tagging*. Given a word in a sentence, information about both the words which precede and those which succeed it is useful for predicting

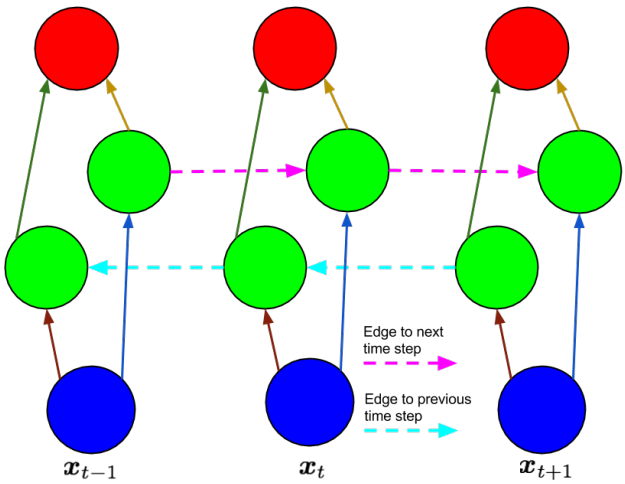

Figure 11: Structure of a bidirectional recurrent neural network as described by Schuster and Paliwal in [51].

that word's part of speech. Karpathy et al. use such a network for generating captions for images [34].

The LSTM and BRNN are in fact compatible ideas. The former introduces a new basic unit from which to compose a hidden layer, while the latter concerns the wiring of the hidden layers, regardless of what nodes they contain. Such an approach, termed a BLSTM was used by Graves et al. to achieve state of the art results on handwriting recognition and phoneme classification [24] [25].

4 Applications of LSTM and BRNN

In the previous sections, we introduced the basic building blocks from which nearly all state of the art recurrent neural networks are composed. Now, we'll look at several application areas where recurrent networks have been employed successfully. We'll quickly introduce the representations used for input and output and the commonly used performance metrics. Then we will survey state of the art results in machine translation, image captioning, video captioning, and handwriting recognition.

4.1 Representing Natural Language for Input and Output

Many applications of RNNs involve text processing. Some applications, e.g. image captioning, involve generating strings of text. Others, such as machine translation and dialogue systems, require both inputting and outputting text.

In this section, we provide the background necessary to understand how text is represented in most recurrent net models.

When words are output at each time step, generally the output consists of a softmax vector $\boldsymbol{y}^{(t)} \in \mathbb{R}^K$ where K is the size of the vocabulary. A softmax layer is like an element-wise logistic function that is normalized so that all of its components sum to 1. Intuitively these outputs correspond to probabilities that each word is the correct output at that time step.

For application where inputs consist of sequences of words, typically the words are fed to the network one at a time as inputs in consecutive time steps. In these cases, the simplest way to represent words is a *one-hot* encoding, using binary vectors with a length equal to the size of the vocabulary, e.g. "1000" and "0100" would represent the first and second words in the vocabulary respectively. Such an encoding is discussed in [17] among others. However, this encoding is inefficient, requiring as many bits as the vocabulary is large. Further, it offers no direct way to capture different aspects of similarity between words in the encoding itself. Thus it is more common to model words with a distributed representation using a *meaning vector*. In some cases, these meanings for words are learned given a large corpus of supervised data, but it is common to initialize the *meaning vectors* using an embedding based on word co-occurence statistics. Freely available code to produce word vectors from co-occurrence stats include *Glove* from Pennington Socher and Manning [49], and *word2vec* [22], which implements a word embedding algorithm from Mikolov et al. [45].

Such distributed representations for symbolic data were described by Hinton in 1986 [28], used extensively for natural language by Bengio et al. in 2003 [7], and more recently brought to wider attention in the deep learning community by Socher, Manning and Ng in a number of papers describing Recursive Auto-encoder (RAE) networks ([57], [54], [58], [56]). For clarity we point out that these *recursive networks*, which are sometimes referred to as "RNNs" *are not recurrent neural nets*. In contrast to RNNs which model arbitrary dependencies between inputs at different points in the sequence, recursive networks assume a tree structure where each word in a sentence is a leaf and each sentence can be represented as a binary tree. In this model, each internal node has a meaning vector which can be calculated by concatenating the meanings of its children and multiplying the composite vector by an encoding matrix. Each meaning vector $m_j \in \mathbb{R}^d$ and the encoding matrix $A_e \in \mathbb{R}^{2d \times d}$. The composite meaning $m_p \in \mathbb{R}^d$ given to the parent p of two child nodes l and r in the syntax tree is

$$m_p = \sigma(A_e[m_l; m_r]^T)$$

This method seems restricted to sentences, and incompatible with the online setting where the full sequence may not be fixed in advance. Additionally, while it has been successfully used for classification tasks, it offers no generative model for composing text.

In many experiments with recurrent neural networks ([17], [61], [69]), input is fed in one character at a time (and output generated one character at a time). While the output is nearly always a softmax layer, many papers omit details

of how they represent single-character inputs. It seems reasonable to infer that characters are encoded with a one-hot encoding. We know of no cases of paper using a distributed representation at the single-character level.

4.1.1 Evaluation Methodology

One serious obstacle to training systems to output variable length sequences of words are the flaws of the available performance metrics. Developed in 2002, *BLEU* score, a common evaluation metric for machine translation, is related to modified unigram precision [**?**]. It is extended by taking the geometric mean of the n-gram precisions for all values of n between 1 and some upper limit N. In practice, 4 is a typical value for N, shown to maximize agreement with human raters. Because precision can be made high by offering absurdly short translations, the authors of *BLEU* introduce a brevity penalty BP. Where q is the length of the candidate translation and r is the length of the reference translations the brevity penalty is expressed as

$$BP = \begin{cases} 1 & \text{if } q > r \\ e^{(1-r/q)} & \text{if } q <= r \end{cases}.$$

Then the *BLEU* score can be calculated as

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

where w_n is a uniform weight $1/N$ and p_n is the modified n-gram precision.

BLEU score is commonly used in recent papers to evaluate both translation and captioning systems. While *BLEU* score does appear highly correlated with human judgments, there is no guarantee that any given translation with a higher *BLEU* score is superior to another which receives a lower *BLEU* score. In fact, while *BLEU* scores agree with human judgement across large sets of translations, they are significantly less accurate at the sentence level.

METEOR, introduced in 2005 by Banerjee and Lavie is an alternative metric intended to overcome these weaknesses of the *BLEU* score [**?**]. *METEOR* is scored based on explicit word to word matches between candidates and references. When multiple references exist, the best score is used. Unlike *BLEU*, *METEOR* exploits known synonyms and stemming. First they compute an F-score

$$F_\alpha = \frac{P \cdot R}{\alpha \cdot P + (1-\alpha) \cdot R}$$

based on single word matches where P is the precision and R is the recall. Then they calculate a fragmenting penalty $Pen \propto ch/m$ where ch is the smallest number of *chunks* of consecutive words such that the words are adjacent in both the candidate and the reference, and m is the total number of matched unigrams yielding the score:

$$METEOR = (1 - Pen) \cdot F_\alpha.$$

Empirically, this metric has been found to agree more with human raters than *BLEU* score.

However, *METEOR* is less straight-forward to calculate than *BLEU*. To replicate the *METEOR* score reported by another party, one must exactly replicate their stemming and synonym matching, as well as the calculations. Both metrics rely upon having the exact same set of candidate translations.

Even in the straightforward case of binary classification, without time dependencies, commonly used performance metrics like F1 give rise to complicated thresholding strategies which may not accord with any reasonable intuition of what should constitute good performance [40]. Along the same lines, given these new performance metrics which are weak proxies for our objectives and which cannot be optimized directly, it may be difficult to differentiate systems which are truly stronger and those which most meticulously game the performance metrics of interest.

4.2 Text Translation

Natural language translation is a fundamental problem in machine learning that resists solutions with shallow methods. Some tasks, like document classification, can be performed successfully with representations like bag-of-words, which ignore word order. But word order is essential in translation. The sentences *"Scientist killed by raging virus"* and *"Virus killed by raging scientist"* have identical bag-of-words representations.

In *Sequence to Sequence Learning*, published in 2014, Sutskever et al. present a translation model using two multilayered LSTMs and demonstrate impressive performance translating from English to French. The first LSTM is used for *encoding* an input phrase from the source language and one for *decoding* the output phrase in the target language. Their model works according to the following procedure (Figure 12):

- The source phrase is fed to the *encode* LSTM, one word at a time. The inputs are *word vectors*. This LSTM does not output anything. In practice, the authors found that significantly better results were achieved when the input sentence was fed into the network in reverse order.
- When the end of the phrase is reached, a special input which indicates the beginning of the sentence is fed to the *decoder* LSTM. This second LSTM gets as input the start token, and also the final state of the encoder This LSTM outputs softmax probabilities over the vocabulary at each time step.
- A word is chosen and fed to the network at the next step. At this time step, the network receives activation along all recurrent edges as well as the word vector of the chosen word.

For training, the true inputs are fed to the encoder, the true translation is fed to the decoder, and loss is propagated back from the outputs of the decoder.

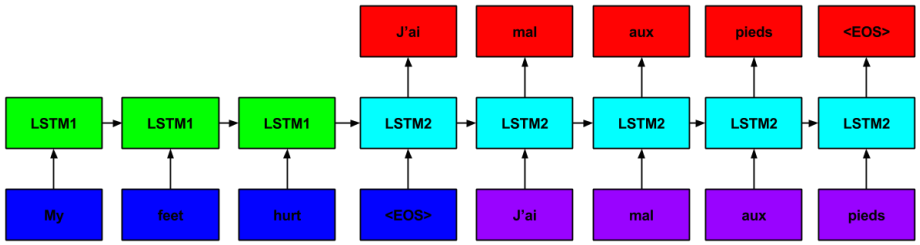

Figure 12: Sequence to sequence LSTM model as described by Sutskever et al. [62]. The input blocks (blue and purple) all correspond to word vectors, which are fully connected to the corresponding hidden state. Red nodes are softmax outputs. The network consists of an encoding model (LSTM1) and a decoding model (LSTM2). Weights are tied among all encoding steps and among all decoding time steps.

The network is trained to maximize the likelihood of the correct translation. At inference time a left to right beam search is used to determine which words to output. A few among the most likely next words are chosen for expansion after each time step. The beam search ends when the network outputs an end-of-sentence (*EOS*) token. They train the model using only stochastic gradient descent, without momentum, halving the learning rate every half epoch, after the first five. Using this approach, they achieve a *BLEU* score of 34.81, which outperforms the best previous neural network NLP systems, and is very close to the best published results, including systems that have explicitly programmed domain expertise. When their system is used to rerank candidate translations from another system, they achieved a *BLEU* score of 36.5

Another RNN approach to language translation is presented by Auli et al. in *Joint Language and Translation Modeling with Recurrent Neural Networks*. Their RNN model uses the word embeddings of Mikolov [2] and a lattice representation of the decoder output to facilitate search over the space of possible translations. In their lattice, each node corresponds to a sequence of words. They report a *BLEU* score of 28.5 on French-English translation tasks. Both papers report results on data from WMT datasets but Sutskever et al. only report results on English to French translation while Auli et al. report French to English translation. Thus it is impossible to directly compare the performance of the two models.

4.3 Image Captioning

Recently, recurrent neural networks have been used successfully for image captioning, [34, 43, 65]. In this task, a training set consists of input images $\boldsymbol{x}$ and target captions $\boldsymbol{y}$. Given a large set of image-caption pairs, a model is trained

to predict the appropriate caption for an image.

Vinyals et al. follow up on the success in language to language translation by considering captioning as a case of image to language translation. Instead of both encoding and decoding with LSTMs, they introduce the idea of encoding an image with a convolutional neural network, and then decoding it with an LSTM. This idea is the core of several papers which address this topic. This work clearly precedes [34] as it is cited throughout, but [43] claim it as an independent innovation. With this architecture, they achieve then state of the art results on the Pascal, Flickr30k, and COCO datasets.

Karpathy et al. similarly use a convolutional neural network to encode images together with a bidirectional neural network to decode translations, using word2vec embeddings as word representations [34]. They consider both full-frame evaluations and a model that captures correspondences between image regions and text snippets. At inference, their procedure resembles the one described for Sutskever et al., where sentences are decoded one word at a time. The most probable word is chosen and fed to the network at the next time step. This process is repeated until an *EOS* token is produced.

4.4 Other Interesting Applications

Handwriting recognition is an application area where bidirectional LSTMs have been used to achieve state of the art results. In work by Liwicki, Graves et al. ([41], [24]), data is collected from a whiteboard using an eBeam interface. The interface collects the (x, y) coordinates of the pen at regularly sampled time steps. In the more recent paper, they use a bidirectional LSTM model, outperforming an HMM model by achieving 81.5% word-level accuracy, compared to 70.1% for the HMM.

In the last year, a number of papers have emerged, that extend the success of recurrent networks for translation and image captioning to new domains. Among the most interesting of these applications are unsupervised video encoding [59], video captioning [64] and program execution [69]. In [64], Venugopalan et al. demonstrate a sequence to sequence architecture similar to that which Sutskever uses for natural language translation. However, instead of both encoding and decoding words, they encode frames from a video and decode words. At each time step the input to the encoding LSTM is the topmost hidden layer of convolutional neural network. At decoding time, the network outputs probabilities over the vocabulary at each time step.

In *Learning to Execute* ([69]), Zaremba and Sutskever experiment with networks which read computer programs one character at a time and predict their output. They focus on programs which output integers and find that for simple programs, including adding two nine-digit numbers, their network, which uses LSTM units, several stacked hidden layers, and makes a single left to right pass through the program, can predict the output with 99% accuracy.

5 Discussion

Over the past thirty-five years, recurrent networks have gone from impractical models, primarily of interest for cognitive modeling and computational neuroscience, to powerful and practical tools for large-scale supervised learning. This progress owes to advances in model architectures, training algorithms, and parallel computing. Recurrent nets are especially interesting because they seem able to overcome many of the extreme restrictions generally placed on data by traditional machine learning approaches. With recurrent nets, the assumption of independence between consecutive examples is broken, the assumption of fixed-dimension input is broken, and yet RNN models perform competitively with or outperform the state of the art on many tasks.

While LSTMs and BRNNs set records, we find it noteworthy that many advances come from novel architectures rather than fundamentally novel algorithms. It seems that research exploring the space of possible models, either via genetic algorithms or a Markov Chain Monte Carlo based approach, should be promising. Along these lines, we note that neural networks seem to offer a wide range of easily transferable techniques. New activation functions, training procedures, initializations procedures, etc. are generally transferable across networks and tasks, often conferring similar benefits. As the number of such techniques grows, the practicality of testing all combinations diminishes. However it seems reasonable to infer that as a community, neural network researchers are exploring the space of a model architectures and configurations much as a genetic algorithm might, mixing and matching techniques, with a fitness function in the form of evaluation metrics on major datasets of interest.

This suggests two things. First, as stated before, this body of research would benefit from automated procedures to explore the space of models. Second, as we build systems designed to perform more complex tasks, we would benefit from improved fitness functions. *BLEU* inspires less confidence than the accuracy reported on a binary classification task. To this end, when possible, it seems prudent to individually test techniques first with classic feedforward networks on datasets with established benchmarks before applying them to recurrent networks in settings with less concrete evaluation criteria.

Lastly, the rapid success of recurrent neural networks on natural language tasks leads us to believe that extensions of this work to longer form text would be fruitful. Additionally, we imagine that dialogue systems could be built along similar principles to the architectures used for translation, encoding prompts and generating responses, retaining the entirety of conversation history as contextual information.

Acknowledgements

My research is funded by generous support from the Division of Biomedical Informatics at UCSD, via a training grant from the National Library of Medicine. This review has benefited from insightful comments from Charles Elkan, John

Berkowitz, Julian MacAuley, Balakrishnan Narayanaswamy, Stefanos Poulis, Sharad Vikram.

References

[1] Lecture 6.5- rmsprop: Divide the gradient by a running average of its recent magnitude.

[2] Michael Auli, Michel Galley, Chris Quirk, and Geoffrey Zweig. Joint language and translation modeling with recurrent neural networks. In *EMNLP*, pages 1044–1054, 2013.

[3] Pierre Baldi and Gianluca Pollastri. The principled design of large-scale recursive neural network architectures–DAG-RNNs and the protein structure prediction problem. *The Journal of Machine Learning Research*, 4:575–602, 2003.

[4] Justin Bayer, Daan Wierstra, Julian Togelius, and Jürgen Schmidhuber. Evolving memory cell structures for sequence learning. In *Artificial Neural Networks–ICANN 2009*, pages 755–764. Springer, 2009.

[5] Richard K Belew, John McInerney, and Nicol N Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In *In*. Citeseer, 1990.

[6] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

[7] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

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

[9] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, page 3. Austin, TX, 2010.

[10] Avrim L Blum and Ronald L Rivest. Training a 3-node neural network is NP-complete. In *Machine learning: From theory to applications*, pages 9–28. Springer, 1993.

[11] Bob Carpenter. Lazy sparse stochastic gradient descent for regularized multinomial logistic regression. *Alias-i, Inc., Tech. Rep*, pages 1–20, 2008.

[12] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[13] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in Neural Information Processing Systems*, pages 2933–2941, 2014.

[14] Wim De Mulder, Steven Bethard, and Marie-Francine Moens. A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1):61–98, 2015.

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[16] Charles Elkan. Learning meanings for sentences.

[17] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[18] Felix Gers. Long short-term memory in recurrent neural networks. *Unpublished PhD dissertation, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland*, 2001.

[19] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE, 2000.

[20] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with LSTM. *Neural computation*, 12(10):2451–2471, 2000.

[21] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, volume 15, pages 315–323, 2011.

[22] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[23] Alex Graves. *Supervised sequence labelling with recurrent neural networks*, volume 385. Springer, 2012.

[24] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(5):855–868, 2009.

[25] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[26] Frederic Gruau et al. Neural network synthesis using cellular encoding and the genetic algorithm. 1994.

[27] Steven A Harp and Tariq Samad. Optimizing neural networks with genetic algorithms. In *Proceedings of the 54th American Power Conference, Chicago*, volume 2, 2013.

[28] Geoffrey E Hinton. Learning distributed representations of concepts.

[29] Sepp Hochreiter. Bridging long time lags by weight guessing and long short-term memory.

[30] Sepp Hochreiter, Yoshua Bengio, and Paolo Frasconi. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

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

[32] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[33] Michael I Jordan. Serial order: A parallel distributed processing approach. *Advances in psychology*, 121:471–495, 1997.

[34] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. *arXiv preprint arXiv:1412.2306*, 2014.

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

[36] John Langford, Lihong Li, and Tong Zhang. Sparse online learning via truncated gradient. In *Advances in neural information processing systems*, pages 905–912, 2009.

[37] B Boser Le Cun, John S Denker, D Henderson, Richard E Howard, W Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a backpropagation network. In *Advances in neural information processing systems*. Citeseer, 1990.

[38] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.

[39] Zachary C Lipton and Charles Elkan. Efficient elastic net regularization for sparse linear models. 2015.

[40] Zachary C Lipton, Charles Elkan, and Balakrishnan Naryanaswamy. Optimal thresholding of classifiers to maximize F1 measure. In *Machine Learning and Knowledge Discovery in Databases*, pages 225–239. Springer, 2014.

[41] Marcus Liwicki, Alex Graves, Horst Bunke, and Jürgen Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In *Proc. 9th Int. Conf. on Document Analysis and Recognition*, volume 1, pages 367–371, 2007.

[42] Andrew L Maas, Quoc V Le, Tyler M O'Neil, Oriol Vinyals, Patrick Nguyen, and Andrew Y Ng. Recurrent neural networks for noise reduction in robust ASR. In *INTERSPEECH*. Citeseer, 2012.

[43] Junhua Mao, Wei Xu, Yi Yang, Jiang Wang, and Alan Yuille. Deep captioning with multimodal recurrent neural networks (m-RNN). *arXiv preprint arXiv:1412.6632*, 2014.

[44] James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040, 2011.

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

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

[47] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012.

[48] Barak A Pearlmutter. Gradient calculations for dynamic recurrent neural networks: A survey. *Neural Networks, IEEE Transactions on*, 6(5):1212–1228, 1995.

[49] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. *Proceedings of the Empiricial Methods in Natural Language Processing (EMNLP 2014)*, 12, 2014.

[50] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[51] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45(11):2673–2681, 1997.

[52] Hava T Siegelmann and Eduardo D Sontag. Turing computability with neural nets. *Applied Mathematics Letters*, 4(6):77–80, 1991.

[53] Yoram Singer and John C Duchi. Efficient learning using forward-backward splitting. In *Advances in Neural Information Processing Systems*, pages 495–503, 2009.

[54] Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems*, pages 801–809, 2011.

[55] Richard Socher, Andrej Karpathy, Quoc V Le, Christopher D Manning, and Andrew Y Ng. Grounded compositional semantics for finding and describing images with sentences. *Transactions of the Association for Computational Linguistics*, 2:207–218, 2014.

[56] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136, 2011.

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

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

[59] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using LSTMs. *arXiv preprint arXiv:1502.04681*, 2015.

[60] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 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, 2013.

[61] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

[62] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[63] Alan M Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.

[64] Subhashini Venugopalan, Marcus Rohrbach, Jeff Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence–video to text. *arXiv preprint arXiv:1505.00487*, 2015.

[65] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

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

[67] Wikipedia. Backpropagation — Wikipedia, the free encyclopedia, 2015. [Online; accessed 18-May-2015].

[68] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[69] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

[70] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[71] Matthew D Zeiler, M Ranzato, Rajat Monga, M Mao, K Yang, Quoc Viet Le, Patrick Nguyen, A Senior, Vincent Vanhoucke, Jeffrey Dean, et al. On rectified linear units for speech processing. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3517–3521. IEEE, 2013.