The Dissertation Committee for Aditya Rawal certifies that this is the approved version of the following dissertation:

Discovering Gated Recurrent Neural Network Architectures

Committee:

Risto Miikkulainen, Supervisor

Scott Niekum

Aloysius Mok

Kay Holekamp

Discovering Gated Recurrent Neural Network Architectures

by

Aditya Rawal,

Dissertation

Presented to the Faculty of the Graduate School

of the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2018

I dedicate this to my late mother, Shamita Rawal.

Acknowledgments

This work wouldn't have been possible without unwavering support of my wife Neha, my father Jatin, my brother Kandarp and my 3-year old daughter Samyra.

Risto, my advisor, has provided me freedom to fail which I found very encouraging. He has provided supported even during the most difficult periods of this endeavor. His guidance helped me pivot my efforts at the right time. He has been a role model and I wish to carry forward my learnings from him as I embark on a new chapter of my life.

Discovering Gated Recurrent Neural Network Architectures

by

Aditya Rawal, Ph.D.

The University of Texas at Austin, 2018

Supervisor: Risto Miikkulainen

Reinforcement Learning agent networks with memory are a key component in solving POMDP tasks. Gated recurrent networks such as those composed of Long Short-Term Memory (LSTM) nodes have recently been used to improve state of the art in many supervised sequential processing tasks such as speech recognition and machine translation. However, scaling them to deep memory tasks in reinforcement learning domain is challenging because of sparse and deceptive reward function. To address this challenge first, a new secondary optimization objective is introduced that maximizes the information (Info-max) stored in the LSTM network. Results indicate that when combined with neuroevolution, Info-max can discover powerful LSTM-based memory solutions that outperform traditional RNNs. Next, for the supervised learning tasks, neuroevolution techniques are employed to design new LSTM architectures. Such architectural variations include discovering new pathways between the recurrent layers as well as designing new gated recurrent nodes. This dissertation proposes evolution of a tree-based encoding of the gated memory nodes, and shows that it makes it possible to explore new variations more effectively than other methods. The method discovers nodes with multiple recurrent paths and multiple memory cells, which lead to significant improvement in the standard language modeling benchmark task. The dissertation also shows how the search process can be speeded up by training an LSTM network to estimate performance of candidate structures, and by encouraging exploration of novel solutions. Thus, evolutionary design of complex neural network structures promises to improve performance of deep learning architectures beyond human ability to do so.

Table of Contents

Chapter 1

Introduction

Imagine artificial agents that can learn from past events and adapt like humans. These agents can then be deployed in challenging unseen environments. Robots with a sophisticated understanding of natural language and conversation skills can carry out many of our daily chores. Better weather forecasting systems that predict natural disasters can save many lives. One common technology that is critical in order to achieve these breakthroughs is AI based memory. With the help of recent many advancements in deep learning, recurrent neural networks have come to the forefront for building such AI based memory. The goal of this dissertation is to devise new techniques to further improve this field and discover better memory networks.

1.1 Motivation

Natural organisms can memorize and process sequential information over long time lags. Chimpanzees and orangutans can recall events that occurred more than a year ago (Martin-Ordas et al. ((2013)). Long term social memory can provide significant survival benefits. For example, bottlenose dolphins can recognize each other's whistle sounds even after decades (Bruck ((2013)). Such a capability allows the dolphin to identify adversaries as well as potential teammates for hunting. First step towards adaptive behavior is to memorize past events and utilize them for future decision making (Stanley et al. (2003)). For example, a group of hyenas, during lion-hyena interactions, modulate their behavior over a period of time through memory-based emotions - transitioning from being fearful initially to becoming risk-taking later (Watts and Holekamp ((2008)). Memory is therefore a key cognitive component and incorporating this capability in artificial agents can make them more realistic (Schrum et al. (2011)).

In the reinforcement learning (RL) domain, the tasks requiring memory can be formally be described as POMDP problems. Traditionally, recurrent neural networks (RNNs) have been the preferred choice for this purpose. However, RNNs leak information and are unable to discover long-term dependencies (Hochreiter

et al. (2001)). Long Short Term Memory (LSTM) (Hochreiter and Schmidhuber (1997a)) successfully overcomes these limitations of RNNs. It consists of memory cells with linear activations. The inflow and outflow of information to and from these cells is controlled by associated input/output gated units.

Such LSTM based memory networks are also used to build chat-bots, speech recognition and forecasting systems. The time-series prediction problem in such settings fall into the category of supervised learning problems. With the recent advances in deep learning, the performance of such LSTM networks has improved drastically (Bahdanau et al. (2015a), Graves and Jaitly ((2014)) but their capabilities cannot match human levels.

New methods are presented in this dissertation that can evolve deep sequence processing recurrent networks to solve RL and supervised memory tasks with long time-dependencies.

While LSTM networks have been used to achieve strong results in the supervised sequence learning problems such as in speech recognition (Graves and Jaitly ((2014)) and machine translation Bahdanau et al. (2015a), their success in POMDP tasks has been limited (Bayer et al. (2009a), Bakker et al. (2003)). A possible reason is that it is difficult to train LSTM units (including its associated control logic) with weak reward/fitness signal. Also, the number of LSTM units in a network is a parameter that is often manually selected. This approach turns out to be inefficient especially in new problems where the memory depth requirements are not clear.

1.2 Challenges

In RL tasks, recurrent networks are often used as policy controllers to determine agent actions given some input observations or as function approximators to predict the value function (Hausknecht and Stone (2017)). However, one key challenge is to find the optimal size of such networks (for e.g. the number of nodes in the layer). This is not a straightforward task and requires separate hyperparameter tuning. Size of the network matters for two reason: computational cost and performance. Larger networks have bigger memory footprint and they take more processor power. Networks used for RL are much smaller and run on CPU than the ones used for supervised learning (that use GPU). It becomes more critical

therefore for RL memory networks to have close to optimal network size. Selecting a model with a large number of LSTM nodes (much more than that required by the task) can lead to overfitting of the model while selecting a very small model can lead to underfitting.

A second challenge in solving RL memory problems could be that of sparse and deceptive rewards. Deceptive fitness landscapes often lead the agent to local optima. The lack of sufficient rewards can prevent the agent to escape local optima.

The LSTM networks used for supervised learing are significantly larger than RL and the parameter size is often in the order of tens of millions. For supervised learning problem, backpropagation through time (BPTT) is a powerful technique to train LSTM network weights. With the availability of large scale labeled data and GPU compute, it is now possible to train LSTMs with mini-batch stochastic gradient descent (SGD). Such networks often consist of couple of LSTM layers stacked together along with some glue logic (for e.g. embedding layer, softmax layer) to construct a full model. These models can then be deployed for language modeling, machine translation, speech recognition etc.

Previously, experimental results in supervised non-recurrent tasks like image classification have demonstrated that deeper and larger feedforward networks often outperform smaller and shallower networks (Saxe et al. (2013)). However, the same idea does not directly scale to deep recurrent networks. While, recurrent networks are naturally deep in time (depth equal to the number of unroll steps in BPTT), stacking more LSTM layers vertically yields diminishing returns (Chung et al. (2015)). Some of the recent human designed variants of stacked LSTMs like recurrent highway networks (Zilly et al. (2016)), grid LSTMs (Kalchbrenner et al. (2015)) and gated-feedback networks (Chung et al. (2015)) suggest that the search space of architecture is large. (Chung et al. (2015)) and (Kalchbrenner et al. (2015)) showed that new ways of stitching up LSTM layers together gives improved performance while (Zilly et al. (2016)) came up with a modifed recurrent node architecture to achieve the same.

The recurrent architectures have grown complex and can no longer be optimized by humans. Therefore, a third challenge, specific to the supervised learning domain, is the automatic design of deep recurrent network architectures.

1.3 Approach

A neuroevolution technique called NEAT (Stanley and Miikkulainen (2002)) is used in dissertation to automatically design the recurrent networks for RL memory tasks. NEAT is a gradient-free optimization methods that can search for both network structure and weights in a non-parametric manner. NEAT gradually complexifies the structure of the network, starting with a small seed network. Due to its complex control logic, training LSTM networks with only NEAT could be challenging. To address this, a phased training approach is developed in this work; where the input and forget gates are trained first and the output gates are trained later.

To overcome the challenge of deception and sparse rewards in RL problems, a key contribution of this dissertation is the design of an unsupervised objective called information maximization (Info-Max). Besides maximizing the reward, the agent maximizes (through evolution) the information stored in the LSTM based memory network. This new objective not only drives the agent to gather new information in the environment but is also used to construct an optimal sized network that stores memory in the most efficient way possible. This hybridization of agent exploration and memorization leads to improved agent behavior in challenging memory tasks such as deep T-maze.

To address the challenge of designing the architecture of recurrent networks for supervised learning problems, variants of NEAT are employed. Recent work in this area suggests that optimizing the network architecture using a meta-learning algorithm like Bayesian optimization (Malkomes et al. (2015)), reinforcement learning (Zoph and Le (2016) Baker et al. (2016)), or evolutionary computation (Miikkulainen and et al. (2017), Real et al. (2017), et al. (2017)) and subsequently training the network using SGD is a promising approach. Neuroevolution algorithms like NEAT could therefore play an important role in search through this architectural space to discover better structures.

The search for LSTM networks using neuroevolution can be divided into two categories: 1) Evolving layer connectivity, 2) Evolving recurrent node architecture. In this dissertation, new technology is developed to solve the problems in each of the category. First, a simpler variant of NEAT is used to discover new connections between LSTM layers while keeping the recurrent node architecture

fixed. One such evolved solution discovered a new recurrent path between the stack of two layers. The new recurrent path doubled the overall depth (sum of feedforward and recurrent depths) of the network. Experimental results in the language-modeling domain show that the evolved connectivity between LSTM layers leads to improved performance at almost no extra cost (since the feedback connection has a fixed weight of 1.0). The results suggest that neuroevolution can be effective in quickly finding deeper and better LSTM network.

Next, the recurrent layer connectivity was fixed and neuroevolution was applied to discover new gated recurrent node architectures. The gated recurrent node can be represented as a tree. While, a LSTM node has a carefully designed gating logic, in this experiment, evolution was tasked to search for a better gating mechanism. A single LSTM node, when represented as a tree, can be considered as a five layer deep network (see Figure 2.2). Evolution enables search for deeper and larger gated recurrent nodes. Thus, the search space for node evolution is orders of magnitude larger than layer evolution. Sopshisticated neuroevolution techniques are needed to search such a large architectural space. A combined genetic programming and NEAT (called GP-NEAT) is used for this purpose. GP-NEAT combines the best ideas from genetic programming (for e.g. structural tree distance and homologous crossover) with the advantages of NEAT (for e.g. speciation). New techniques are proposed in this dissertation to exhaustively compare tree structures and deduce their similarity, thus avoiding redundant trees in the population. Additionally, there were three key innovations that were developed in this dissertation for evolving gated recurrent nodes.

First, a new mechanism to encourage exploration of architectures was devised. An archive of already-explored areas (called Hall of Shame) is maintained during the course of evolution. This archive is used to drive architecture search towards new un-explored regions. The effect is similar to that of novelty search (Lehman (2012)), but does not require a separate novelty objective, simplifying the search.

Second, experiments were conducted to show that evolution of neural network architectures in general can be speeded up significantly by using an LSTM network to predict the performance of candidate neural networks. After training the candidate for a few epochs, such a Meta-LSTM network predicts what per-

formance a fully trained network would have. That prediction can then be used as fitness for the candidate, speeding up evolution fourfold in these experiments. While network performance prediction has recently gained some attention (Baker et al. (2017)), this is the first study that actually deploys fitness prediction mechanism to discover new and significantly improved gated recurrent architectures.

Third, taking inspiration from the results of layer evolution, where a new feedback connection led to performance improvement, an extra recurrent connection is introduced within the node. Evolution is tasked with finding the appopriate gating logic within the recurrent node. Combining all these innovations speeds up evolution, resulting in the discovery a of new gated recurrent node that outperforms LSTMs and other state-of-the-art recurrent nodes like NAS (Zoph and Le (2016)) and RHN (Zilly et al. (2016)) in the language-modeling domain (See Figure 5.9).

Finally, the results from evolution were transfered to the music domain. Here, the best evolved recurrent node was used for polyphonic music generation. The data set used for training and evaluation consists of classical piano tunes. Interestingly, in this domain, the evolved nodes were outperformed by the LSTMs. This suggest that evolution discovered a recurrent architecture customized for the language-modeling problem. An interactive music generation application was created that takes a few input notes from the user and then uses the gated recurrent networks to automatically compose a new musical piece.

1.4   Guide to the Reader

Chapter 2 describes the concepts used across this dissertation. It also discusses previous work in gated recurrent network architecture search. Chapter 3 looks at RL based memory problems and proposes solutions to solve them using evolved LSTMs. Gated recurrent networks are evolved for supervised learning in Chapter 4 and 5. In chapter 4, the connectivity between two LSTM layer is evolved. Chapter 5 looks at evolving gated recurrent nodes while keeping the layer connectivity fixed. In Chapter 6, the best evolved solution from Chapter 5 is transfered to the music domain. This chapter also includes details about an application created to compose music using recurrent networks. Chapter 7 lists

possible future direction for this research. The contributions of this research and conclusions are provided in Chapter 8.

Chapter 2

Background and Related Work

Recurrent Neural Networks are generative models that produce outputs which can be used as inputs in future time steps. The experiments performed in this work demonstrate how to build such memory networks for RL and supervised learning tasks. The challenges in these two kinds of problems are different. This chapter provides detail on the concepts used across the dissertation. For e.g. RNNs, LSTMs and their applications. Recently, techniques like regularization, batch normalization and architecture modifications have been proposed to improve the performance of LSTMs on supervised learning tasks. Since the LSTM networks for RL problems are much smaller than the ones used for supervised tasks, these techniques do not yield as much performance improvements. RL problems come with their own set of challenges like deceptive rewards.

This chapter also briefly describes the existing neuroevolution technologies that are extended for LSTM evolution in later chapters. RL tasks in deep T-maze are used for evaluation in Chapter 3 and language and music tasks are used for supervised learning in subsequent chapters.

2.1 Vanishing Gradients in Recurrent Neural Networks

One major limitation of RNNs is that they are not able to maintain contexts for longer time sequences. This occurs because of two reasons. First, the memory state of a RNN keeps getting updated at each time step with new feedforward inputs. This means that the network does not have the control of what and how much context to maintain over time. This behavior negatively effects RNNs even when their weights are trained with gradient-free methods like evolution. Second, when RNNs are trained with backpropagation through time, they are not able to properly assign gradients to previous time steps due to squashing non-linearities. This is called as the vanishing gradient problem.

As shown in Figure 2.1, the gradient flows back through time in the unrolled RNN. Equation 2.1 describe the feedforward activation in each step of the RNN. Equation 2.2 and 2.3 describe the error that is backpropagated from time step $t+1$
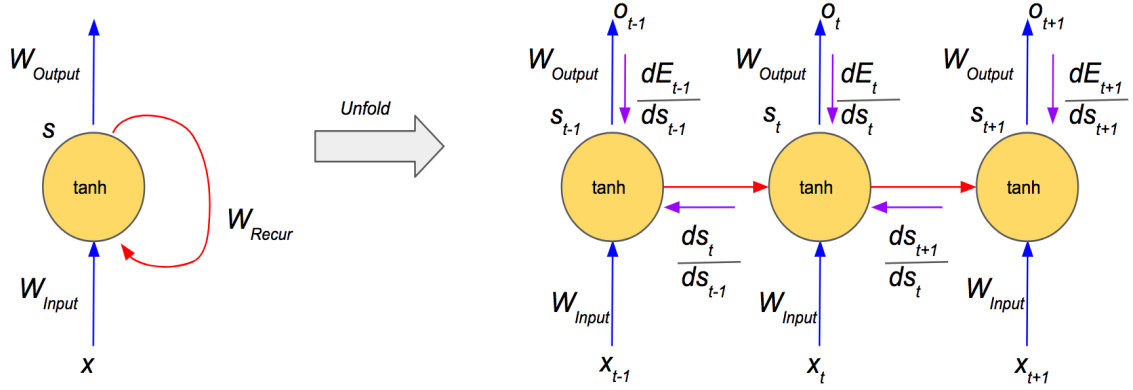
Figure 2.1: Gradient flow in RNNs: on the left is a simple RNN and on the right is its unrolled version. Purple arrows depict gradient flow back in time.

to the time step $t - 1$. Since the derivative of $tanh$ is bounded between 0 and 1, the gradient can vanish during backpropagation (Pascanu et al. (2013) This vanishing gradient problem leads to poor performance of RNNs in deep sequential tasks.

$$s_t = tanh(W_{input}x_t + W_{recur}s_{t-1} + b) \tag{2.1}$$

$$\frac{\partial E_{t+1}}{\partial s_{t-1}} = \frac{\partial E_{t+1}}{\partial s_{t+1}} \frac{\partial s_{t+1}}{\partial s_t} \frac{\partial s_t}{\partial s_{t-1}} \tag{2.2}$$

$$\frac{\partial s_t}{\partial s_k} = \prod_{k+1}^{t} \frac{\partial s_i}{\partial s_{i-1}} = \prod_{k+1}^{t} W_{recur} diag(tanh'(s_{i-1})) \tag{2.3}$$

2.2 LSTM

LSTMs were designed to overcome the vanishing gradient problem by controlling gradient flow using extra control logic and by providing extra linear pathways to transfer gradient without squashing Hochreiter and Schmidhuber (1997b). LSTM include three types of control gates: write control that determines the input to the memory state (with linear activation), forget gate that controls how much of the stored memory value is transferred to the next time step, and output gate which regulates the output of the memory cell. In addition, LSTM units can include extra peephole connections to probe the internal memory state. The peep-

hole connections allow the LSTM gates to be modulated based on value stored in the internal memory state. The output activation function is tanh. Structure of a single LSTM unit is depicted in Figure 2.2. Equations 2.4, 2.5, 2.6, 2.7, 2.8, 2.9 are the for the feedforward path. During backpropagation, the gradient for $\frac{\partial s_t}{\partial s_k}$ still goes through a couple of $tanh$ non-linearities. However, there is an additional linear pathway through memory cell c_t that preserves the gradient based on the control logic. Gated Recurrent Unit (GRU) is a variant LSTM in which the input and forget gates are merged for effeciency. Thus, gated recurrent networks outperform RNNs in long sequence processing task and are therefore applicable in many domains.

$$i_t = \sigma(W_{input_i}x_t + W_{recur_i}s_{t-1} + b_i) \tag{2.4}$$

$$\tilde{c}_t = tanh(W_{input_c}x_t + W_{recur_c}s_{t-1} + b_c) \tag{2.5}$$

$$f_t = \sigma(W_{input_f}x_t + W_{recur_f}s_{t-1} + b_f) \tag{2.6}$$

$$o_t = \sigma(W_{input_o}x_t + W_{recur_o}s_{t-1} + b_o) \tag{2.7}$$

$$c_t = c_{t-1} * f_t + \tilde{c}_t * i_t \tag{2.8}$$

$$s_t = tanh(c_t) * o_t \tag{2.9}$$

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t \tag{2.10}$$

$$\frac{\partial c_t}{\partial c_k} = \prod_{i=k+1}^{t} \frac{\partial c_i}{\partial c_{i-1}} = \prod_{i=k+1}^{t} f_i \tag{2.11}$$

$$\frac{\partial E}{\partial i_t} = \frac{\partial E}{\partial c_t}\frac{\partial c_t}{\partial i_t} = \frac{\partial E}{\partial c_t}\tilde{c}_t \tag{2.12}$$

$$\frac{\partial E}{\partial f_t} = \frac{\partial E}{\partial c_t}\frac{\partial c_t}{\partial f_t} = \frac{\partial E}{\partial c_t}c_{t-1} \tag{2.13}$$

$$\frac{\partial E}{\partial \tilde{c}_t} = \frac{\partial E}{\partial c_t}\frac{\partial c_t}{\partial \tilde{c}_t} = \frac{\partial E}{\partial c_t}i_t \tag{2.14}$$

$$\frac{\partial E}{\partial c_{t-1}} = \frac{\partial E}{\partial c_t}\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial E}{\partial c_t}f_t \tag{2.15}$$
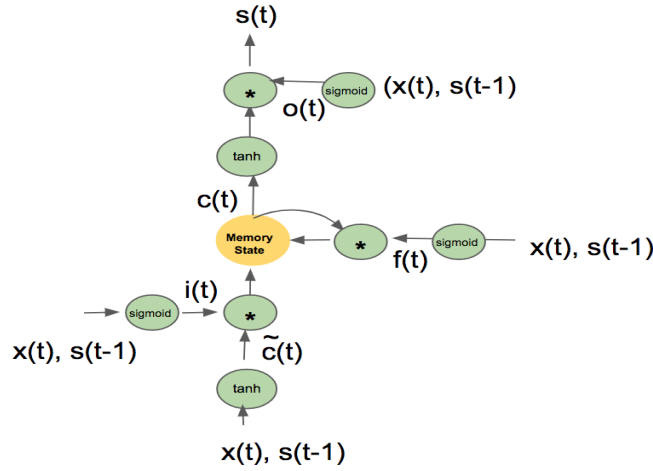
Figure 2.2: **LSTM node architecture:** LSTM include three types of control gates: write control that determines the input to the memory state (with linear activation), forget gate that controls how much of the stored memory value is transferred to the next time step, and output gate which regulates the output of the memory cell. For backpropagation, LSTM node is unrolled in time similar to RNN.

2.3 Applications of LSTMs

Gated recurrent networks like LSTMs and GRUs have made many real-world sequence processing applications possible, in particular those that include supervised training with time-series data. For e.g. they are widely used for supervised sequence learning problems like machine translation (Bahdanau et al. (2015a)), image captioning (Vinyals et al. (2015)), web traffic forecasting (Suilin (2017)), speech recognition (Bahdanau et al. (2015a)), handwriting recognition, automatic music transcription (Ycart and Benetos (2017)), sentiment analysis and stock market prediction.

Converting a sentence from English to French is an example of a Machine translation problem. Sequence-to-sequence LSTM models are often used for this purpose Sutskever et al. (2014). They consist of two parts: an encoder and a decoder. The input sentence is fed into the encoder one word at a time. The decoder outputs a sequence of words in the target language. The encoder outputs are ignored while the decoder outputs are used to compute the network loss. The advantage of such sequence to sequence model is that they are effective even when the input and output sequences are of varying length. One such encoder-decoder

model is used in this work for network learning curve prediction (see Section 5.7).

Image captioning is an example of LSTMs being used as a generative model. Given an input image, the goal is to generate a sequence of words describing that image. The model often consists of a convolutional network to process the image followed by a layer of LSTM. The LSTM layer functions as a type of language model that takes the image embeddings in its first time-step and generates a sequence of words in the subsequent time-steps (see Section 2.7.1 for more details on the language model).

Problems like speech and hand-writing recognition often use bi-directional LSTM Schuster and Paliwal (1997). Bi-directional LSTM process the data in both directions with two separate hidden layers, which are then fed forwards to the same output layer Graves and Jaitly ((2014).

LSTMs can also used be used for classification tasks like sentiment analysis. A set of text articles, tweets or reviews can be fed into the LSTM and the hidden value activation (s_t) of the final layer at the last time-step can be treated as an embedding for the whole article. This embedding can then be used to predict the sentiment of the article.

2.3.1   LSTMs for Reinforcement Learning problems

Although LSTMs are commonly used in supervised tasks, they can be used in RL as well, as both policy controller and function approximator. For example, LSTM based network was used as a function approximator in the robot navigation task Bakker et al. (2003). In this work, the input sequence to LSTM was pre-processed to capture salient information from the environment. An unsupervised event extraction was performed by classifying stream of inputs into a variable number of distinct classes. Any change in input stream class is considered an event, and is fed into a RL model consisting of LSTM function approximator. One drawback of this method is that it can ignore sequential information that remains fixed during a trial but changes across trials. Wierstra et al. ((2010) applied policy gradient algorithm to train LSTM networks resulting in deeper memories for POMDP tasks. LSTM layers can be combined with Deep Q-Network (DQN) to integrate information over time and perform better in Atari games Hausknecht and Stone (2017). LSTMs can be trained through policy gradient algorithm to solve

meta-learning problems like network achitecture search Zoph and Le (2016) and learning new learning rules for gradient descent Andrychowicz et al. (2016).

Bayer et al.Bayer et al. (2009a) evolved custom LSTM memory cells using mutation operators. These custom cells are then manually instantiated to construct LSTM network for solving the T-maze problem. Their result suggests that evolving LSTMs can lead to interesting solutions to POMPD problems that are difficult to solve otherwise. Yet the evolved memory is not deep enough to be useful in real-world AI tasks (like deep T-maze or modeling hyena emotions). New methods are required to scale the evolution of LSTM to such tasks. Powerful neuroevolutionary techniques (like NEAT) is one candidate approach to achieve complex memory solutions. NEAT can evolve both the topology and weights of a LSTM network in a non-parametric manner. Chapter 3 describes one such approach of evolvng LSTMs for RL.

2.4   Improvements in LSTMs

In recent past, several techniques have been developed to improve the performance of LSTMs. For e.g. weight regularization, activation regularization, dropouts, modifed stochastic gradient descent, variable length back propagation through time and new recurrent node and network architectures. Such variations are usually developed] on supervised learning problems like language modeling but can be extended to reinforcement learning domain. These improvements are often independent and combining them gives additive gain in performance. The following sections describe such improvements in detail.

2.4.1   Regularization

Regularization is a widely used machine learning technique to prevent the overfitting of model to the training data. It is only applied during training and not during inference. There are several ways to regularize neural network models. The traditional approaches for regularization include adding $L1$ and $L2$ weight penalty to the network loss function. For e.g when used in logistic regression, $L1$ penalty leads to sparse connectivity and $L2$ penalty leads to smaller weight values. Regularization through dropout in deep networks is a more recent advancement

and leads to better generarlization. In this method, a randomly selected subset of neurons are excluded from participating in the forward and backward pass. The model is forced to work even in the absence of dropped neurons and thus creates an ensemble effect and more robust model.

Initial experiments in applying standard feedforward dropouts to recurrent networks showed degradation in performance. Zaremba et al. (2014) proposed not dropping recurrent connections (W_{recur} in Figure 2.1) and applying dropouts only to feedforward paths (termed as vanilla recurrent dropout). Later, Gal and Gharamani (2015) showed that recurrent connections can be dropped but the same dropout mask should be applied across time steps within a sequence (termed as variational recurrent dropout). They also showed applying the same principle to input sequence dropout further improves generalization. Another approach is to limit updates to LSTM's hidden state by dropping out updates to control gates instead of the hidden unit themselves Semeniuta et al. (2016).

Smaller models often generalize better. To take advantage of this fact, Press and Wolf (2016) showed that for a language model, sharing input embedding with output weights reduces overall parameter count and makes it easier to train even larger models.

Another form of regularization modifies the hidden unit activations including a technique called recurrent batch normalization (Cooijmans et al. (2016)). Here, the hidden unit activations are whitened by scaling them to have zero mean and unit variance. The model activations thus stay in linear range of non-linearities like $tanh$ and thus prevent vanishing gradients.

Experiments in Chapter 3 do not use any regularization since the models are relatively small. Layer evolution (Chapter 4) uses vanilla dropouts as proposed in Zaremba et al. (2014) and node evolution (Chapter 5) uses variational dropouts (Gal and Gharamani (2015)), $L2$ weight regularization and shared embeddings(Press and Wolf (2016)).

2.4.2   Improvements through Architecture Modifications

Experiments in feedforward networks have previously indicated that architecture matters. Recent work suggests that the same could be true for recurrent networks. Grid-LSTMs (Kalchbrenner et al. (2015)) and gated feedback recurrent

networks (Chung et al. (2015)) are examples of LSTM variants with modified layer connectivity. Zilly et al. (2016) came up with a new kind of recurrent node called the recurrent highway network. It consists of a 10-layer deep network within each recurrent node. The internals of the weights are trained during backpropagation (unlike LSTM node where internal weights are fixed to 1.0).

Automatic design of networks is an emerging area of research and often described as meta-learning. The 'meta' algorithm could be either a reinforcement learning algorithm, evolutionary algorithm or a recurrent network itself.

Initial work in discovering new recurrent architectures did not yield promising results (Klaus et al. (2014)). However, a recent paper from Zoph and Le (2016) showed that policy gradients can be used to train a LSTM network to find better LSTM designs. In Zoph and Le (2016), a recurrent neural network (RNN) was used to generate neural network architectures, and the RNN was trained with reinforcement learning to maximize the expected accuracy on a learning task. This method uses distributed training and asynchronous parameter updates with 800 graphic processing units (GPUs) to accelerate the reinforcement learning process. Baker et al., (2017) have proposed a meta-modeling approach based on reinforcement learning to produce CNN architectures. A Q-learning agent explores and exploits a space of model architectures with an $\epsilon-$greedy strategy and experience replay. These approaches adopt the indirect coding scheme for the network representation, which optimizes generative rules for network architectures such as the RNN. Suganuma et al. (2017) propose a direct coding approach based on Cartesian genetic programming to design the CNN architectures.

2.5 Evolutionary Techniques - Genetic Programming, NEAT

Evolutionary algorithms fall into the category of gradient-free global search techniques that are very effective for solving non-differentiable problems or problems with very weak gradient information. For e.g. in reinforcement learning domain, evolutionary algorithms like NEAT are quite competitive with Q-learning and policy gradient methods (Stanley et al. (2003)). With the advent of deep learning, the policy gradient methods have made a comeback. REINFORCE trick allows a non-differentiable reward to be converted into a differentiable surrogate that can

be used as an objective to train the policy neural network using stochastic gradient ascent (Williams (1992)). One advantage of NEAT is that unlike policy gradient that only trains weights, NEAT can be used to evolve both the structure and the weights simultaneously.

Genetic Programming (GP) is another evolutionary technique that modifies tree structure instead of graphs (as in NEAT). Both NEAT and GP have variable length genotype encoding. The following two sub-section describe NEAT and GP in more detail.

2.5.1 NEAT

NEAT is a neuroevolution method that has been successful in solving sequential decision making tasks (Stanley and Miikkulainen (2002; 2004), Lehman and Miikkulainen (2014)). When used for RL, the evolved network is used as a policy controller that receives sensory inputs and outputs agent actions at each time step. NEAT begins evolution with a population of simple networks and gradually modifies the network topology into diverse species over generations. As new genes are added through mutations, they are assigned unique historical markings called innovation number. During crossover, genes with the same historical markings are recombined to produce offsprings. The population of networks is divided into different species based on number of shared historical markings. Speciation protects structural innovations in networks and maintains diversity by reducing competition among different species. The historical markings and speciation thus allow NEAT to construct complex task-relevant features. Memory can be introduced into the network by adding recurrent connections through mutations. NEAT is an ideal choice to evolve networks with memory in a non-parametric manner.

In Chapter 3, NEAT is used to evolve both the structure and weights of a LSTM policy controller network. In Chatper 4, a simplified version of NEAT with no speciation and crossover is used to evolve connectivity between two LSTM layers. NEAT is combined with GP to evolve gated recurrent nodes in Chapter 5.

2.5.2 Genetic Programming

Genetic Programming (GP) is an Evolutionary Computation technique that builds solutions represented as expresssions/programs. Similar to NEAT, it can start from a simple expression that eventually grows over time. At the start of evolution, a population of randomly generated simple expressions are evaluated. Genetic programming iteratively transforms a population of parent expressions into a new generation of the offspring by applying genetic operations like crossover and mutations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness.

The expression in the tree is often represented as a tree with left and right branches. For e.g. in Chapter 5, the gated recurret node is represented as a tree and therefore can evolved using GP. A variant of GP, called cartesian genetic programming, allows evolution of graphs as well (Suganuma et al. (2017)).

One of the challenges in scaling GP is that it can suffer from bloating i.e. programs can grow unnecessarily large without fitness improvements. To overcome this problem, the idea of diversity maintenance through speciation in NEAT can be combined with GP (Tujillo et al. (2015)).

2.6 Diversity in Evolution

Fitness landscape in many high-dimensional problems is deceptive that can lead to local optima. When evolving topologies of the network, this effect can lead to bloating Tujillo et al. (2015). Speciation in NEAT addresses this problem by maintaining different kinds of networks in the population. The same ideas can applied to genetic programming (as shown in Chapter 5).

Diversity can be used to characterize the behavior of the network in RL or in supervised learning domains. For e.g. the set of actions that the agent takes within an episode could be its behavioral vector and the agent's behavioral novelty can be measured in terms of the vector distance from agents' actions (Lehman (2012)). In the supervised problem like language modeling defining network diversity is more compute intensive because of the presence of vastly more data points. Therefore, diversity of network structure is often measured and maximized. Novelty

search is one such objective that can applied to either maximize behavioral or structural diversity. Howerver, it converts the problem into multi-objective problem. In Chapter 5, structure diversity of recurrent nodes is maximized using a Hall of Shame. This includes keeping a fixed size archive of already explored regions in the architectural search space and preventing reproduction from spawning individuals in those areas. Thus, we can induce structural novelty without explicitly maximizing it.

For RL problems, the problem of deception is more pronounced due to sparser rewards. For e.g. in a maze search problem, where a small reward can be found at a nearby corner from the agent and a high reward placed far away, the agent learns to greedily act and settle for smaller reward. To overcome this, Chapter 3 presents a new information maximization objective that motivates the agent to explore areas that provides new information. The information maximization objective quanitfies the novetly of information in terms of entropy. This additional objective is closely aligned with NEAT such that the evolved networks have maximial information stored in LSTM nodes in the most efficient way possible.

2.7 Problem Domains

Three benchmark problems are used to evaluate the technology developed in this work. Deep T-maze is a RL problem used in Chapter 3. Chapter 4 and 5 use language modeling as a benchmark problem. The solution evolved in Chapter 5 for language modeling is then transfered to music data.

2.7.1 Language

The goal of language modeling is to estimate the probability distribution of words $P(w)$ in a English language sentence.

This has application in speech recognition (Graves and Jaitly ((2014)) and machine translation(Bahdanau et al. (2015b)) and text summarization. Training better language models (LM) improves the underlying metrics of the downstream task (such as word error rate for speech recognition, or BLEU score for translation), which makes the task of training better LMs valuable by itself. Further, when trained on vast amounts of data, language models compactly extract knowledge
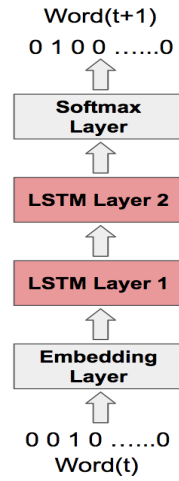
Figure 2.3: **LSTM based Neural Network Language Model**. Words are encoded as one-hot vectors and fed into a embedding layer. Two LSTM layers are stacked on top of the embedding layer and the final layer is softmax. The network outputs word probabilities for the next word in the sentence. In this manner, the network can be used for next word prediction.

encoded in the training data. For example, when trained on movie subtitles, these language models are able to generate basic answers to questions about object colors, facts about people, etc. (Józefowicz et al. (2016))

Statistical language models have been previously used where the model learns probability of word occurrence based on examples of text. Bi-grams and tri-grams are examples of such N-gram models where word transition probabilities are computed by counting frequency of co-occurence of words in the training data. Such models are simple to design and work well for short sentences but are not powerful enough to capture dependencies in longer sentences.

Recently, the use of neural networks in the development of language models has become very popular, to the point that it may now be the preferred approach. Nonlinear neural network models solve some of the shortcomings of traditional language models due to their ability to generalize across longer, variable length sequences. Figure 2.3 shows example of one such LSTM model. In every time step, the model takes in a one hot encoded word as input. A linear embedding layer converts the discrete word input into a real-valued vector to represent each word in the projection space. A stack of two LSTM layers combine the word embedding

with the context stored in their memory. Finally the softmax layer outputs word probabilities. Such a model can be trained using stochastic gradient descent (SGD).

Recurrent Neural Networks based LMs employ the chain rule to model joint probabilities over word sequences:

$$p(w_1, w_2, ...w_n) = \prod_{i=1}^{N} p(w_i|w_1, w_2...w_{i-1}) \tag{2.16}$$

where the context of all previous words is encoded with an LSTM, and the probability over words uses a Softmax (see Figure 2.3).

A good language model is one which assigns higher probability to grammatically correct or frequently observed sentences than ungrammatical or rare ones. One metric to measure this performance is called word perplexity.

$$Perplexity(w_1, w_2, ...w_n) = P(w_1, w_2, ...w_n)^{\frac{-1}{n}} \tag{2.17}$$

$$Perplexity(w_i) = exp(Average\ Cross\ Entropy\ Loss\ per\ Word) \tag{2.18}$$

$$Average\ Cross\ Entropy\ Loss\ per\ Word = \sum_{i=0}^{N-1} -P_{i_{target}}log(P_{i_{predicted}}) \tag{2.19}$$

A completely random model that assigns every word in the sentence equal probability ($p = \frac{1}{N}$, where N is the vocabulary size) will have perplexity of N. This is the worst case performance of the model. Thus, a model with lower perplexity value on the test data is considered better.

In this word, the experiments are focused on the task of predicting the next word in the Penn Tree Bank corpus (PTB), a well-known benchmark for language modeling Marcus et al. (1993). LSTM architectures in general tend to do well in this task, and improving them is difficult Zaremba et al. (2014) Jozefowicz et al. (2015) Gal and Gharamani (2015). The dataset consists of 929k training words, 73k validation words, and 82k test words, with a vocabulary of 10k words. During training, successive minibatches of size 20 are used to traverse the training set

sequentially.

2.7.2 Music

Music consists of a sequence of notes that often exhibit temporal dependence. Predicting future notes based on the previous notes can therefore be treated as a sequence prediction problem. Similar to natural language, musical structure can be captured using a music language model (MLM). Just like natural language models form an important component of speech recognition systems, polyphonic music language model are an integral part of Automatic music transcription (AMT). AMT is defined as the problem of extracting a symbolic representation from music signals, usually in the form of a time-pitch representation called piano-roll, or in a MIDI-like representation. This remains a challenging problem, in particular in the case of polyphonic music (Lewandowski et al. ((2012), Lavrenko and Pickens (2003.)).

MLM predict the probability distribution of the notes in the next time step. Multiple notes can be turned-on at a given time step for playing chords. The architecture of MLM is very similar to the one shown in Figure 2.3. One key difference is that the output layer in case of MLM consists of a sigmoid layer (for chords). The loss function used for training the network is cross-entropy between predicted note and the target notes (see equation 2.19). The metric used to evaluate MLM is called the F-measure. F-measure is computed on the test data by taking the geometric mean of precision and recall.

The input is a piano-roll representation, in the form of an $88XT$ matrix M, where T is the number of timesteps, and 88 corresponds to the number of keys on a piano, between MIDI notes A0 and C8. M is binary, such that $M[p, t] = 1$ if and only if the pitch p is active at the timestep t. In particular, held notes and repeated notes are not differentiated. The output is of the same form, except it only has $T - 1$ timesteps (the first timestep cannot be predicted since there is no previous information).

Piano-midi.de dataset is used as the benchmark data. This dataset currently holds 307 pieces of classical piano music from various composers. It was made by manually editing the velocities and the tempo curve of quantised MIDI files in order to give them a natural interpretation and feeling (Ycart and Benetos (2017)).

MIDI files encode explicit timing, pitch, velocity and instrumental information of the musical score.

2.7.3   RL Memory Tasks

Most real-world RL applications are POMDPs and require some form of recurrency to solve the problem of state aliasing. In order to sucessfully solve such problems, a combination of sophisticated RL algorithm and simple memory architecture is required. For e.g. in the atari video game, while the overall agent performance improves with the addition of an LSTM layer, the same improvement can be achieved by feeding multiple consecutive frames to a feedforward network (Hausknecht and Stone (2017)). Since the goal of experiments in Chapter 3 is to solve deep memory problems, a customized task is designed for this purpose called deep T-maze.

Standard T-mazes are widely used as testbed for RL problems. At the beginning of each trial, the agent observes one light. As the agent moves forward in the aisle towards the T-junction, it no longer has access to the light. The color of the light (red/green) indicates the direction (right/left) that the agent should take at T-junction in order to reach the goal. Therefore, to be successful, the agent needs to memorize the color of the light that was shown at the start. The problem of T-maze can be easily scaled to a Deep T-maze. In this case, there are multiple input lights at the start corresponding to the multiple T-junctions. In order to be successful, the agent is required to recall the input light sequence in the correct order at each T-junction. This task requires deeper memory than the simple T-maze (See Figure 3.2 for details).

Chapter 3

Evolving LSTM Network Structure and Weights using Unsupervised Objective - InfoMax

In this chapter [1], NEAT (Neuroevolution of Augmenting Topologies) Stanley and Miikkulainen (2002) algorithm is extended to incorporate LSTM nodes (NEAT-LSTM). Since NEAT algorithm can evolve network topologies, it can discover the correct amount of memory units for the task. NEAT-LSTM outperform RNNs in two distinct memory tasks. However, NEAT-LSTM solutions do not scale as the memory requirement of the task increases. To overcome this problem, a secondary objective is used that maximizes the information stored in the LSTM units. The LSTM network is first evolved during the pre-training phase with this unsupervised objective to capture and store relevant features from the environment. Subsequently, during the task fitness optimization phase, the stored LSTM features are utilized to solve the memory task. This approach yields LSTM networks that are able to solve deeper memory problems. Two memory tasks are used to compare the performance of different algorithms: sequence recall and sequence classification.

3.1 Problem of Deception

From an optimization perspective, since the problem of evolving memory is deceptive, extra objectives (to promote solution diversity) can be used to overcome deception Lehman and Miikkulainen (2014), Ollion et al. (2012). In such approaches, the evolutionary optimization problem is often cast as a multi-objective problem with two objectives - primary objective (task fitness) and secondary diverstiy objective (like novelty search). However, there are no guarantees that such diversity objectives can aid the learning algorithm to capture and store useful historical information from the environment. Since the secondary diversity objective is unrelated to the task fitness, the network can also undergo unsupervised pre-

---

[1] this chapter is based on the paper Rawal and Miikkulainen (2016). Co-author Risto Miikkulainen provided feedback through discussions.

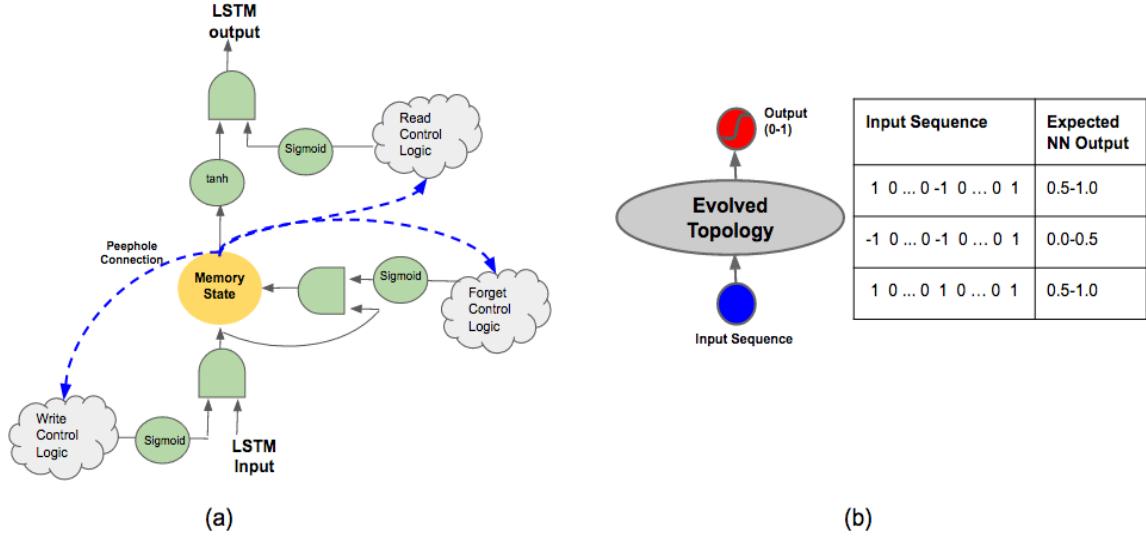|	Input Sequence	Expected NN Output
	1 0 ... 0 -1 0 ... 0 1	0.5-1.0
	-1 0 ... 0 -1 0 ... 0 1	0.0-0.5
	1 0 ... 0 1 0 ... 0 1	0.5-1.0

(a)   (b)

Figure 3.1: (a) **The LSTM node with peephole connection:** a single LSTM unit is comprised of three components: (1) green-colored multiplication gates and sigmoid/tanh activation function, (2) yellow-colored internal memory state with linear activation, and (3) blue-colored peephole connections. There are three multiplication gates (write, forget and read) that control the flow of information through the LSTM unit. The peephole connections allow the internal memory state of the LSTM unit to be probed. The peripheral control logic (shown in gray cloud-shaped boxes) and the peephole connection weights are modified during the course of evolution. (b) **The Sequence-Classification Task:** This is a binary classification task where the network has to count whether the number of 1s in the input sequence exceeds the number of -1s (0s are ignored). At the start of evolution, the network consists of one input node and one output node. NEAT evolves hidden layer topology. A network requires memory in order to succeed in this task. Example of input-output sequence values are shown in the table.

training to optimize this objective. One such unsupervised objective is presented in this dissertation that maximizes the total theoretic information stored in the LSTM network.

3.2   Unsupervised Training of LSTM

In supervised learning domain, unsupervised pre-training of neural networks to initialize its parameters has been shown to improve the overall network task performance significantly. This idea of pre-training the network with unsupervised objective can also be extended to the RL domain. However, not much literature exists on the topic of unsupervised pre-training of LSTM for RL POMDP tasks. On a related note, Klapper-Rybicka et al. (2001) performed unsupervised clustering of musical data using LSTM networks. In this work, Binary Information Gain Optimization (BINGO, Schraudolph and Sejnowski (1993)) was used as an unsupervised objective. BINGO maximizes the information gained from observing the output of a single layer network of logistic nodes, interpreting their activity as stochastic binary variables. One limitation of BINGO is that it searches only for uncorrelated binary features (thus the solution ends up having zeros and ones in equal parts), which limits the amount of information the network can store. Instead, LSTM features with maximal information are evolved in this work. Specifically, the LSTM networks are evolved (using NEAT) to first extract and store independent (and highly informative) real-valued features. These features are then later used to solve the memory task. Storing independent features in the LSTM ensures that the network has maximal information from a theoretic perspective. This information-theoretic objective to train LSTM network is similar to the info-max approach published in Bell and Sejnowski (1995). However, the main difference is that there is no underlying assumption on the number of mixed features and the linearity of the mixture.

3.3   Memory Tasks

This section describes two memory tasks. Both the tasks are situated in a discrete maze where the agent moves one-step at each time-step of the trial. The
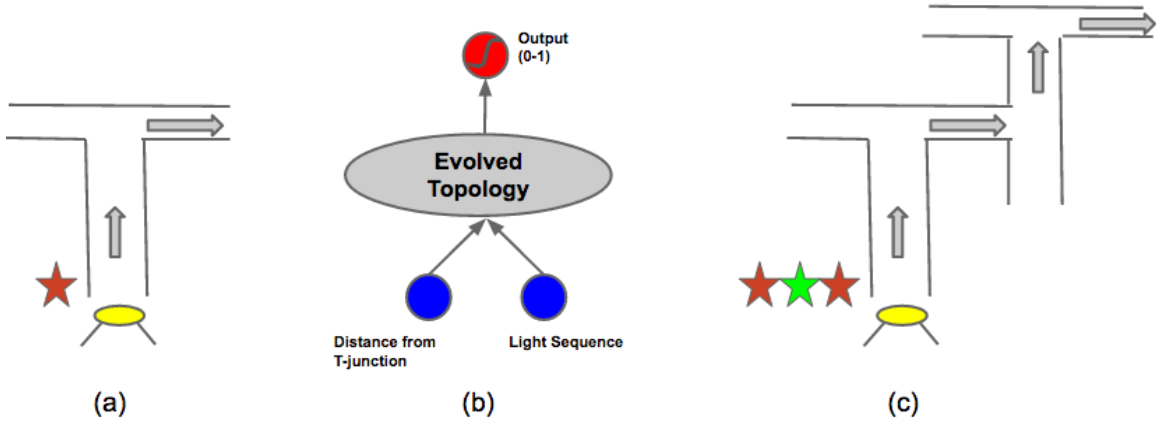
Figure 3.2: (a) Sequence Recall in simple T-maze: At the beginning of each trial, the agent observes one light. As the agent moves forward in the aisle towards the T-junction, it no longer has access to the light. The color of the light (red/green) indicates the direction (right/left) that the agent should take at T-junction in order to reach the goal. Therefore, to be successful, the agent needs to memorize the color of the light that was shown at the start. (b) Network architecture: In this task, the network has two inputs: one input represents the distance to the T-junction and the second input is the light sequence (active only during the first few time steps) (c) Deep T-maze: In the Deep T-maze, there are multiple input lights at the start corresponding to the multiple T-junctions. In order to be successful, the agent is required to recall the input light sequence in the correct order at each T-junction. This task requires deeper memory than the simple T-maze.

first task, sequence classification, is a binary classification problem of streaming input. The second task, sequence recall, requires the agent to recall a previously provided instruction input and use it to make future decisions (turn left or right at the T-junction). In both the tasks, the agent is required to store and to utilize the past events in order to be successful in the task.

3.3.1 Sequence Classification

This is a binary classification task, where given an input sequence of 1 and -1 (interleaved with 0s), the network needs to determine whether it received more 1s than -1s. The number of interleaved 0s in the input sequence is random and ranges between (10-20 time steps) . This task can be visualized as a maze positioning task. An agent situated at the center of a one-dimensional maze is provided instructions to move either left/right. It is expected to move left (west) when its input is -1 and move right (east) when its input is 1. When its input is zero, it is not expected to move. At the end of the sequence, the agent needs to identify whether it is on the right side of the maze or the left side i.e. has it taken mores steps towards east than west. Note, that input 0 does not affect this decision but only serves to confuse the agent. The number of turns (left/right) that the agent takes during a trial is defined as the depth of the task. For example, a sequence of four turns (total four 1/-1 inputs interleaved with 0 in the input sequence) is termed as 4-deep. Different sequence classification experiments are carried out by varying the depth of the task (4/5/6 deep). The network architecture and example input-output combinations are shown in Figure 3.1b.

3.3.2 Sequence Recall

In the simple T-Maze, at the beginning, an agent receives an instruction stimulus (like red/green light) (see Figure 3.2a). The agent then travel a corridor until it reaches the T-junction. At the junction, the path splits into two branches (left/right) and the agent needs to take the correct branch in order to reach the reward. The position of the reward is indicated by the instruction stimulus it received at the beginning. For example, red light can indicate presence of reward on the right-branch and green light can indicate the reward on left branch. A suc-
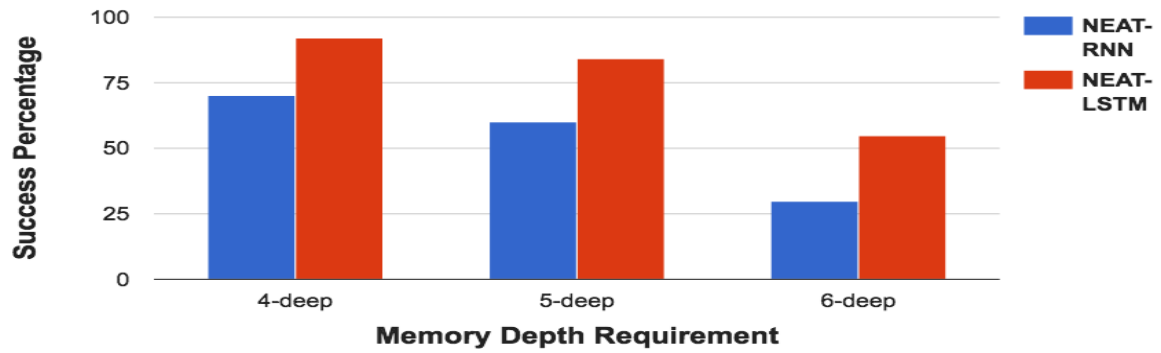
Figure 3.3: NEAT-LSTM vs. NEAT-RNN comparison on the Sequence-Classification task. The memory depth requirement is varied on the x-axis. The y-axis values represents the success rate of each method in 50 runs. The performance of NEAT-RNN and NEAT-LSTM is comparable for 4-deep sequence classification. As the task depth is increased to 6-deep, NEAT-LSTM significantly outperforms NEAT-RNN. Successful solutions to the sequence-classification task should be able to retain an aggregate of previous inputs and should also continuously update this aggregate with new incoming inputs. The performance results indicate that LSTM-based networks can memorize information over longer intervals of time as compared to RNNs.

cessful agent thus can only maximize collecting the reward by memorizing and utilizing its stimulus instruction. T-Maze has been widely used as a benchmark problem for evaluating agents with memory Bakker (2002), Bayer et al. (2009a), Lehman and Miikkulainen (2014), Ollion et al. (2012).

This simple T-maze (with one T-junction) can then be extended to a more complex deep T-maze which consists of a sequence of independent T-junctions (Figure 3.2c.). Here, the agent receives a sequence of ordered instructions (one corresponding to each T-junction decision) at the start of trial and it has to utilize the correct instruction at every T-junction in order to reach the goal. Risi et al. Risi et al. (2010.) used one such T-maze extension (double T-maze) to test plastic neural networks. The distracted sequence recall task used in Monner and Reggia (2012) is another variation of the deep T-maze recall, but it uses supervised training to learn the LSTM parameters. However, the approach presented in this dissertation uses a weak fitness signal (proportional to the number of correctly recalled input instructions) to train the memory network. Scaling the memory depth of network to recall long sequences in such RL settings has been a challenge.

3.4   Experiments

The following experiments have a three goals. First, to compare the performance of RNNs vs. LSTM in the memory tasks. Since LSTMs outperform RNNs in supervised domain, the hypothesis is that they can do the same in the RL domain as well. Second, the LSTMs based networks are scaled to deeper memory tasks to understand their bottlenecks. Finally, a new unsupervised objective called Info-Max is presented that enables deeper LSTMs.

In each experiment, a population of 100 networks is evolved using NEAT for 15,000 generations. Each individual in the population is evaluated on all possible input sequences. Deeper tasks therefore require more evaluations than shallow ones. The length of each trial also depends on the task-depth. For example, a 4-deep sequence classification task consists of at least four time steps (corresponding to four 1/-1) and 40 time steps of 0 inputs interleaved between 1/-1. During evolution, out of a total fitness of 100, the network receives a fraction for correctly predicting a part of the problem. For example, in a 4-deep sequence classification
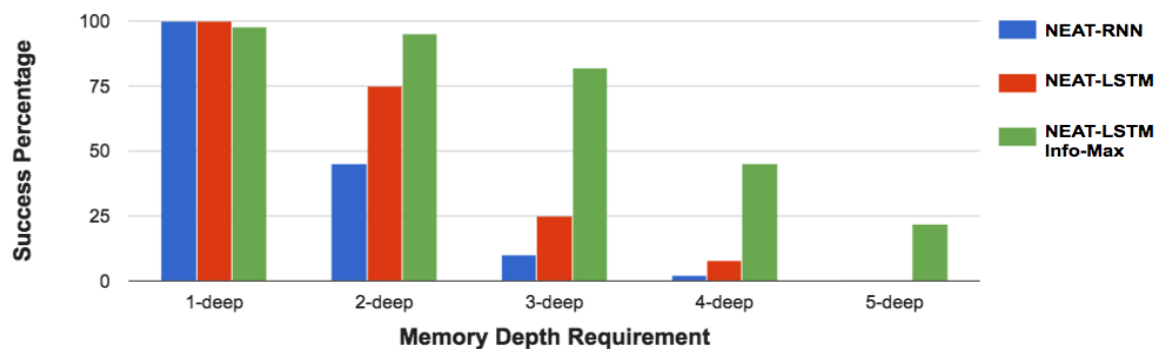
Figure 3.4: NEAT-LSTM, NEAT-RNN and NEAT-LSTM Info-max comparisons on Sequence-Recall task. Success percentage of each method is plotted for T-maze with varying depth. Both NEAT-RNN and NEAT-LSTM can quickly find the solution to simple one-step T-maze. As the maze becomes deeper, NEAT-LSTM outperforms NEAT-RNN. However, beyond 3-deep, the problem becomes too complex for both the methods. A solution to the deep T-maze problem requires memorizing the input light sequence in its correct order for several hundred time-step. NEAT-LSTM Info-max can successfully find solutions for even 5-deep recall. This suggests that pre-training of LSTM networks with unsupervised Info-max objective results in the capture of useful information from the environment that can later be used to solve the memory task.

problem, if the agent correctly predicts its position in the maze (left or right side) after two turns, then it receives 50 fitness points. It is expected that such partial reward will shape the network towards evolving optimal behavior.

Data is collected from 50 runs of each experiment type and the average success rate (percentage of runs that yield solution with maximum fitness) is measured to compare performance of different methods. At each time step, the network under evaluation is activated once. The value at the input of a node is propagated to its output in a single time step. Therefore, the number of time steps it takes for the network input to reach the output is equal to the shortest path between input and output. This setup is critical in ensuring that the network captures the input sequence values and their order correctly in the recall task.

3.4.1   Experiment 1: Comparing RNNs vs. LSTM

First, RNNs are evolved using standard NEAT algorithm (labeled as NEAT-RNN). A user-defined parameter controls the probability (set to 0.1 in these experiments) of adding a new recurrent link. The recurrent links can be either self-loops or a longer loop consisting of several intermediate nodes. Next, NEAT is extended to evolve LSTM-based networks that are compared with RNNs.

Method: NEAT-LSTM

Standard NEAT algorithm can add new nodes (sigmoid or rectified linear units) through mutation. In this work, NEAT is extended such that during its search process, it can add new LSTM units (probability of adding a new LSTM unit is 0.01). On being first instantiated, LSTM gates have default values - always write, always read and always forget. This setting ensures that initially, newly added LSTM units do not affect the functionality of the existing network. Each new instantiation of a LSTM unit is associated with the corresponding addition of a minimum of six network parameters - three connections from external logic to the control gates (depicted as cloud shaped gray boxes in Figure 3.1a) and three peephole connections (blue-colored links in Figure 3.1a). During the course of evolution, the existing parameters can be modified/removed and new ones can be added to suit the requirements of the task. No recurrent connections are allowed

except the recurrency that exists within the LSTM unit.

Results

As shown in Figure 3.3, both NEAT-RNN and NEAT-LSTM can solve the 4-deep sequence classification task easily. As memory depth requirement increases to six, their success rate gradually decreases. NEAT-LSTM significantly outperforms NEAT-RNN in all the cases (t-test; $p<0.01$). The sequence-classification task is relatively simpler than the sequence-recall task. The agent is required to update and store its internal state with each new valid input (1/-1). Therefore, the successful networks have simple architecture (consisting of only a few recurrent neurons in the case of NEAT-RNN and a single LSTM in the case of NEAT-LSTM).

In the sequence-recall task, both NEAT-RNN and NEAT-LSTM quickly find the solution to one-step T-maze (Figure 3.4). This result is expected and it matches the outcomes of previous papers that focused on this problem (Lehman and Mikkulainen (2014), Wierstra et al. ((2010)). However, as the T-maze becomes deeper, the successful solutions are required to store input light sequence information for longer durations and in its correct order. Therefore, beyond 3-deep T-maze, the problem becomes too complex for both NEAT-RNN and NEAT-LSTM to solve.

3.4.2 Experiment 2: Scaling NEAT-LSTM

In the harder task of sequence recall, the agent needs to store the entire input sequence in correct order. Incremental evolution approach was applied in order to solve deeper recall problems (>4-deep). Networks were first evolved for 2-, 3-, and 4-deep recall problems; these networks were then used as a starting point (in the NEAT algorithm) to solve the more complex problems of increasing depth (>4-deep). However, this approach did not yield much success. The problem may be that as the length of the input sequence increases, the number of parameters to be evolved also increase (with each additional LSTM units). Also, the incremental evolution approach requires detailed knowledge of the problem domain that is not inline with the goal of this dissertation (i.e. to solve the memory task with limited domain knowledge).
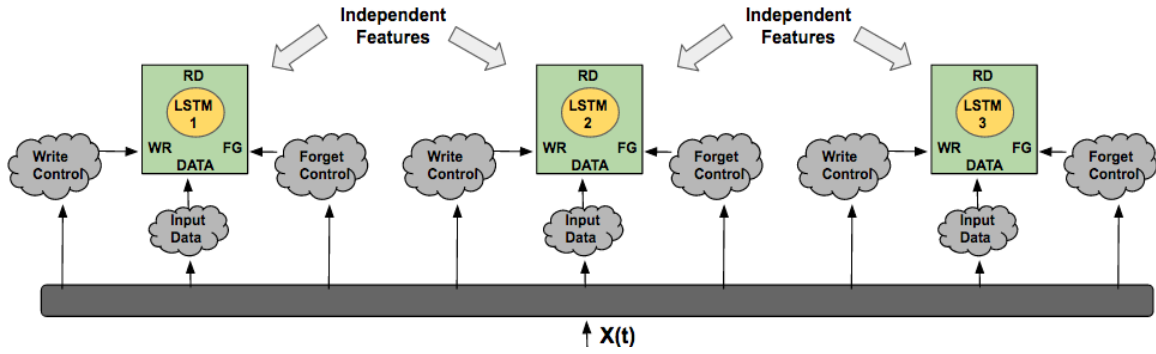
Figure 3.5: Unsupervised Pre-training using the Information Maximization objective. Highly informative, independent LSTM features are incrementally added by modifying the write control, forget control and input data logic using NEAT. Unsupervised pre-training is carried out until evolution stops discovering independent features or at the end of 10000 generations. By the end of this pre-training phase, salient information observed by the agent is captured and stored in the LSTM network. The pre-training aids evolution of deeper memory solutions in two ways. First, it reduces the problem of deception by directing evolutionary search towards landscapes that provide new information. Second, by incrementally adding independent features, it avoids the problem of training a large number of LSTM parameters simultaneously.

Method: Information Maximization Objective

One way to overcome the problem of deception is by ensuring that evolution discovers unique time-dependencies by not re-discovering existing information. In solving POMDP tasks that require memory, the agent can benefit by capturing salient historical information from the environment and storing it in its neural-network controller. As the agent moves in the environment over multiple trials, it comes across a lot of information. For example, it can observe the wall (which are mostly static across trials) or it can observe a binary light (red/green in T-maze), which can provide possible clues for the location of the goal. It is difficult to discern which information should be stored for later use (blinking light) and which should be discarded (static walls).

One solution could be to store inputs (in their native form or in combination with other inputs) such that the total information stored in the network is maximized. From a theoretical perspective, the information stored in a set of random variables is maximized when their joint entropy is maximized. The joint entropy of two random variables X and Y can be defined as

$$H(X,Y) = H(X) + H(Y) - I(X,Y), \tag{3.1}$$

$$H(X) = \sum P(X) \log P(X), \tag{3.2}$$

where $H(X)$, $H(Y)$ are the individual entropies of X and Y respectively and $I(X, Y)$ is the mutual information between X and Y. Thus, to maximize the information stored in the network, the individual entropy of its hidden unit activations should be maximized and their mutual information minimized. The mutual information can be expressed as Kullback-Leibler divergence:

$$I(X,Y) = \sum P(X,Y) \log \frac{P(X,Y)}{P(X)P(Y)} \tag{3.3}$$

Random variables with zero mutual information are statistically independent. Computing and storing highly informative, maximally independent features (i.e. features with high individual entropy and low mutual information) in the network is the unsupervised objective that will be used for NEAT (Info-max). The features are stored in the hidden LSTM units, since these units can retain informa-
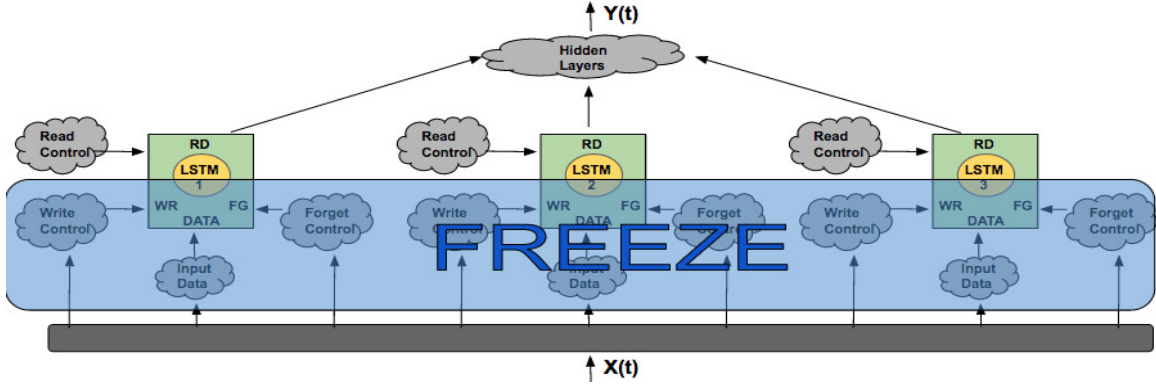
Figure 3.6: RL training using the fitness objective. During the RL phase, the stored/memorized features are utilized to solve the memory task. The write control, forget control and input data logic of the LSTM units (that store independent features) is frozen and the read control logic is evolved using NEAT. NEAT can add new hidden layers over the top of existing LSTM network.

tion over several time steps. Evolving independent features allows for pre-training the network and only augments the NEAT-LSTM approach as outlined in Section 4.1.1. The new learning algorithm now consists of two steps: The first one is an unsupervised objective phase where the independence of LSTM hidden units is maximized (see Figure 3.5), and the second is an RL phase where the fitness objective is maximized (Figure 3.6).

A feature vector is constructed by concatenating the memory state values (yellow-colored circle in Figure 3.1) of the LSTM unit from each neural network activation across different trials. There is one distinct feature vector corresponding to each LSTM hidden unit. To compute entropy and mutual information, the feature vectors are treated as random variables. The real-valued features are partitioned into 10 equal-sized bins, and a histogram is constructed by counting the number of elements in each bin. Entropy and pairwise mutual information (an approximation of total mutual information) of feature histograms is then calculated using equation 3.2 and equation 3.3 respectively.

Evolving multiple independent features simultaneously can be challenging. This problem can be broken down using incremental evolution Gomez and Mikkulainen (1997) (without the need for domain knowledge). Independent features (with their values stored in individual LSTM units) can be discovered one at a time

35

using NEAT. Since the primary goal of this work is to build networks with memory, one simplifying assumption can be introduced: environment is not dynamic, i.e. it does not change over time. With this assumption, independent features once discovered and stored in LSTM need not change over a period of time. The stationarity assumption also entails that during the unsupervised training phase, only the write control logic, forget control logic and input data logic of the LSTM unit need to be modified. Once the independent features have been discovered, no more changes to the write, forget and input logic of the LSTM units are required (i.e. they are frozen). For the remainder of evolutionary search, NEAT can only add outgoing connections from the frozen network (to facilitate re-use of frozen logic). Often, there exists multiple solutions to the problem of finding networks with maximum information. Some of these network solutions could be large. To bias NEAT towards evolving smaller solutions during unsupervised objective optimization, a regularization factor is introduced that penalizes larger networks. The size of the evolved network (equal to the number of network connections) is weighted by a regularization parameter (value varies between 0 and 1), and the resulting penalty term is subtracted from the unsupervised objective value. Unsupervised training is stopped either when NEAT cannot find any more independent features or at end of 10000 generations.

Subsequently, during the RL phase, the LSTM outputs are provided as extra inputs (in addition to the sensor inputs) to the NEAT algorithm. NEAT evolves the read control logic of the frozen LSTM units to utilize the stored features appropriately as deemed fit for the task. This approach makes neuroevolution computationally more tractable.

Results

NEAT-LSTM with the information maximization objective (NEAT-LSTM Info-max) was evaluated on sequence-recall task, since it is the harder of the two memory tasks. As shown in Figure 3.4, NEAT-LSTM and NEAT-RNN outperform NEAT-LSTM Info-max in the 1-deep task. This is probably because NEAT-LSTM and NEAT-RNN are powerful enough to quickly find a solution to the shallow memory problem. As the memory depth requirement increases however, NEAT-LSTM Info-max consistently outperforms NEAT-LSTM and NEAT-RNN (t-test;

p<0.01). NEAT-LSTM Info-max is able to solve 5-deep sequence recall problem about 20% of the time.

In the sequence-recall task, the networks evolved using NEAT-LSTM Info-max method can preserve information (in correct order) over hundreds of time steps. This result suggests that pre-training of LSTM network with information maximization objective facilitates the capture of useful information from the environment. To confirm this hypothesis, pairwise mutual information was computed between LSTM feature vectors and the network inputs. Since each input light in the input sequence is independent of the other, maximum-information objective should often yield solutions such that different LSTM units capture information from distinct previous inputs. This was often found to be true. The info-max objective drives the network evolution towards gathering maximum possible information. Note that unlike novelty search, info-max is not open-ended since it depends on the richness of observable information in the environment. Further comparisons between Info-max and novelty search on memory tasks can be found at the demo page (link provided in Section 4) . While the idea of maximizing agent information is proposed to increase the depth of the agent's memory, it can indirectly lead to exploratory behaviors. Artificial curiosity Schmidhuber (2010) is one such concept, where the agent is explicitly rewarded for exploring areas in the environment that provide more information.

When neuroevolution is used to discover features, a rich feature set may accumulate. Recently, feature accumulation through evolution has been successful in solving both supervised learning problems and RL problem. For example, Szerlip et al. Szerlip et al. (2015) used novelty search as an unsupervised objective to train HyperNEAT networks for MNIST digit recognition problem. Koutnik et al. Koutnik et al. (2014) used a similar diversity objective (during unsupervised training) to train Convolutional Networks for simulated car racing. The Info-max objective presented in this dissertation is customized for memory tasks and therefore, should work better in POMDP problems. The idea of maximizing information gain for advancing the complexity of behaviors is biologically plausible as well Adami (2012).

3.5 Conclusions

Incorporating memory into artificial agents in a non-parametric manner is a challenging problem Doshi (2009). As a solution to this problem, LSTM networks are evolved using NEAT (NEAT-LSTM). NEAT discovers the appropriate number of LSTM units suitable for a given task. Evaluation on two memory tasks indicate that LSTM networks outperform RNNs in shallow POMDP tasks. To further scale the evolution of LSTM to deeper memory problems, a new information maximization (Info-max) objective is devised. The LSTM networks are pre-trained by optimizing for this unsupervised objective. During the course of pre-training over several generations, LSTM units incrementally capture and stored unique features from the environment. After this pre-training phase, the LSTM network is evolved to solve the memory task. The strong performance of NEAT-LSTM Info-max on deep sequence recall task indicates its utility in building generic AI agents that can solve both MDPs and POMDPs.

Next, LSTM networks are evolved for supervised learning tasks like language-modeling. LSTM networks for supervised learning are large and face different kinds of challenges than the one evolved for RL. However, the main insight from this chapter, that evolution can be an effective tool to automatically construct LSTM networks, is extended to new kinds of problems in the next chapter.

Chapter 4

Evolving Multi-layered LSTM structures as Graphs

Recent research in LSTMs has been broadly focused in two directions - first, in finding variations of individual LSTM memory unit architecture (Bayer et al. (2009b), Jozefowicz et al. (2015), Cho et al. (2014)) and second, in discovering new ways of stitching LSTM layers into a network (Kalchbrenner et al. (2015), Zilly et al. (2016)). The first approach has led to marginal performance improvements over the vanilla LSTMs (Klaus et al. (2014)) while the second approach has met with more success (Chung et al. (2015)). In this chapter [1], the two approaches are combined and a hybrid is proposed - i.e. neuroevolution is used to search for both new LSTM units and their connectivity across multiple layers at the same time.

Specifically, multiple LSTM layers are flattened into a neural network graph that can then be modified by neuroevolution. In each generation of evolution, a population of networks with LSTM variants is created through mutation, and the individual networks are trained/tested on benchmark task of language modeling. At the end of training for this benchmark task, the test perplexity scores for each network is its negative fitness value (in the language modeling task, lower perplexity value is better). Thus, neuroevolution only modifies the network architecture while the stochastic gradient descent updates the network weights over several epochs of training. During selection, binary tournament is used to select individuals for mutuations. There are two types of mutation possible - first, that disables a network connection and second, that adds a skip connection between two nodes. Recently, skip connections have led to performance improvements in deep neural networks, which suggests that they could be useful for LSTM networks as well.

4.1 Evolution Search Space

Two stacked LSTM nodes are opened up and presented as input for evolution to modify. For e.g. Figure 4.1 shows two layered stacked LSTM on the left. The stacked LSTM layers are shown along with their internal logic on the right. A

---

[1]based on the paper Miikkulainen et al. (2018). My contribution included designing and running experiments to evolve LSTMs and running the image captioning experiments.
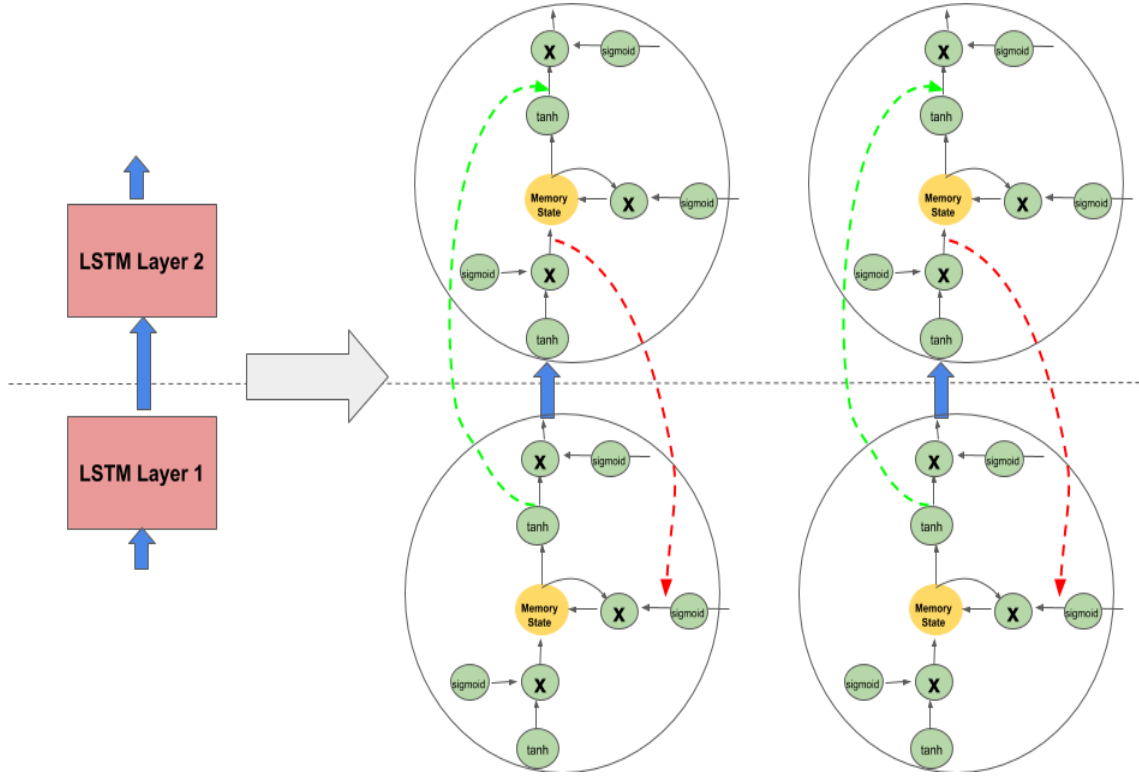
Figure 4.1: **Evolution search space**: On the left is a two layered stacked LSTM with each layer having two hidden neurons. On the right, the stacked LSTM layers (each of width=2) are shown along with their internal details. The internals of the two LSTM layers are exposed to evolution. Evolution, through mutation, can add feedforward skip connection (shown in green) and feedback skip connections (shown in red) between the two layers. The figure show one such example where a feedforward and feedback connection has been added. Evoution is constrained such that the skip connection can be added only between layers and not within a layer. Evolution modifies the connection between two nodes in a single column. That modification is then repeated across the layer. Constraining the evolution in this manner reduces its search space and makes it more tractable.

single column of stacked nodes on the right is evolved and the evolved solution is repeated across the layer. Notice, therefore that the two columns of nodes on the right side of Figure 4.1 are exactly the same.

Evolution, through mutation can add feedforward and feedback connections between the internals of the two stacked LSTM nodes. It can also disable connections added through evolution. In this manner, the underlying LSTM node structure is not modified but only extended through evolution. Evolution is also constrained to not add any skip connections within the internals of an LSTM node. Limiting the evolution search space in this manner keeps evolution tractable and also drives it towards discovering new paths that do not already exist in the LSTM.

There are 11 internal points within each node where evolution can either add an incoming (feedforward for higher layer node and feedback for lower layer node) or outgoing (feedback for higher layer node and feedforward for lower layer node) skip connection. Evolution can add multiple such skip connection between two stacked nodes. Thus the arachitectural search space is exponentially large.

The experimental setup and the language modeling task are described in the next section.

4.2 Experimental Setup

Word-level prediction experiments were conducted on the Penn Tree Bank (PTB) dataset (Marcus et al. (1993)), which consists of 929k training words, 73k validation words, and 82k test words. It has 10k words in its vocabulary.

A population of 50 LSTM networks was initialized uniformly in [-0.05, 0.05] (Figure 4.2). Each network consisted of two recurrent layers (vanilla LSTM or its variants) with 650 hidden nodes in each layer. The network was unrolled in time upto 35 steps. The hidden states were initialized to zero. The final hidden states of the current minibatch was used as the initial hidden state of the subsequent minibatch (successive minibatches sequentially traverse the training set). The size of each minibatch is 20. For fitness evaluation, each network was trained for 39 epochs with SGD. A learning rate decay of 0.8 is applied at the end of every six epochs and the dropout rate is 0.5. The gradients are clipped if there maximum norm (normalized by minibatch size) exceeds 5. Training single network takes
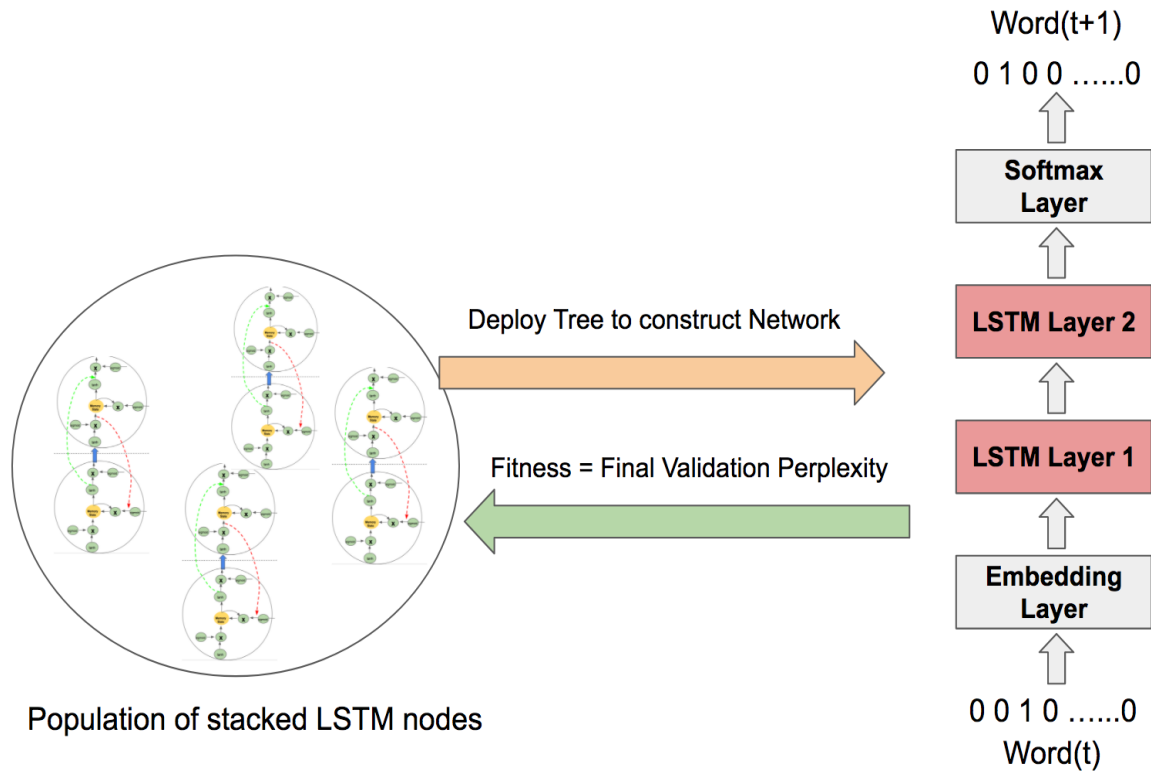
Figure 4.2: A population of stacked LSTM nodes are evolved using mutation operators (shown on the left). Each newly created offspring is duplicated to create a two layered stacked LSTM network as shown on the right. An input embedding layer and an output softmax layer are added to the network to construct the complete language model. The input to the model is the word in the current time step and the output is the expected word in the next time step. The final epoch validation perplexity of the model is used as its fitness during evolution.

about 200 minutes on a GeForce GTX 980 GPU card. [Add details about dropout] The validation perplexity of the network reached after the final epoch of training is treated as the fitness of stacked LSTM node that were used to construct this network. Binary tournament is used as a selection mechanism. This approach ensures that individuals with higher fitness get higher chance to reproduce offspring. Reproduction involves selecting a parent and mutating it. There are three kinds of mutations possible: 1) Mutation to add a feedforward connection between any two randomly selected points in the two stacked LSTM nodes , 2) Mutation to add a skip connection between any two randomly selected points in the two stacked LSTM nodes and 3) Mutation to remove a connection added through evolution. The three mutation probabilities are fixed to 0.5, 0.5, and 0.8 respectively throughout evolution.

4.3   Results

After 25 generations of neuroevolution, the best network improved the performance on PTB dataset by 5% (test-perplexity score is 80.4) as compared to the medium regularized LSTM (Zaremba et al. (2014)) (see Table 4.1). As shown in Figure4.3, this LSTM variant consists of a feedback skip connection between the memory cells of the two LSTM layers. When the size of the network is increased to the large setting (66 Million parameters), the evolved network still give improvements over the vanilla LSTM (77.5 vs. 78.4).

This result is encouraging because it suggests that even small modification in the layer connectivity can make a difference in the overall performance. However, the result is not very surprising because a similar recurrent network (designed by hand) has been shown to outperform vanilla LSTM (Chung et al. (2015)). (Chung et al. (2015)) additionally includes a sophisticated gating mechanism to control the flow of information from the higher layer LSTM layer to the lower LSTM layer.

The evolved layer connectivity adds an extra recurrent pathway in the network. The memory cell update equation for the recurrent nodes (See 2.8 for original) in the lower layer is modified as shown below:
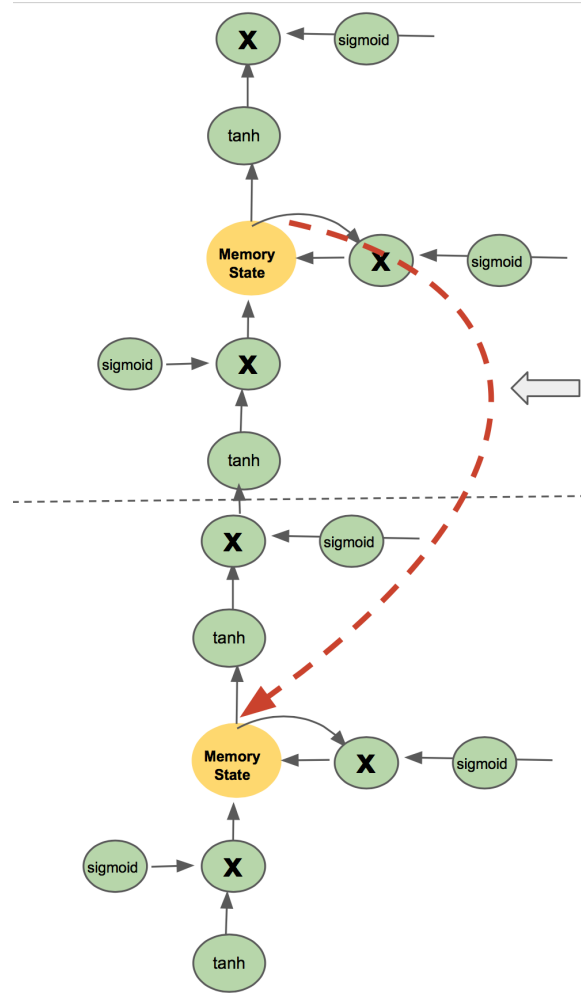
Figure 4.3: **Evolved LSTM layer connectivity:** the best performing LSTM variant after 25 generations of neuroevolution. It consists of a feedback skip connection between the two memory cells. Notice, while the vanilla LSTM has feedforward path from lower layer to the higher layer and recurrent paths within a layer, there is no pathway from the higher layer to the lower layer. Evolution discovered this pathway since it leads to improved network performance.

Table 4.1: **Single Model Perplexity on Test set of Penn Tree Bank:** evolved layers are the ones discovered through evolution. The experimental setting are kept exactly the same when comparing against results from Zaremba (2015) and only the network layer connectivity is swapped. Evolved layers outperform fixed layers in both medium and large setting.

Model	Parameters	Test Perplexity
Zaremba (2015) - Medium regularized LSTM	32M	82.7
Zaremba (2015) - Large regularized LSTM	66M	78.4
Evolved layer - Medium regularized LSTM	32M	80.4
Evolved layer - Large regularized LSTM	66M	77.5

$$c_t = c_{t-1} * f_t + \tilde{c}_t * i_t + d_{t-1}, \tag{4.1}$$

where:

d_{t-1} = Memory cell output of the higher layer from the previous time-step

The memory cell update equation for the lower layer remains unchanged with the feedback connection (same as Equation 2.8). The gradient flow equation for the higher layer now becomes:

$$\frac{\partial E}{\partial d_{t-1}} = \frac{\partial E}{\partial d_t}\frac{\partial d_t}{\partial d_{t-1}} + \frac{\partial E}{\partial c_t}\frac{\partial c_t}{\partial d_{t-1}} = \frac{\partial E}{\partial d_t}f_t + \frac{\partial E}{\partial c_t} \tag{4.2}$$

The gradient flow equation for the lower layer remains unchanged (Equation 2.15). In the vanilla LSTM case, backpropagation ensures that the lower layer adapts itself to the behavior of the higher layer. Howerver, this extra recurrent path also drives the higher layer to modify itself based on the lower layer.

When the evolved network was repeatedly trained (outside of evolution), an interesting phenomena was observed. The network loss during training sometimes explode leading to very large perplexity values. One reason for this could be the additive nature of the gradient for the higher layer (see the newly introduced component $\frac{\partial E}{\partial c_t}$ in Equation 4.2). Further study is required to prevent this exploding gradient problem. One approach could be to ensure that whenever evolution adds a new skip connection, it is always accompanied with some gating logic (Chung et al. (2015)). Other approaches could be lowering the threshold for gradient clipping, regularizing the memory cell values through a $L2$ penalty or using batch

normalization (Cooijmans et al. (2016)).

4.4 Conclusions

The initial results from evolving LSTM layer connectivity are promising, suggesting that a simple neuroevolution with only two possible mutations can automatically discover a new improved LSTM variant. It is quite possible that making neuroevolution richer by adding more forms of mutation and crossover can lead to even lower test perplexity scores. Additionally, such multi-layered LSTMs (as shown in Figure 4.3) can be used as components in evolving blueprints.

In the experiments described above, the LSTM node architecture was fixed and the layer connectivity was evolved. In the next chapter, the overall layer inter-connectivity is kept fixed and the recurrent node architecture is evolved. In this manner, we can understand the individual contributions of layer and node evolution in overall performance improvement.

Chapter 5

Evolving Recurrent Nodes

Recurrent nodes like LSTM and GRU are different from feedforward nodes in terms of the complexity of their hidden neuron activations. As described in the Section 2.1, LSTM node consists of gating logic that controls the information flow across time-steps. This additonal control alleviates the problem of vanishing gradients (Pascanu et al. (2013)) in recurrent networks and thus improve their performance. Another technique to improve performance is to increase their depth (Saxe et al. (2013)). Recurrent networks like LSTMs and GRUs are deep in the time context i.e. the recurrent connection allows information to be transformed through the node in each time step. However, the recurrent nodes are usually not themselves deep. This chapter [1] formulate techniques to build deep recurrent nodes and demonstrate how this technology could lead to improved performance.

The recurrent node is represented as a tree and GP is used to evolve its structure. Some modifications to standard GP are proposed to maintain diversity and improve evolutionary search. Repetition of evolved recurrent node yields a recurrent layer. The recurrent layers can then be stacked together to construct a recurrent network (an input embedding layer and output softmax layer can also be added). Training of the constructed recurrent network is costly i.e it requires ample GPU resources. A fitness prediction mechanism called Meta-LSTM is presented to reduce the training time of the networks without comprimising accuracy.

5.1 Tree Based Representation of Recurrent Node

As shown in Figure 5.1(a), a recurrent node can be represented as a tree structure, and GP can therefore be used to evolve it. However, standard GP may not be sufficiently powerful to do it. In particular, it does not maintain sufficient diversity in the population. Similar to the GP-NEAT approach by Tujillo et al. (2015), it can be augmented with ideas from NEAT speciation.

A recurrent node usually has two types of outputs. The first, denoted by

---

[1] this chapter is based on the paper Rawal and Miikkulainen (2018). The co-author Risto Miikkulainen provided feedback through discussions.
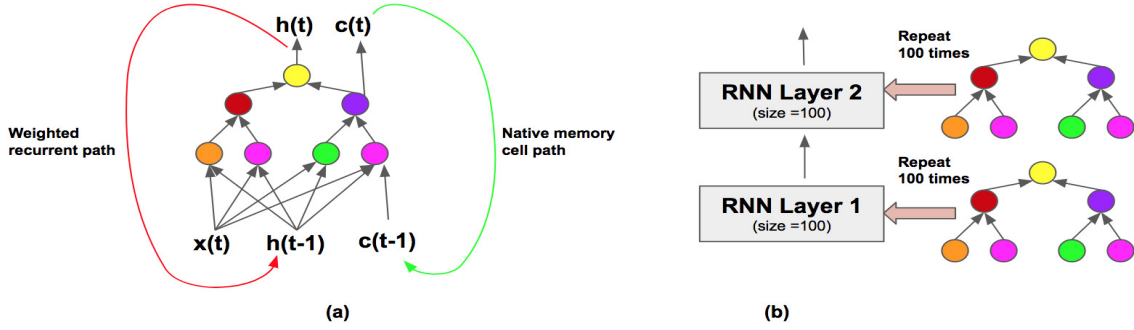
Figure 5.1: (a)**Tree-based representation of the recurrent node:** Tree outputs $h(t)$ and $c(t)$ are fed as inputs in the next time step. Recurrent paths are shown in green and red. The red connection is weighted while the green connection has a fixed weight of 1.0. (b) **Node to layer:** In recurrent networks, the tree node is repeated several times to create each layer in a multi-layered network. Different node colors depict various element activations. In this chapter, recurrent tree structures are evolved and deployed into the recurrent network through repetition.

symbol h in Figure5.1 (a), is the main recurrent output. The second, often denoted by c, is the native memory cell output. The h value is weighted and fed to three locations: (1) to the higher layer of the network at the same time step, (2) to other nodes in the network at the next time step, and (3) to the node itself at the next time step. Before propagation, h is combined with weighted activations from the previous layer, such as input word embeddings in language modeling, to generate eight node inputs (termed as base eight by Zoph and Le (2016)). In comparison, the standard LSTM node has four inputs (see Figure5.9(a)). The native memory cell output is fed back, without weighting, only to the node itself at the next time step. The connections within a recurrent cell are not trainable by backpropagation and they all carry a fixed weight of 1.0.

Thus, even without an explicit recurrent loop, the recurrent node can be represented as a tree. There are two type of elements in the tree: (1) linear activations with arity two (add, multiply), and (2) non-linear activations with arity one (tanh, sigmoid, relu).

One limitation of standard tree is that it can have only a single output: the root. This problem can be overcome by using a modified representation of a tree that consists of Modi outputs (Zhang and Zhang (2004)). In this approach, with

48

some probability p (termed modirate), non-root nodes can be connected to any of the possible outputs. A higher modi rate would lead to many sub-tree nodes connected to different outputs. A node is assigned modi (i.e. connected to memory cell outputs c or d) only if its sub-tree has a path from native memory cell inputs.

This representation allows searching for a wide range of recurrent node structures with GP.

5.2 GP-NEAT: Speciation

Genetic programming suffers from bloating i.e. unnecessary growth in tree size without corresponding increasing in fitness. One way to overcome, as proposed by Tujillo et al. (2015), is to divide the population into NEAT like species. The individuals in the population (trees) are topologically compared using a tree distance metric (described in Section 5.3) and similar trees are assigned to the same species. After the population is divided into species, the individual fitnesses are normalized such that larger species (with more individuals) are penalized. After normalization, the median fitness of the species is used as representative of the species fitness. Averaging these species median fitnesses yields population average fitness.

Each species get to spawn offspring proportional to their median fitness. If the species median fitness is higher than the population average fitness, the species get more spawns and vice-versa. Algorithm 1 describes these steps in detail. Each species carries out its own reproduction independently. The speciation procedure is executed at the end of each generation.

One key difference between NEAT and GP-NEAT is that in NEAT, the population is divided into species based on their shared historical markings. In contrast, in GP-NEAT, individuals are topologically compared before they are assigned species.

5.3 GP-NEAT: Crossover and Mutation

One-point crossover is the most common type of crossover in GP. However, since it does not take into account the tree structure, it can often be destructive. An alternative approach, called homologous crossover Francone et al. (1999), is

Algorithm 1 Speciation: Divide population into m species, normalize fitness and compute spawns

Require:

   N: population size, pop: population of trees

   $thresh$: maximum distance between two trees within the same species

   $SpeciesList$: List of species; empty at the start of evolution

   M: Number of species in the $SpeciesList$

   S_j: Individual species

   T_i: Individual tree representation of node. Includes a species pointer denoting its membership

   $T_i.fitness$: Fitness of the network constructed by repeating tree node T_i

   $S_j.representative$: Individual tree in the species with the best fitness.

1: **for** $i = 1$ to N **do**
2:   **for** $j = 1$ to M **do**
3:     **if** $\delta(S_j.representative, T_i) < thresh$ **then**
4:       S_j.add(T_i) {See equation 5.1 for δ.}
5:       $T_i.species = S_j$
6:     **end if**
7:   **end for**
8: **end for**
9: **for** $i = 1$ to N **do**
10:   **if** $T_i.species == None$ **then**
11:     Create new species S_{M+1} with T_i as its first member
12:   **end if**
13: **end for**

14: **for** $j = 1$ to M **do**
15:   **for** $k = 1$ to $length(S_j)$ **do**
16:     $T_k.Normfitness = T_k.fitness * length(S_j)$ {Normalize fitness to prevent large species}
17:   **end for**
18:   Compute species median normalized fitness: $S_j.medianfitness$
19: **end for**
20: $AvgFitness = \frac{1}{m} \sum_{i=1}^{m} S_j.medianfitness$ {Average normalized population fitness}

21: **for** $j = 1$ to M **do**
22:   **if** $S_j.medianfitness < AvgFitness$ **then**
23:     $S_j.spawns = length(S_j) * 1.1$ {More offsprings for better species}
24:   **else**
25:     $S_j.spawns = length(S_j) * 0.9$
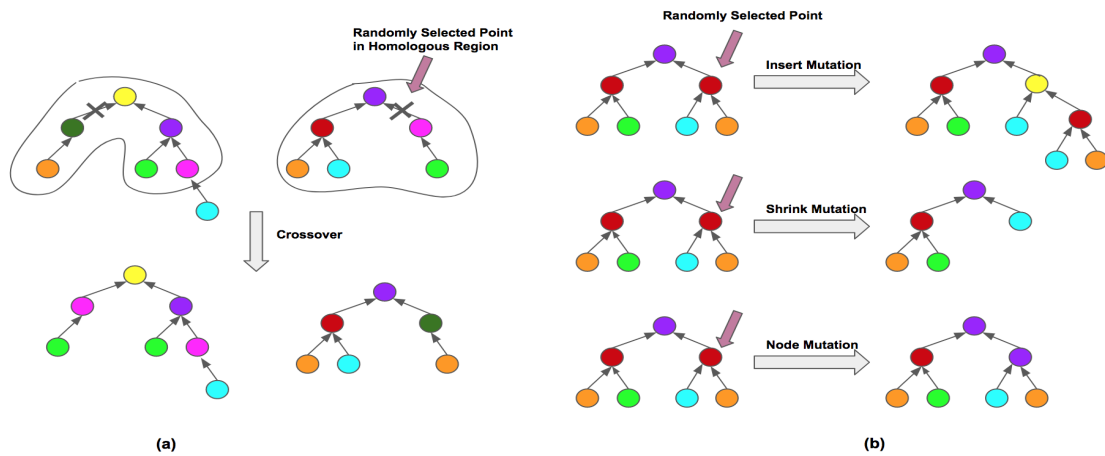26:   **end if**
27: **end for**

Figure 5.2: (a) **Homologous crossover in GP:** the two trees on the top look different but in-fact they are almost mirror images of each other. These two trees will therefore belong in the same species. The line drawn around the trees marks the homologous regions. A crossover point is randomly selected and one point crossover is performed. The bottom two networks are the resultant offspring. (b) **Mutation in GP:** Three type of mutation operators are shown - insert, shrink and node mutations. The crossover and mutation operators are applied to parent trees to create offspring trees.

designed to avoid this problem by crossing over the common regions in the tree. Similar tree structures in the population can be grouped into species, as is often done in NEAT Tujillo et al. (2015). Speciation achieves two objectives: (1) it makes homologous crossover effective, since individuals within species are similar, and (2) it helps keep the population diverse, since selection is carried out separately in each species. A tree distance metric of Tujillo et al. (2015) is used to determine how similar two trees T_i and T_j are:

$$\delta(T_i, T_j) = \beta \frac{N_{i,j} - 2n_{S_{i,j}}}{N_{i,j} - 2} + (1 - \beta) \frac{D_{i,j} - 2d_{S_{i,j}}}{D_{i,j} - 2}, \tag{5.1}$$

where:

n_{T_x} = number of nodes in GP tree T_x,

d_{T_x} = depth of GP tree T_x,

$S_{i,j}$ = shared tree between T_i and T_j,

$N_{i,j} = n_{T_i} + n_{T_j}$,

$D_{i,j} = d_{T_i} + d_{T_j}$,

$\beta \in [0, 1]$,

$\delta \in [0, 1]$.

On the right-hand side of Equation 5.1, the first term measures the difference with respect to size, while the second term measures the difference in depth. Thus, setting $\beta = 0.5$ gives an equal importance to size and depth. Two trees will have a distance of zero if their structure is the same (irrespective of the actual element types).

In most GP implementations, there is a concept of the left and the right branch. A key extension in this work is that the tree distance is computed by comparing trees after all possible tree rotations, i.e. swaps of the left and the right branch. Without such a comprehensive tree analysis, two trees that are mirror images of each other might end up into different species. This approach reduces the search space by not searching for redundant trees. It also ensures that crossover can be truly homologous Figure5.2 (a).

There are three kind of mutation operations in the experiments: (1) Mutation to randomly replace an element with an element of the same type, (2) Mutation to randomly insert a new branch at a random position in the tree; the subtree at the

---

Algorithm 2 Reproduction: Create Offspring from Parent

Require:

    $parents$: one or two parents

    $p_{crossover}$: probability of homologous crossover

    $p_{mutateNode}$: probability of replacing an element with another element of same arity

    $p_{mutateInsert}$: probability of randomly inserting a new branch at a random position in the tree

    $p_{mutateShrink}$: probability of randomly replacing a branch with one of its argument

  1: Sample p from a uniform distribution such that $p \in [0,1]$
  2: **if** $p < p_{crossover}$ **then**
  3:    $offsprings = HomologousCrossover(parents)$
  4: **else**
  5:    $offsprings = parents$ {Copy parents to offsprings}
  6: **end if**
  7: **if** $p < p_{muateNode}$ **then**
  8:    $offsprings = MutateNode(offsprings)$
  9: **end if**
10: **if** $p < p_{mutateShrink}$ **then**
11:    $offsprings = MutateShrink(offsprings)$
12: **end if**
13: **if** $p < p_{mutateInsert}$ **then**
14:    $offsprings = MutateInsert(offsprings)$
15: **end if**
16: **return** $offsprings$

---

chosen position is used as child node of the newly created subtree. (3) Mutation to shrink the tree by choosing a branch randomly and replacing it with one of the branch's arguments (also randomly chosen). Algorithm 2 presents the reproduction in detail.

5.4 Hall of Shame

The structural mutations in GP, i.e. insert and shrink, can lead to recycling of the same strcuture across multiple generations. In order to avoid such repetitions, a key contribution of this dissertation is to introduce an archive called Hall
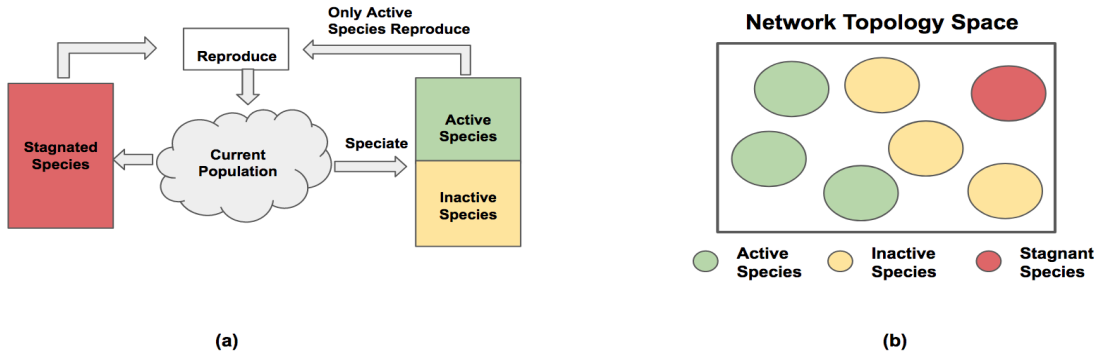
Figure 5.3: (a)**Hall of Shame:** an archive of stagnant species called Hall of Shame (shown in red) is built during evolution. This archive is looked up during reproduction, to make sure that newly formed offsprings do not belong to any of the stagnant species. At a time, only 10 species are actively evaluated (shown in green). This constraint ensures that active species get enough spawns to ensure a comprehensive search in its vicinity before it is added to the Hall of Shame. Offsprings that belong to new species are pushed into a inactive species list (shown in yellow) and are only moved to the active list whenever an active species moves to Hall of Shame. (b) **Regions in architectural search space:** the color of circles (green/yellow/red) indicates the category of species (active/inactive/stagnant) within that search space. Thus, hall of shame induces the search for novel architectures without an explicit objective.

of Shame (Figure5.3(a)). This archive consists of individuals representative of stagnated species, i.e. regions in the architecture space that have already been discovered by evolution but are no longer actively searched. During reproduction, new offsprings are repeatedly mutated until they result in an individual that does not belong to Hall of Shame. Mutations that lead to Hall of Shame are not discarded, but instead used as stepping stones to generate better individuals. Such memory based evolution is similar to novelty search. However, unlike novelty search (Lehman (2012)), there is no additional fitness objective, simply an archive.

As shown in Figure5.3(b), the population consists of three types of species - active, inactive and stagnant. While a species is active, constraints are added to its crossover and mutation to ensure that its offsprings lie within its space (shown as green circles in Figure5.3(b)). Once a species has been stagnated i.e. its best fitness does not improve over X generations (X is a evolution parameter), it is marked stagnated and it is added to the Hall of Shame. This stagnated species undergoes severe mutations to reproduce new architectures (shown as yellow circles in Figure5.3(b)). At a time, only 10 species are actively evaluated (shown in green). This constraint ensures that active species get enough spawns to ensure a comprehensive search in its vicinity before it is added to the Hall of Shame. Offspring that belong to new species are pushed into a inactive species list (shown in yellow) and are only moved to the active list whenever an active species moves to Hall of Shame.

5.5 Search Space: Node

GP evolution of recurrent nodes starts with a simple fully connected tree. During the course of evolution, the tree size increases due to insert mutations and decreases due to shrink mutations. The maximum possible height of the tree is fixed at 15. However, there is no restriction on the maximum width of the tree.

The search space for the nodes is more varied and several orders of magnitude larger than in previous approaches. More specifically, the main differences from the state-of-the-art Neural Architecture Search (NAS; Zoph and Le (2016)) are: (1) NAS searches for trees of fixed height 10 layers deep; GP searches for trees with height varying between six (the size of fully connected simple tree) and 15
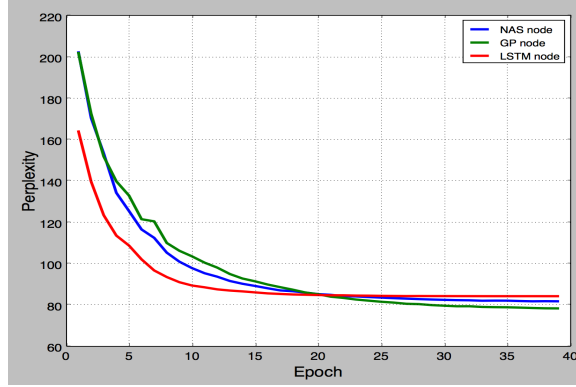
Figure 5.4: **Learning curve comparison of LSTM node, NAS node and GP nodes:** Y-axis is the validation perplexity (lower is better) and X-axis is the epoch number. Notice that LSTM node learns quicker than the other two initially but eventually settles at a larger perplexity value. This graph demonstrates that the strategy to determine network fitness using partial training (say based on epoch 10 validation perplexity) is faulty. A fitness predictor model like Meta-LSTM can overcome this problem.

(a constraint added to GP). (2) Unlike in NAS, different leaf elements can occur at varying depths in GP. (3) NAS adds several constraints to the tree structure. For example, a linear element in the tree is always followed by a non-linear element. GP prevents only consecutive non-linearities (they would cause loss of information since the connections within a cell are not weighted). (4) In NAS, inputs to the tree are used only once; in GP, the inputs can be used multiple times within a node. Thus, GP is allowed greater flexibility to search for recurrent node architectures.

5.6 Extra Recurrent Memory Cells

Most gated recurrent node architectures consist of a single native memory cell (denoted by output c in Figure5.1(a)). This memory cell is the main reason why LSTMs perform better than simple RNNs. One key innovation introduced in this dissertation is to allow multiple native memory cells within a node. The memory cell output is fed back as input in the next time step without any modification, i.e. this recurrent loop is essentially a skip connection. Adding another memory cell in the node therefore does not effect the number of trainable parameters: It only adds to the representational power of the node.

56

5.7   Meta-LSTM: Speeding up Evolution using Fitness Prediction

In both node and network architecture search, it takes about two hours to fully train a network until 40 epochs. With sufficient computing power it is possible to do it: for instance Zoph and Le (2016) used 800 GPUs for training multiple such solutions in parallel. However, if training time could be shortened, no matter what resources are available, those resources could be used better.

A common strategy for such situations is early stopping (Real et al. (2017)) i.e. selecting networks based on partial training. For example in case of recurrent networks, the training time would be cut down to one fourth if the best network could be picked based on the 10th epoch validation loss instead of 40th. Figure5.4 demonstrates that this is not a good strategy, however. Networks that train faster in the initial epochs often end up with a higher final loss.

Recent approaches to network performance prediction include Bayesian modeling (Klein et al. (2017)) and regression curve fitting Baker et al. (2017). The learning curves for which the above methods are deployed are much simpler as compared to the learning curves of structures discovered by evolution Figure 5.5.

To overcome costly evaluation and to speed up evolution, a Meta-LSTM framework for fitness prediction was developed in this dissertation. Meta-LSTM is a sequence-to-sequence model (Sutskever et al. (2014)) that consists of an encoder RNN and a decoder RNN (Figure 5.6(a)). Such sequence-to-sequence models have produced state-of-the-art results in challenging time-series prediction problems and generally outperform bayesian and regression approaches Suilin (2017).

In the Meta-LSTM, validation perplexity of the 8, 9 and 10 epochs is provided as sequential input to the encoder, and the decoder is trained to predict the validation loss at epoch 40 (Figure5.6). Experiments showed that only three encoder inputs are enough to predict the full learning curve. Training data for these models is generated by fully training sample networks (i.e. until 40 epochs). The loss is the mean absolute error percentage at epoch 40. This error measure is used instead of mean squared error because it is unaffected by the magnitude of perplexity (poor networks can have very large perplexity values that overwhelm MSE). The hyperparameter values of the Meta-LSTM were selected based on its performance in the validation dataset.

Due to the larger search space of evolution, the validation perplexity values
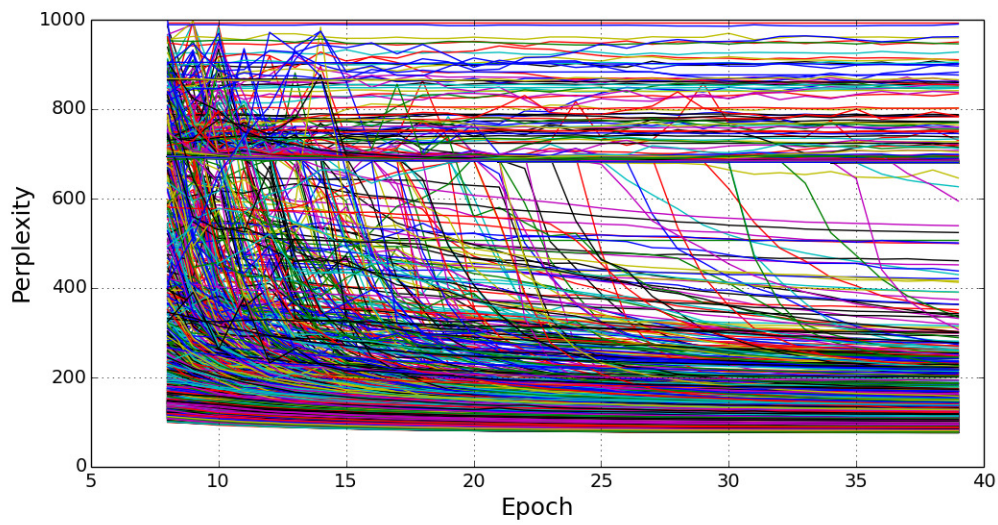
Figure 5.5: **Example learning curves:** learning curves for LSTM networks with different node architectures are plotted (Penn Tree Bank dataset). All the curves are plotted for the same hyper-parameter setting. Each curve corresponds to a network constructed from a unique recurrent node architecture. The curves are non-linear and therefore not easy to predict. This shows as fitness prediction to evolved networks is challenging and require sophisticated methods.
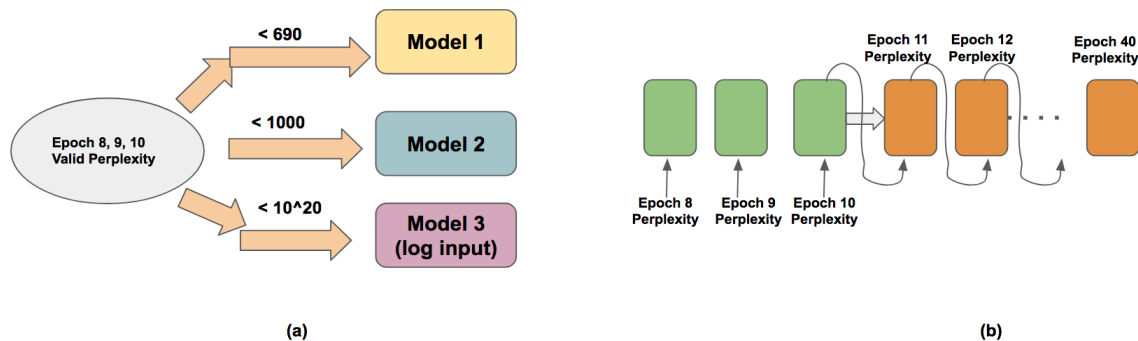
Figure 5.6: (a) **Three Meta-LSTM models:** three separate Meta-LSTM models are trained based on the range of validation perpelxity values. All three models are alike except for the fact that Model 3 takes logarithmic inputs. (b) **Meta-LSTM model:** this is a sequence-to-sequence (seq2seq) model that takes the validation perplexity of the 8th, 9th and 10th epochs as sequential input and predicts the validation perplexity at epoch 40. The green rectangles denote the encoder and the orange rectangles denote the decoder. Thus, the encoder length is 3 and the decoder length is 30. Decoder loss is computed at each step and averaged to generate a single final loss. Once trained and deployed, Meta-LSTM provides 4x speedup during network evaluation.

(i.e. training data of Meta-LSTM) have a high variance (from 70 to 10^{20} as shown in Figure 5.5). Additionally, it is more important to have accurate epoch 40 perplexity prediction for smaller values than for the larger values. As a result, three separate models of meta-lstm are trained depending upon the value of the input. As shown in Figure 5.6 (a), Model 1 is used for validation perplexity values (for epoch 8, 9 and 10) less than 690. Model 2 is used for values in the range 690-1000 and Model 3 is used for the values in the range 1000-10^{20}. Inputs to model 3 are fed after taking their logarithm, in order to properly scale them. Other than the difference in the input data range, all three models are trained in the exact same manner.

Note that Meta-LSTM is trained separately and only deployed for use during evolution. Thus, networks can be partially trained with a $4\times$ speedup, and assessed with near-equal accuracy as with full training.

The Meta-LSTM network consists of two layers, 40 nodes each. To generate training data for it, 1000 samples from a preliminary node evolution experiment was obtained, representing a sampling of designs that evolution discovers. Each

of these sample networks was trained for 40 epochs with the language modeling training set; the perplexity on the language modeling validation set was measured in the first 10 epochs, and at 40 epochs. The Meta-LSTM network was then trained to predict the perplexity at 40 epochs, given a sequence of perplexity during the first 10 epochs as input. A validation set of 500 further networks was used to decide when to stop training the Meta-LSTM, and its accuracy measured with another 500 networks.

In line with Meta-LSTM training, during evolution each candidate is trained for 10 epochs, and tested on the validation set at each epoch. The sequence of such validation perplexity values is fed into the trained meta-LSTM model to obtain its predicted perplexity at epoch 40; this prediction is then used as the tness for that candidate.

The procedure of evolution of recurrent node is described in Algorithm 3.

5.8   Experimental Setup and Results

The following sections describes the experimental setting for network training and evolution.

5.8.1   Network Training

During evolution, each network has two layers of 540 units each, and is unrolled for 35 time steps. The hidden states are initialized to zero; the final hidden states of the current minibatch are used as the initial hidden states of the subsequent minibatch. The dropout rate is 0.4 for feedforward connections and 0.15 for recurrent connections Gal and Gharamani (2015). The network weights have L2 penalty of 0.0001. The evolved networks are trained for 10 epochs with a learning rate of 1; after six epochs the learning rate is decreased by a factor of 0.9 after each epoch. The norm of the gradients (normalized by minibatch size) is clipped at 10. Training a network for 10 epochs takes about 30 minutes on an NVIDIA 1080 GPU. The following experiments were conducted on 40 such GPUs.

The Meta-LSTM consists of two layers, 40 nodes each. To generate training data for it, 1000 samples from a preliminary node-evolution experiment was obtained, representing a sampling of designs that evolution discovers. Each of these

Algorithm 3 Evolution of Recurrent Nodes

Require:

    G: Maximum number of generations to run

    $startgene$: Initial seed tree gene representing a recurrent node

    N: population size

    M: Number of species in the $SpeciesList$

    pop: new population of size N spawned by mutating startgene

    S_j: Individual species

    T_i: Individual tree representation of node. Includes a species pointer denoting its membership

    $T_i.fitness$: Fitness of the network constructed by repeating tree node T_i

1: **for** $g = 1$ to G **do**
2:   **for** $i = 1$ to N **do**
3:     Construct a recurrent network by repeating the tree gene T_i
4:     Train the recurrent network for 10 epochs on the training data using Backprop
5:     Compute Validation Perplexity for each of the 10 epoch
6:     $T_i.fitness$ = meta-lstm(validation perplexity of first 10 epochs as input sequence)
7:   **end for**

8:   Speciate population into m species:S_1, S_2, ..., S_M {See Algorithm 1}

9:   $offspring = []$, $count = 0$
10: **for** $j = 1$ to M **do**
11:    **for** $k = 1$ to $S_j.spawns$ **do**
12:     $parents$: Use binary tournament selection to pick parent trees within species S_j
13:     $offspring[count] = reproduce(parents)$ {Reproduce within species: See Algorithm 2}
14:     $count = count + 1$
15:    **end for**
16:   **end for**

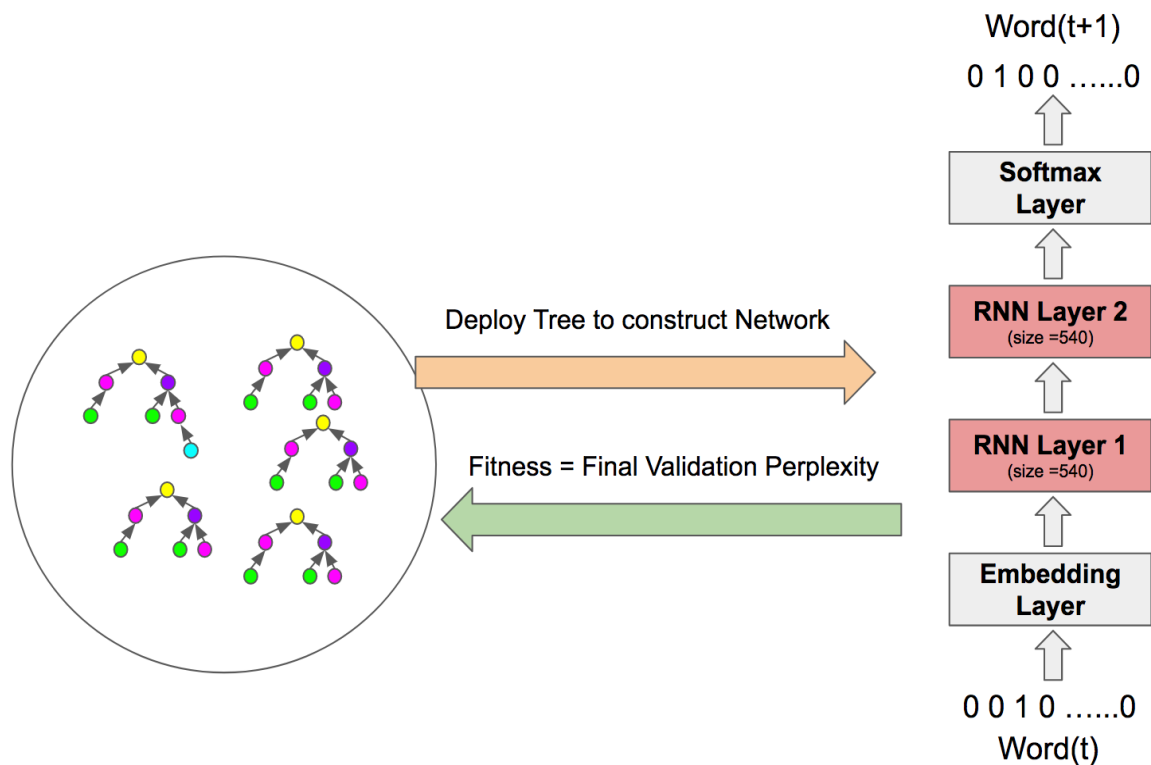17: $pop \leftarrow$ offspring

18: **end for**

Figure 5.7: **Evolution of recurrent nodes:** a population of recurrent tree nodes are evolved (shown on the left). Each newly created offspring is duplicated to create a two layered stacked RNN network as shown on the right. An input embedding layer and an output softmax layer are added to the network to construct the complete language model. The input to the model is the word in the current time step and the output is the expected word in the next time step. The final epoch validation perplexity of the model is used as its fitness during evolution. The best evolved network (with the best validation performance) is later selected and fine tuned with hyperparameter optimization.

sample networks was trained for 40 epochs with the language-modeling training set; the perplexity on the language modeling validation set was measured in the first 10 epochs, and at 40 epochs. The Meta-LSTM network was then trained to predict the perplexity at 40 epochs, given a sequence of perplexity during the first 10 epochs as input. A validation set of 500 further networks was used to decide when to stop training the Meta-LSTM, and its accuracy measured with another 500 networks.

In line with Meta-LSTM training, during evolution each candidate is trained for 10 epochs, and tested on the validation set at each epoch. The sequence of such validation perplexity values is fed into the trained meta-LSTM model to obtain its predicted perplexity at epoch 40; this prediction is then used as the fitness for that candidate. The individual with the best fitness after 30 generations is scaled to a larger network consisting of 740 nodes in each layer. This setting matches the 32 Million parameter configuration used by Zoph and Le (2016). A grid search over drop-out rates is carried out to fine-tune the model. Its performance after 180 epochs of training is reported as the final result (Table 5.1).

Loss Function

Perplexity is used as the loss function for training the language modeling network. As was described in equation 2.18, perplexity is exponentiated cross entropy. Cross entropy loss function is commonly used in training deep networks for classfication tasks (equation 2.19).

The loss function for the Meta-LSTM is mean absolute percentage error.

For mini-batch stochastic gradient descent, the loss is computer per word by averaging over both the batch size and the unroll length of the network (35 time steps in this case).

Regularization

To prevent over-fitting of the model to the training data, regularization is used in machine learning models. There are several prescribed regularization techniques for recurrent networks (as discussed in Chapter 2). The language-modeling network used three such techniques: $L2$ penalty on the weights, variational drop-

pout (Gal and Gharamani (2015)), and shared input, output embeddings (Press and Wolf (2016)).

During experiments, it was observed that dropping out inputs of the first recurrent layer made the network unstable leading to diverging results. Thus, the variational droppout (Gal and Gharamani (2015)) method was modified and the first recurrent layer inputs were not dropped.

Batch Normalization

Batch normalization is used in deep networks to scale the hidden neuron activations such that their distribution remains the same throughout the training process. It prevent covariate shift within hidden layers and improves gradient flow (Cooijmans et al. (2016)).

Adding a recurrent batch-norm layer to the language-modeling network gave poor results. One reason for this degradation could be due to regularization effect of batch-norm. Since the language model already has three types of regularization, adding batch-norm leads to spurious effects. Further understanding this problem could be an area of future study.

Optimization Algorithm

The language-modeling network is trained with mini-batch SGD. At the end of each batch, all the gradients are averaged within the batch and across timesteps, and then applied to the network weights. The weight update rule is given by:

$$w = w - \eta \frac{\partial Q(w)}{\partial w},\tag{5.2}$$

where:

$Q(w) = \frac{1}{n} \sum_{i=1}^{N} Q_i(w)$

The Meta-LSTM is trained with ADAM optimizer. ADAM is similar to SGD except that it keeps a running mean and variance of the gradients and scales the gradients accordingly.

5.8.2   Evolution Parameters

A population of trees of size 100 was evolved for 30 generations with a crossover rate of 0.6, insert and shrink mutation probability of 0.6 and 0.3, respectively, and modi rate (i.e. the probability that a newly added node is connected to memory cell output) of 0.3. A compatibility threshold of 0.3 was used for speciation; species is marked stagnated and added to the Hall of Shame if the best fitness among its candidates does not improve in four generations. Each node is allowed to have three outputs: one main recurrent output (h) and two native memory cell outputs (c and d). See Figure 5.7 for details.

The evolution parameters were not optimized since that requires running each parameter variation several times. Running each experiment is costly and therefore, tuning of these parameters is future work.

5.8.3   Meta-LSTM training

Two layered LSTM network with 80 hidden nerurons in each layer. After training, model 1 achieves an accuracy of $97 \pm 3.1\%$, model 2 achieves an accuracy of $85 \pm 10.5\%$ and model 3 achieves an accuracy of $60 \pm 35\%$. Adam optimizier Algorithm 3

5.8.4   Distribution Methodology

The following experiments were run on the 80 NVIDIA 1080Ti GPUs. Since these machines were a shared resource, custom work-distribution framework was developed to find an idle machine and utilize it for training. This framework included server-client setup: server would run the evolution code on the CPU to produce offspring networks which were then distributed on client GPUs for training.

5.8.5   Results

In the following sub-sections, the impact of new features like Hall of Shame and Meta-LSTM is evaluated. Then the performance of the evolved node is compared against other state-of-the-art methods on Penn Tree Bank.
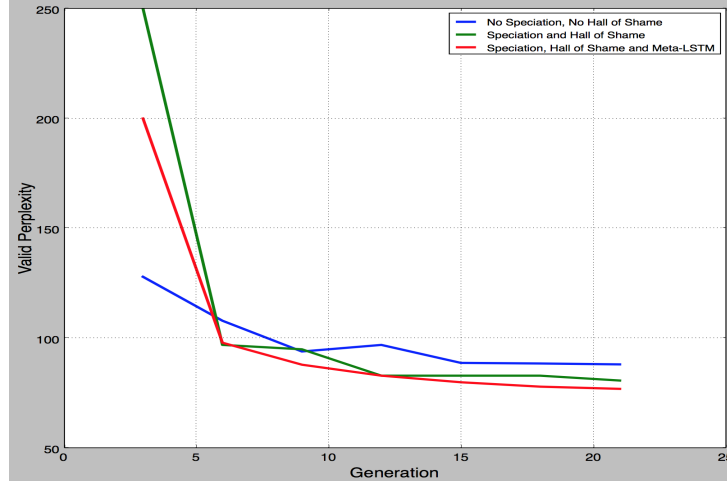
Figure 5.8: **Evaluating the effect of Speciation, Hall of Shame and Meta-LSTM:** three evolution experiments are conducted. In the first experiment, speciation, Hall of Shame and Meta-LSTM are disabled (blue curve). In the second, experiment, speciation and Hall of Shame are enabled (green curve). In the third experiment, speciation, Hall of Shame and Meta-Lstm are enabled (red curve). For all the three experiments, each network is partially trained till epoch 10. The epoch 10 validation perplexity is used as the fitness in the first two experiments. For the third experiment, Meta-LSTM is used to predict epoch 40 validation perplexity which is then used as the network fitness. After evolution has completed, the best individuals from each generation are picked and fully trained till epoch 40. For all three experiments, this graph plots the epoch 40 performance of the best network in a given generation. The plot shows that as evolution progresses, speciation and hall of shame select better individuals (green curve v/s. blue curve). Further enabling Meta-LSTM leads to selection of best networks (red curve). The results suggests that inclusion of Speciation, Hall of Shame, and Meta-LSTM in the evolution can improve the overall run-time and benchmark performance.

Table 5.1: Single Model Perplexity on Test set of Penn Tree Bank. Node evolved using GP outperforms the node discovered by NAS (Zoph(2016)) and Recurrent Highway Network (Zilly et al. (2016)) in various configurations.

Model	Parameters	Test Perplexity
Gal (2015) - Variational LSTM	66M	73.4
Zoph (2016)	20M	71.0
GP Node Evolution	20M	68.2
Zoph (2016)	32M	67.9
GP Node Evolution	32M	66.5
Zilly et al. (2016) , shared embeddings	24M	66.0
Zoph (2016), shared embeddings	25M	64.0
GP Evolution, shared embeddings	25M	63.0

Understanding the Impact of Speciation, Hall of Shame and Meta-LSTM

It is also important to understand the impact of using Speciation, Hall of Shame and Meta-LSTM in evolution. For this purpose, three evolution experiments were conducted: 1) Single Species with no speciation and no Hall of Shame, 2) With Speciation and Hall of Shame enabled 3) Speciation, Hall of Shame and Meta-LSTM enabled. In each experiment, the individual in the population was assigned a fitness equal to its 10th epoch validation perplexity. As evolution progressed, in each generation, the best individual was trained fully till epoch 40. The epoch 40 validation perplexity of best individuals in these three experiments has been plotted in Figure 5.8. This figure demonstrates incremental gains with each feature. For e.g. enabling Speciation and Hall of Shame results in better champions at the end of 21 generations. Subsequently, enabling Meta-LSTM gives the networks with the best epoch 40 validation perplexity. This comparison shows that individuals that are selected based upon meta LSTM prediction perform better than the ones selected using only partial training.

Evolved Nodes

The best evolved node is shown Figure 5.9. The evolved node reuses inputs as well as utilize the extra memory cell pathways. As shown in Table 5.1, the evolved node (called GP Node evolution in the table) achieves a test performance of 68.2 for 20 Million parameter configuration on Penn Tree Bank. This is 2.8 per-
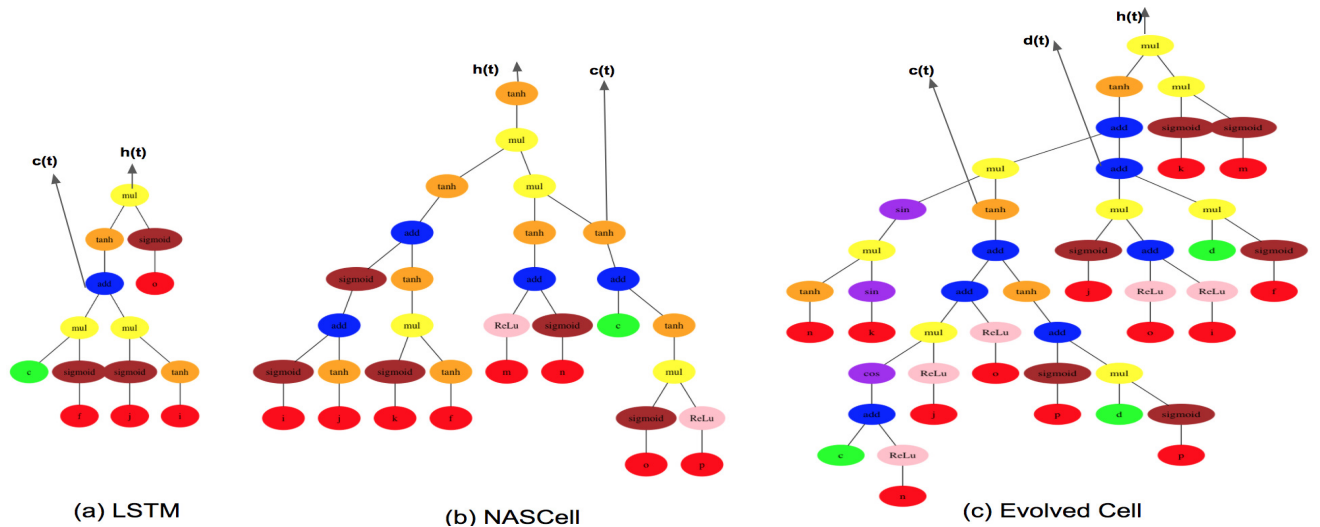
Figure 5.9: (a) **Comparing Evolved recurrent node with NASCell and LSTM:** the green input elements denote the native memory cell outputs from the previous time step (c, d). The red colored inputs are formed after combining the node output from the previous time step $h(t-1)$ and the new input from the current time step $x(t)$. In all three solutions, the memory cell paths include relatively few nonlinearities. The evolved cell utilizes the extra memory cell in different parts of the node. GP evolution also reuses inputs unlike the NAS and LSTM solution. With these discovered features, the evolved cell outperforms both LSTM and NASCell in the language-modeling task.

plexity points better than the test performance of the node discovered by NAS (Zoph(2016) in the table) in the same configuration. Evolved node also outperforms NAS in the 32 Million configuration (68.1 v/s. 66.5). Recent work has shown that sharing input and output embedding weight matrices of neural network language models improves performance (Press and Wolf (2016)). The experimental results obtained after including this method are marked as shared embeddings in Table 5.1.

One interesting aspect of the evolved node is the discovery of LSTM like output gating. Figure 5.9 (a) shows the LSTM node with control logic (o input followed by a $sigmoid$ activation) used for output gating. A similar output gating was discovered by evolution in Figure 5.9 (c) (See inputs k and m, each followed by a $sigmoid$). The result demonstrates that evolution can find solutions that not only outperform human designs, but also employ similar underlying principles.

5.9 Conclusions

Recent studies on metalearning methods such as neural architecture search and evolutionary optimization have shown that LSTM performance can be improved by complexifying it further (Zoph and Le (2016) Miikkulainen and et al. (2017)). This chapter develops a new method along these lines, recognizing that a large search space where significantly more complex node structures can be constructed could be beneficial. The method is based on a tree encoding of the node structure so that it can be efficiently searched using genetic programming. Indeed, the approach discovers significantly more complex structures than before, and they indeed perform significantly better: Performance in the standard language modeling benchmark, where the goal is to predict the next word in a large language corpus, is improved by 6 perplexity points over the standard LSTM (Zaremba et al. (2014)), and 0.9 perplexity points over the previous state of the art, i.e. reinforcement-learning based neural architecture search (Zoph and Le (2016)).

GP-NEAT is an effective tool to evolve such nodes. The search can be made more efficient, by keeping an archive of stagnated species called Hall of Shame. The network evaluation time can be reduced by using Meta-LSTM that predicts the learning curve of each network. Thus, 4x speed-up in evolution can be achieved.

More analysis of the evolved node is required to understand whether all nodes are necessary.

In the next chapter, the node evolved for language-modeling is evaluated in the music-modeling domain. This can give us insights into whether the discovered solution is transferrable to other domains.

Chapter 6

Recurrent Networks for Music

The problem of pattern detection in music involves predicting future musical notes, given the notes that have occurred in the past. Music has temporal patterns and previous work (in Lewandowski et al. ((2012); Lavrenko and Pickens (2003.); Eck and Schmidhuber ((2002)) demonstrated that these patterns are challenging to learn using traditional machine learning techniques. Learning these patterns often requires human intervention for feature generation. In this chapter [1], the best evolved node from the node evolution experiment in Chapter 5 is used for music note prediction. Musical data consists of melodies which requires short term memory and repetitions that require long term memory.

6.1   Music Language Model

The problem domain considered here is the polyphonic music prediction (See Figure 6.1). Music has several possible representations (e.g. .wav, .mp3 etc.). This work uses MIDI (www.midi.org) representation of music, although the models proposed here can applied to other representations of music as well. In MIDI files, the onset, duration, and pitch of every note in a piece of music is known. The pitches are encoded as numbers, ranging from 1 to 128. The durations are not symbolic, but instead are given as millisecond integers. The onset times also are not symbolic, but occur at millisecond integer locations. Since only piano datasets are used for this study, the maximum number of notes present at a given time is 88 i.e. the key range of the piano. Open source packages like pretty-midi can be used to convert the MID representation to the piano roll.

Monophonic music is such that, only one note is turned on at any given time. In polyphonic music, there is no such restriction. Any note may start or finish before any other note finishes. Polyphonic MIDI can therefore be viewed as a two-dimensional graph, with time along the x-axis, and MIDI note number (21 to 108) along the y-axis. At any point along the y-axis, notes turn on, remain on for

---

[1]this chapter is based on the paper Rawal and Miikkulainen (2018). The co-author Risto Miikkulainen provided feedback through discussions.
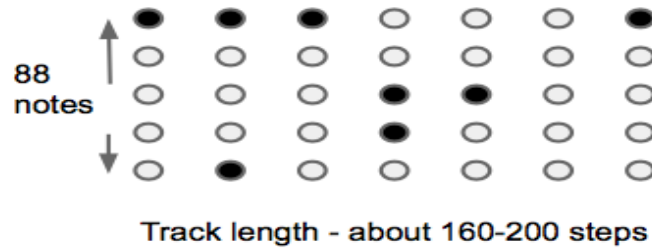
71

Track length - about 160-200 steps

Figure 6.1: **Polyphonic music data in piano roll format:** a 2D representation of a single musical track. Dark circles represent notes which are played at a given time instant. There are 88 notes in total corresponding to 88 piano keys. Thus musical sounds can be represented as discrete symbols.

a particular duration, and then turn back off again as shown in Figure 6.1. Black circles represent notes being on. White circles represent notes being off. With this data representation, the problem statement can be described concretely - given a set of musical note history, predict the note values (1/0) at the next time step.

6.1.1 Experimental Setup

As described in Section 2.7.2, Piano-midi.de dataset is used as a benchmark data. This data is divided into train (60%), test (20%) and validation (20%) sets. The music language model consists of a single recurrent layer of width 128 (See Figure 6.4). The input and output layers are 88 wide each. The network is trained for 50 epochs with Adam at a learning rate of 0.01. The network is trained by minimizing cross entropy between the output of the network and the ground truth. For evaluation, F1 score is computed on the test data. F1 score is the harmonic mean of precision and recall (higher is better). Since the network is smaller, regularization is not required.

Note, this setup is similar to the one presented in Ycart and Benetos (2017). The goal of this experiment is not to achieve state-of-the-art results but instead is to perform apples-to-apples comparison between LSTM node and evolved node (discovered for language) in a new domain i.e. music.

Table 6.1: F1 scores computed on Piano-Midi dataset. LSTM outperforms both the evolved node and NAS node for language, but not the node evolved specifically for music, demonstrating that the approach discovers solutions customized for the task.

Model	F1 score
LSTM	0.548
Zoph (2016)	0.48
GP Node Evolution	0.49
GP Node Evolution (Music)	0.599

6.1.2   Results

In this transfer experiment, three networks were constructed: first with LSTM nodes, second NAS node and the third with evolved node. All the three networks were trained under the same setting as described in the previous section. The F1 score of each of the three models is shown in Table 6.1. LSTM node outperforms both NAS and evolved nodes. This result is interesting because both NAS and evolved nodes significantly outperformed LSTM node in the language-modeling task. This result suggests that NAS and evolved nodes are custom solution for a specific domain, and do not necessarily transfer to other domains. In the next section, custom nodes will be evolved for the music domain.

6.2   Evolution of Recurrent Node for Music

The framework developed for evolving recurrent nodes for natural language can be transferred to other domains like music. The setup could be very similar i.e. at each generation a population of recurrent nodes represented as trees will be evaluated for their performance in the music domain (see Figure 6.2). The validation performance of the network constructed from the respective tree node will be used as the node fitness. The performance measure of the network in music domain is the F1 score, therefore, it is used as the network fitness value.

The evolution parameters are the same as that used for natural language modeling (see Section 5.8.2). Meta-lstm is not used for this evolution experiment because the run-time of each network is relatively small (< 600 seconds). The network training parameters are the same as the ones described in Section 6.1.1.
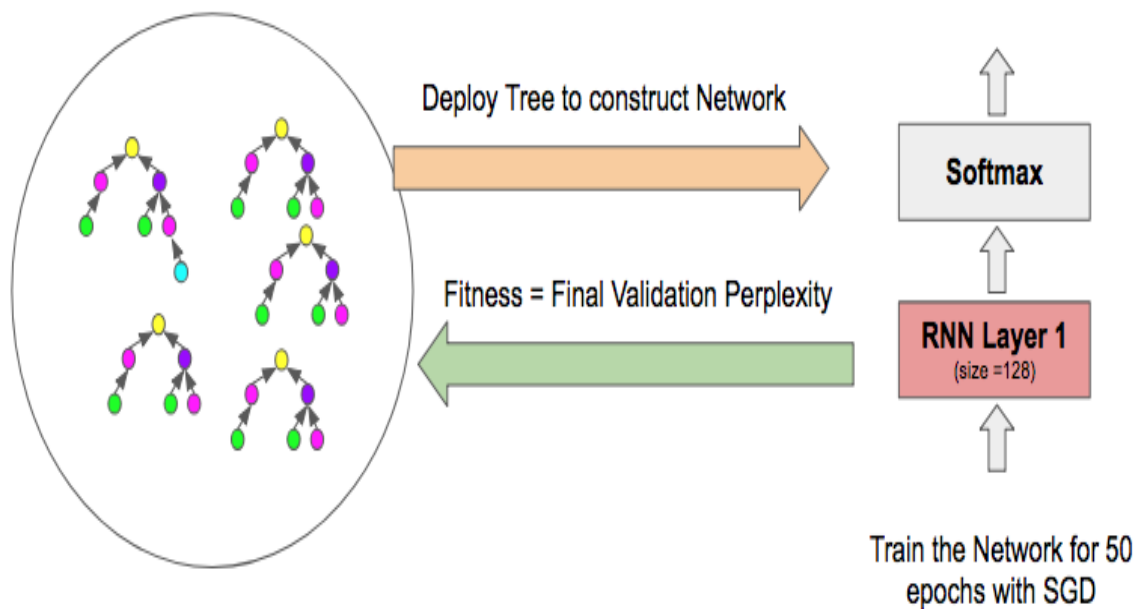
Figure 6.2: **Evolving recurrent node for music:** The figure on the right describes the architecture of the music language model. It is similar to the natural language-model as shown in Figure 2.3. There are two key differences - first, the input vector is not one-hot in music due to the presence of chords; second, there is no embedding layer that was present in the natural language model. The figure on the left shows a population of recurrent tree nodes that are evaluated on the music prediction task.
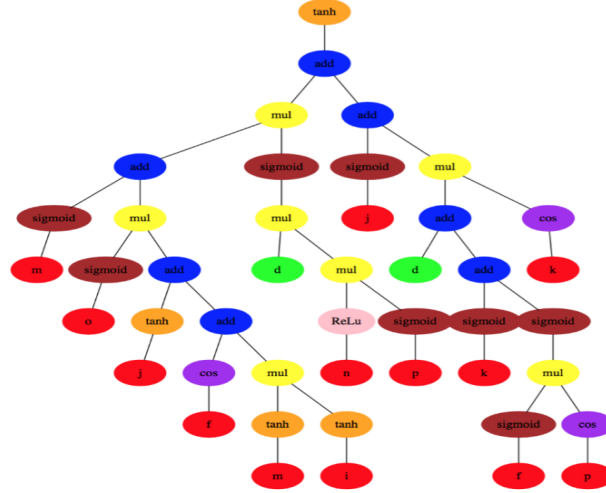
Figure 6.3: **Evolved Node for Music**. The node evolved to solve the music task is very different from the node for the natural language task. For example, this node only uses a single memory cell (green input d in the figure) unlike the language node that used both c and d. This results indicates that 'architecture does matter' and that custom evolved solution perform better than hand-designed ones.

Result from evolving custom node for music is shown in Table 6.1. The custom node (GP Evolution (Music)) achieves an improvement of five points in F1 score over LSTM (see Figure 6.3). Thus, evolution was able to discover custom structure for the music modeling domain as well—and it was different from structure in the language domain. The results demonstrate that it is not simply the added complexity in the nodes that matter, but that it is the right kind, i.e. complexity customized for each task.

6.3 AI Music Composer

To demonstrate the power of recurrent networks, an interactive music compostion application was developed. The trained music language model from the above experiment was used in a generative setting. The user can enter a few notes using their keyboard (See Figure 6.4 on left). The timing does not matter; the sequence is converted into a sequence of quarter notes automatically. These notes are fed as a sequence into the music language model on the right. The music model then takes over and generates new outputs. The network outputs are then fed into

its input in the next time-step. Thus the network works as a generative model.

This is a simple demonstration of how humans and AI can colloborate to create new music. Here is the weblink for the application: Click here
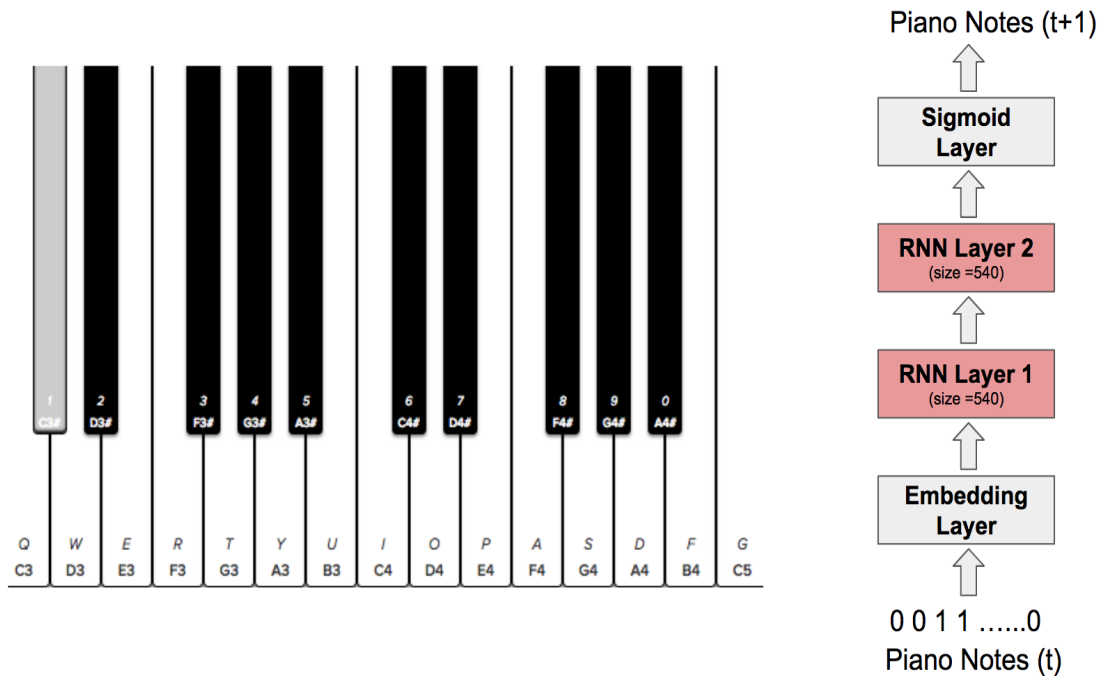
Figure 6.4: **Music Language Model used for Generation:** The figure on the right describes the architecture of the music language model. It is similar to the natural language-model as shown in Figure 2.3. There are two key differences - first, the input vector is not one-hot in music due to the presence of chords; second, the output layer is a sigmoid here unlike the softmax layer in the natural language case. The sigmoid layer allows generation of chords at the network output. The figure on the left is a snapshot of the music generation demo that was developed to exhibit the power of recurrent networks. It shows a set of white and black keys similar to the ones present on an actual piano. The user can enter a few notes using the keyboard. These notes are fed as a sequence into the music language model on the right. The music language then takes over and generates new outputs. The network outputs are then fed into its input in the next time-step. Thus the network works as a generative model.

Chapter 7

Future Work

There are five main experiments that can be performed in the future.

First, the Info-Max objective can be combined with the idea of artificial curiosity (Schmidhuber (2010)). While the idea of maximizing agent information is proposed to increase the depth of the agent's memory, it can indirectly lead to exploratory behaviors. Artificial curiosity is one such concept, where the agent is explicitly rewarded for exploring areas in the environment that provide more information.

Second, instead of training weights for each new offspring from scratch, the offspring can inherit the weights of one of the parent Real et al. (2017). This would allow partial training of the offspring and thus reduce evaluation time.

Third, the ideas from Chapter 4 and Chapter 5 can be combined i.e. both layer connectivity and recurrent node structure can be evolved simultaneously using bi-level methods (Miikkulainen and et al. (2017)).

Fourth, the ideas from recurrent highway networks (Zilly et al. (2016)) can be borrowed and employed here. In Zilly et al. (2016), the internal weights of the recurrent node are trained using backpropagation. This was made possible due to the presence of highway circuitry with every internal non-linearity. Including such highway logic in the recurrent tree nodes can allow the internal node weights to be trained.

Fifth, the results in language and music domain suggest that evolutionary search of neural architecture discovers regularities in each task and develops custom node structures that take advantage of them. Analyzing what those regularities are and how the structures encode them is an interesting direction of future work.

Chapter 8

Conclusions

In many areas of engineering design, the systems have become so complex that humans can no longer optimize them, and instead, automated methods are needed. This has been true in VLSI design for a long time, but it has also become compelling in software engineering: The idea in "programming by optimization" is that humans should design only the framework and the details should be left for automated methods such as optimization (Hoos (2012)). Recently similar limitations have started to emerge in deep learning. The neural network architectures have grown so complex that humans can no longer optimize them; hyperparameters and even entire architectures are now optimized automatically through gradient descent (Andrychowicz and et al. (2016)), Bayesian parameter optimization (Malkomes et al. (2015)), reinforcement learning (Zoph and Le (2016) Baker et al. (2016)), and evolutionary computation (Miikkulainen and et al. (2017) Real et al. (2017), et al. (2017)). Improvements from such automated methods are significant and suggest that the structure of the network matters.

This dissertation shows that the same approach can be used to improve architectures that have been used essentially unchanged for decades. The case in point is the Long Short-Term Memory (LSTM) network (Hochreiter and Schmidhuber (1997b)). It was originally proposed in 1992; with the vastly increased computational power, it has recently been shown a powerful approach for sequential tasks such as speech recognition, language understanding, language generation, and machine translation, in some cases improving performance 40% over traditional methods (Bahdanau et al. (2015b)). The basic LSTM structure has changed very little in this process, and thorough comparisons of variants concluded that there's little to be gained by modifying it further (Klaus et al. (2014) Jozefowicz et al. (2015)).

New techniques are proposed in this dissertation to automatically design LSTM networks for problems in RL domain and supervised learning domain. The next section lists key contributions.

8.1 Contributions

1. The first contribution of this work is an understanding of the impact of evolving networks consisting of RNNs, LSTMs in RL based memory problems. LSTMs outperform RNNs in tasks that require shallow memory. However, for deeper memory tasks, the performance of evolved LSTMs do not scale. The reward function in deeper memory problems is sparse, thus limiting the agent to sub-optimial behaviors.

2. To overcome this problem, a second contribution is the design of a new objective called Information-Maximization. This is an unsupervised objective and therefore does not depend on careful design of the environment. The training of the agent network is split into two phases. In the first phase, the agent captures maximal information from the environment and stores it in an efficient manner in the LSTM layer. Subsequently, during the reward maximization phase, the agent can utilize its stored information and escape sub-optimal behaviors. Information maximization within an agent is in-fact a general idea that can be combined with other RL methods like policy gradient or Q-learning as well.

3. For supervised learning problems like language modeling, only the architecture of the network is evolved and the weights are trained using BPTT. The third contribution of this research is to demonstrate how a simple evolutionary algorithm can be applied to discover novel connections between LSTM layers. One such evolved solution consisted of a feedback skip connection from the higher layer LSTM to the lower layer LSTM. This pathway doubled the network depth and facilitated gradient flow, thus improving the overall network performance in the language-modeling task.

4. Fourth contribution of this work is to deploy a combination of GP and NEAT (GP-NEAT) to evolve gated recurrent nodes. Modifications are introduced in GP-NEAT so that tree structural comparisons are more accurate, thus preventing redundant trees in the population.

5. Inspired by the encouraging results in Chapter 4, an extra recurrent connection was introduced within a node. Evolution is then allowed to construct

the appropriate glue logic surrounding the two recurrent connections.

6. Sixth contribution is the design of an archive called Hall of Shame. This archive drives evolution towards new architectural search space without explicitly maximizing a novelty objective.

7. A Meta-LSTM was designed that predicts the final performance of the network under evaluation. It uses a sequence-to-sequence model that takes in the learning curve values from the first few epochs and then predicts the remaining curve. This innovation significantly speeds up network evaluation, thus saving both money and time.

8. GP-NEAT used speciation, Hall of Shame and Meta-LSTM to quickly discover new gated recurrent node that outperforms both LSTM and NAS. The evolved node is shown in Figure 5.9.

9. Transfering the node evolved for the language modeling task to the music prediction task yields poorer results (when compared to LSTM). This provides new insights into the power of architecture search. Although, both evolved and NAS node significantly outperformed LSTMs in language-modeling, they were customized for that domain only. Evolving nodes for the music domain led to the discovery of a new recurrent node that outperformed LSTMs.

10. The final contribution of this work is to create an interactive AI music composer. Human user and the music-modeling network can collaborate to create new songs.

8.2 Conclusions

Supervised learning problems with large data and RL problems with sparse rewards, both face a common challenge: how to automatically design networks for these problems? Experiments presented in this work demonstrate the power of evolutionary techniques to solve such problems. While the human memory is potentially infinite, the same is not true for artificially created memory through recurrence in RL agents. Gathering and storing information in the recurrent networks in an efficient manner is therefore important. A new unsupervised object

called Info-max is developed in this research. When combined with neuroevolution, this objective can be used to simultaneously drive the agent exploration and complexification of its recurrent network in a non-parameteric manner.

Neuroevolution can also discover new pathways in large and deep recurrent networks used for supervised learning. It can be applied to construct new gated recurrent nodes that are deep networks within a network. Results from Chapter 6 indicate that novel architectures discovered for language-modeling are not easily transferrable to other domains like music. However, with new innovations presented in Chapter 5 that speed up architecture search, a custom solution can be quickly discovered in new domains. Finally, the results from this dissertation indicate that letting an AI system design another AI system is a promising area of research and can lead to discoveries that are beyond human imagination.

Bibliography

C. Adami. The use of information theory in evolutionary biology,. *Annals NY Acad. Sci.*, 1256:49–65, 2012.

M. Andrychowicz and et al. Learning to learn by gradient descent by gradient descent. In *NIPS*, 2016.

Marcin Andrychowicz, Misha Denil, Sergio Gómez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 3981–3989. Curran Associates, Inc., 2016. URL http://papers.nips.cc/paper/6461-learning-to-learn-by-gradient-descent-by-gradient-descent.pdf.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *In ICLR*, 2015a.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In *In ICLR*, 2015b.

B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. 2016. URL https://arxiv.org/pdf/1611.02167v2.pdf.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Practical neural network performance prediction for early stopping. *CoRR*, abs/1705.10823, 2017. URL http://arxiv.org/abs/1705.10823.

B. Bakker. Reinforcement learning with long short-term memory. In *Advances in Neural Information Processing Systems 14*, pages 1475–1482, 2002.

B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber. A robot that reinforcement-learns to identify and memorize important previous observations. In *In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2003*, pages 430–435, 2003.

J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber. Evolving memory cell structures for sequence learning. In *Proc. ICANN*, pages 755–764, 2009a.

J. Bayer, D. Wierstra, J. Togelius, and J. Schmidhuber. Evolving memory cell structures for sequence learning. In *In Artificial Neural Networks ICANN*, pages 755–764, 2009b.

A. J. Bell and T. J. Sejnowski. An information-maximisation approach to blind separation and blind deconvolution. *Neural Computation*, pages 1129–1159, 1995.

J.N. Bruck. Decades-long social memory in bottlenose dolphins. *Proceedings of the Royal Society B: Biological Sciences*, 280, (2013).

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Gated feedback recurrent neural networks. *arXiv preprint*, arxiv/1502.02367, 2015.

Tim Cooijmans, Nicolas Ballas, César Laurent, and Aaron C. Courville. Recurrent batch normalization. *CoRR*, abs/1603.09025, 2016.

F. Doshi. The infinite partially observable markov decision process. In *NIPS,2009*, 2009.

D. Eck and J. Schmidhuber. Finding temporal structure in music: Blues improvisation with lstm recurrent networks. In *Neural Networks for Signal Processing, IEEE workshop*, (2002).

C. Fernando et al. Pathnet: Evolution channels gradient descent in super neural networks. 2017. URL https://arxiv.org/abs/1701.08734.

F. Francone, M. Conrads, W. Banzhaf, and P. Nordin. Homologous crossover in genetic programming. In *GECCO*, 1999.

Y. Gal and Z. Gharamani. A theoretically grounded application of dropout in recurrent neural networks. 2015.

F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5:317–342, 1997.

A. Graves and N. Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In *In Proc. 31st ICML*, pages 1764–1772, (2014).

M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint*, arxiv/1507.06527, 2017.

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

S. Hochreiter and J. Schmidhuber. Long short term memory. *Neural Computation*, 1997b.

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In *A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press*, 2001.

Holger Hoos. Programming by optimization. *Communications of the ACM*, 55:70–80, 2012.

R. Jozefowicz, W. Zaremba, and I. Sutskever. An empirical exploration of recurrent network architectures. In *In Proceedings of the 32nd International Conference on Machine Learning*, pages 2342–2350, 2015.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016. URL http://arxiv.org/abs/1602.02410.

Nal Kalchbrenner, Ivo Danihelka, and Alex Graves. Grid long short-term memory. *CoRR*, abs/1507.01526, 2015. URL http://arxiv.org/abs/1507.01526.

M. Klapper-Rybicka, N. N. Schraudolph, and J. Schmidhuber. Unsupervised learning in lstm recurrent neural networks. In *ICANN*, pages 684–691. Springer-Verlag, 2001.

G. Klaus, R. Srivastava, J. Koutnik, R. Steunebrink, and J. Schmidhuber. Lstm: A search space odyssey. *arXiv preprint*, arxiv/1503.04069, 2014.

Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. In *ICLR*, 2017.

J. Koutnik, J. Schmidhuber, and F. Gomez. Evolving deep unsupervised convolutional networks for vision-based reinforcement learning. In *In Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, (GECCO 2014)*, pages 541–548, 2014.

V. Lavrenko and J. Pickens. Polyphonic music modeling with random fields. In *ACM MM*, pages 120–129, 2003.

J. Lehman. Evolution through the search for novelty. 2012.

J. Lehman and R. Miikkulainen. Overcoming deception in evolution of cognitive behaviors. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2014)*, Vancouver, BC, Canada, July 2014.

N. B. Lewandowski, Y. Bengio, and P. Vincent. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. In *International Conference on Machine Learning*, (2012).

G. Malkomes, C. Schaff, and R. Garnett. Bayesian optimization for automated model selection. In *NIPS*, 2015.

M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2), 1993.

G. Martin-Ordas, D. Berntsen, and Call J. Memory for distant past events in chimpanzees and orangutans. *Current Biology*, 23(15):1438–1441, (2013).

R. Miikkulainen and et al. Evolving deep neural networks. 2017. URL `https://arxiv.org/abs/1703.00548`.

Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. In Robert Kozma, Cesare Alippi, Yoonsuck Choe, and Francesco Carlo Morabito, editors, *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Amsterdam: Elsevier, 2018. URL `http://nn.cs.utexas.edu/?miikkulainen:chapter18`.

D. D. Monner and J. A. Reggia. A generalized lstm-like training algorithm for second-order recurrent neural networks. *Neural Networks*, 25:70–83, 2012.

C. Ollion, T. Pinville, and D. Stephane. With a little help from selection pressures: evolution of memory in robot controllers. In *In Artificial Life, volume 13*, pages 407–414, 2012.

Razvan. Pascanu, Tomas. Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *ICML*, Atlanta, GA, USA, June 2013.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint*, arxiv/1608.05859, 2016.

Aditya Rawal and Risto Miikkulainen. Evolving deep lstm-based memory networks using an information maximization objective. In *Genetic and Evolutionary Computation Conference (GECCO 2016), Colorado, USA*, 2016. URL `http://nn.cs.utexas.edu/?rawal:gecco2016`.

Aditya Rawal and Risto Miikkulainen. From nodes to networks: Evolving recurrent neural networks. *CoRR*, abs/1803.04439, 2018. URL `http://arxiv.org/abs/1803.04439`.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V.Le, and Alexey Kurakin. Large-scale evolution of image classifiers. 2017. URL `https://arxiv.org/abs/1703.01041`.

S. Risi, C. E. Hughes, and K. O. Stanley. Evolving plastic neural networks with novelty search. *Adaptive Behavior*, 18(6):470–491, 2010.

Andrew M. Saxe, James L. McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *CoRR*, abs/1312.6120, 2013. URL `http://arxiv.org/abs/1312.6120`.

J. Schmidhuber. Formal theory of creativity, fun, and intrinsic motivation (1990-2010). In *IEEE Transactions on Autonomous Mental Development*, volume 2(3), pages 230–247, 2010.

N. N. Schraudolph and T. J. Sejnowski. Unsupervised discrimination of clustered data via optimization of binary information gain. *Advances in Neural Information Processing Systems*, 5:499–506, 1993.

J. Schrum, I. Karpov, and R. Miikkulainen. Ut2: Human-like behavior via neuroevolution of combat behavior and replay of human traces. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2011)*, pages 329–336, 2011.

. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. In *IEEE Transactions on Signal Processing*, 1997.

Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. Technical report, 2016. URL `http://arxiv.org/abs/1603.05118`.

K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *Journal of Artificial Intelligence Research*, 21:63–100, 2004.

K. O. Stanley, B. Bryant, and R. Miikkulainen. Evolving adaptive neural networks with and without adaptive synapses. In *Proceedings of the 2003 Congress on Evolutionary Computation*, Piscataway, NJ, 2003. IEEE. URL `http://nn.cs.utexas.edu/?stanley:cec03`.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. URL `http://nn.cs.utexas.edu/?stanley:ec02`.

M. Suganuma, S. Shirakawa, and T. Nagao. A genetic programming approach to designing convolutional neural network architectures. In *GECCO*, 2017.

Arthur Suilin. Web traffic time series forecasting. `https://www.kaggle.com/c/web-traffic-time-series-forecasting/discussion/43795`, 2017. Accessed: 2017.

I. Sutskever, O. Vinyals, and Q. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

P. A. Szerlip, G. Morse, J. K. Pugh, and K. O. Stanley. Unsupervised feature learning through divergent discriminative feature accumulation. In *In Proc. of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

L. Tujillo, L. Munoz, E. Lopez, and S.Silva. neat genetic programming: Controlling bloat naturally. In *Information Sciences*, 2015.

O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. In *Proc. of CVPR*, pages 3156–3164, 2015.

H. Watts and K. E. Holekamp. Interspecific competition influences reproduction in spotted hyenas. *Journal of Zoology*, 276(4):402–410, (2008).

D. Wierstra, A. Foerster, J. Peters, and J. Schmidhuber. Recurrent policy gradients. *Logic Journal of IGPL*, 18(2):620–634, (2010).

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. 1992.

Adrien Ycart and Emmanouil Benetos. A study on lstm networks for polyphonic music sequence modelling. In *ISMIR*, 2017.

W. Zaremba, I. Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint*, arxiv/1409.2329, 2014.

Y. Zhang and M. Zhang. A multiple-output program tree structure in genetic programming. In *Complex Systems*, 2004.

Julian G. Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *CoRR*, abs/1607.03474, 2016. URL http://arxiv.org/abs/1607.03474.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. 2016. URL https://arxiv.org/pdf/1611.01578v1.pdf.