Convex Approximation techniques for Recurrent Neural Networks

Sapienza University of Rome

Sibghat Ullah

October 12, 2018

Dedication

I dedicate this dissertation to my parents for their unconditional love and support throughout my academic career…

Declaration

This dissertation is submitted for the Masters degree in Data Science at Sapienza University of Rome. I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments. This dissertation contains 53 pages including bibliography and equations.

Sibghat Ullah
October 12, 2018

Acknowledgements

I would first like to thank my thesis advisor Professor Aurelio Uncini. His supervision and guidance in my research meant a great deal to me. I would also like to thank Dr. Simone Scardapane for his valuable advice, continuous overseeing and time throughout the process of researching and writing this thesis.

Finally, I would express my gratitude to my parents for providing me with unfailing support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

Sibghat Ullah
October 12, 2018

Abstract

Deep Learning has been at the core of research in machine learning and artificial intelligence for the last decade or so. Many different deep learning architectures, models and algorithms have been proposed and investigated thoroughly. Deep learning architectures have been used on a variety of applications including computer vision, text mining and signal processing. As powerful as deep learning is, the learning process in deep networks still demands a lot of attention to make deep learning more robust, stable, heterogeneous and attractive. This is due to the nature of the problem and its vast applicability.

Recurrent networks are intuitively the most difficult networks to train and at the same time remain one of the widely used architectures in deep learning. More specifically, recurrent networks have been applied quite recently in different sequence learning problems including image, signal and natural language processing. They have reached greater accuracy than many of the state of the art machine learning models including other deep learning architectures i.e. feed forward neural networks.

As powerful as recurrent neural networks are, they are extremely difficult to train. Thus, the dissertation aims at optimizing the parameters and hyper-parameters of learning algorithm on recurrent network. More specifically, I propose to investigate Successive convex approximation (SCA) as the training algorithm for recurrent neural networks and Gaussian processes for the hyper-parameters optimization of Successive convex approximation.

I first describe the neural network as a computational model. Next, I describe the state of the art and the most common issues related to recurrent neural networks. I then move forward to describe Successive convex approximation and recurrent neural networks optimization. After that, the role of hyper-parameters and their optimization is discussed. Finally, the experimental results on a benchmark classification problem using Long short

term memory neural networks which are a special class of recurrent networks are discussed. Some important notes and future research line on Successive convex approximation is also presented in the final section of the dissertation.

Contents

Chapter 1

Introduction

In this chapter, I discuss the basics necessary to understand the dissertation. More specifically, I discuss the idea of neural networks, the feed forward and recurrent neural network architectures and the optimization in recurrent neural networks. The notations introduced in this chapter will be used throughout the dissertation.

1.1 Neural Networks

An artificial neural network denoted here as ANN is a computational model that is loosely inspired by the way biological neural networks in the human brain process information. Artificial neural networks have generated a lot of excitement in machine learning research and industry, thanks to many breakthrough results in speech recognition, computer vision and text processing. In this dissertation, I investigate the optimization in recurrent neural network which is one of the widely used neural network architectures.

The formal definition of artificial neural network is in agreement to the informal introduction above. More specifically, an artificial neural network can be defined as a computing system made up of a number of simple, highly interconnected processing elements which process information by their dynamic state response to the external inputs. The small interconnected computational elements are known as artificial neurons, nodes or processing units. These small units are the basis of a bigger network when they are connected with each other.

A single artificial neuron receives input from some other neurons, or from an external source and computes an output. Each input has an associated weight which is assigned on the basis of its relative importance to other inputs. The neuron applies a function to the weighted sum of its inputs. This function is often referred to as activation function. The intuition of neural network as a computational model is that the weights associated to each external or internal input are configurable which means that changing the weights result in changing the behavior of these artificial neurons. Thus, if one is able to optimize the weights in a neural network, the network will be able to perform in desired way.

1.2 Neural Network Architecture

Summarizing the above discussion we can conclude that a neural network is made of neurons which are interconnected with each other forming a network topology. The weights of these connections can be learned to optimize the behavior of the entire network. To add to that, the fundamental responsibility of any neuron is to compute its activation function. Thus intuitively, we can solve the most challenging tasks with neural network once we are able to learn the weights of the connections in the network. We refer to them as the parameters of the network.

Next, I will briefly introduce the important features of the architecture of a feed forward neural network aka multilayer perceptron.

A feed forward neural network aka a vanilla neural network can be divided in three logical layers. A layer is a vertical series or block of neurons. The three types are often referred to as the input layer, the hidden layer and the output layer. The input layer takes input data from some external source. No computation is performed in input layer, it just passes the information to the next layer. The next layer is called the hidden layer. A neural network can have multiple hidden layers. The intermediate processing or computation of neural network is done in hidden layer(s). The third type of layer is the output layer. Here, we receive the output of the neural network.

The network consists of connections, each connection transferring the output of a neuron i to the input of a neuron j. In this sense i is the predecessor of j and j is the successor of i, each connection is assigned a weight W_{ij}.

It is evident from the above discussion that each neuron computes an activation function. The activation function of a neuron defines the output

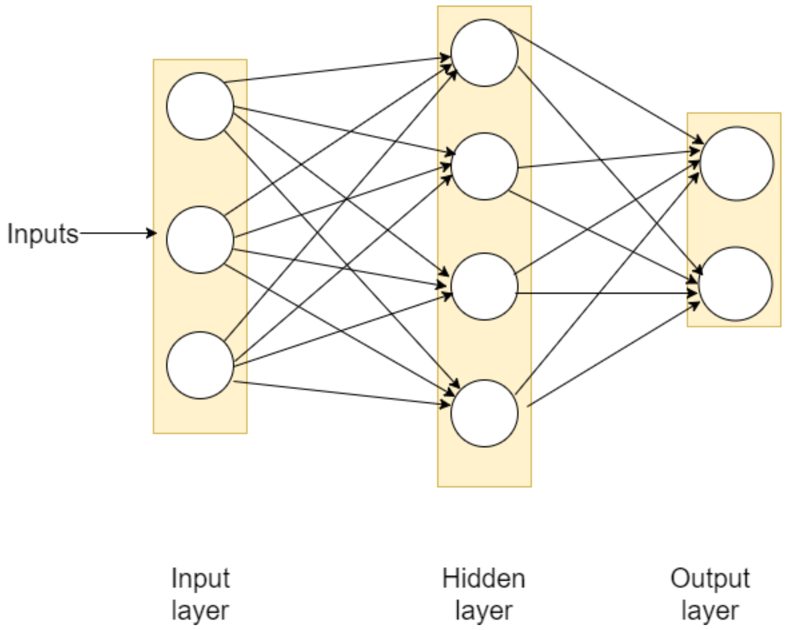


Figure 1.1: An example of a simple feed forward neural network aka multi-layer perceptron with three layers.

of that neuron given an input or set of inputs. The choice of activation function is wide. However, it should be intuitive that only the choice of a non linear activation function can result in making the neural network extremely powerful. One of the common choices of activation functions is a sigmoid function.

Once we understand the architecture of the network, we can understand the concept of learning rule. The learning rule is a method to modify the weights of the connections in the network. In other words, the learning rules determines the optimization of the so called parameters of the network. Figure 1.1 presents a simple feed forward neural network with one input layer having three neurons, a hidden layer with four neurons and finally an output layer with two neurons. In this case, the network is fully connected however flexible architectures are also possible.

1.3 Recurrent Neural Network

A recurrent neural network denoted as RNN throughout the dissertation is a class of artificial neural networks where the links between neurons form a recurrent connection i.e loop. This makes the recurrent networks more powerful as they can show dynamic behavior for a given input sequence. This sequence is typically based on time dependence [1] however other sequences based on natural phenomenon(s)and/or the properties of problem at hand can also be used. Recurrent network is a generic class of neural networks that can be specialized to do a variety of tasks. Some of the major classes of recurrent networks are Bi-directional,Gated recurrent units and Long short term memory networks.

As opposed to feed forward neural networks (as presented in figure 1.1), recurrent neural networks can use their internal state (aka memory) to process the time dependent inputs. Examples for such inputs are common in the field of image processing, signal processing, natural language processing, time series prediction, computational biology and computational chemistry.

1.4 Recurrent Neural Network Architecture

Recurrent neural network is a class of supervised machine learning models. The only peculiar thing about them is that they are made of neurons which are connected to each other forming a loop. These loops are sometimes referred to as sequences. Training a recurrent neural network involves minimizing the difference between the output which is estimated from training data and the actual output given in training data. This is done by optimizing the weights of these artificial neurons. A simple example of a recurrent network is given in figure 1.2.

A simple recurrent neural network has three layers known as the input, the hidden and the output layer respectively. The input layer has K input units [6]. More specifically, the input to this layer at each time step t is a vector denoted as $\mathbf{x}_t$. Similarly, I can denote all input vectors through different time steps as

$$\mathbf{x}_t = (x_1, x_2, ..., x_k) \tag{1.1}$$

In the case of fully connected recurrent network, all the neurons of the input layer are connected to all the neurons of the hidden layer. These

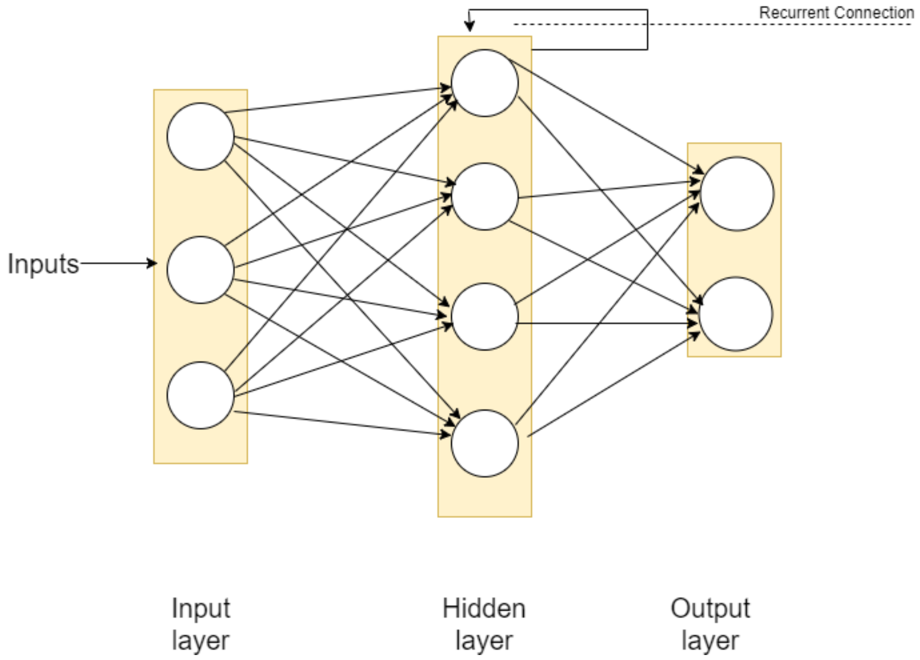


Figure 1.2: An example of a recurrent network with only one hidden layer. The network is presented in folded fashion with a generic recurrent connection providing the previous hidden state at each time step similar to the relationship in (1.4).

connections from input to the hidden layer are defined with a weight matrix denoted as $\mathbf{W}_{IH}$. The hidden layer has N neurons

$$\mathbf{h}_t = (h_1, h_2, ...h_N) \tag{1.2}$$

which are connected to each other through sequences of time. The hidden layer controls the so called memory of the network through its activation function. This can be defined as

$$\mathbf{h}_t = f_H(\mathbf{o}_t) \tag{1.3}$$

where as

$$\mathbf{o}_t = \mathbf{W}_{IH}\mathbf{x}_t + \mathbf{W}_{HH}\mathbf{h}_{t-1} + \mathbf{b}_h \tag{1.4}$$

And f_H is the activation function for the hidden layer, $\mathbf{W}_{HH}$ is the weight of all the connections from the previous hidden state and $\mathbf{b}_H$ is the bias vector of the hidden units. These neurons of the hidden layer are connected to the output layer with weighted connections denoted by $\mathbf{W}_{HO}$. The output layer has M units

$$\mathbf{y}_t = (y_1, y_2, ...y_M) \tag{1.5}$$

which can be computed by

$$\mathbf{y}_t = f_O(\mathbf{W}_{HO}\mathbf{h}_t + \mathbf{b}_o) \tag{1.6}$$

Where f_O is the activation function for the output layer and $\mathbf{b}_o$ is the bias vector of the output units.

1.5 Learning in Recurrent Neural Network

As it has already been discussed, recurrent network is a generic class of neural networks that can learn dynamic sequences in a data set. Typically, we use recurrent networks in the fields of image, signal and natural language processing however, they can be seamlessly extended to many other domains. Since all versions of recurrent networks have one peculiar thing in specific i.e. recurrent sequence, I use that to express the learning process in recurrent networks.

The first step in recurrent networks is the forward pass. The interpretation is that given a vector of input $\mathbf{x}_j$ at time j, one can compute the hidden state and the output of the neural network at that time step using the relationships presented in (1.3), (1.4) and (1.6). This simply means that having the input $\mathbf{x}_j$ at time j, one can compute $\mathbf{o}_j$ in (1.4) by specifying the weight matrices $\mathbf{W}_{IH}$, $\mathbf{W}_{HH}$ and the bias vector $\mathbf{b}_h$. Here $\mathbf{W}_{IH}$ denotes the weight matrix for the connections of the input layer while $\mathbf{W}_{HH}$ denotes the weights of the recurrent connections i.e. previous hidden state. To adjust the bias, we add the vector $\mathbf{b}_h$ which provides additional parameters of the network.

After the vector $\mathbf{o}_j$ is computed, next step is to compute the activation function provided in (1.3) which will give us the value of all the neurons in the first hidden layer. Similarly, all the hidden layers (if more than one) can be computed with the output of the previous hidden layer acting as the input and the previous hidden state $\mathbf{h}_{j-1}$ using (1.3) and (1.4). Once all the hidden layers at time j are computed, the network produces the current hidden state denoted as $\mathbf{h}_j$.

After having the hidden state of the network denoted as $\mathbf{h}_j$, we can use the relationship in (1.6) to compute the output of the network denoted as $\mathbf{y}_j$ by specifying the weight matrix $\mathbf{W}_{Ho}$ and the bias vector $\mathbf{b}_o$. Here, the weight matrix $\mathbf{W}_{Ho}$ contains the weights of the connections coming from the current hidden state.

Finally, the activation function of the output layer is computed which produces the value of the output neurons. At this point, we have computed the output of the network for one training example at time j which can be compared to the actual output provided in the data set to compute the error. This is done using the error or the so called loss function.

The choice of the loss function is wide however it's a common practice in neural network literature to use a convex loss function. For regression problems, normally squared error is used while for classification problems, cross entropy loss is used. These loss functions are used in combination with l_1 or l_2 regularization. For the sake of optimization, the expressions in (1.3), (1.4) and (1.6) can be substituted with an arbitrary recurrent neural network function denoted as $f(\mathbf{w}; j, \mathbf{x}_i)$ which takes two arguments. The first argument is the time step j while the second argument is the real valued vector of inputs $\mathbf{x}_i \in R^d$ i.e. ith training example at time j. This function gives us the output of the network at time j for ith training example and covers the expressions in (1.3), (1.4) and (1.6).

Now, assuming we have a training data set of N sequences where each such sequence can be denoted as $S = \{\mathbf{x}_i, y_i\}$. Also assuming to have fixed the number of time steps which is denoted by M and using an arbitrary convex loss function $l(.,.)$ combined with an arbitrary choice for regularization, the learning of the recurrent neural network is solving the following optimization problem.

$$min(U(\mathbf{w})) = \frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M} l(y_{ij}, f(\mathbf{w}; j, \mathbf{x}_i)) + \lambda r(w) \tag{1.7}$$

In (1.7), $\mathbf{w}$ shows all the parameters of the network as presented in (1.3), (1.4) and (1.6), y_{ij} is the actual output for the sequence i at time step j, $f(\mathbf{w}; j, \mathbf{x}_i)$ is the corresponding estimated output, and $l(.,.)$ is a convex, smooth loss function. I assume here that it meets all the assumptions provided by Successive convex approximation which are discussed in detail in chapter 3, and can be used for optimization. $r(\mathbf{w})$ is the regularization term which is weighted by a scalar λ. The optimization problem in (1.7) is non convex since we have a smooth convex loss function $l(.,.)$ with the non convex neural network function $f(\mathbf{w}; j, \mathbf{x}_i)$. The neural network function is non convex since we assume the activation functions of the layers to be non convex.

The simplest intuition of (1.7) is that we have to find the set of parameters $\mathbf{w}^*$ so as to minimize the value of the loss function.

Once we compute the aggregated loss of the neural network function for all $\mathbf{x}_i$ where $i \in 1, ..., N$ and for all time steps j where $j \in 1, ..., M$ as presented in (1.7), we can adjust the parameters so as to have a smaller value of the loss function at the next iteration. This is done by a concept known as Back-propagation. The main idea behind Back-propagation is to find the gradient of the total aggregated loss in (1.7) with respect to the parameters of the network. This is implemented by the chain rule in differential calculus. For the case of recurrent networks, this concept takes the name of Back-propagation through time and is presented in detail in [3].

Back-propagation through time first unrolls the recurrent network. After that, we are able to compute the aggregated error and run the normal Back-propagation algorithm same as a feed forward network. However, this process generally results in a much deeper architecture and the process is thus called Back-propagation through time since the network is deep due to the sequence length. This also implies that the network is much harder to train because of

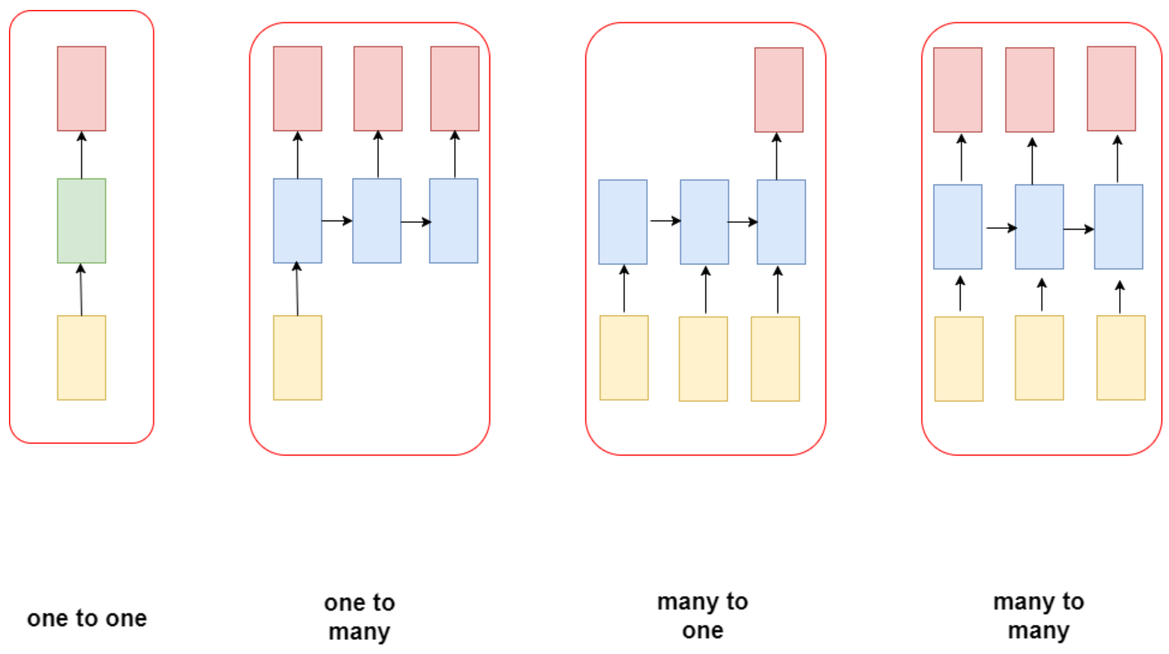


Figure 1.3: Different sequence types in recurrent networks

the presence of huge number of parameters and the number of hidden layers. The latter gives birth to the concept of vanishing and exploding gradient problem which is discussed in detail in the next chapter.

Recurrent networks can vary in that they can have different sequence types. The four normal categories of sequence types (number of inputs as compared to the number of outputs in the network) are given in figure 1.3.

In the next chapter, I will discuss state of the art in deep learning and recurrent networks including the long short term memory architecture and the training algorithms namely the Stochastic Gradient Descent, Adagrad and Adam.

Chapter 2

State of the Art

In this chapter, I discuss the state of the art in machine learning concerning the recurrent neural networks. More specifically, I discuss the most important problem in recurrent neural network and the possible solutions including the famous architecture known as Long short term memory neural networks. I then move forward and discuss the state of the art training algorithms in neural networks including Stochastic Gradient Descent, Adagrad and Adam.

2.1 Vanishing and Exploding Gradient Problem

The previous discussion gives us an introduction to the optimization in recurrent neural networks. As we can clearly see, the architecture of recurrent networks allows us to consider the dependency or persistence in the input. Examples of such inputs are encountered frequently in image processing, signal processing, time series analysis, and many other natural phenomenon which care for the organization of the input as sequences. As powerful as recurrent networks are, they still face some serious problems.

More specifically, if the input sequence is short, recurrent networks are able to consider the dependency structure. However, once the length of the sequence increases, the network starts to forget the previous dependency structure. This happens because of a very important concept known in literature as vanishing gradient problem [2]. The elaboration of vanishing gradient problem is that once the network becomes very deep, the gradient starts to vanish. This happens because the gradient received from the output layer starts to decrease exponentially as it reaches back to the input layer in the Back-propagation step [3]. In other words, the gradient starts to decline as an exponential function of the number of layers present in the network. This is the most important challenge that recurrent neural network faces and often results in poor local optimum.

As opposed to vanishing gradient problem, the recurrent neural network faces another challenge known as exploding gradient problem. This means that gradient starts to increase/explode as an exponential function of the number of layers present in the network. The simplest reason for vanishing and exploding gradient problem is the presence of chain rule multiplication of the gradients. The problems were discovered by Sepp Hochreiter [2] in 1991 and since then many solutions have been proposed including an extremely regulated structure such as Long short term memory neural networks [4].

An important thing to note is that the vanishing and exploding gradient problems do not theoretically mean that there is no optimization. Rather, in the case of vanishing gradient problem, although theoretically neural network is optimizing, the problem is much more numeric. This means that practically the gradient is approaching to zero and consequently, the optimization is not sufficient and can not result in a good local minimum. In the case of the exploding gradient problem, it means that gradient is exploding i.e. reaching to a very high number. As a result, the optimization is not stable.

A common practice to counter the exploding gradient problem is the clipping method [5]. Another interesting thing to note is that the vanishing gradient problem is much more difficult to diagnose and treat than exploding gradient problem although both problems are of the same nature. Combining all these points, we can safely conclude that a vanilla recurrent neural network can't effectively be optimized in the presence of vanishing or exploding gradient problem even with very sophisticated algorithms such as Back-propagation [3]. As such, it leads to a structure which can regulate the gradient explicitly giving birth to Long short term memory neural networks aka LSTM as presented in [4].

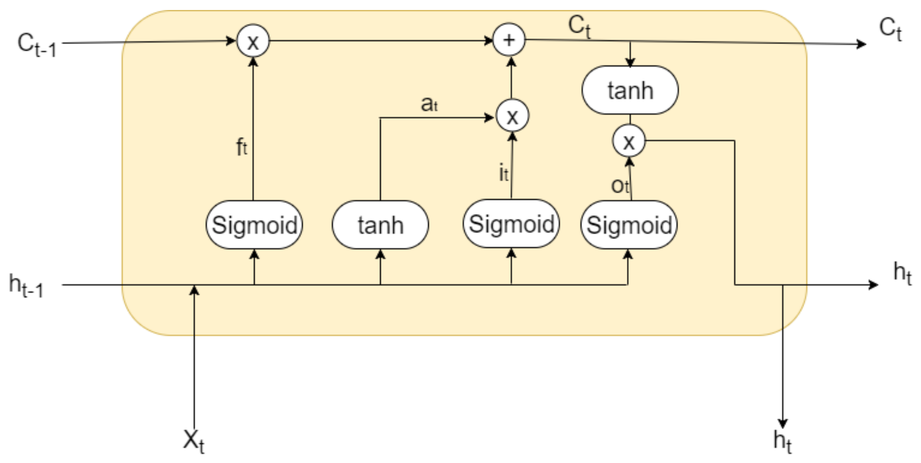


Figure 2.1: The internal structure of an LSTM unit as described in expressions (2.1-2.6)

2.2 Long Short Term Memory Neural Networks

Long short term memory neural network aka LSTM is a special class of recurrent neural networks. LSTMs are specially designed to solve the issue of long term dependencies i.e. vanishing and exploding gradient problem. As a result, LSTM is the most widely accepted neural network structure currently in the field of signal, image and natural language processing.

It is evident that all recurrent neural networks have the recurrent connection i.e. loop to themselves. The difference between vanilla recurrent neural network and LSTM thus is the internal structure of node. In the case of vanilla recurrent neural networks which have been discussed previously, each neuron has a very simple structure in that it takes an input and previous hidden state and computes the activation function on the weighted sum of both terms.

The internal structure of an LSTM unit differs in that it computes what part of the input sequence the network should forget and what new information it should add. It does this by the explicit small computing units called gates which are present in every neuron. Three common choices for these gates are the input, the forget and the output gate although many variations on the number and types of gates exist. The internal structure of a modern LSTM unit or cell is presented in figure 2.1 which is slightly different to the one in [4].

I now move forward to describe the internal structure of an LSTM cell that requires the introduction of some concepts. More specifically, the above discussed long term memory takes the name of cell state or unit state in LSTM. The recurrent connection allows the information from previous time steps to be stored in the LSTM cell. A modern LSTM cell has four gates called the input activation gate, input gate, forget gate and the output gate. The cell state is modified by the forget gate, input activation gate and input gate. Forget gate decides which part of the previous cell state needs to be deleted. The input activation gate and the input gate combine together to select some part of the input to be added to the cell state. Finally, the hidden state is computed using the cell state and the output gate. This hidden state is used to compute what to forget, input and output in the cell at the next time step. The notations and expressions to cover the entire discussion so far are presented below.

$$\mathbf{a}_t = tanh(\mathbf{W}_{IA}\mathbf{x}_t + \mathbf{W}_{HA}\mathbf{h}_{t-1} + \mathbf{b}_a) \tag{2.1}$$

$$\mathbf{i}_t = \sigma(\mathbf{W}_{II}\mathbf{x}_t + \mathbf{W}_{HI}\mathbf{h}_{t-1} + \mathbf{b}_i) \tag{2.2}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{IF}\mathbf{x}_t + \mathbf{W}_{HF}\mathbf{h}_{t-1} + \mathbf{b}_f) \tag{2.3}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{IO}\mathbf{x}_t + \mathbf{W}_{HO}\mathbf{h}_{t-1} + \mathbf{b}_o) \tag{2.4}$$

In (2.1-2.4), $\mathbf{x}_t$ is the input vector at current time step while $\mathbf{h}_{t-1}$ is the hidden state at previous time step. $\mathbf{W}_{IA}$, $\mathbf{W}_{II}$, $\mathbf{W}_{IF}$, $\mathbf{W}_{IO}$ are the weights of the connections coming from the input. Similarly, $\mathbf{W}_{HA}$, $\mathbf{W}_{HI}$, $\mathbf{W}_{HF}$, $\mathbf{W}_{HO}$ are the weights of the recurrent connection i.e. connection from the previous hidden state. Additionally, a bias vector is added for all four gates to adjust the bias. Finally, the activation function is computed for all four gates. Normal activation function for input activation gate presented in (2.1) is tanh. For the input, forget and output gates presented respectively in (2.2), (2.3) and (2.4) sigmoid is used as the activation function.

Finally, we can conclude that the input activation, input ,forget and output gates all take similar inputs and compute their activation functions accordingly.

Once we have computed the value of all four gates, we can compute the cell state aka cell memory, internal memory or long term memory and the hidden state aka the output of the unit. The equations for both are presented below.

$$\mathbf{C}_t = \mathbf{a}_t * \mathbf{i}_t + \mathbf{f}_t * \mathbf{C}_{t-1} \tag{2.5}$$

$$\mathbf{h}_t = tanh(\mathbf{C}_t) * \mathbf{o}_t \tag{2.6}$$

In (2.5) and (2.6) $\mathbf{C}_t$ denotes the cell state, cell memory, internal memory or the long term memory. As it is obvious in (2.5), $\mathbf{C}_t$ depends on input activation, input and forget gates which depend on the current input $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$. The current hidden state (2.6) denoted as $\mathbf{h}_t$ depends on the current cell state $\mathbf{C}_t$ and the output gate $\mathbf{o}_t$.

The relationships in (2.1-2.6) describe the computations necessary for a single LSTM cell in the forward pass. The input gate along side the input activation gate decide which part of the input i.e. new information should be added to the cell memory. The forget gate decides which previous information needs to be forgotten from the cell memory. The output gate decides what should be the output based on what information was added and forgotten. Together, these gates explicitly decide what part of the sequence is important to remember. As a result, the network is able to remember the short term dependencies in a very long sequence thus giving rise to the term long short term memory.

Thus, we can easily understand as to why these networks are so good at remembering short term memory for a long period of time which is a common happening in image processing, text processing, language models and protein protein interaction . The most important thing is that within each neuron, we have a special mechanism of regulating the information flow.

One important thing to note is that, like any other neural network, the weights of the connections to these gates and the biases are the parameters of the network and have to be learned from the external input. This also means that we roughly have three to four times more parameters to learn as compared to a vanilla recurrent network. This can be problematic since many optimization algorithms like Successive convex approximation (discussed in next chapter) save the parameters of the network in a matrix which becomes difficult computationally for a large number of parameters. Further, inverting a matrix also becomes problematic as the number of elements grow.

In the next section, I describe the state of the art in training neural networks. More specifically, I discuss the first order approaches like Stochastic Gradient Descent, Adagrad [12] and Adam [7] for neural networks training. I also briefly discuss some other approaches for neural networks training.

2.3 Gradient Descent based Optimization Algorithms

In this section, I present a brief overview of the most famous algorithms based on Gradient descent which are used in deep learning. I present the idea of Gradient descent and then move to discuss Stochastic gradient descent. Next, I discuss algorithms based on Momentum including Adagrad [12] and Adam [7] in much more detail.

Gradient descent is a way to minimize the value of an arbitrary objective function $f(\boldsymbol{\theta})$ with $\boldsymbol{\theta} \in R^z$ acting as the vector set of parameters. More specifically, we have to find the values of parameters $\boldsymbol{\theta}^*$ such that f takes the (local) minimum value. Gradient descent does this by adjusting the parameters in the opposite direction of the gradient of the objective function which can be denoted as $\delta_\theta f(\boldsymbol{\theta})$. The learning rate η determines the scale of adjustment aka the learning rate or update rate. This process is employed untill we reach a (local) minimum value.

There are three common variations of Gradient descent known as Batch gradient descent, Stochastic gradient descent and Mini-batch gradient descent. These three types vary in the amount of data used for a single update of the parameters.

Batch gradient descent aka vanilla Gradient descent computes the gradient of the objective function with respect to the set of parameters $\boldsymbol{\theta}$ for the entire data set. This is expressed in (2.7).

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta * \delta_\theta f(\boldsymbol{\theta}) \tag{2.7}$$

It is evident that Batch gradient descent is not efficient for large scale deep learning since the set of parameters $\boldsymbol{\theta}$ becomes huge and is thus difficult to manage. Further, computing gradient becomes expensive and unworthy since it involves redundant computation. For problems related to low dimensions, it is used in practice but we are mostly concerned with high dimensional set up which is a common happening in large scale machine learning. Now, I will briefly discuss Stochastic gradient descent for deep learning as presented in [10] although the original work for Stochastic estimation was done in 1952 and is presented in [9].

2.4 Stochastic Gradient Descent

Stochastic gradient descent is used to avoid the computational challenges of Batch gradient descent. Stochastic gradient descent does this by updating the parameter $\boldsymbol{\theta}$ of the objective function f at each iteration by using only a single input instance which is denoted here as $(\mathbf{x}_i, y_i)$. This is expressed in (2.8).

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta * \delta_\theta f(\boldsymbol{\theta}; \mathbf{x}_i; y_i) \tag{2.8}$$

It is understandable from the above expression that Stochastic gradient descent is much faster than the Batch gradient descent. This is since Stochastic gradient descent only uses a single input example for each update. This becomes computationally feasible and avoids redundant computation for calculating the gradients for similar examples which is a common happening in Batch gradient descent. An additional property of Stochastic gradient descent is that it can be used in online learning.

Although Stochastic gradient descent is much cheaper to compute, it has its draw backs. Stochastic gradient descent oscillates while converging to the (local) minimum. This is due to the intrinsic nature of the algorithm since it only uses one input instance at every iteration. Thus, it is also affected by the noise presented in the data set and is essentially a noisy version of the Batch gradient descent. This oscillation becomes a serious problem in Stochastic gradient descent convergence. Recently, some solutions have been proposed to avoid this issue including the Mini-batch gradient descent based algorithms which are discussed in the next section.

2.5 Mini-Batch Gradient Descent

Mini-batch gradient descent is used to counter the problems faced by Batch gradient descent and Stochastic gradient descent. It does this by combining the both concepts. More specifically, it takes the mini batch of size L from the entire data set at each iteration. It then uses this mini batch to update the parameters at that iteration. This is formally expressed in (2.9).

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta * \delta_\theta f(\boldsymbol{\theta}; \mathbf{x}_{(i,i+L)}; y_{(i,i+L)}) \tag{2.9}$$

Using a mini-batch in this fashion has many advantages. The most important advantage is that it is computationally cheaper compared to the entire data set since the size of mini batch is very small. It also removes the disadvantage of Stochastic gradient descent since it is not affected by the noise the way Stochastic gradient descent is affected. Thus, Mini-batch gradient descent has become the most important Gradient descent based approach in deep learning. All of the state of the art neural networks training algorithms including AdaGrad [12] and Adam [7] are based on Mini-batch gradient descent.

Mini-batch gradient descent however has some challenges. The most important challenge is to choose the learning rate η in a very smart way. This challenge is addressed by Adagrad [12] which is discussed in the next section.

2.6 Adagrad

Adagrad [12] is one of the most efficient algorithms to train neural networks. It is inspired by the concept of Momentum [22]. As we have already seen, Stochastic gradient descent faces the challenge of oscillations around the local optimum which can be a serious problem in convergence. Momentum [22] is a concept known to move Stochastic gradient descent in relevant direction only, thus significantly reducing the oscillations. It accomplishes this by adding the information of past updates at each iteration. Intuitively, by having the information of past updates, Stochastic gradient descent is not affected by the current noise in the input and thus moves only in the relevant direction of local minimum. As a result, convergence is faster. An image representation of Stochastic gradient descent with and with out Momentum is presented in figure 2.2.

Formally, we can express this in (2.10)

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \{\beta \mathbf{v}_{t-1} + \eta \delta_\theta f(\boldsymbol{\theta})\} \tag{2.10}$$

In the above equation $\mathbf{v}_{t-1}$ is the update vector at previous iteration aka Momentum where as β is the scalar weighting the contribution of Momentum in the current update. β typically takes a higher value such as 0.9.

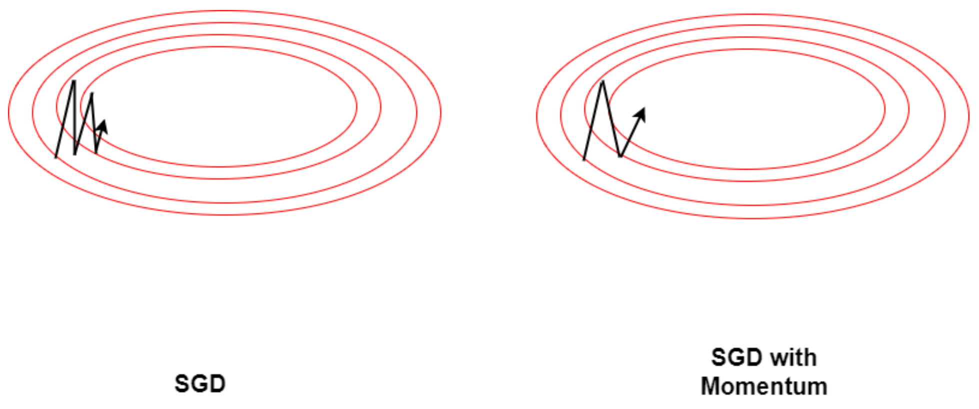


Figure 2.2: Stochastic gradient descent with and without Momentum

This means Stochastic Gradient Descent is greatly affected by the previous updates and results in smooth optimization.

Another concept related to Momentum [22] and presented in [12] is that within a data set, some features are rare yet more important than the others. Thus, the Momentum also needs to be adaptive referring it should consider the sparsity in the data set. The solution gives birth to a very famous family of algorithms known as Adagrad [12]. Adagrad adapts the learning rate to the parameters as it performs larger updates for rare features and small updates for frequent features.

Formally, we can denote the gradient of a single parameter i at kth iteration as g_{ik}. Lets also assume θ_{ik} to be the value of parameter i at this iteration. Using this gradient in next iteration can accelerate Stochastic gradient descent in only relevant direction of the local minimum with respect to the parameter i. This is formally expressed in (2.11)

$$\theta_{ik+1} = \theta_{ik} - \eta g_{ik} \tag{2.11}$$

The final learning rule for a single parameter then becomes

$$\theta_{ik+1} = \theta_{ik} - g_{ik} * \frac{\eta}{\sqrt{G_{iik} + \epsilon}} \tag{2.12}$$

Where $\mathbf{G}_k \in R^{z*z}$ is a diagonal matrix where each diagonal entry iik is the sum of the squares of the gradient with respect to θ_i upto iteration k and ϵ is the term used to avoid the numerical problem of division by zero.

The update rule for all the parameters at each iteration based on (2.12) is given in (2.13)

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{g}_k * \frac{\eta}{\sqrt{\mathbf{G}_k + \epsilon}} \tag{2.13}$$

The expression in (2.13) is the final representation of Adagrad as taken from [10] and [12]. Adagrad has proven to be a nice candidate to handle sparsity but still has many drawbacks including the positive summation of gradients in the denominator which keeps growing and results in little to no learning in extreme cases. In the next section, I describe Adam which is state of the art algorithm to train neural networks and is inspired from Momentum [22] and Adagrad [12].

2.7 Adam

Adam : adaptive moment estimation is one of the most efficient algorithms to train neural networks. It was fundamentally inspired by Momentum. Momentum is used to overcome the oscillation problem of Stochastic gradient descent[9]. More specifically, Stochastic gradient descent oscillates across the slopes of the cost function making very little progress to reach the local solution. This is intuitive, since Stochastic gradient descent only uses very little data i.e one input example at every iteration to optimize the cost function. Using very little information like that, it's essentially a noisy version of the Batch gradient descent [10] and thus oscillates while converging. This oscillation can be a serious problem in rate of convergence for Stochastic gradient descent. To improve the performance of Stochastic gradient descent, a new term Momentum [22] was introduced in machine learning literature.

The basic idea behind Momentum is to accelerate the Stochastic gradient descent in the direction of local optimum by minimizing the number of oscillations. This can be done by including the information of the past gradients at each time step. With the information regarding the gradients at previous time step, Stochastic gradient descent can reduce the oscillations.

Intuitively, we can understand that Momentum increases the stability in optimization by accelerating only in the relevant direction. Fundamentally, we can think that by using information of previous gradients, we can achieve convergence in less number of iterations since the optimization is much smoother.

2.8 Neural Networks Optimization via Adam

Adaptive Moment Estimation : Adam is a class of algorithms that use adaptive learning rate for each parameter. The first notable algorithm in neural networks literature which used adaptive learning rate was Adagrad [12] which has been discussed in the last section. Later on, similar algorithms like Adadelta [13],RMSprop [10], Adamax [7] and Nadam [10] were introduced. Adam can also be considered as an extension of this family in the sense that it uses adaptive learning rate for each parameter. More specifically,

Let $f(\boldsymbol{\theta})$ be the stochastic objective function with $\boldsymbol{\theta}$ acting as the vector of parameters. We are interested in minimizing the expected value of this function denoted as $E[f(\boldsymbol{\theta})]$. We assume here that $f(\boldsymbol{\theta})$ is differentiable with respect to its parameters $\boldsymbol{\theta}$. Let us also introduce the vector of partial derivatives of $f(\boldsymbol{\theta})$ at time step t as

$$\mathbf{g}_t = \delta f(\boldsymbol{\theta}) \tag{2.14}$$

Now, we can estimate the 1st and 2nd moment (mean and uncentered variance) of gradient at time step t as

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t \tag{2.15}$$

and

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \tag{2.16}$$

respectively where β_1 and β_2 are the hyper-parameters in the region $[0, 1)$ controlling the contribution of exponential decaying average of the 1st and 2nd moment of the past gradients. Here $\mathbf{g}_t^2$ represents the element wise square of $\mathbf{g}_t$. The algorithm, thus updates the exponential decaying average of the gradient $\mathbf{m}_t$ and squared gradient $\mathbf{v}_t$. In [7], author initialize it with zeros and explain that because of this initialization, these estimates are biased towards zero. Thus, two new equations are introduced for the unbiased estimates of $\mathbf{m}_t$ and $\mathbf{v}_t$.

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t} \tag{2.17}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \tag{2.18}$$

Where β_1^t and β_2^t are the hyper-parameters discussed earlier at time step t. Finally, the update rule for the vector of parameters $\boldsymbol{\theta}$ is given below

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}\hat{\mathbf{m}}_t \tag{2.19}$$

where η is the learning rate.

Adam is one of the most sophisticated neural networks training algorithms and can be used to train a variety of networks. One important thing to note is that all the algorithms discussed above are first order Gradient descent algorithms. Neural networks training has gained significant attention in last decade or so and many other techniques as opposed to first order Gradient descent have been investigated. Examples for such work include the famous Real time recurrent learning (RTRL) [23] as opposed to Back-Propagation through time, Quasi newton [24] and Hessian free optimization of recurrent networks [25].

In the next chapter, I present Successive convex approximation as taken from [8] and is adapted here for general recurrent neural network architecture.

Chapter 3

Optimization in Recurrent Networks via SCA

In this chapter, I discuss the optimization in recurrent neural networks with newly proposed Successive convex approximation [8]. More specifically, I discuss the idea of Successive convex approximation [8], the set up as previously discussed in the first chapter and then move to introduce some notations for a general recurrent neural network. I then provide the necessary relationships to understand the entire optimization process including the loss function and regularization which are adapted from [8].

3.1 Successive Convex Approximation

Successive convex approximation [8] is a newly proposed family of algorithms used to train neural networks. It has been previously discussed in detail in [8] where authors have used it to train feed forward neural networks. In this section, I introduce some new notations specific for a general recurrent neural network. More specifically, I provide the concept of Successive convex approximation as taken from [8]. I then move to describe the necessary expressions for Successive convex approximation as an optimization tool for recurrent neural networks.

The idea behind Successive convex approximation is to replace the original non convex, high dimensional optimization problem as discussed in the first chapter with a series of convex approximations. This is done by substituting the original loss function of optimization with a surrogate loss function. In [8], authors propose the linear approximation of the neural network function. The surrogate loss function, in general, is designed to simplify the optimization problem.

More specifically, I introduce the the following notations. Let $f(\mathbf{w}; j, \mathbf{x}_i)$ be a generic recurrent neural network function which takes two arguments. The first argument is the time step j while the second argument is the real valued vector of inputs $\mathbf{x}_i \in R^d$ i.e. ith training example at time j and $\mathbf{w}$ is a vector set of configurable parameters of the neural network and has to be learned from the external data set.

Assuming we have a training data set of N sequences where each such sequence can be denoted as $S = \{\mathbf{x}_i, y_i\}$. Also, assuming to have fixed the number of time steps which is denoted by M. Using the results directly provided in [8] and the optimization problem discussed in the first chapter, the learning of the recurrent neural network becomes.

$$min(U(\mathbf{w})) = \frac{1}{N}\frac{1}{M}\sum_{i=1}^{N}\sum_{j=1}^{M} l(y_{ij}, f(\mathbf{w}; j, \mathbf{x}_i)) + \lambda r(\mathbf{w}) \tag{3.1}$$

In (3.1), y_{ij} is the actual output for the sequence i at time step j, $f(\mathbf{w}; j, \mathbf{x}_i)$ is the corresponding estimated output, and $l(.,.)$ is a convex, smooth loss function as discussed in the first chapter. $r(\mathbf{w})$ is the regularization term which is weighted by a scalar λ.

Successive convex approximation [8] implies certain assumptions on the activation functions of the neurons and the surrogate functions for $l(.,.)$ which can be used in the optimization. These assumptions are discussed in the next section.

3.2 Assumption Set in Successive Convex Approximation

In this section, I provide assumptions and constraints on recurrent neural network function and the loss function as taken from the existing literature on Successive convex approximation [8]. This is to provide the theoretical guarantees of convergence for Successive convex approximation. These assumptions are provided below.

- f is continuously differentiable with respect to it's parameters denoted as $\mathbf{w}$
- f has Lipshitz continuous gradient with respect to $\mathbf{w}$ for some constant $D > 0$
- $\hat{l_{ij}}$ is convex and differentiable with respect to $\mathbf{w}$ anywhere.
- $\delta\hat{l_{ij}}(\mathbf{w};\mathbf{w}) = \delta l_{ij}(\mathbf{w})$ for any $\mathbf{w}$
- $\hat{l_{ij}}$ has Lipshitz continuous gradient with respect to $\mathbf{w}$ for some constant $E > 0$

Satisfying these assumptions means the surrogate loss $\hat{l_{ij}}$ keeps the first order properties of the original loss l_{ij} which is the fundamental building block of the optimization. This also means that the neural network function f has some constraints on the choice of the activation functions. Thus, activation functions with non differentiable points such as ReLu [26] and maxout neurons [27] are unable to be optimized using Successive convex aproximation at this point. A final point in this regard is that these assumptions are highly flexible and can result in a variety of surrogate loss functions to be used in optimization.

We can now understand that the simplest way to solve the optimization problem in (3.1) is to use Stochastic gradient descent SGD [10] which has been discussed in the previous chapter. The update rule for Stochastic gradient descent is provided in (3.2).

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \delta U(\mathbf{w}_k) \tag{3.2}$$

Here $\mathbf{w}_k$ is a very generic notation describing all the configurable parameters of the neural networks at iteration k, and α_k is slowly diminishing step size.

There can be a lot of variants of Stochastic gradient dsescent which have their advantages and disadvantages.

Theoretically, Stochastic gradient descent only uses the first order information of the cost function $U(.)$ which typically results in slow convergence. Successive convex approximation on the other hand tries to build the approximation of this convex cost function which can preserve as much as its convexity. More specifically, Successive convex approximation builds a strong convex approximation for each loss l_{ij} with a surrogate loss $\hat{l_{ij}}$ which keeps the first order properties of the former as specified in the assumption set. Next, Successive convex approximation solves this surrogate optimization problem which will be formally expressed below. The new estimate of the parameters $\mathbf{w}$ then becomes a convex combination of the previous estimates and the solution to the surrogate optimization problem. Taking the problem specified in (3.1), I now formally introduce some notations for Successive convex approximation.

Let $l_{ij}(\mathbf{w}) = l(y_{ij}, f(\mathbf{w}; j, \mathbf{x}_i))$ be a single loss term in (3.1) and let $\hat{l_{ij}}(\mathbf{w}; \mathbf{w}_k)$ be a surrogate loss of l_{ij} which meets the certain assumptions by Successive convex approximation discussed above. Then, the update rule at iteration k by using a mini batch B_k becomes

$$\begin{aligned}\hat{\mathbf{w}}_{k+1} = argmin : \mathbf{w}\{\frac{\rho_k}{NM}.\sum_{i=1}^{N}\sum_{j=1}^{M}\hat{l_{ij}}(\mathbf{w}; \mathbf{w}_k) + \lambda r(\mathbf{w}) \\ +(1-\rho_k)\mathbf{d}_k^T(\mathbf{w}-\mathbf{w}_k) + \tau\|\mathbf{w}-\mathbf{w}_k\|_2^2\}\end{aligned} \tag{3.3}$$

Here

- N is the number of data points i.e. input example(s) in the mini batch.
- M is the number of time steps in unfolded recurrent neural network. M can be considered a variable but here I assume it's fixed to a scalar.
- $\mathbf{d}_k \in R^Q$ is the smoothed average of gradients untill iteration k.
- ρ_k is scalar used to weight the information of current mini batch B_k w.r.t historical information kept in $\mathbf{d}_k$.
- The last term with τ is used to make the optimization problem strongly convex.

The above equation gives us an estimate of weights for the next iteration. This estimate depends only on the current mini batch and the smoothed average of previous gradients. We use this estimate of weights for the next iteration with current weights. Both terms are weighted by an iteration dependent step size α_k. The equation for this is given as

$$\mathbf{w}_{k+1} = (1 - \alpha_k)\mathbf{w}_k + \alpha_k \hat{\mathbf{w}}_{k+1} \tag{3.4}$$

While the equation to update the variable d_k becomes

$$\mathbf{d}_{k+1} = (1 - \rho_k)\mathbf{d}_k + \frac{\rho_k}{NM} \sum_{i=1}^{N} \sum_{j=1}^{M} \delta l_{ij}(\mathbf{w}_k)) \tag{3.5}$$

I skip here the proof of convergence presented in [8] to $\mathbf{w}^*$, which is a local solution of (3.1).

3.3 Surrogate Optimization in Successive Convex Approximation

Now that we understand how Successive convex approximation updates the weights of the neurons, I discuss how we can make the surrogate optimization. Fundamentally, we have to approximate the convex cost function at each iteration, in a way such that the approximation is easier to work with. Although in principle we can choose any surrogate loss function which satisfies the assumption set in Successive convex approximation framework, it's quite intuitive that we want to preserve the convexity of the loss function as much as possible. Thus, the fundamental linear approximation of the loss function given as

$$\hat{l_{ij}}(\mathbf{w}; \mathbf{w}_k) = l_{ij}(\mathbf{w}_k) + \delta l_{ij}(\mathbf{w}_k)(\mathbf{w} - \mathbf{w}_k) \tag{3.6}$$

can not preserve the entire hidden convexity of the loss function. Thus, in [8], authors provide a reasonable way to preserve the convexity of loss function which is done by substituting the original neural network function with first order linear approximation. The neural network function $f(\mathbf{w}_k; j, \mathbf{x}_i)$ is approximated by

$$\hat{f_{ij}}(\mathbf{w}; \mathbf{w}_k) = f(\mathbf{w}_k; j, \mathbf{x}_i) + \mathbf{J}_{ij,k}(\mathbf{w} - \mathbf{w}_k) \tag{3.7}$$

whereas

$$\mathbf{J}_{ij,k} = \frac{\delta(f(\mathbf{w}_k; j, \mathbf{x}_i)}{\delta \mathbf{w}_k} \tag{3.8}$$

is the vector of partial derivatives of single neural network output with respect to current parameters. In [8], authors describe it as weight jacobian. Here, we have total of $M * N$ such jacobian vectors at every iteration k. The total space required to store these matrices is in the order of $M * N * Q$ where M, N and Q have been discussed above.

After having all the necessary ingredients for the approximation of the loss function, it can be defined as

$$\hat{l_{ij}}(\mathbf{w}; \mathbf{w}_k) = l(y_{ij}, \hat{f_{ij}}(\mathbf{w}; \mathbf{w}_k)) \tag{3.9}$$

It can be shown in [8] that this surrogate loss function satisfies all the assumptions. Thus, it can be used for optimization and convergence can be achieved theoretically.

In the next section, I discuss two possible choices for regularization namely L_1 and L_2 same as [8].

3.4 Successive Convex Approximation with L2 Regularization

In this section, I describe how to use Successive convex approximation for a mean squared loss with l_2 regularization. I describe

$$l(a, b) = (a - b)^2 \tag{3.10}$$

$$r(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|_2^2 \tag{3.11}$$

where (3.10) shows a squared loss function whereas (3.11) describes an l_2 regularization. Because we have very strong convex regularization term, we can set τ to be zero in (3.3). A residual term $r_{ij,k}$ is described in the next equation which will be used in algebra manipulations same as [8].

$$r_{ij,k} = y_{ij} - f(\mathbf{w}_k; j, \mathbf{x}_i) + \mathbf{J}_{ij,k}^T \mathbf{w}_k \tag{3.12}$$

Using $r_{ij,k}$ and defining

$$\mathbf{A}_k = \frac{\rho_k}{NM}.\sum_{i=1}^{N}\sum_{j=1}^{M}\mathbf{J}_{ij,k}\mathbf{J}_{ij,k}^T \tag{3.13}$$

$$\mathbf{b}_k = \frac{\rho_k}{NM}.\sum_{i=1}^{N}\sum_{j=1}^{M}\mathbf{J}_{ij,k}r_{ij,k} - \frac{(1-\rho_k)}{2}.\mathbf{w}_k \tag{3.14}$$

the optimization problem in (3.3) can be written as

$$\hat{\mathbf{w}}_{k+1} = argmin : \mathbf{w}\{\mathbf{w}^T\left(\mathbf{A}_k + \lambda\mathbf{I}\right)\mathbf{w}_i - 2\mathbf{b}_k^T\mathbf{w}\} \tag{3.15}$$

where $\mathbf{I}$ is the identity matrix. The optimization problem in (3.15) is quadratic. The solution of this quadratic problem (3.15) is given by,

$$\hat{\mathbf{w}}_{k+1} = \left(\mathbf{A}_k + \lambda\mathbf{I}\right)^{-1}\mathbf{b}_k \tag{3.16}$$

Note that (3.16) gives us a solution to optimize a surrogate loss function. This solution involves inversion of a square matrix with dimensions $Q * Q$ where Q denotes the number of entries in the vector set of parameters i.e total number of parameters in the entire neural network. At times, this solution is difficult to obtain since for problems related to computer vision, we may have as many as a few million parameter. As such, keeping a square matrix with dimensions in millions is practically impossible. In that case, authors in [8] describe a decomposition strategy to solve this efficiently. I skip those details here.

Another important comment about the solution in (3.16) is the possibility of numerical problems. If that happens, we can always set $\tau > 0$ and we will get the solution

$$\hat{\mathbf{w}}_{k+1} = \left(\mathbf{A}_k + (\lambda+\tau)\mathbf{I}\right)^{-1}\left(\mathbf{b}_k + \tau\mathbf{w}_k\right) \tag{3.17}$$

which is in principle the same as (3.16) and converges to the local solution of (3.1).

3.5 Successive Convex Approximation with L1 Regularization

We can denote l_1 regularization as

$$r(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_{i \in Q} |\mathbf{w}_i| \tag{3.18}$$

where Q is the vector of parameters.

The surrogate optimization problem using l_1 regularization can be written as

$$\hat{\mathbf{w}}_{k+1} = argmin : \mathbf{w}\{\mathbf{w}^T \mathbf{A}_k \mathbf{w}_i - 2\mathbf{b}_k^T \mathbf{w} + \lambda \|\mathbf{w}\|_1\} \tag{3.19}$$

The problem in (2.25) can be solved by a wide range of proximal gradient methods such as FISTA [14], NESTA [15] , SPGL1 [16]and GPSR [17]. The formulation in (3.19) can be extended to group sparse [18] which is a structured from of regularization. Most algorithms which solve (3.19) can also solve the group sparse regularization.

3.6 Effecient Jacobian Computation

Successive convex approximation is a surrogate approximation of problem in (3.1). Successive convex approximation has nice theoretical properties including the convergence of surrogate optimization and the choice of regularization. The biggest challenge in Successive convex approximation from a practical point of view is the efficient computation of (3.8). It is evident from the relationships in (3.12), (3.13) and (3.14) that the term weight jacobian which is present in (3.8) is the fundamental building block of the entire optimization process. Thus, in this section I explain some practical insight into the computation of this term.

The relationship in (1.4) and (1.6) tell us that the hidden state of the neural network at any time step t can be computed using only the input, the previous hidden state and the set of parameters required for the connections. Thus, at every time step, we can save an auxiliary variable z in the memory which is the current hidden state $\mathbf{h}_t$ and avoid the re-computation of this term when we compute the next hidden state $\mathbf{h}_{t+1}$.

Once we compute the hidden state at any time step, computing the output at that time step is trivial and is thus skipped. The pseudo-code for efficient hidden state computation is given in Algorithm 1. This algorithm can be used to compute the expression in (3.12), (3.13) and (3.14).

Algorithm 1 Pseudo-code to compute hidden state at every time step

```
$\mathbf{h}_0 \leftarrow 0$
$j \leftarrow 1$
$z \leftarrow \mathbf{h}_0$
while $j < m$ do
    Compute $\mathbf{h}_j$ from (1.4) Using the auxiliary variable $z$ instead of $\mathbf{h}_{j-1}$
    Update the auxiliary variable as $z \leftarrow \mathbf{h}_j$
end while
```

In the pseudo-code, m denotes the number of time steps. Finally, to compute the term in (3.8), we have to use the Back-propagation through time aka BPTT which is a standard algorithm to calculate the gradients in recurrent neural networks. I will skip the analytic derivation of BPTT which is present in [3].

Chapter 4

Hyper-parameters Optimization

In this chapter I briefly discuss the concept of global optimization also known as hyper-parameters optimization. Hyper-parameters optimization is one of the few concepts in machine learning literature lacking intuition, theoretical understanding and practical guidelines. This is due to the intrinsic nature of the problem since hyper-parameters optimization is built on top of parameters optimization i.e learning and as such demands higher computational resources.

In this chapter, I discuss the hyper-parameters optimization using Gaussian processes. I apply the concept to practical test using python module GPyOpt [21]. More specifically, I use GPyOpt which is a python module that uses Gaussian processes to find out the best hyper-parameters for two training algorithms Adam and Successive convex approximation within fixed number of iterations. I then use these hyper-parameters in Adam and Successive convex approximation and compare their performance results which are discussed in detail in the next chapter.

4.1 Hyper-parameters Optimization

The hyper-parameters optimization also referred to as global optimization points to finding the best set of parameters that any arbitrary function takes to process some input. Hyper-parameters optimization is one of the most challenging problems in machine learning. This is due to the fact that the under lying machine learning model e.g neural network takes a lot of time for it's own learning. Thus, it's difficult to find the best hyper-parameters within very few iterations and almost always, the default hyper-parameters are used.

Hyper-parameters Optimization in machine learning is a subject lacking theoretical insight and empirical evidence. Untill recently, grid search was the most obvious choice regarding hyper parameter optimization. It simply meant that the designer of the machine learning algorithm presented some choices of the hyper-parameters based on his/her previous knowledge. Randomly changing the hyper-parameters was another choice however both were unfeasible. This was due to the fact that the former required too much technical expertise on the subject while the latter lacked meaningful intuition. Recently, some work on hyper-parameters optimization using non parametric models such as Gaussian processes has gained considerable attention.

4.2 Hyper-parameters Optimization via GPy-Opt

In this section, I discuss the hyper-parameters optimization for two algorithms namely the state of the art Adam and the newly proposed Successive convex approximation. More specifically, I use Gaussian processes (GP) [19] which are implemented in Python module called GPyOpt [20].

Gaussian processes (GP) [19] are defined as a finite collection of jointly Gaussian distributed random variables. For regression problems these random variables represent the values of a function $f(\mathbf{x})$ at input points $\mathbf{x}$. Prior beliefs about the properties of the latent function are encoded by the mean function $m(\mathbf{x})$ and co variance function $k(\mathbf{x}, \mathbf{x}_0)$. Thereby, all function classes that share the same prior assumptions are covered and inferences can be made directly in function space.

The Python module GPyOPt implements the Gaussian processes for hyper-parameters optimization [20]. It requires the definition of a function, the hyper-parameters to be optimized, and some prior belief on the optimal value of the parameters. Using this information, it tries to optimize the hyper-parameters of the function.

In the case of Adam, there are four hyper-parameters as presented in (2.17), (2.18) and (2.19). For Successive convex approximation, I try to optimize the initial step size α_0, the scalar to update the gradients ρ and the term to make the optimization problem strongly convex τ as presented in (3.3) and (3.4). The function to be optimized is the mean classification score on test data set. The initial priors are set randomly. The entire process is described below.

I run the GPyOpt for 50 iterations for both training algorithms Adam and Successive convex approximation. The initial beliefs are fed randomly. At every iteration, GPyOpt uses the prior to guess the best possible hyper-parameters. It uses those parameters in the training algorithm and runs the training process. Once the learning process ends, it calculates the mean classification score on test data set and saves the hyper-parameters and score in the memory. In the next iteration, it uses the initially provided and the previously used hyper-parameters as priors and repeats the same process. After 50 iterations, it returns the best hyper-parameters based on the best mean classification score. This is done for all 8 experiments that are mentioned above.

More specifically, the process is done for two data sets i.e. MNIST and Fashion MNIST, for two training algorithms i.e. Adam and Successive convex approximation and for two input representations e.g. standard and column wise. Once the best hyper-parameters are learned within 50 iterations for each of the 8 choices, I use those hyper-parameters to finally compare the performance of both training algorithms. The experimental results on the performance comparison of Adam and Successive convex approximation with two settings i.e. with default hyper-parameters and with hyper-parameters received from Gaussian Processes (GP) as implemented in GPyOpt are discussed.

Chapter 5

Experimental Results

In this chapter, I present some experimental evidence on the performance comparison of Successive convex approximation with state of the art neural networks training algorithm Adam on benchmark classification problems. More specifically, I compare the performance comparison in two settings i.e. with default hyper-parameters and with hyper-parameters received from Gaussian Processes (GP) as implemented in GPyOpt. I use two data sets Mnist and Fashion Mnist, two training algorithms Adam and Successive convex approximation, and two input representations called standard and column wise through out this chapter respectively. Finally, I present the conclusion of the dissertation and the future research line on Successive convex approximation.

5.1 Experimental Results

In this section, I present some experimental insight into the performance comparison of Adam and Successive convex approximation. There were total of 16 experiments. This was due to the fact that I used two data sets namely the famous hand written digits database MNIST and zalando's article images database Fashion MNIST. Further, the performance comparison involved two algorithms Adam and Successive convex approximation. The two variations of input representation namely row wise (called standard throughout the chapter) and column wise resulted in 8 experiments.

The total experiments rose to 16 because of the hyper-parameters optimization as discussed in the previous chapter. More specifically, the above set up of 8 experiments was run for two cases. The first case where we used the default hyper-parameters of both training algorithms while the later concerned with the hyper-parameter optimization of both algorithms using Gaussian processes.

I will briefly discuss the both data sets and the two choices for input representations. Next, I will present the performance comparison using default hyper-parameters for both algorithms.

5.2 Data Sets and Input Representation

MNIST is a database of hand written images (digits) and is one of the widely used synthetic data sets for classification. The data set has a total of 60,000 instances. In the experiments, I divide the training data set of MNIST in two disjoint sets. The first set containing the first 50,000 instances is called the training set. The second set containing the last 10,00 instances is called test set. The same process is repeated for Fashion MNIST since both data sets are identical.

For MNIST, all the images have been size normalized and centered in a fix size image. More specifically, the original grey scale images from NIST have been sized normalized so they can fit in a 20*20 pixel box keeping their aspect ratio intact. Later, the digits were centered in a 28*28 image by computing the center of the mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field. Each image in MNIST belongs to one of the ten classes 0-9.

Fashion MNIST is a data set of Zalando's article images with 60,000 instances. Each example is a 28x28 gray scale image, associated with a label from 10 classes. As it is evident from the name, Fashion MNIST images belong to one of the 10 Fashion classes. Fashion MNIST is considered to be relatively difficult to classify compared to MNIST.

For the sake of these experiments, I explicitly convert the labels of the images in both data sets in one hot encoding form. The LSTM architecture which I'm using for these experiments has 2 hidden layers each with 14 neurons. The input layer has 28 neurons and the sequence length is also 28. This means that for a single image, neural network will receive 28 pixels at a single time step, process them into 2 hidden layer each with 14 neurons and produce hidden state and output of the network. This process will be repeated 28 times. At the last time time step, the final output of the hidden

Algorithm	Input Representation	Median	Mean	Max	Min
Adam	Standard	92.5	91.69	100.0	77.5
SCA	Standard	85.0	84.14	97.5	67.5
Adam	Column	90.0	90.62	100.0	75.0
SCA	Column	90.0	89.34	100.0	70.0

Table 5.1: Adam vs Successive convex approximation performance comparison on MNIST Using default hyper-parameters

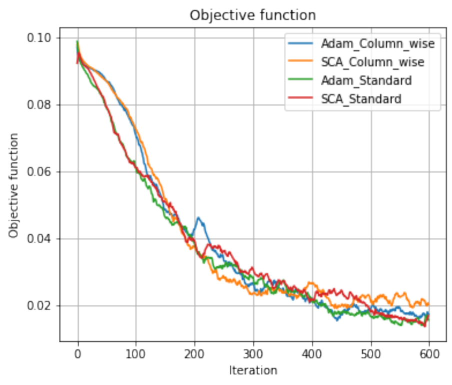
layers will be received. The output layer of the neural network will take this output of the hidden layers as input and convert them into the number of classes. The size of mini batches for both training algorithms is 40.

Now that we have a gentle idea of both data sets and neural network architecture, I can proceed to the way the input is fed to the LSTM networks. More specifically, every instance in both data sets is a 28*28 matrix. This matrix can be presented to LSTM as input without doing anything else. I refer to this choice as row wise or standard choice. This means that LSTM will receive 28 pixels of a row at single time step. At next time step, it will receive another 28 pixels for the next row and so on to receive all 784 pixels. As an opposite to this, we can feed the neural network with 28 pixels of a column at input layer and repeat this process for all columns untill all 784 pixels have been used in the input. I refer to this choice as column wise representation.

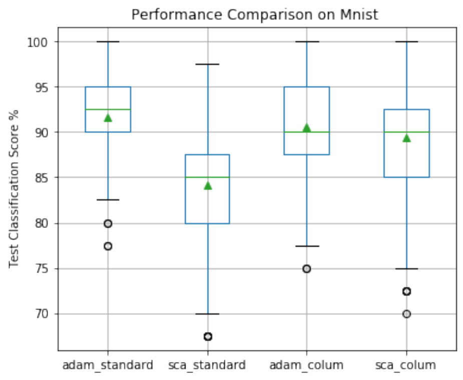
5.3 Performance Comparison using Default Hyper Parameters

Now, I will present the performance results for both training algorithms on MNIST data set with default hyper-parameters. The default hyper-parameters for Adam and Successive convex approximation are presented in [7] and [8] respectively. The regularization parameter λ is set to zero. The error function and test accuracy are presented in figure 5.1.

It is evident that both algorithms are competitive and produce really good results. The mean test scores for Adam and Successive convex approximation with standard input representation are 91.69 %and 84.14 % respectively. For



(a) Training Phase

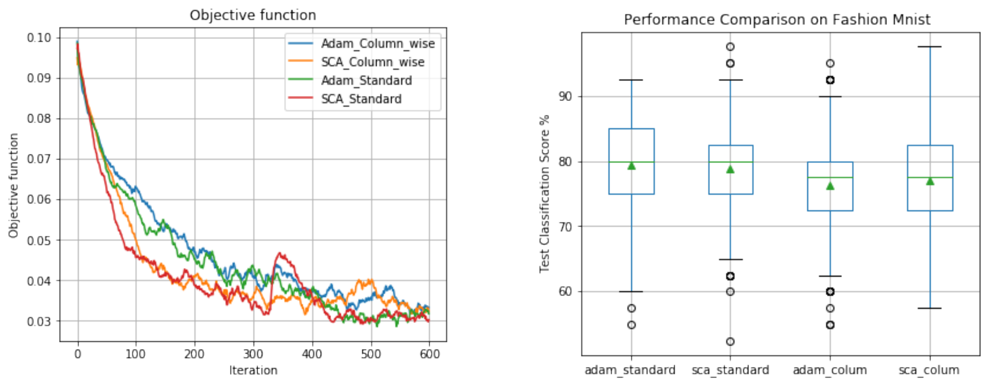


(b) Test Phase

Figure 5.1: Performance comparison on Mnist using default hyper-parameters

column wise representation, the mean test scores for Adam and Successive convex approximation are 90.62 %and 89.34 % respectively. The maximum test score for Adam and Successive convex approximation in standard representation is 100 % and 97.5 %. For column wise representation, this score is 100 % for both algorithms. The minimum test score for Adam and Successive convex approximation in standard representation is 75 % and 67.5 % while for column wise representation, the scores are 77.5 % and 70 %. Both algorithms produce good test accuracy, and can be seen that Successive convex approximation is competitive with Adam. The mean, median, maximum and minimum test score for both Adam and Successive convex approximation on MNIST are presented in Table 5.1.

Next, I will describe the experimental results on Fashion MNIST using the same default hyper-parameters. These results are presented in figure 5.2. Both algorithms produce good results but slightly less than MNIST. This is evident since Fashion MNIST is used as a relatively difficult data set for classification than MNIST. Using Fashion MNIST, the mean test scores for Adam and Successive convex approximation with standard input representation are 79.35 %and 78.9 % respectively. For column wise representation, the mean test scores for Adam and Successive convex approximation are 76.31 %and 76.925 % respectively. The maximum test score for Adam and Successive convex approximation in standard representation is 92.5 % and 97.5 %. For column wise representation, this score is 95.0 and 97.5 % respectively.



(a) Training Phase (b) Test Phase

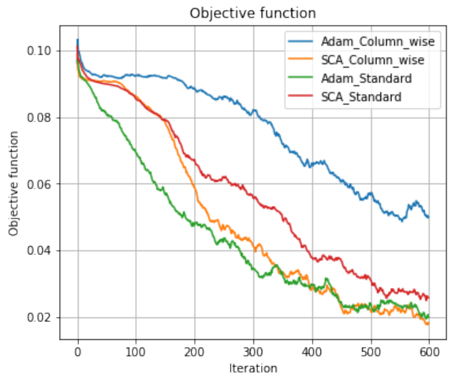
Figure 5.2: Performance comparison on Fashion MNIST using default hyper-parameters

Algorithm	Input Representation	Median	Mean	Max	Min
Adam	Standard	80.0	79.35	92.5	55.0
SCA	Standard	80.0	78.9	97.5	52.5
Adam	Column	77.5	76.31	95.0	55.0
SCA	Column	77.5	76.925	97.5	57.5

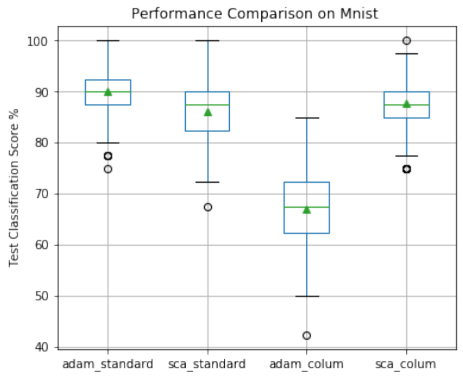
Table 5.2: Adam vs Successive convex approximation performance comparison on Fashion MNIST using default hyper-parameters

The minimum test score for Adam and Successive convex approximation in standard representation is 55 % and 52.5 % while for column wise representation, the scores are 55 % and 57.5 %. These experimental results are also presented in Table 5.2. It can be seen that Successive convex approximation is almost as good as Adam as the median test classification score is exactly the same. Successive convex approximation also has better mean score in column wise representation.

As a final note, we can safely conclude that both training algorithms produce competitive results using default hyper-parameters with Successive convex approximation surpassing Adam in some cases. In the next section, performance comparison involves hyper-parameters optimization.



(a) Training Phase



(b) Test Phase

Figure 5.3: Performance comparison on MNIST using hyper-parameters from Gaussian processes

Algorithm	Input Representation	Median	Mean	Max	Min
Adam	Standard	90.0	90.008	100.0	75.0
SCA	Standard	87.5	86.19	100.0	67.5
Adam	Column	67.5	67.13	85.0	42.5
SCA	Column	87.5	87.66	100.0	75.0

Table 5.3: Adam vs Successive convex approximation performance comparison on MNIST using hyper-parameters from Gaussian processes

5.4 Performance Comparison using Hyper-parameters Optimization

In this section, I compare the performance of both Adam and Successive convex approximation using the best hyper-parameters achieved within 50 iterations with GPyOpt. Using the data set MNIST, the results are presented in figure 5.3.

As it is evident from the training phase, Successive convex approximation is performing better than Adam in this setting. The mean test scores for Adam and Successive convex approximation with standard input representation are 90.08 %and 86.19 % respectively. For column wise representation, the mean test scores for Adam and Successive convex approximation are

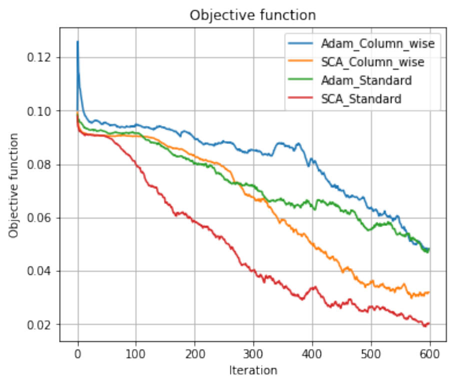
67.13 %and 87.66 % respectively. The maximum test score for both Adam and Successive convex approximation in standard representation is 100 %. For column wise representation, this score is 85 % and 100 % respectively. The minimum test score for Adam and Successive convex approximation in standard representation is 75 % and 67.5 % while for column wise representation, the scores are 42.5 % and 75 %. Although both algorithms produce good test accuracy, we can conclude that Successive convex approximation produces slightly better results in this experiment. The median,mean ,maximum and minimum test scores for both algorithms with hyper-parameters received from Gaussian processes on MNIST are also presented in Table 5.3

The above set up is run for Fashion MNIST and the results achieved are presented in figure 5.4. It is evident that Adam is not training well due to the poorly fed hyper-parameters resulted from GPYOpt. Successive convex approximation on the other hand receives very impressive hyper-parameters and as a result outperforms Adam in this experiment. As was the case with default hyper parameters, both algorithms perform relatively poor than MNIST since Fashion MNIST is a challenging benchmark data set. The median, mean,maximum and minimum test scores for both algorithms with hyper-parameters received from Gaussian processes on Fashion MNIST are also presented in Table 5.4.

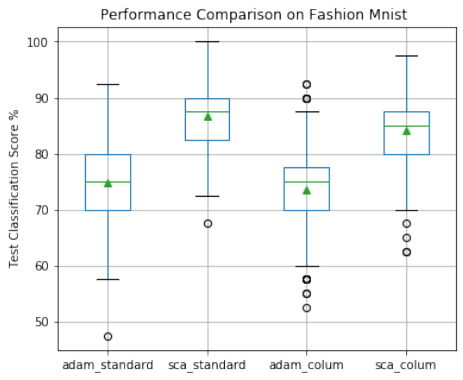
The mean test scores for Adam and Successive convex approximation with standard input representation are 74.88 %and 86.79 % respectively. For column wise representation, the mean test scores for Adam and Successive convex approximation are 73.55 %and 84.14 % respectively. The maximum test score for both Adam and Successive convex approximation in standard representation is 92. 5 and 100 % respectively. For column wise representation, this score is 92.5 % and 97.5 % respectively. The minimum test score for Adam and Successive convex approximation in standard representation is 47.5 % and 67.5 % while for column wise representation, the scores are 52.5 % and 62.5 %. Although both algorithms produce decent test accuracy, we can conclude that Successive convex approximation outperforms Adam in this experiment due to the presence of good hyper parameters.

5.5 Conclusion

In this dissertation, I studied the optimization in recurrent neural networks. As has been discussed above, learning in deep vanilla recurrent networks is



(a) Training Phase



(b) Test Phase

Figure 5.4: Performance comparison on Fashion MNIST using hyper-parameters from Gaussian processes

Algorithm	Input Representation	Median	Mean	Max	Min
Adam	Standard	75.0	74.88	92.5	47.5
SCA	Standard	87.5	86.79	100.0	67.5
Adam	Column	75.0	73.55	92.5	52.5
SCA	Column	85.0	84.14	97.5	62.5

Table 5.4: Adam vs Successive convex approximation performance comparison on Fashion MNIST Using hyper-parameters from Gaussian processes

still very much challenging due to the famous vanishing and exploding gradient problem. The state of the art on the approaches to solve vanishing and exploding gradient problem including the regulated architecture Long short term memory was also discussed. Later, I presented an overview of the famous Gradient descent based first order algorithms including the Batch gradient descent, Stochastic gradient descent and Mini-batch gradient descent. Additionally, some material on the algorithms based on Momentum including Adagrad and Adam were presented. After the state of the art, I discussed the newly proposed Successive convex approximation for neural networks training and adapted the notations and concept to the recurrent architecture. I also briefly discussed the roles of hyper-parameters and the concept of hyper-parameters optimization. Finally, some experimental insight into the performance of Successive convex approximation with state of the art training algorithm Adam was presented. As can be seen from the figures and tables, Successive convex approximation is competitive with Adam and produces better results on some occasions. The future research line is to investigate Successive convex approximation for a variety of networks including convolutional, Bi-directional recurrent and Gated recurrent unit networks and on a variety of academic and industrial benchmark data sets.

Bibliography

[1] **Jeffrey L.Elman**. *Finding structure in time.* Cognitive Science - Volume 14 , Issues : 2 , Pages 179-211 , 1990.

[2] **Sepp Hochreiter**. *Untersuchungen zu dynamischen neuronalen Netzen (German) [Investigations on dynamic neural networks].* Diploma Thesis, TU Munich , 1991.

[3] **P.J. Werbos**. *Backpropagation through time: what it does and how to do it.* Proceedings of the IEEE - Volume: 78, Issue: 10, 1990.

[4] **Sepp Hochreiter and Jürgen Schmidhuber**. *Long Short-Term Memory.* Neural Computation - Volume 9 , Issue 8 , 1997.

[5] **Razvan Pascanu , Tomas Mikolov and Yoshua Bengio**. *On the difficulty of training recurrent neural networks.* ICML'13 Proceedings of the 30th International Conference on Machine Learning - Volume 28 , Pages III-1310-III-1318, 2013.

[6] **Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee**. *Recent Advances in Recurrent Neural Networks.* arXiv preprint arXiv:1801.01078 , 2017.

[7] **Diederik P. Kingma and Jimmy Ba**. *Adam: A Method for Stochastic Optimization.* International Conference for Learning Representations, 2015.

[8] **Simone Scardapane and Paolo Di Lorenzo**. *Stochastic Training of Neural Networks via Successive Convex Approximations.* IEEE Transactions on Neural Networks and Learning Systems - Volume: PP, Issue: 99 , 2018.

[9] **J. Kiefer and J. Wolfowitz**. *Stochastic Estimation of the Maximum of a Regression Function.* The Annals of Mathematical Statistics - Volume: 23, Issue: 3 , Pages 462-466 , 1952.

[10] **Sebastian Ruder**. *An overview of gradient descent optimization algorithms.* arXiv:1609.04747 , 2016

[11] **Ilya Sutskever, James Martens, George Dahl and Geoffrey Hinton**. *On the importance of initialization and momentum in deep learning.* ICML'13 Proceedings of the 30th International Conference on Machine Learning - Volume 28 ,Pages III-1139-III-1147, 2013

[12] **J. Duchi, E. Hazan, and Y. Singer**. *Adaptive subgradient methods for online learning and stochastic optimization.* Journal of Machine Learning Research - Volume 12, Pages 2121–2159, 2011

[13] **Matthew D. Zeiler**. *ADADELTA: An Adaptive Learning Rate Method.* arXiv:1212.5701, 2012

[14] **Amir Beck and Marc Teboulle**. *A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems.* SIAM Journal on Imaging Sciences , Volume 2, Issues 1 , 2009

[15] **Stephen Becker, Jérôme Bobin and Emmanuel J. Candès**. *NESTA: A Fast and Accurate First-Order Method for Sparse Recovery.* SIAM Journal on Imaging Sciences , Volume 4, Issues 1 , Pages 1-39 , 2009

[16] **E. van den Berg and M. P. Friedlander**. *SPGL1: A solver for large-scale sparse reconstruction.*

[17] **Zhang Ning, Yunho Jung, Yan Jin and Kee-Cheon Kim**. *Route Optimization for GPSR in VANET.* IEEE International Advance Computing Conference , 2009.

[18] **Simone Scardapane, Danilo Comminiello, Amir Hussain and Aurelio Uncini**. *Group sparse regularization for deep neural networks.* Neurocomputing - Volume 241, Issue C, Pages 81-89, 2017

[19] **Leslie Foster, Alex Waagen, Nabeela Aijaz, Michael Hurley, Apolonio Luis, Joel Rinsky, Chandrika Satyavolu, Michael J. Way, Paul Gazis and Ashok Srivastava**. *Stable and Efficient Gaussian Process Calculations*. Journal of Machine Learning Research - Volume 10, 2009.

[20] **Emmanuel Kieffer, Grégoire Danoy, Pascal Bouvry and Anass Nagih**. *Bayesian optimization approach of general bi-level problems*. GECCO '17 Proceedings of the Genetic and Evolutionary Computation Conference Companion , Pages 1614-1621, 2017.

[21] **The GPyOpt authors**. *GPyOpt: A Bayesian Optimization framework in Python*. GPyOpt, 2016.

[22] **Ning Qian**. *On the momentum term in gradient descent learning algorithms*. Neural networks : the official journal of the International Neural Network Society, 12(1):145–151, 1999

[23] **R. Williams, and D. Zipser**. *A learning algorithm for continually running fully recurrent neural networks*. Neural Computation - Volume 1 , Issues 2, Pages 270-280, 1989.

[24] **J. Sohl-dickstein, B. Poole, and S. Ganguli**. *Fast large-scale optimization by unifying stochastic gradient and quasi-newton methods*. 31st International Conference on Machine Learning (ICML), Pages 604-612 , 2014.

[25] **James Martens and Ilya Sutskever**. *Learning recurrent neural networks with hessian-free optimization*. ICML'11 Proceedings of the 28th International Conference on International Conference on Machine Learning, Pages 1033-1040 , 2014.

[26] **X. Glorot, A. Bordes, and Y. Bengio**. *Deep Sparse Rectifier Neural Networks*. 14th International Conference on Artificial Intelligence and Statistics (AISTATS), Pages 315–323, 2011

[27] **I. J. Goodfellow, D. Warde-farley, M. Mirza, A. Courville, and Y. Bengio**. *Maxout networks*. Proc. 30th International Conference on Machine Learning (ICML), Pages 1319–1327, 2013