On Neural Networks and the Parity Function

Martin Georgiev

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh
2019

Abstract

Neural Networks are artificial intelligence models which have recently become very popular after solving major problems in image recognition and language processing. They have been performing better than humans on tasks which have been notoriously difficult for machine learning systems to solve. Unfortunately, neural networks sometimes struggle with very simple problems. Furthermore, it is very difficult to understand and predict their learning behavior and underlying representations. In this report, I investigate the strengths and weaknesses of neural networks while solving a simple arithmetic problem - the parity function. Various modern architectures and hyperparameters are explored and the best performing configurations are highlighted. Learning the parity function using neural networks turned out to be a difficult task due to the instability of the models. While most of the configurations struggled to learn the parity function, one model managed to achieve perfect generalization quickly and efficiently.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Artificial Neural Network (ANN) machine learning models have been growing in popularity due to their recent successes in image recognition (Krizhevsky et al., 2012), language processing (Kumar et al., 2015) and other practical domains. Some of the more interesting outcomes of using neural networks include ImageNet (He et al., 2015) - beating human performance on labeling images and AlphaGo (Silver et al., 2016) - beating a world champion at the popular board game.

Despite being considered a contemporary model, artificial neural networks have been around since the 1960s. A few factors allowed for their recent rapid development. Neural networks require vast amounts of computational power which became easier to obtain due to the exponential growth of the capacity of integrated circuits. Furthermore, ANNs tend to rely on large amounts of data which fell in cost and became widely available in recent years.

The aim of the project is to investigate the strengths and limitations of neural networks using a simple arithmetic problem. More generally, the task at hand could be considered a problem of learning algorithms. The project explores the process of creating, evolving and optimizing neural network models which solves a specific arithmetic problem. Moreover, it illustrates the robustness of Artificial Neural Networks when dealing with such tasks. Multiple types of models were investigated, including Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

When choosing which mathematical problems to explore, I decided to go in depth rather than breadth. While focusing on multiple problems could lead to some insights, due to time constraints, the problems would only be explored very briefly and the underlying workings of the neural networks with its best performing parameters would remain unknown. I chose to focus on the parity function problem because of its simple definition but inherent difficulty for neural networks to solve. Any bit change in the input changes the output of the function dramatically. Moreover, there are good theoretical results on the topic as the function has been frequently used to investigate the

complexity of threshold circuits, which are essentially simple neural networks.

The parity function is a boolean function that returns 1 if there are an odd number of ones in a bitstring and 0 otherwise. For example:

```
f('10101') = 1          f('10100') = 0          f('11111') = 1
f('00100') = 1          f('00000') = 0          f('11000') = 0
```

More formally, the n-variable Parity function is the Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ and $f(x) = 1$ if and only if the number of ones in the vector $\{0,1\}^n$ is odd. Another way to define the parity function is:

$$f(x) = x_1 \oplus x_2 \cdots \oplus x_n$$

where $\oplus$ is the XOR boolean function. As further discussed in Section 2.8, the Parity Function has been solved in numerous ways using specifically crafted network architectures or manually calculated weights. My approach to the problem is to try and solve it using common configurations which can be applied to any other arithmetic problems. In other words, I start with a general neural network and carefully adjust its architecture and parameters using insight from theory and experiments.

While solving the parity function problem using neural networks is not a novel idea on its own, it could definitely be beneficial in comprehending ANN and how they interact with arithmetic problems. I believe that this understanding can help us develop efficient models that might ultimately be used to solve more difficult and important problems. An example of such problem is integer prime factorization. There has been an interesting attempt by (Jansen & Nakayama, 2005) to solve the problem using neural networks but it had limited success. If this becomes possible, it could have long-lasting effects on cryptographic protocols and computer security in general. We are very far from solving such problems using neural networks but I hope this report can contribute to this long-term vision.

1.2 Main contributions

- Broad overview of neural networks, including fully-connected networks, back-propagation, convolution, pooling, recurrent networks, LSTM, activation functions, weight initialization, gradient descent optimization and regularization.
- Designed 15 original graphics to illustrate key concepts in neural networks and experimental architectures.
- Created a data generating script to produce parity function examples for training and evaluation.
- Experimented with 272 distinct configurations in order to understand how neural networks learn the parity function.

- Created a recurrent neural network model that learns the parity function with perfect generalization using few training examples and scaling linearly with the size of the input.

1.3 Structure of the report

Chapter 2 - provides the necessary background in neural networks needed for the rest of the report. It also gives an overview of the previous work on the topic.

Chapter 3 - describes the research questions the thesis wants to address. The technology and data generation method used and the training and evaluation procedures are also given.

Chapter 4 - consists of preliminary experiments which vary neural network hyperparameters in order to find a stable learning model for the parity function problem.

Chapter 5 - investigates deeper and wider networks and how the size of the network impacts learning of the parity function.

Chapter 6 - shows the application of one dimensional convolutional neural networks to the parity function problem. Hyperparameters specific to convolutions are tested and models are compared.

Chapter 7 - two recurrent neural network models are tested and compared. One general and one specifically designed for the parity function problem. The effects of using various recurrent layer sizes are also explored.

Chapter 8 - conclusion and limitations of the research conducted including directions for further work on the topic.

Chapter 2

Background

Ever since the invention of programmable machines, people have tried to create computers that can think. Many of the early Artificial Intelligence (AI) models have been focusing on hard-code knowledge about the world in formal languages (knowledge base approach). More recently, with the introduction of machine learning, AI systems developed the ability to acquire their own knowledge, by learning patterns from raw data. The Artificial Neural Network model is one of many such systems. It is loosely inspired by the way the human brain seems to process information (Grossberg, 1988) and the smallest building block of this model is the Perceptron.

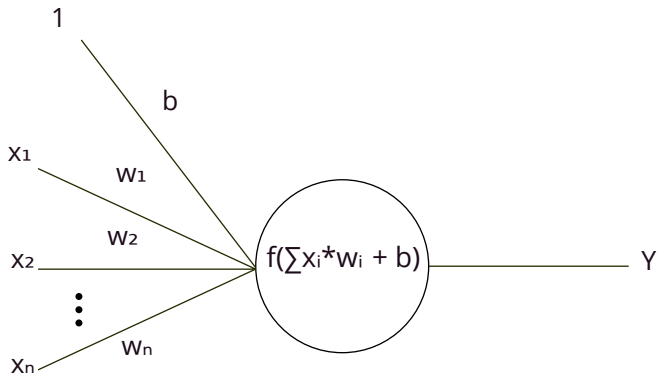

Figure 2.1: A simple perceptron (neuron).

Dating back to the 1950s, it is a simple structure that takes n inputs $x_1, x_2, ..., x_n$, a bias **b** and n weights $w_1, w_2, ..., w_n$. All the inputs are multiplied by their respective weights, the bias is added and an activation function is applied to the final result. The inputs and weights are treated as vectors and the perceptron is defined as follows:

$$Y = \begin{cases} 0 & \text{if } w * x + b \leq 0 \\ 1 & \text{if } w * x + b > 0 \end{cases} \tag{2.1}$$

This simple structure can be used to compute logical functions such as AND, OR and NAND. In fact, we can use networks of perceptrons to compute any logical function

due to the universality of the NAND gate. Modern neural networks utilize many of these simple structures to learn the optimal weights and biases that would approximate any target function. In fact, from the Universal Approximation Theorem (UAT) (Hornik et al., 1989) we know that a shallow neural network with a finite amount of neurons can approximate any function. Unfortunately, this does not give us guarantees for the number of neurons needed, the time it would take for the network to train and whether training would be successful at all since learning can get stuck in local minima.

The perceptron's step function is obsolete since small changes in the input can cause drastic changes in the output and is not differentiable at 0. Instead of the step function described above, modern neural networks use more elaborate activation functions such as the Sigmoid function and Rectified Learning Unit (ReLU) function (Nair & Hinton, 2010). Sometimes the apparent practical success of an activation function is not theoretically understood very well as is the case with ReLU. Further explanation of activation functions and their applications can be found in Section 2.4.

$$sigmoid(x) = \frac{1}{1+e^{-x}} \qquad relu(x) = \max(0,x)$$

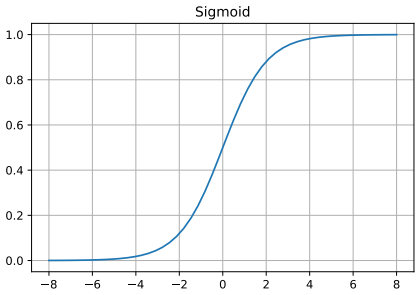

Figure 2.2: Sigmoid function.

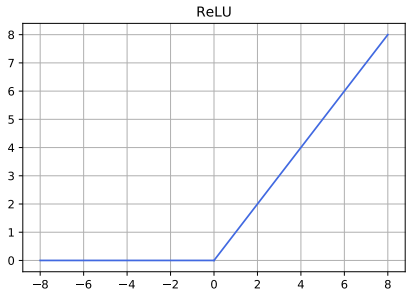

Figure 2.3: ReLU function.

2.1 Feed-forward Networks and Learning

Feed-forward networks, as mentioned earlier, are simply connected layers of neurons. A layer is a group of neurons which are not connected to each other but to other groups of neurons. The leftmost layer of a network is used for the input and the rightmost one for the output. Any number of hidden layers can be instantiated in between. Moreover, each hidden layer could have an arbitrary number of neurons.

In order to train our network, we need to provide many examples containing inputs representing the problem and desired outputs which are the solution to the problem. Usually, some of these examples are set aside so that we can evaluate our model on unseen data after training is complete.

To learn the best weights and biases for a specific problem, an algorithm known as back-propagation is used. Depending on the task at hand the output layer can have

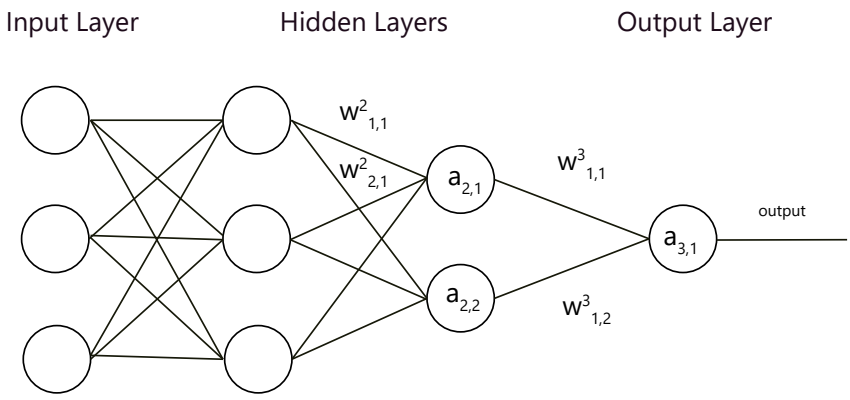

Figure 2.4: Fully-connected feedforward network.

a multitude of activation functions. The most common ones are Sigmoid for binary classification, Softmax for multi-class classification and Linear for regression. A cost (or loss) function measuring the difference between the predicted and the actual output is used. The goal of the neural network then is to minimize this cost by appropriately updating the weights and biases. This is done by iteratively calculating gradients for each layer using the chain rule.

Lets assume that the activation function of the output neuron is a sigmoid and the cost function is the Squared Error Function $E = \frac{1}{2}(target - output)^2$. From our neural network and neuron definitions, $output = sigmoid(a_{3,1})$ and $a_{3,1} = a_{2,1} * w^3_{1,1} + a_{2,2} * w^3_{1,2} + b$. We want to know how much a change in $w^3_{1,1}$ affects the total error, in other words $\frac{\partial E}{\partial w^3_{1,1}}$.

$$\frac{\partial E}{\partial w^3_{1,1}} = \frac{\partial a_{3,1}}{\partial w^3_{1,1}} * \frac{\partial output}{\partial a_{3,1}} * \frac{\partial E}{\partial output}$$

We can apply practically the same formula to calculate the gradient of the bias term of $b_{3,1}$:

$$\frac{\partial E}{\partial b_{3,1}} = \frac{\partial a_{3,1}}{\partial b_{3,1}} * \frac{\partial output}{\partial a_{3,1}} * \frac{\partial E}{\partial output}$$

Finally, we can iteratively calculate each of the gradients and update the weights using the gradient descent formula:

$$w := w - \eta * \frac{\partial E}{w}$$

where η is a parameter called the learning rate which defines the size of the changes we are applying to our weights. In other words, it determines how fast the neural network is learning. A very small η could mean that we are not learning at all but a too large one could mean that the changes are drastic and we can end up overshooting the

global minimum. More about how the learning rate is chosen and optimized in neural networks can be found in Section 2.6.

All of these equations allow us to calculate the updates for a single example and thus perform Stochastic Gradient Descent. In practice, this could be too computationally demanding and therefore, we divide the training set into smaller sets of examples and average out the calculated weight updates. This technique is known as Mini-Batch Gradient Descent.

2.2 Convolutional Neural Networks

Convolutional Neural Networks were perhaps the leading factor for the recent increase in interest in neural networks. Although they were first conceived by Yan LeCun (Lecun et al., 1998) in the 1990s, due to the computational limitations of the time, they were not widely adopted. One of the first successful model utilizing CNNs was AlexNet (Krizhevsky et al., 2012). Since then, they have solved major problems where other machine learning algorithms have failed.

Although while solving the parity function problem I use 1D convolution, CNNs are easier to understand when using them with images in 2 dimensions. One of the domains where CNNs excel is image recognition. Fully-connected neural networks cannot easily work with images. Take for example a standard colored HD image with a resolution of 1,280 by 720 pixels. That amounts to 2,764,800 neurons at the input layer making it too computationally expensive for feed-forward training. Furthermore, when flattening the image many of the underlying connections between adjacent pixels are lost. Convolutional Neural Networks solve these problems using a few simple techniques.

Traditionally, in feed-forward networks, each pixel of the image would be connected to all weights of the next layer. Convolutional neural networks take a different approach in order to solve this inefficiency. There is a filter (sometimes called a kernel bank) which is a 2D matrix of weights that convolves over the image and applies an activation function to produce a feature map. An activation function is applied to each calculated value. The goal of the CNN then is to minimize the error by updating this filter's weights.

More formally, the convolution operation is defined as follows:

$$h_{i,j} = sigmoid(\sum_{k=0}^{m-1}\sum_{l=0}^{m-1} w_{k,l} * x_{i+k,j+l} + b)$$

where $h_{i,j}$ is an entry in the feature map, $w_{k,l}$ are the weights of the filter and $x_{i+k,j+l}$ are the pixel values of the image. In practice, the neural network has several filters, each one detecting a certain characteristic of an image.

We can adjust the size of the Feature Map by changing the size of the kernel, padding the image or changing the stride. The stride is a hyperparameter that defines how many

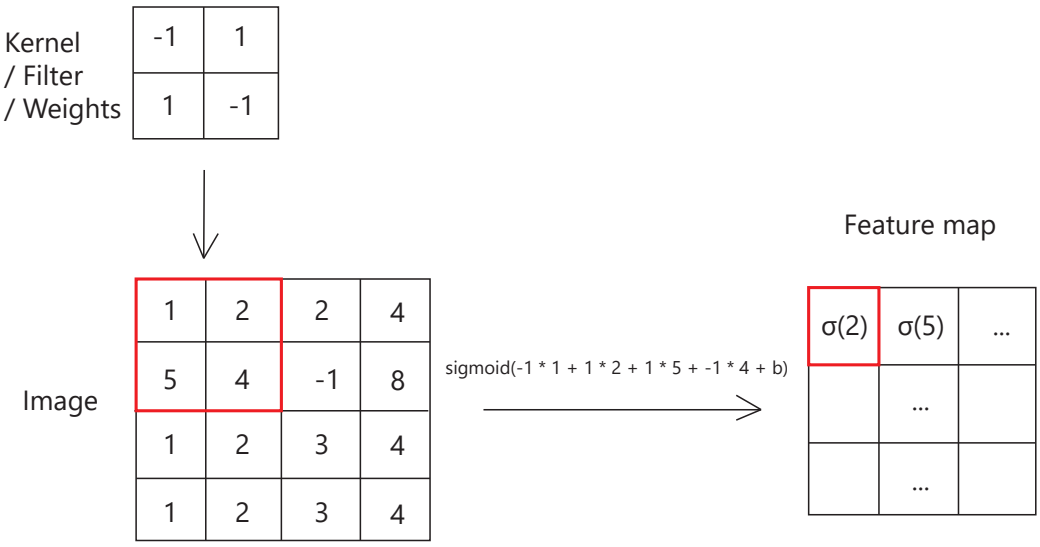

Figure 2.5: Convolutional layer.

pixels we jump while sliding the kernel. In Figure 2.5 the stride is set to 1 and changing that to 2 will shrink the size of the feature map matrix to 2x2.

Convolutional processing of an image works in higher dimensions as well. For instance, if the picture is colored, it will be represented in 3 dimensions - red, green and blue for each pixel. In that case, we just need to apply the kernel on each dimension and sum the results to produce the feature map.

2.2.1 Pooling Layer

The pooling layer, also known as subsampling, is applied after the convolutional layer. Its function is to reduce the size of the feature map and the number of parameters, thus optimizing and regularizing the network. There are many types of pooling layers such as Average, Sum and L_p but the most widely used one is Max Pooling. It combines adjacent entries of the feature map obtained from the convolutional layer, into one value by taking the maximum value of that region of the matrix as shown in Figure 2.6. Usually, we make the stride large enough such that the pooling regions do not overlap.

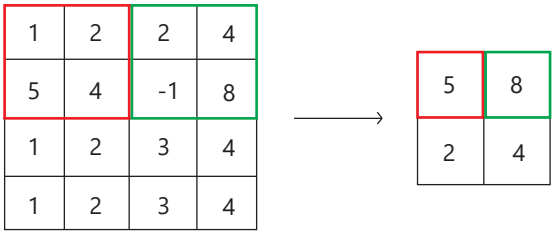

Figure 2.6: Pooling layer.

The depth dimension of the convolutional layer output remains unchanged. In other words, pooling is applied to each feature map independently.

2.2.2 Fully-connected Layer

Convolutional and pooling layers are repeated until there is small enough number of features that can be handled by a feed-forward network. Subsequently, each entry of the feature maps is used as an input to a fully connected network and continues as described in the previous section.

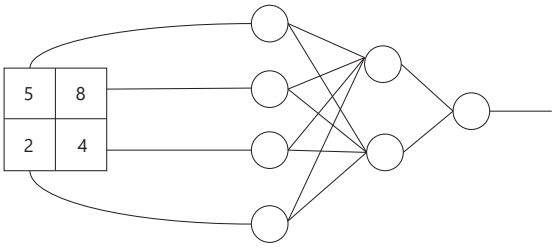

Figure 2.7: Flattening layer.

2.3 Recurrent Neural Networks

Recurrent Neural Networks (RNN) are a different type of neural network models that achieve impressive results when dealing with sequential data. They are designed to remember important features about the previous input they have received, putting the next prediction into context. This is particularly useful when handling sequences of data such as speech, audio, video and financial movements.

Similarly to the fully-connected and convolutional models, RNNs are relatively old. They were initially introduced in the 1980s (Hopfield, 1982) but due to the computational limitations at the time and other issues with training, they were not adopted until recently. Some of the areas recurrent neural networks have excelled include Machine Translation (Bahdanau et al., 2014), Speech Recognition (Graves et al., 2013), Image Captioning (Vinyals et al., 2014), Language Modeling and Text Generation (Graves, 2013).

Recurrent networks are distinguished from feedforward networks by a loop connecting their past outputs to the next prediction as shown in Figure 2.8. The meaning of each parameter of the recurrent layer is described below:

- X - a sequence of inputs $x_1, x_2..x_n$
- Y - a sequence of outputs $y_1, y_2..y_n$. They are calculated in the same way as a regular feedforward network. For instance, outputting binary examples would be calculated as follows - $y_i = \text{sigmoid}(V * s_i)$.
- s_i - is the hidden state of the network at input x_i. It is calculated based on the previous hidden state and the input at the current step: $s_t = f(U * x_i + W * s_{i-1})$ where f is an activation function such as ReLU or Sigmoid. The state s_{-1}, which is required to calculate the first hidden state, is typically initialized to zero.

- U, V, W - an RNN shares the same parameters (U, V, W) throughout all steps since it is essentially performing the same task but with different inputs. This keeps the RNN efficient by lowering the number of parameters.

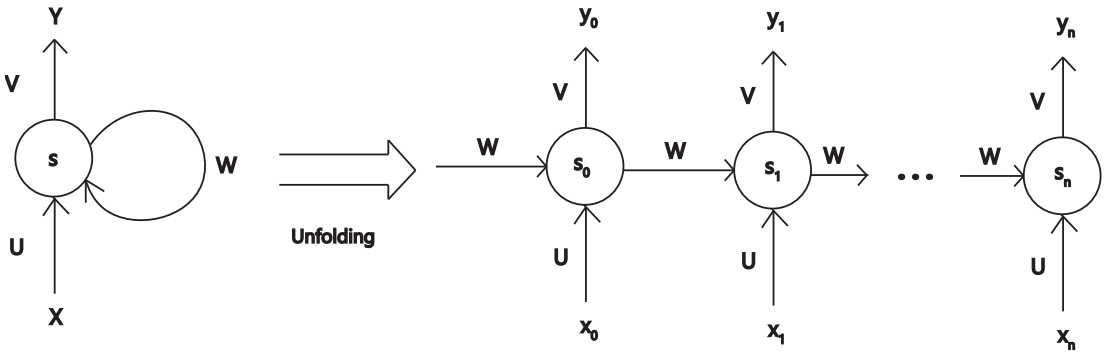

Figure 2.8: Unfolding in time for a recurrent layer.

Its worth mentioning that depending on the task at hand, the recurrent layer could have a different number of inputs and outputs. The important part of the RNN is its hidden state and how it interacts with the previous and next time steps of the network. As shown in Figure 2.9, we can have various RNN structures such as - one to one, one to many (Text Generation), many to one (Parity Function), many to many (Machine Translation).

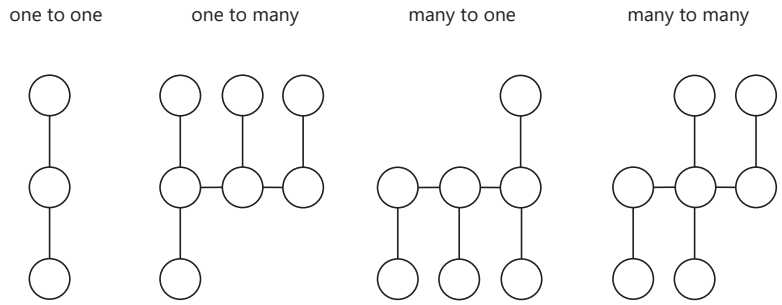

Figure 2.9: Types of recurrent architectures.

Training an RNN is similar to training a feed-forward network. In this case, the algorithm is called Backpropagation Through Time (BTT). Due to the fact that outputs at any time step depend on states of previous time steps, we first need to unfold the RNN as shown in Figure 2.8. Once unfolded, we can treat the network as a long feed-forward network and calculate the gradients and updates as usual. For example, to calculate the gradients for the output y_3 we would need to unfold the network, backpropagate 2 steps, sum up the gradients of W and U and then update them using gradient descent.

Unfortunately, due to the increased number of iterative multiplications caused by the unfolding of the network, two major problems arise - vanishing and exploding gradients. In the exploding gradients problem, the weight updates become too big due to the repetitive multiplication of many gradients that are bigger than 1.00. Fortunately, there is an easy fix for that problem - we can truncate the gradients to a specific value, such that they cannot get too large.

The vanishing gradients problem, on the other hand, occurs when gradients become too small and the weights updates are minimal. It can be partially mitigated by using non-linearities such as ReLU and carefully selecting the appropriate weight initialization technique. This problem considerably affects RNNs since they tend to become less affected by previous inputs, essentially turning into a feed-forward neural network.

2.3.1 LSTM

Long Short Term Memory (LSTM) is a type of RNN architecture first introduced in 1997 (**?**). The network architecture is modified such that the vanishing gradient problem is explicitly avoided, whereas the training algorithm is left unchanged. In particular, when values are back-propagated from the output, derivatives are able to flow arbitrarily far back without decaying significantly. This allows the model to learn longer dependencies than a traditional RNN.

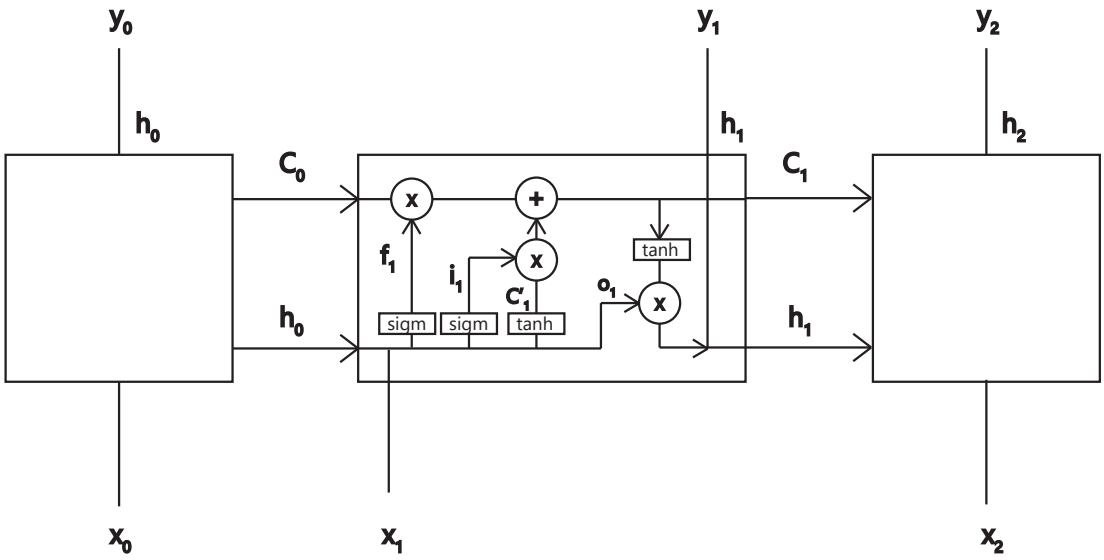

Figure 2.10: Underlying workings of an LSTM layer.

With a standard RNN we have the hidden state defined as $s_t = f(U * x_i + W * s_{i-1})$. The way LSTMs define this hidden state is slightly different. A new cell state C_i is introduced which allows the LSTM to remove or add information to its "memory", by using carefully regulated structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation. Moreover, they output a number between 0 and 1, essentially describing how much information the layer should let through to the cell state. Instead of using only one weight matrix W, now we have 4 (W_f, W_i, W_g, W_o), one for each of the gates.

A summary of the inner workings of the LSTM and how each component is calculated is given below. The $\odot$ operation is the hadamard product (entrywise product) of two matrices.

- **Forget Gate Layer:** decides which components are erased or kept on the cell state received from the previous time step. It outputs a number between 0 ("completely erase") and 1 ("completely keep") for each wight in the cell state:

$$f_t = sigmoid(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- **Input Gate Layer:** controls how much of the current input x_t and the previous output h_{t-1} will enter into the new cell:

$$i_t = sigmoid(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- **Cell Update Transformation Layer:** creates candidate values that could potentially be added to the memory cell.

$$g_t = \tanh(W_g \cdot [h_{t-1}, x_t] + b_g)$$

- **Cell State Update:** computes the next cell state using the previous cell state, the transformation gate and the gated input:

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

- **Output Gate:** scales the output from the cell:

$$o_t = \sigma(W_0 \cdot [h_{t-1}, x_t] + b_o)$$

- **Final LSTM Output:** output of the LSTM scaled by a tanh transformation of the current state:

$$h_t = o_t \odot \tanh(c_t)$$

2.4 Activation Functions

As described in the introduction of this chapter, activation functions are applied to a neuron after weights and inputs (or outputs from previous layers) are multiplied and the bias is added - $f(w * x + b)$. The choice of an appropriate activation function for a specific problem is a difficult problem and usually involves experimentation. In this section, I am going to give a brief overview of the most widely used activation functions and outline their strengths and weaknesses.

Linear

The linear activation function simply returns the number it receives. In other words, the input passes through to the output without any modification. This activation function is commonly used at the output of linear regression networks as the output of such networks has to be a real number. Its derivative with respect to x is 1 and the gradient

$$linear(x) = x$$

$$linear'(x) = 1$$

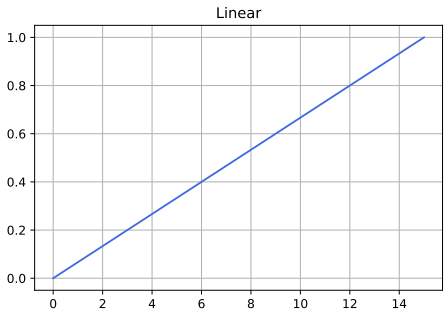

Figure 2.11: Linear function.

has no relationship with x, therefore it does not make any sense to use it at the hidden layers.

Sigmoid

The sigmoid activation function looks like the step function in the perceptron but can also express a probability (or confidence) between 0 and 1. It can be used at the output layer of a network predicting binary or probabilistic values.

It has also been frequently used at the hidden layer but there are 3 main problems with it:

1. Vanishing gradients - during backpropagation through the network with sigmoid activation, the gradients in neurons whose output is near 0 or 1 are nearly 0, thus the weights in these neurons update too slowly.
2. Not zero centered - the output of the function is always positive.
3. Computationally expensive - the exp() operation is difficult to compute compared to other activation functions.

$$sigm(x) = \frac{1}{1+e^{-x}}$$

$$sigm'(x) = sigm(x)(1-sigm(x))$$

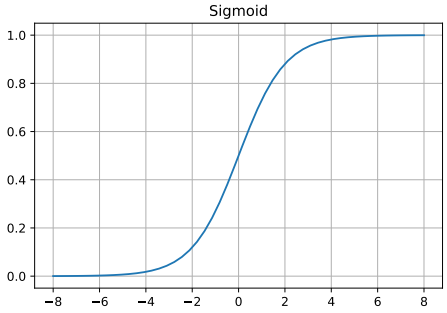

Figure 2.12: Sigmoid function.

Tanh

The tanh (hyperbolic tangent) activation function is similar to a Sigmoid. It exhibits the same properties but has one major advantage when used at the hidden layers. It is zero-centered and the output ranges between -1 and 1 as shown in Figure 2.13.

$$tanh(x) = 2sigm(2x) - 1$$

$$tanh'(x) = 1 - tanh^2(x)$$

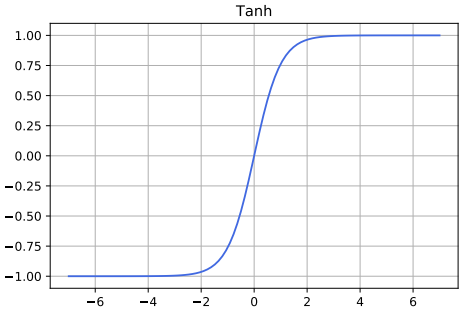

Figure 2.13: Hyperbolic tangent function.

ReLU (Rectified Linear Unit)

ReLU (Nair & Hinton, 2010) has many advantages that have made it one of the most widely used activation function, especially at the hidden layer. It reduces the likelihood of the gradient vanishing as it has a constant value for $x > 0$. Another benefit is that the gradient computation is fairly simple which can speed up the training of the network. Most importantly, empirical results show consistently better network performance when using ReLU compared to other activation functions.

Unfortunately, ReLU units can become problematic during training. Sometimes large gradients flowing throw them cause the weights update to happen in such a way that they never get activated again. This could lead to the gradient irreversibly becoming zero. This is known as the ”dying ReLU problem”

$$relu(x) = \max(0, x)$$

$$relu'(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}$$

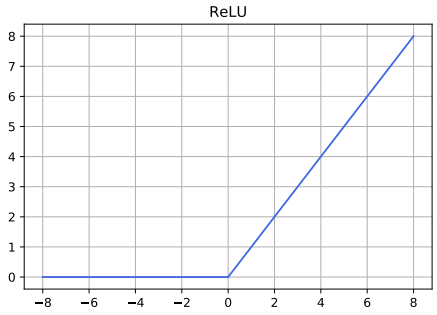

Figure 2.14: ReLU function.

SELU (Scaled Exponential Linear Unit)

The SELU (Klambauer et al., 2017) activation function is one of the derivatives of ReLU designed to overcome the ”dying ReLU” problem. It induces self-normalizing properties which allow the training of deeper networks with many layers. It also employs strong regularization schemes, thus making learning highly robust. Furthermore, for activations not close to unit variance, there exist an upper and lower bound on the variance, making vanishing and exploding gradients impossible.

$$\lambda = 1.0507 \text{ and } \alpha = 1.67326$$

$$selu(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$$

$$selu'(\alpha, x) = \lambda \begin{cases} selu(\alpha, x) + \alpha & x \leq 0 \\ 1 & x > 0 \end{cases}$$

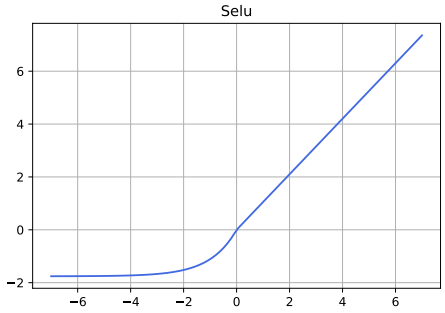

Figure 2.15: SELU function.

Softsign

The Softsign activation function is another nonlinearity which can be considered an alternative to tanh. It is significantly easier to compute than tanh and yielded good experimental results on the parity function problem.

$$softsign(x) = \frac{x}{1 + |x|}$$

$$softsign'(x) = \frac{1}{(1 + |x|)^2}$$

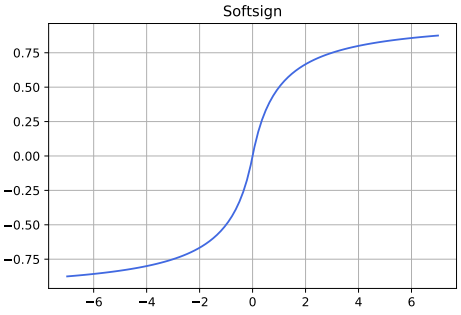

Figure 2.16: Softsign function.

2.5 Weight and bias initialization

Initializing the weights of a neural network can have profound effects both on the convergence rate and final quality of the network. There are many types of weight initialization techniques available. Listed below are some of the most popular ones that I have used in this report. Biases are typically initialized to 0.

Constant

The easiest way to initialize the weights of the network is to set them to a constant value. For instance, we could set them to 0. This could lead to many problems since every neuron in the network would output the same value and during backpropagation undergo the exact same parameter updates. In other words, asymmetry is desired for more efficient learning.

Random Normal or Uniform

In order to avoid the asymmetry problem, we can draw values for each weight at random from a normal or uniform distribution. This will allow the network to learn better since each weight will compute distinct updates. The difference between using normal or uniform random distributions seems to be negligible. Typically, relatively small, zero-centered random weights are preferred.

Lecun Normal or Uniform

One problem with the above method is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. (LeCun et al., 1998) suggest that we can normalize the variance of each neuron's output by scaling its input weights by the square root of its number of inputs (n_{in}). The paper suggests using:

$w_i \sim U(-\sqrt{3/n_{in}}, \sqrt{3/n_{in}})$

where U is the uniform distribution. A normal distribution can be used as well.

Glorot (Xavier) Normal or Uniform

Glorot initialization (Glorot & Bengio, 2010) is very similar to Lecun but it also takes into account the backpropagated signal, constraining the estimated variance of a unit's gradient to be independent of the number of outgoing connections (n_{out}). Taking both consideration into account, the paper suggests weight initialization as follows:

$w_i \sim U(-\sqrt{6/(n_{in}+n_{out})}, \sqrt{6/(n_{in}+n_{out})})$

Orthogonal

(Saxe et al., 2013) suggests that the weight matrix should be chosen as a random orthogonal matrix (a square matrix W for which $W^T W = I$). There are two properties of orthogonal matrices that aid learning:

- Norm preserving - at least at the beginning of training the initialization will keep the norm of the input constant, thus helping with the problem of exploding and vanishing gradients.
- Columns and rows are orthonormal - encourages weights to learn different input features

Batch Normalization

Batch Normalization (Ioffe & Szegedy, 2015) is a recently developed technique that minimizes the need for appropriate weight initialization. Normalizing examples before training is a common pre-processing step to make data comparable across features. During training, weights and parameters adjust those values, sometimes making the data too big or too small again - a problem the authors of the algorithm refer to as "internal covariate shift". Batch normalization applies normalization at every layer before the activation function is applied.

If u_i is the multiplication between the weights and inputs (or outputs from previous layers), then $u_i = w_i \cdot x$ and

$$\hat{u}_i = \frac{u_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

where μ_i and σ_i^2 are the mean and variance of each hidden unit activation across the minibatch of size M:

$$\mu_i \leftarrow \frac{1}{M}\sum_{m=1}^{M} u_i^m$$
$$\sigma_i^2 \leftarrow \frac{1}{M}\sum_{m=1}^{M} (u_i^m - \mu_i)^2$$

The final output form a hidden unit then becomes:

$$h_i = \text{activation\_function}(\gamma_i \cdot \hat{u}_i + \beta_i)$$

where γ and β are learnable parameters and β takes the role of a bias. This layer is differentiable and there is no problem including it in the network.

Batch Normalization helps the network train faster, makes initialization of weights easier, allows for training of deeper networks and provides a form of regularization. Moreover, it also permits the use of higher learning rates. In practice batch normalization has actually shown better performance when applied after the activation function.

2.6 Optimizers

As described in Section 2.1 the most common way to find the right weights and minimize a loss function ($J(\theta)$) of a neural network is by updating them in the opposite direction of the gradient of that function $\theta = \theta - \eta \cdot \nabla J(\theta)$ with a factor of η (learning rate). There are a few challenges with using this simple formula:

- The learning rate is very difficult to choose. If it is too small, the network will converge very slowly and if it is too large, it is going to fluctuate around the minimum and never quite get there.
- The same learning rate applies to all parameters. We might want to update certain parameters with larger rate than others.
- It is easy to get stuck in local minima and very difficult to escape one.

Momentum

Momentum is a method that helps accelerate gradient descent in the relevant direction and reduces oscillations while learning. It does this by adding a fraction γ of the previous update vector to the current one:

$$v_t = \gamma \cdot v_{t-1} + \eta \cdot \nabla J(\theta)$$
$$\theta = \theta - v_t$$

Adagrad

Adagrad (Duchi et al., 2010) is another optimization algorithm that adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. This makes it suitable for dealing with sparse data. Since we are updating each parameter separately, we can set we $g_{t,i}$ to be the gradient of the objective function w.r.t. the parameter θ_i at time step t:

$$g_{t,i} = \nabla_\theta J(\theta_{t,i})$$

Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \varepsilon}} \cdot g_{t,i}$$

where $G_t \in \mathbb{R}^{d \times d}$ is a diagonal matrix where each diagonal element i,i is the sum of the squares of the gradients w.r.t. θ_i up to time step t and ε is a small value that avoids division by zero. The main advantage of Adagrad is that its adaptive and eliminates the need to manually choose the learning rate. Its main weakness is that the learning rate is monotonically decreasing.

Adadelta

Adadelta (Zeiler, 2012) is an extension of the Adagrad algorithm which tends to remove the decaying learning rate problem. It limits the number of past gradients to a fixed size **w** and instead of storing all previous gradients it as a defines the sum of the gradients as a decaying average of all past squared gradients. The running average $E[g^2]_t$ is then defined as follows:

$$E[g^2]_t = \gamma \cdot E[g^2]_{t-1} + (1-\gamma) \cdot g_t^2$$

where γ is usually set to 0.9. For clarity we can rewrite the update formula to:

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

We can replace Adagrad's diagonal matrix $G_{t,ii}$ with $E[g^2]_t$:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} \cdot g_t$$

The authors further optimize the algorithm by making sure the update has the same hypothetical units as the parameter. To realize this, they define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

$$E[\Delta\theta^2]_t = \gamma \cdot E[\Delta\theta^2]_{t-1} + (1-\gamma) \cdot \Delta\theta_t^2$$

Therefore the final Adadelta update rule becomes:

$$\Delta\theta_t = -\frac{\sqrt{\mathrm{E}[\Delta\theta^2]_{t-1} + \varepsilon}}{\sqrt{\mathrm{E}[g^2]_t + \varepsilon}} \cdot g_t$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Adadelta, therefore, removes the need to set an initial learning rate.

RMSProp

RMSProp is an optimization algorithm very similar to Adadelta but independently developed during the same time. It is identical to the first idea of Adadelta:

$$E[g^2]_t = 0.9 \cdot E[g^2]_{t-1} + 0.1 \cdot g_t^2$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$

RMSProp is often used as the default optimization algorithm for recurrent neural networks.

Adam

Adam (Adaptive Moment Estimation) is an optimization algorithm that combines RMSProp and Momentum. It stores an exponentially decaying average of past squared gradients v_t similar to RMSProp and an exponentially decaying average of past gradients m_t, similar to Momentum.

$$\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \\ v_t &= \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2 \end{aligned} \tag{2.2}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively. They are initialized as vectors of

0s, thus are biased towards zero, especially during the initial time steps and when the decay rates are small (i.e. β_1 and β_2 are close to 1). The authors counteract these biases by computing bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.3}$$

Yielding the final Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t$$

The authors suggest values of 0.9 for β_1 and 0.999 for β_2. Adam achieves impressive empirical results and is the default optimization algorithm for many deep learning libraries.

2.7 Regularization

Overfitting is a problem in machine learning which occurs when the model learns the training data too well and cannot generalize to new, unseen data. In a sense, the model only remembers the examples from the dataset, rather than understanding the underlying function. Underfitting occurs when the model is too general but it can be easily solved in neural networks by introducing more neurons or layers to the model.

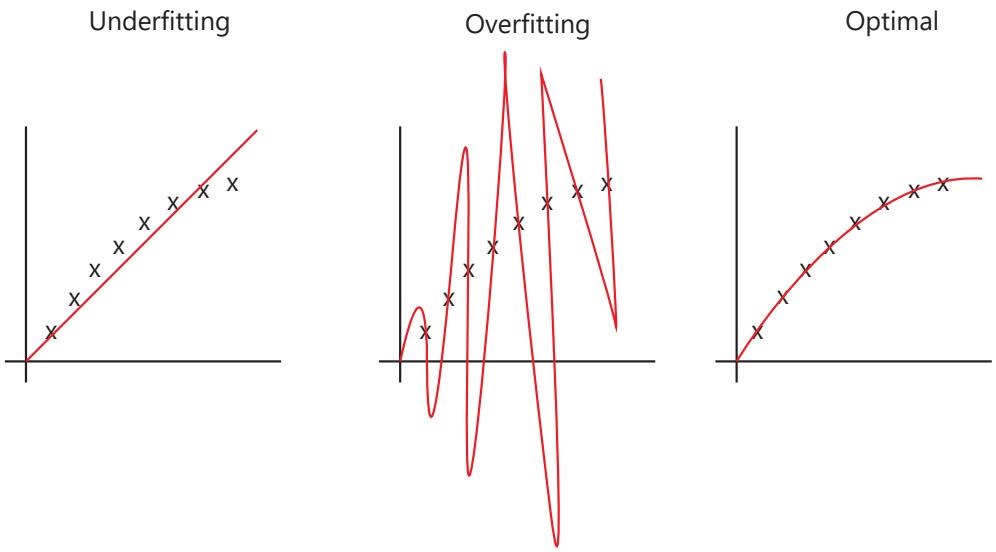

Figure 2.17: Overfitted and underfitted networks compared with an optimal solution.

There are several ways of controlling the capacity of a neural network:

L2

Weight decay is a standard way for regularization used in other machine learning models as well. L2 regularization penalizes the squared magnitude of all weights directly in the loss function. More specifically, for every weight w_i in the network, we add the term $\frac{1}{2}\lambda w^2$ to the objective, where λ is the regularization strength. The factor $\frac{1}{2}$ is added because then the gradient of this term with respect to the parameter w_i is simply λw_i.

L2 regularization works like a "spring" on the weights. If training data puts a consistent force on a weight, it will outweigh weight decay. Otherwise, the weight will decay to 0. This has the appealing property of encouraging the network to use all of its inputs a little, rather than some of its inputs a lot.

L1

L1 regularization works in a similar way to L2 but adds a different term: $\lambda \mid w \mid$ to the loss function. It tends to shrink some weights to 0, leaving a few large important connections, thus encouraging sparsity. It is possible to combine the L1 and L2 regularization, known as elastic net regularization.

Dropout layer

Dropout (Srivastava et al., 2014) is a recently developed regularization technique, specifically designed for neural networks. At each training stage, individual nodes are either deactivated with probability 1-p or kept with probability p, so that a reduced network is left. Incoming and outgoing edges to a deactivated node are also removed. This technique reduces overfitting by preventing complex co-adaptations on training data because the nodes become more insensitive to the weights of other nodes.

2.8 Previous work

The parity function is arguably one of the most important boolean functions in computer science. It can be used to detect error codes, calculate the last bit of the sum of n integers and is notable for its role in the theoretical investigation of circuit complexity. Therefore, there exists a significant body of literature focusing on the parity function solutions using neural networks. Most of the research done focuses on reducing the number of neurons needed to solve the problem and utilizes threshold circuits, which use an activation function similar to the step function of the perceptron.

The theoretical studies on the topic, investigate and prove the circuit (network) sizes needed in order to solve the parity function problem under different conditions. If input-output connections are allowed and the activation function is a sigmoid, (Minor, 1993) proved that in order to solve the N-bit parity function we would need at least $\lfloor \frac{N}{2} \rfloor$ neurons in the hidden layer. (Fung & Li, 2001), shows that if we are using only connections between adjacent layers and applying the threshold activation function, only N neurons in the hidden layer are sufficient for $N <= 4$.

Many papers also describe practical solutions to the parity function problem. For instance, (Stork & Allen, 1993) uses a specifically crafted activation function to solve the N-bit parity function using only 2 neurons in the hidden layer. Furthermore, while (Hohil et al., 1999) and (Arslanov et al., 2002) criticize the use of specifically crafted activation functions and use the simpler threshold (or step) activation function, they also design overly-specific networks that, for example, allow direct connections from the input layer to the output. There are many other similar studies such as (Franco & Cannas, 2001), (Liu et al., 2002) and (Wilamowski et al., 2003) that describe specific structures which solve the parity function problem with very few neurons.

Research has also been undertaken in investigating the more general issue of solving arithmetic problems using neural networks. (Franco & Cannas, 1998) describes solutions for simple arithmetic problems such as the addition of N numbers, multiplication of two numbers and left and right shifts. The resulting networks are, again, specifically crafted for each problem. More recently, (Graves et al., 2014) and (Kaiser & Sutskever, 2015) define more complex neural network architectures that show a lot of potential on solving arithmetic problems such a addition and multiplication, without the need to tweak them for each problem.

The common pattern in the more practical work on the topic is that it uses feed-forward neural networks and computes the weights of the network manually. Others, prove that certain, overly-specific architectures would work. My goal for this study is to use more general architectures, suitable for any problem and use backpropagation in order to see whether the weights can actually be learned. Furthermore, I explore other recently successful architectures, such as Convolutional Neural Networks and Recurrent Neural Networks. The work by (Setiono, 1997) has the most similar approach to the problem. He uses a sigmoid activation function on a fully connected network with a single hidden layer. He also shows how to calculate the weights by solving a system of linear equations.

Chapter 3

Problem description and setup

The parity function is the most widely used boolean function to investigate learning algorithms due to its relatively simple definition but great complexity. As described in the introduction, the parity function returns 1 if the number of ones in a bitstring is an odd number and 0 otherwise. The difficulty of solving the parity function using neural networks comes from the fact that small changes in the input can cause the output to fluctuate unpredictably. For instance, flipping any of the bits from the input causes the output to flip as well. In order to test the learning and generalization properties of the experimental network architectures, I create datasets with bit lengths of:

- **10** - There are only 1024 (2^{10}) examples possible with this bit length. It is easy to work with this input size since the number of neurons is low and training time is fast. Furthermore, we could try to overfit the network in order to decide whether it is even possible to achieve accuracy above the baseline when using a particular architecture.
- **20** - There are a little over a million (2^{20}) examples with bit size of 20. While it is possible to generate and work with this amount of data, training would be slow and unstable. In my experiments, I generate only around 10% of that data (100,000 examples) and it would be a considerable achievement if a network can learn and generalize the function without being exposed to a large portion of the potential training set.
- **30** - The amount of example possible with bit size of 30 is over 1 billion (2^{30}). I will try to learn the function of this bit length using only 100,000 examples (1% of the total possible dataset). Only very strong models should be able to generalize the parity function of this length, especially with such small number of examples present it this dataset.

3.1 Research questions

I wanted to tackle the parity function problem using a variety of neural network concepts and architectures. I devised a plan with research questions and hypotheses I

wanted to explore. Naturally, many of the ideas were revised and updated throughout the experimentation phase.

I started with shallow network architecture with activation functions and a minimum number of neurons as suggested by literature and tried to determine whether a standard gradient descent optimization algorithm could learn the appropriate weights for the solution. Although certainly possible, I believed that it would be highly unlikely for this simple architecture to learn the parity function.

Literature suggests that using an appropriate activation function is extremely important for learning. I hypothesized that varying the activation function at the hidden layer would allow for better generalization, thus higher validation accuracy. I wanted to examine a number of different options to understand which ones perform better than others in this context.

Minor changes in the input of the parity function can cause drastic changes in the output. I suspected that due to this inherent difficulty of the problem, training would be heavily dependent on the initial conditions of the network and the learning procedure. For this reason, I also create a set of experiments where I try weight initializations, optimizations and batch normalization, in hope to find a stable learning model.

After discovering hyperparameters which allow for stable learning I create a set of architectures with an increased number of neurons and layers. My hypothesis is that both the depth and breadth of the network will help achieve better generalization. Making the network too large might lead to overfitting which can be avoided by using the standard regularization techniques described in Section 2.7.

One dimensional Convolutional Neural Networks have never been used to solve the parity function. I created a set of experiments where I vary the number of filters and their size in order to understand the representational power of the network. I also investigate whether CNNs could be more computationally efficient than fully connected feed-forward networks.

Finally, I hypothesized that Recurrent Neural Networks will perform very well on the parity function problem because of its sequential nature. I proposed two RNN models and tested their performance. It could be argued that one of these networks is crafted specifically for the parity function but in general RNNs sometimes require preprocessing of the data in order to achieve optimal performance.

3.2 Technology

Many of the popular machine learning libraries and frameworks are using Python as their main programming language. Some of them are built specifically for Deep Learning applications and include many of the concepts described in Chapter 2. Tensorflow (Abadi et al., 2015) is such a library, designed for numerical computation using data flow graphs. It is a relatively low-level framework, optimized for distributed computing and extremely flexible. Unfortunately, it is also relatively difficult to use.

For my experiments, I decided to use Keras (Chollet et al., 2015) - a neural network framework built on top of Tensorflow. It presents a higher-level, more intuitive set of abstractions that make it easy to create and configure neural networks. It is perfect for testing and building standard architectures. All of the layers, activation functions, initializers, optimizers and regularizers described in Chapter 2 are build into Keras and I use them out of the box in order to design the network architectures needed for my experiments. Some other essential libraries and tools I have used include:

- Scikit-learn - a machine learning library for Python. It could be used for general preprocessing tasks such as train-test splitting and one-hot encoding. (Pedregosa et al., 2011)
- Pandas - a Python package providing fast, flexible, and expressive data structures designed to make working with data both easy and intuitive (McKinney)
- Numpy - a library for Python which adds support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays. (Walt et al., 2011)
- Github - a version control service used to store the code for my experiments and the resulting data. The repository can be found at `https://github.com/Thinkorswim/nn_arithmetic`.

3.3 Data generation

In order to make the training data required for learning, I created a simple script that produces parity function examples. The generating function takes as an input the amount of examples desired, their bit length and whether repetition of examples is permitted. Every line of the output file contains an example generated at random. They are given in binary form where each bit is separated by a space and the last bit represents the output of the parity function. A sample generated file with bit strings of length 5 with 5 examples and no repetition is shown below.

```
0 1 1 1 1 0
1 1 1 1 0 0
1 1 1 0 0 1
0 1 1 1 0 1
1 1 0 0 0 0
```

3.4 Training, validation and evaluation procedure

I divide my dataset into 80% training and 20% validation portions. During training, I feed the network batches of examples exclusively from the training set. One epoch of training means that the network has gone through all of the examples in the dataset once.

After learning is completed, I check the accuracy of the predictions of the network on the validation set, which is composed of examples the network has never seen before. Some machine learning tasks utilize a third collection of examples called the test set. It is used to validate the network after all of the hyperparameters are tuned so that the final network is not biased towards the validation set. Since I randomize the dataset every time I run experiments and do not allow repeating examples, I decided that a test set is not necessary.

In order to evaluate a particular architecture, I take into account the percentage of accurately predicted training and validation examples. If the percentage difference between the two is too big, this could mean that the network is overfitting. It is worth mentioning that the baseline accuracy for this problem is 50%, since predicting only 0s would guarantee this success rate.

Confusion matrices are square matrices where $c_{i,j}$ is an element of the matrix containing the number of examples which have a true label of i but are classified as a label j. Their diagonal contains the number of correctly predicted examples for each class. This could be useful in further understanding of the learned representations. For example, it could reveal that the network is biased towards predicting 1s and never makes mistakes when predicting a 0.

To further understand how the architectures learn the parity function and in which cases they are prone to making mistakes, I also look into wrongly classified examples. Many such examples may be similar to each other and reveal a common generalizing weakness of the network. Furthermore, we can assess the confidence of the network predictions. In other words, whether they are closer to 0.5 (not confident) or 1 and 0 (confident).

Chapter 4

Preliminary Experiments

From the work of (Setiono, 1997) we know that the N-bit parity problem can be solved using a fully connected feed forward network with a single hidden layer consisting of $(N/2)+1$ units if N is even and $(N+1)/2$ if N is odd. The activation function used in the hidden and output layers is sigmoid. The author gives a proof that this architecture solves the problem and shows how the weights of the network can be obtained by solving a system of linear equations.

My objective for these preliminary experiments was to determine whether I can replicate the described minimal architecture with sigmoid activation function and establish whether gradient descent can learn the appropriate weights for the solution. Furthermore, I wanted to experiment with various hyperparameters to establish a strong model to be used with deeper configurations.

4.1 Architecture

The parity function of two inputs is also known as the XOR function (Figure 4.1). The separating hyperplane of a single layer perceptron is given by the formula $w \cdot x + b = 0$, which is a linear function of input vector x. As shown in Figure 4.2 the positive and negative points cannot be separated by a line, or effectively, there does not exist a line that can separate the two classes.

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

Figure 4.1: The XOR function definition.

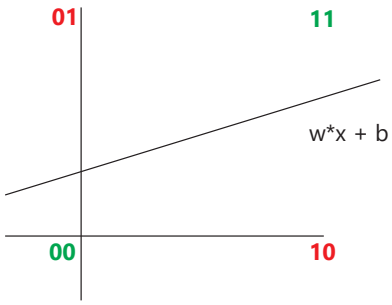

Figure 4.2: Visual representation of XOR.

The inability of perceptrons to learn the XOR function has been a long-established problem. This means that the network architecture needed to solve the parity function would require at least one hidden layer. Figure 4.3 shows an example of a fully-connected architecture where $x_1, x_2 ... x_n$ are the bit inputs and y is the output of the network which can be any value between 0 and 1. If y is above 0.5 the output is considered a 1 else a 0.

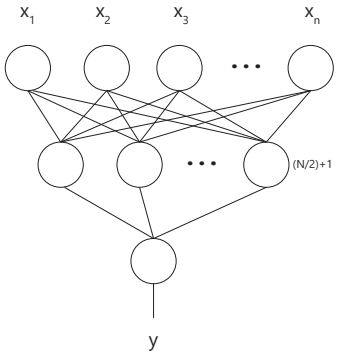

Figure 4.3: Shallow architecture described by (Setiono, 1997).

4.2 Description

For these preliminary experiments, I started with the simple architecture described in the previous section. Unless otherwise stated, I set the loss function to 'binary crossentropy' which is used for binary output, the optimizer to Adam and the kernel initializer to glorot uniform as they are currently the default suggested by the Keras library. I also set the batch size to 10, 100, 100 for experiments with bit lengths of 10, 20 and 30 respectively. The biases are initialized to 0. Figures 4.4 and 4.5 show the exact number of inputs and outputs for the 10 and 20-bit architectures.

InputLayer	input:	(None, 10)
	output:	(None, 10)
Dense	input:	(None, 10)
	output:	(None, 6)
Dense	input:	(None, 6)
	output:	(None, 1)

Figure 4.4: Layer sizes for 10-bit input.

InputLayer	input:	(None, 20)
	output:	(None, 20)
Dense	input:	(None, 20)
	output:	(None, 11)
Dense	input:	(None, 11)
	output:	(None, 1)

Figure 4.5: Layer sizes for 20-bit input.

Sigmoid Experiments

For the first set of experiments, I used the sigmoid activation function. I tested the system with bit lengths of 10 and dataset size of 1024 (819 training, 205 validation), which encompasses all possible examples of this length. Furthermore, I varied the epoch sizes between 500, 1000, 5000 and 10000, in order to test the importance of training time. I ran each experiment 3 times in order to avoid conclusions based on randomness.

Following, the success of the experiments with bit length of 10, I decided to try the same architecture on bit length of 20 with data size of 100,000 (80,000 training and 20,000 validation). I also reduced the epoch sizes to 100, 250 and 500, since the data and batch size is now larger.

The 20-bit parity problem was clearly a more difficult problem and I decided to focus this input size until I have chosen a stable architecture. Unless otherwise stated all of the following experiments in this chapter use the same 20-bit dataset and epoch sizes.

Activation Functions Experiments

In the third set of experiments, I varied the activation function applied to the hidden layer. I tested some of the most widely used activation functions - tanh, relu, selu and softsign. I expected that they will perform better than sigmoid for the reasons given in Section 2.4.

Initialization Experiments

In order to avoid the unstable learning problem and ensure consistency in my experiments, I decided to vary the weight initialization procedure for the two best performing activation functions - selu and softsign. The initialization methods I considered were:

- Random Uniform and Normal
- Lecun Uniform and Normal
- Glorot Uniform and Normal
- Orthogonal

I also set the epoch size to 500 as training with fewer epochs seemed redundant at this point.

Optimization Experiments

For this set of experiments, I chose the best performing weight initializers for each of the activation functions and varied the gradient descent optimization algorithm between:

- Adagrad
- Adadelta
- RMSProp
- Adam

I also tested a network which does not use any gradient descent optimization. My goal in this investigation was to discover which optimizer would perform best for the parity function problem and whether the authors claimed improvements hold true.

Batch Normalization Experiments

For my last set of experiments in this chapter, I chose to test the effectiveness of batch normalization when trying to solve the parity function. For this purpose, I added batch normalization to the following architectures:

- Sigmoid + Glorot Uniform + Adam
- Softsign + Lecun Normal + Adadelta
- SELU + Lecun Uniform + Adadelta

The sigmoid configuration, although possible according to (Setiono, 1997), could not learn the parity function. I wanted to test whether batch normalization will remedy the situation. The other two cases represent the best performing configurations from the previous experiments. I was hoping that batch normalization would increase their accuracy and stability.

4.3 Results

Sigmoid Experiments

From the results in Table 4.1, it is clear that training the network for 500 and 1000 epochs is not sufficient in order to solve the parity function using this architecture and hyperparameters. It is interesting to note that the validation accuracies for these two epoch sizes are significantly lower than the baseline. From the learning graph of the 1000 epochs model in Figure 4.6, we can see that the training accuracy is increasing slightly but the validation one is decreasing rapidly. These results suggest an instance of anti-learning and in fact (Roadknight et al., 2013) gives the XOR function as an example where the idea can be applied. This discovery can be a point of interest for future investigation of the parity function.

EPOCHS	TRAIN ACCURACY	VALIDATION ACCURACY
500	0.531	0.394
1000	0.527	0.345
5000	0.930	0.927
10000	0.904	0.880

Table 4.1: Average training and validation accuracies for 10 bit inputs.

Training for a larger amount of epochs show accuracies of more than 90% on the validation test, suggesting the capability of the network to learn the parity function, at least with bit length up to 10. The difference between the accuracies of 5000 and 10000 epochs are insignificant. In fact, from Figure 4.7 it seems that the network accuracies level off at around epoch 5000 and it stops learning. This could mean that

it is getting stuck in local minima. Nevertheless, the results seemed promising to test the architecture on longer input lengths.

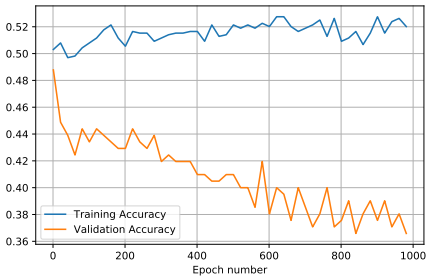

Figure 4.6: Training and Validation accuracies for 1000 epochs with 10 bit inputs.

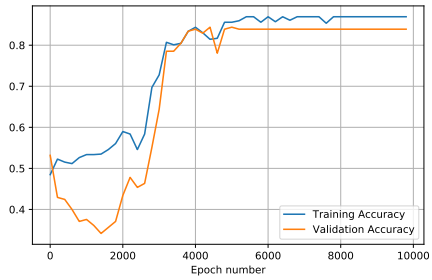

Figure 4.7: Training and Validation accuracies for 10000 epochs with 10 bit inputs.

The network does not seem to be biased towards predicting any of the two possible outputs as illustrated by the confusion matrices in Figure 4.8 and Figure 4.9. Furthermore, I could not perceive any obvious patterns in the wrongly predicted examples.

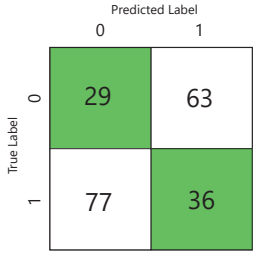

Figure 4.8: Confusion matrix for 10-bit inputs - Trial 1.

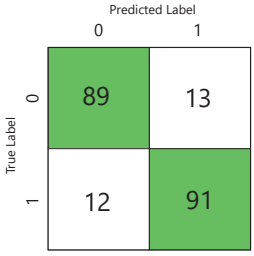

Figure 4.9: Confusion matrix for 10-bit inputs - Trial 2.

Unfortunately, The shallow network using sigmoid activation function could not learn the parity function of bit length 20 as shown in Table 4.2. While the training accuracy increases with the number of epochs, this is probably due to the network memorizing some of the examples. The validation accuracy oscillates around the baseline of 50%. This pattern was prevalent in all of the experiments, regardless of the epoch size. Finally, the network did not exert any significant bias towards any of the classes.

EPOCHS	TRAIN ACCURACY	VALIDATION ACCURACY
100	0.515	0.499
250	0.521	0.495
500	0.525	0.498

Table 4.2: Average training and validation accuracies for 20-bit inputs.

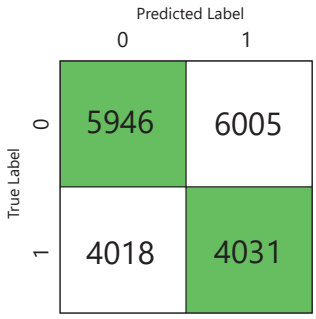

Figure 4.10: Confusion matrix for 10-bit inputs - Trial 1.

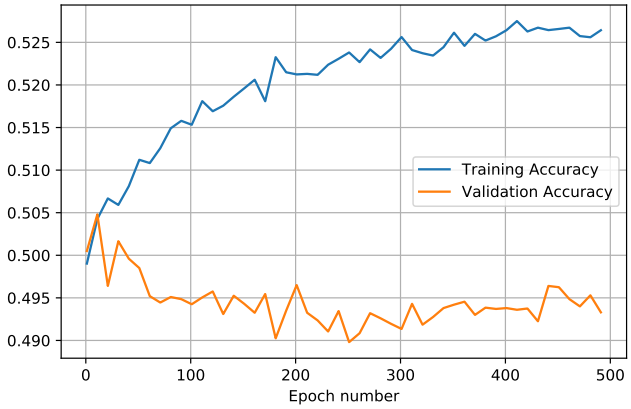

Figure 4.11: Training and Validation accuracies for 500 epochs with 20 bit inputs.

Activation Functions Experiments

- *Tanh:* The hyperbolic tangent activation function is very similar to the sigmoid one. The results were also nearly identical, with tanh having a slight increase in training accuracy over sigmoid as shown in Table 4.3. Perhaps training for more epochs might result in better accuracy, but it is apparent that the current network configurations are not learning the underlying structure of the parity function.

Epochs	Train Accuracy	Validation Accuracy
100	0.521	0.498
250	0.526	0.500
500	0.529	0.501

Table 4.3: Average training and validation accuracies for the tanh activation function.

- *ReLU:* As expected, relu achieved better results than sigmoid and tanh. Table 4.4 shows the 3 attempts at running this architecture. Accuracies are higher than the baseline and the network seems to learn some representation of the function. Unfortunately, learning was very unstable. Sometimes the network achieves accuracies of almost 90% while others barely go above the baseline. Epochs do not seem to have a large influence on learning.

  In this case, I am going to display all my 3 trials of the experiment with 500 epochs. As illustrated in Figure 4.12, learning happens very spontaneously in a step like manner. This inconsistency is prevalent in many of the future experiments and it is influenced by the initial weight parameters and the optimization algorithm.

Epochs	Tr(1)	Val(1)	Tr(2)	Val(2)	Tr(3)	Val(3)	**Avg Tr**	**Avg Val**
100	0.765	0.755	0.532	0.512	0.521	0.548	**0.615**	**0.605**
250	0.722	0.744	0.890	0.880	0.590	0.572	**0.734**	**0.733**
500	0.707	0.716	0.650	0.647	0.720	0.709	**0.692**	**0.691**

Table 4.4: Training (Tr) and validation (Val) accuracies for each ReLU experiment trial.

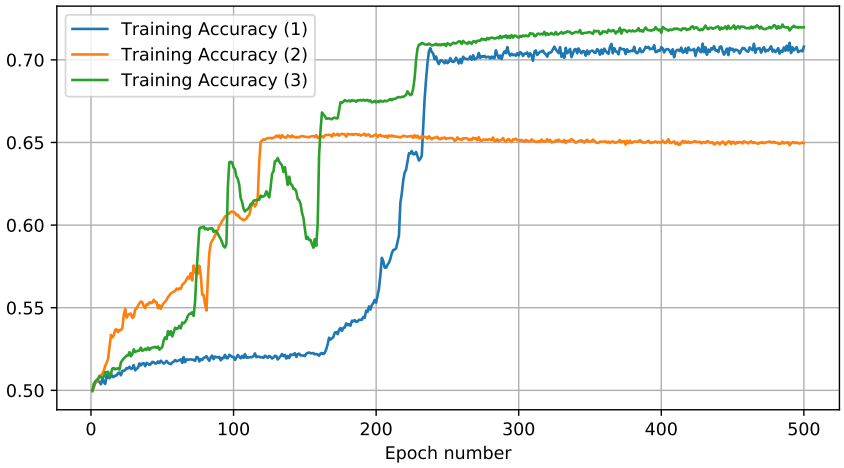

Figure 4.12: Training accuracy for each trial when using a ReLU activation.

- *SELU:* Due to the self-normalizing properties of selu, a more consistent learning is observed. Increasing the number of epochs leads to more accuracy but there exists a difference between each trial when using the same epoch size. This is probably based on the initial conditions of the network. Figure 4.13 displays much smoother learning, although the accuracy seems to drop spontaneously for a few epochs. Nevertheless, as expected, selu is the most promising activation function so far.

EPOCHS	TR(1)	VAL(1)	TR(2)	VAL(2)	TR(3)	VAL(3)	**AVG TR**	**AVG VAL**
100	0.562	0.565	0.617	0.648	0.532	0.520	**0.570**	**0.578**
250	0.795	0.804	0.614	0.614	0.619	0.626	**0.676**	**0.682**
500	0.868	0.886	0.641	0.632	0.759	0.705	**0.756**	**0.741**

Table 4.5: Training (Tr) and validation (Val) accuracies for each SELU experiment trial.

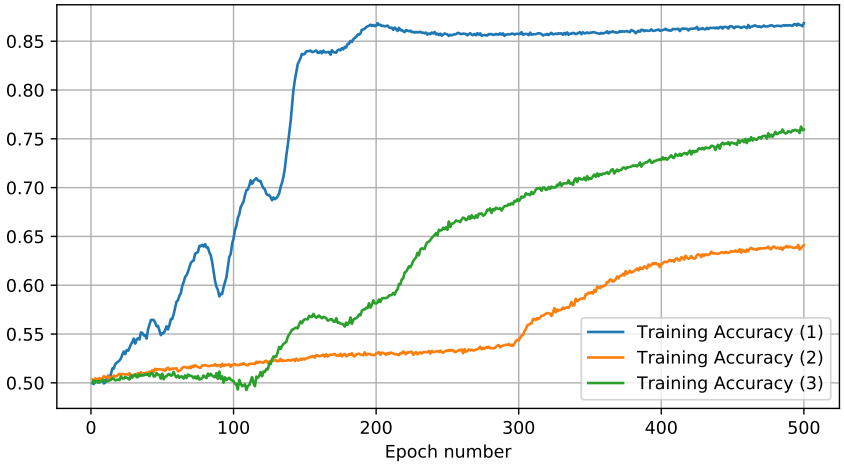

Figure 4.13: Training accuracy for each trial when using a SELU activation.

- *Softsign:* Surprisingly, softsign achieves the highest validation accuracy and is the most stable activation function of all tested. Given enough epoch, it learns the parity function of bit length 20 with an accuracy of up to 98%. There are many ups and downs in accuracy while learning but generally, it is increasing rapidly until it levels off. I believe that expanding the number of neurons in the hidden layer would allow for accuracies closer to 100%.

EPOCHS	TR(1)	VAL(1)	TR(2)	VAL(2)	TR(3)	VAL(3)	AVG TR	AVG VAL
100	0.817	0.815	0.710	0.722	0.523	0.503	**0.683**	**0.680**
250	0.943	0.943	0.956	0.959	0.941	0.938	**0.946**	**0.947**
500	0.980	0.978	0.872	0.874	0.879	0.880	**0.910**	**0.911**

Table 4.6: Training (Tr) and validation (Val) accuracies for each softsign experiment trial.

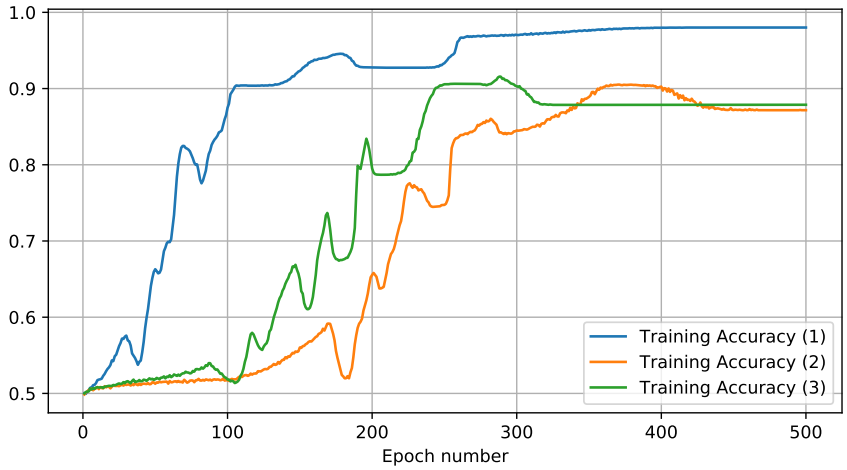

Figure 4.14: Training accuracy for each trial when using a SELU activation.

Initialization Experiments

- *Softsign:* Table 4.7 summarizes the results from this set of experiments. As expected drawing the weights at random from uniform and normal distributions results in poor generalization of the network. The variance scaling performed by LeCun and Glorot initializations resulted in higher accuracy and consistency. The orthogonal initialization also yielded promising results. Unfortunately, sometimes the network failed to learn the function as is the case with the first trial when using Glorot Uniform. These inconsistencies exemplify the importance of weight initialization in solving the parity function problem.

  Figure 4.15 illustrates how the network behaved while learning for each of the initialization techniques on their 3rd trial. The step-like learning behavior continues to appear during learning, regardless of the initialization method used. Overall, the best performing and consistent initializer for the softsign activation function is LeCun Normal, although the differences are insignificant and the slight edge might be due to the randomness introduced while picking the weights from the distributions.

Initializer	Tr(1)	Val(1)	Tr(2)	Val(2)	Tr(3)	Val(3)	**Avg Tr**	**Avg Val**
Random Uniform	0.553	0.539	0.905	0.909	0.528	0.502	**0.662**	**0.650**
Random Normal	0.526	0.498	0.527	0.492	0.565	0.553	**0.540**	**0.515**
Lecun Uniform	0.964	0.965	0.530	0.496	0.921	0.911	**0.805**	**0.791**
Lecun Normal	0.904	0.895	0.938	0.932	0.856	0.846	**0.900**	**0.891**
Glorot Uniform	0.899	0.899	0.704	0.707	0.946	0.937	**0.850**	**0.848**
Glorot Normal	0.543	0.526	0.942	0.943	0.946	0.948	**0.811**	**0.806**
Orthogonal	0.988	0.987	0.891	0.894	0.778	0.770	**0.886**	**0.883**

Table 4.7: Training and validation accuracies for softsign initialization experiments.

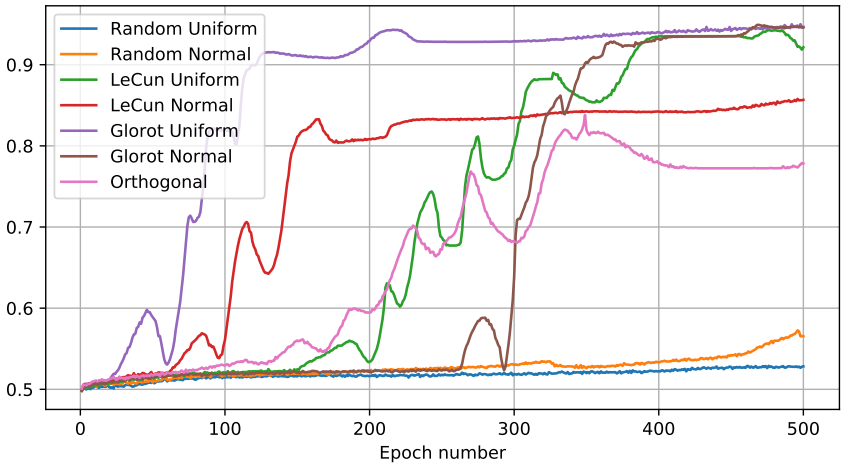

Figure 4.15: Validation accuracy during training for Softsign initialization experiments.

- *SELU:* The results from varying the weight initializers while using SELU as an activation function are presented in Table 4.8. Overall, SELU seems to achieve lower and more inconsistent results than softsign. Similarly, Lecun seems to be the best performing method for weight initialization, but in this case, the uniform distribution performs better.

Initializer	Tr(1)	Val(1)	Tr(2)	Val(2)	Tr(3)	Val(3)	**Avg Tr**	**Avg Val**
Random Uniform	0.505	0.498	0.504	0.498	0.505	0.498	**0.504**	**0.498**
Random Normal	0.510	0.501	0.510	0.495	0.506	0.500	**0.509**	**0.499**
Lecun Uniform	0.906	0.906	0.868	0.860	0.722	0.690	**0.832**	**0.819**
Lecun Normal	0.816	0.821	0.732	0.728	0.539	0.534	**0.696**	**0.695**
Glorot Uniform	0.877	0.902	0.787	0.798	0.640	0.651	**0.768**	**0.784**
Glorot Normal	0.930	0.921	0.920	0.920	0.526	0.505	**0.792**	**0.782**
Orthogonal	0.816	0.819	0.598	0.609	0.515	0.496	**0.643**	**0.642**

Table 4.8: Training and validation accuracies for SELU initialization experiments.

As discussed during the activation function experiments, the architectures using selu tend to learn a lot smoother than other activation functions. Nevertheless, there are still some sudden jumps in accuracy, which perhaps can be mitigated by changing the gradient descent optimization algorithm.

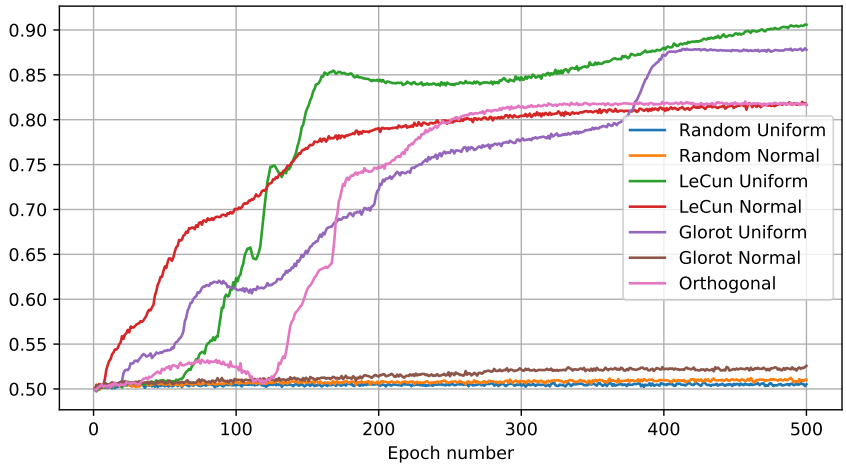

Figure 4.16: Validation accuracy during training for each SELU initialization experiment.

Optimization Experiments

- *Softsign with Lecun Normal:* The results presented in Table 4.10 illustrate the importance of the optimizer used during the learning of the N-bit parity function. This particular architecture achieves best results when using Adadelta and Adam, which is expected as they are more developed than the rest of the optimizers. In particular, Adadelta performs very well on each of its trials. Unfortunately, Figure 4.17 shows that the optimizers have not solved the step-like learning behavior and the network only starts learning at around epoch 100.

Optimizer	Tr(1)	Val(1)	Tr(2)	Val(2)	Tr(3)	Val(3)	**Avg Tr**	**Avg Val**
None	0.525	0.4952	0.980	0.980	0.979	0.979	**0.828**	**0.818**
Adagrad	0.513	0.499	0.515	0.495	0.516	0.501	**0.515**	**0.498**
RMSProp	0.537	0.514	0.940	0.932	0.849	0.854	**0.776**	**0.767**
Adadelta	0.945	0.939	0.954	0.977	0.882	0.882	**0.927**	**0.933**
Adam	0.943	0.944	0.525	0.499	0.880	0.882	**0.782**	**0.775**

Table 4.9: Training and validation accuracies for softsign optimizer experiments.

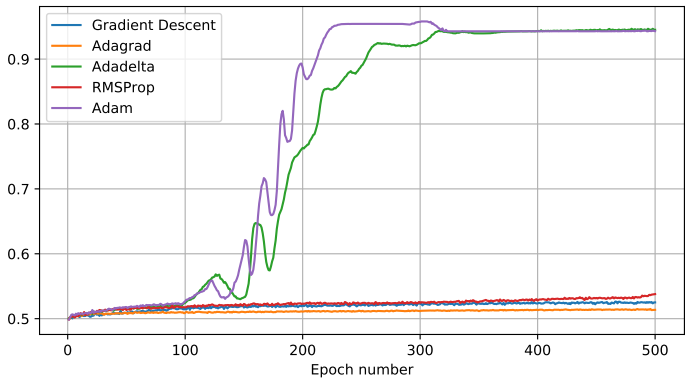

Figure 4.17: Validation accuracy during training for Softsign optimization experiments.

- *SELU with Lecun Uniform:* Although the overall accuracy of this architecture is lower than the softsign one, learning seems to be a lot more stable. In particular, when using Adadelta, learning starts immediately and the accuracy increases steadily. Form the confusion matrices for the Adadelta experiments shown in Figures 4.19 and 4.20 we can see that the network tends to achieve higher accuracy for only one of the classes. This configuration could be a key to stable learning of the parity function.

OPTIMIZER	TR(1)	VAL(1)	TR(2)	VAL(2)	TR(3)	VAL(3)	**AVG TR**	**AVG VAL**
NONE	0.520	0.509	0.579	0.569	0.613	0.632	**0.571**	**0.570**
ADAGRAD	0.510	0.498	0.513	0.504	0.551	0.537	**0.525**	**0.513**
RMSPROP	0.735	0.755	0.899	0.898	0.790	0.811	**0.808**	**0.821**
ADADELTA	0.920	0.919	0.878	0.889	0.567	0.551	**0.788**	**0.786**
ADAM	0.862	0.851	0.937	0.933	0.870	0.875	**0.890**	**0.887**

Table 4.10: Training and validation accuracies for SELU optimizer experiments.

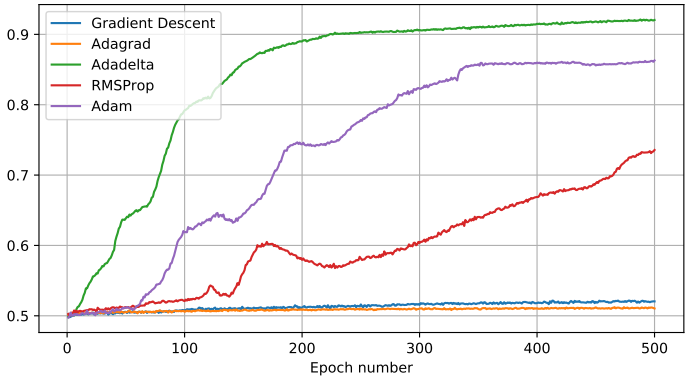

Figure 4.18: Validation accuracy during training for SELU optimization experiments.

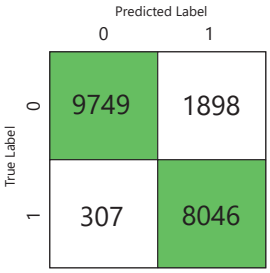

Figure 4.19: Confusion matrix for SELU with Adadelta - Trial 1.

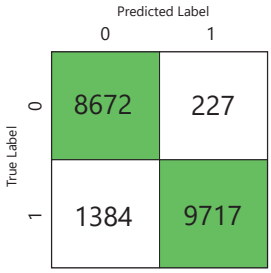

Figure 4.20: Confusion matrix for SELU with Adadelta - Trial 2.

Batch Normalization Experiments

The results from this set of experiments were slightly disappointing. As shown in Table 4.11, the sigmoid configuration could not learn the parity function and the other architectures did not benefit from an increase in accuracy or stability. Furthermore, learning does not appear to be any smoother than previous experiments as illustrated

in Figure 4.21. Batch Normalization is most effective when used in deep networks with many layers. That might be one of the reasons it did not improve these shallow architectures.

ARCHITECUTRE	TR(1)	VAL(1)	TR(2)	VAL(2)	TR(3)	VAL(3)	AVG TR	AVG VAL
SIGMOID	0.527	0.492	0.528	0.499	0.524	0.494	**0.526**	**0.495**
SOFTSIGN	0.943	0.925	0.776	0.773	0.867	0.870	**0.862**	**0.856**
SELU	0.939	0.951	0.619	0.632	0.884	0.893	**0.814**	**0.825**

Table 4.11: Training (Tr) and validation (Val) accuracies for each Batch Normalization experiment trial.

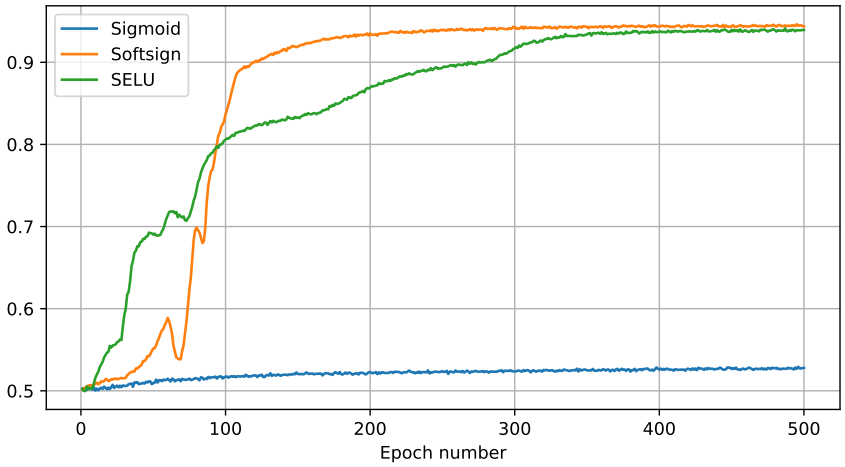

Figure 4.21: Validation accuracy during training for Batch Normalization experiments.

4.4 Discussion

In this chapter, I created a set of preliminary experiments in order to understand which neural network hyperparameters work best when solving the parity function. I used a shallow architecture with few neurons and varied the activation function, weight initialization method and the gradient descent optimization algorithm.

The network with Sigmoid activation had no problem learning the 10-bit parity function given enough epochs, but could not successfully do that when using 20 bit inputs. It is possible that increasing the number of epochs might lead to better performance, but there is certainly an instability in learning.

Some activation functions performed better than others and showed the potential to learn the 20-bit parity function. In particular, Softsign and SELU had the best validation accuracy and the most stable performance. This could be due to their advantages described in Section 2.4 but in general, it is a very difficult problem to understand why a specific activation function perform better for a certain problem.

The next set of experiments illustrated the importance of initialization while solving this problem. Variance scaling methods such as Lecun and Glorot performed best but Orthogonal initialization also showed promising results. It is unclear whether there is a significant difference between using Uniform or Normal distribution. Interestingly, the best performing initializer for this problem seems to be Lecun, although there was not a notable distinction compared to Glorot.

The more sophisticated gradient descent optimization algorithms such as Adam and Adadelta achieved significantly better results than others. In particular, Adadelta had the most stable learning, suggesting that the momentum added to Adam might be degrading the performance of the network. Batch normalization did not show any improvement in performance but it is best applied to deeper networks so I will keep it for further experimentation.

Overall, SELU using Lecun Uniform initialization and Adadelta optimizer achieved the most stable performance. Its worth mentioning that even when running 3 trials for each of the experiments, many of my decisions might have been based on randomness. Nevertheless, the results are optimistic and I will use this configuration for further experiments.

Interestingly, overfitting did not seem to be a problem as the training and validation accuracies were always close to each other. In fact, most of the times, validation accuracy had slightly better performance. This rendered the use of any regularization in these experiments unnecessary.

Chapter 5

Deep Fully-Connected Network Experiments

Deep architectures which utilize many stacked layers have been very successful in solving certain problems. For instance, Microsoft used a network of 152 layers to win the ILSVRC 2015 image classification challenge (He et al., 2015). While this is an extreme example, in this chapter I wanted to explore whether the depth of the network matters when solving the parity function problem.

In all the previous experiments, I held the number of neurons constant with sizes that are proven to work when using sigmoid activation by (Setiono, 1997). In this set of experiments, I will also vary the number of neurons at the hidden layer in order to study its effects on learning. Increasing either the depth or the width of a network should potentially improve performance in learning but in some cases might also lead to overfitting.

5.1 Architecture

The architecture for this set of experiments is very similar to the preliminary one. The only difference being the number of hidden layers and the number of neurons at each of these layers.

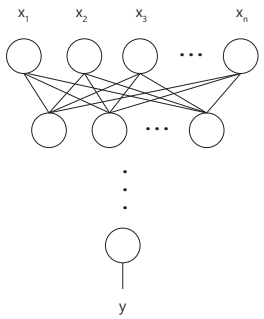

Figure 5.1: Network structure used for experiments in this chapter.

5.2 Description

For this set of experiments, I used the best performing configuration from the preliminary investigation. That was a network using SELU as an activation function at the hidden layer, batch normalization, Lecun Uniform for weight initialization and Adadelta as a gradient descent optimizer. Unless otherwise stated, I use a 'binary crossentropy' loss function, batch size of 100 and epoch sizes of 1000, 100 and 100 for bit lengths of 10, 20 and 30 respectively.

Depth Experiment

In these experiments I varied the depth of the network, starting from one hidden layer and increasing it up to 5 hidden layers. The number of neurons at each layer was held constant at (N/2) + 1 where N is the number of inputs. I ran this experiments for bit lengths of 10, 20 and 30.

Layer Size Experiment

In this set of experiments, I used only one hidden layer but varied its size (number of neurons). For N-bit inputs, the number of neurons I tested was $(N/2)+1, N, 2N, 3N, 4N$ and $5N$. I also ran these experiments for bit lengths of size 10, 20 and 30.

Experiments with bit length of 30

The results for bit lengths of 30 from the previous experiments in this chapter were discouraging, as they could not achieve validation accuracy above the baseline. For this set of experiments, I combined both depth and layer size to understand what kind of architecture would best learn longer bit sequences and whether it is even possible. Moreover, I increased the number of epochs to 1000 for all of the experiments.

5.3 Results

Depth Experiment

From the summary of the results presented in Table 5.1, we can conclude that there is a positive correlation between the number of layers in the network and the achieved accuracy on the parity function problem at least for bit lengths of 10 and 20. Unfortunately, this configuration could not learn the 30-bit parity function. It is possible that the number of epochs or neurons in the hidden layer was not enough. Further experimentation with that bit length is provided in this chapter.

Figure 5.2 illustrates the performance of each neural network depth during learning. We can see that the network with 2 hidden layers is achieving the best stability in learning as the validation accuracy increases steadily from the first epoch. Having more layers might increase the final validation accuracy but from the graph, we can also see that learning starts at a later epoch with each subsequently added layer.

LAYERS	TR(10)	VAL(10)	TR(20)	VAL(20)	TR(30)	VAL(30)
1	0.619	0.424	0.511	0.503	0.525	0.498
2	0.752	0.756	0.947	0.983	0.537	0.494
3	0.896	0.907	0.919	0.988	0.541	0.541
4	0.953	0.985	0.812	0.952	0.547	0.499
5	0.970	0.967	0.887	0.985	0.548	0.501

Table 5.1: Training (Tr) and validation (Val) accuracies each depth and input size.

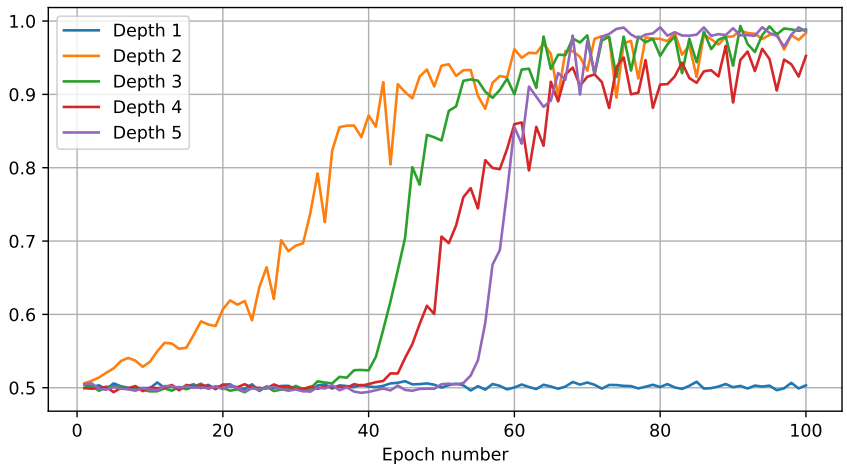

Figure 5.2: Validation accuracy during training for depth experiments.

Layer Size Experiment

These experiments yielded results very similar to the depth variation ones. There is a positive correlation between the layer size and the validation accuracy of the network but this architecture could not learn the 30-bit parity function as shown in Table 5.2.

LAYER SIZE	TR(10)	VAL(10)	TR(20)	VAL(20)	TR(30)	VAL(30)
N/2+1	0.650	0.560	0.639	0.645	0.525	0.501
N	0.610	0.512	0.764	0.785	0.534	0.500
2N	0.709	0.585	0.817	0.865	0.541	0.501
3N	0.629	0.753	0.928	0.957	0.556	0.503
4N	0.697	0.841	0.954	0.965	0.560	0.495
5N	0.692	0.866	0.955	0.971	0.566	0.496

Table 5.2: Training (Tr) and validation (Val) accuracies for each layer and input size.

Increasing the number of layers also increases the rate at which the network is learning. There is a noticeable increase in network learning speed between layer sizes of 2N and 3N. This is apparent in both bit lengths of 10 and 20. These results suggest that using layer sizes 3 times larger than the input might provide a good trade-off between efficiency and accuracy.

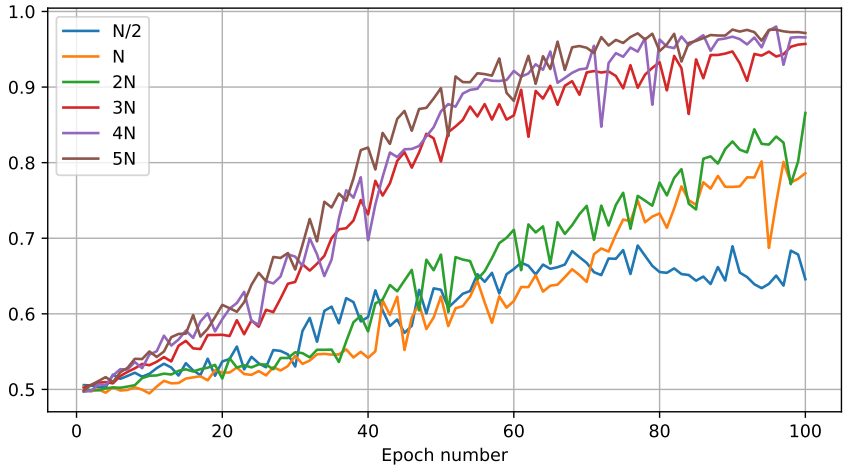

Figure 5.3: Validation accuracy during training for layer size experiments.

Experiments with bit length of 30

The results from these experiments are presented in Table 5.3. Although most of the configurations could not learn the 30-bit parity function, there were a couple that managed to achieve a reasonable accuracy. The best performing configuration was using Depth 2, which was also best in the depth experiments. Moreover, it also had neurons equal to three times the input, which yielded the best improvement in the layer size experiments. I believe that running a network of depth 2 and layer size of 3N for enough epochs would be able to consistently learn the parity function of bit length 30.

Layer Size	Depth 2	Depth 3	Depth 4	Depth 5
N	0.499	0.497	0.493	0.498
2N	0.503	0.499	0.492	0.500
3N	**0.983**	0.500	0.496	0.493
4N	0.496	0.502	0.500	0.500
5N	0.501	**0.813**	0.506	0.494

Table 5.3: Validation accuracy for experiments with bit length of 30.

5.4 Discussion

In this chapter, I explored deeper and wider architectures to understand to what extent they improve the performance of networks when solving the parity function problem. Furthermore, I test the best performing configuration from the preliminary experiments on the 30-bit parity function problem.

It was pretty clear from the depth and layer size experiments that larger networks result in better validation accuracy for input sizes of 10 and 20. Unfortunately, these configurations could not learn the 30-bit parity function. That was the reason for conducting the last set of experiments where I increased the number of epochs and tested a combination of depth and layer size to find a suitable architecture that could be able to scale

to longer bit lengths. While most of these experiments failed, some were able to learn the function with high accuracy. In particular, a network of depth 2 and layer size of 90 neurons was able to learn the function up to a 98% accuracy. This depth and layer size seems to be working very well as further confirmed by the respective experiments.

The N-bit parity function seems to become harder to solve with the increase of N, especially when the number of examples stays constant. It is difficult to ensure that the network starts learning, as once it the validation accuracy begins to increase, the network has no problem approximating the parity function to a high degree. Applying the right hyperparameters as described in Chapter 4 and running the network for long periods of times are two methods that increase the likelihood of learning the parity function.

Chapter 6

Convolutional Network Experiments

Convolutional Neural Networks have been very successful in solving various artificial intelligence problems, especially when dealing with images (Krizhevsky et al., 2012) and other large 2 dimensional inputs. The parity function input is just a one-dimensional array, therefore can apply one-dimensional CNNs. Such 1D architectures have shown promising results in problems such as classification of ECG signals (Kiranyaz et al., 2015) and fault detection (Ince et al., 2016).

CNNs are usually more practical when dealing with large examples as they tend to extract the more important features needed for learning. Although the parity function input is not very large, using such architecture might help with scaling the network to learn longer bit lengths which was a substantial problem for fully-connected networks.

6.1 Architecture

The architecture for this set of experiments is going to consist of convolutional layers and fully-connected layers. As described in Section 2.2, a number of filters will convolve over the input producing feature maps. Figure 6.1 illustrates how a convolutional layer would be applied to a one-dimensional input. Typically, after a convolutional layer, a pooling layer is applied to further reduce the size of the input and extract the more important features from the convolution. Finally, the remaining feature map from the pooling operation is fed into a fully-connected layer and training continues as in Chapter 5.

This type of architecture allows us to tune many hyperparameters. We can vary the stride of the CNN, set the size of the filter between 1 and the number of inputs and apply any amount of filters, in hope to extract different feature representations. Furthermore, we can stack multiple convolutional, pooling and fully-connected layers.

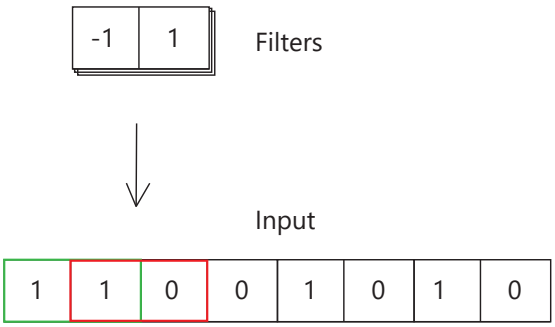

Figure 6.1: One dimensional convolution.

6.2 Description

As I mentioned in the previous section, there are many architectural hyperparameters that can be tuned. I decided to focus on the filter size, number of filters and the depth of the network. Both convolutional and fully-connected layers use SELU as an activation function, Lecun Uniform as the weights initializer and Adadelta as the gradient descent optimizer. Unless otherwise stated, I use 'binary crossentropy' as the loss function, batch size of 100 and epoch sizes of 1000, 500 and 500 for bit lengths of 10, 20 and 30 respectively. Moreover, I use 1 convolutional layer followed by two fully-connected layers with neurons equal to the number of inputs and an output layer consisting of a single neuron with a sigmoid activation function. I do not use a pooling operation since the network seems to lose too much information in the process and reduction in complexity is not needed for the short inputs of the parity function problem.

Filter size Experiment

In this set of experiments, I vary the size of the filter applied to the input. I examine five different filter sizes equally spaced between 1 and the size of the input - (1, $\frac{N}{4}$, $\frac{N}{2}$, $\frac{3N}{4}$, N). I experiment using bit lengths of 10, 20 and 30 and set the number of filters equal to the size of input - 10, 20 and 30 respectively.

Filter count Experiment

For this set of experiments, I study the importance of the number of filters applied to the input. I set the bit length to 20, the filter size to 20 and remove one of the fully-connected layers since I suspected that the network would be getting near perfect results as established from the previous experiments. I explore 5 different filter counts - 1, 5, 10, 20 and 40. Using a single filter with size as big as the input is essentially the same as using a fully-connected layer.

Experiments with bit length of 30

The filter parameter experiments failed to solve the parity function problem of bit length 30. For this set of experiments, I increased the number of epochs, added 2 fully connected layers and tried a variety of smaller and larger filter counts. In particular, I experimented with epoch sizes of 1000, 2000 and 3000 and filter sizes of N/4, N and 4N where N is the size of the input. Part of the reason for the variety of parameters

was to establish whether its even possible to learn the 30-bit parity function with such an architecture, even if its due to chance.

6.3 Results

Filter size Experiment

The results from these experiments are presented in Table 6.1, Figure 6.2 and Figure 6.3. Convolutional Neural Networks achieved close to a perfect accuracy on the parity function problem for bit lengths of 10 and 20. Although accuracy was high for most of the filter sizes, the best and most consistent experiments were the ones with filter size equal to the size of the input. The 30-bit parity function turned out to be a difficult problem for this architecture as well, as it was not able to learn anything above the baseline.

Filter Size	Tr(10)	Val(10)	Tr(20)	Val(20)	Tr(30)	Val(30)
1	1.0	0.975	0.999	0.998	0.582	0.495
N/4	0.997	0.990	0.576	0.504	0.638	0.500
N/2	0.832	0.639	0.998	0.997	0.673	0.502
3N/4	1.0	0.990	0.999	0.999	0.683	0.500
N	0.997	0.995	0.999	1.0	0.598	0.498

Table 6.1: Training (Tr) and validation (Val) accuracies for each filter and input size.

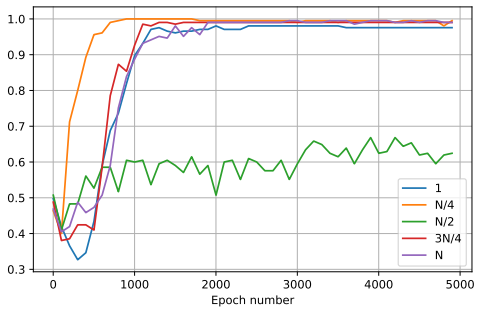

Figure 6.2: Validation accuracy during training for filter size experiments of 10-bit input.

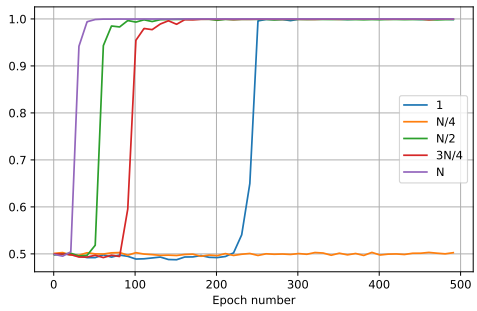

Figure 6.3: Validation accuracy during training for filter size experiments of 20-bit input.

Filter count Experiment

This set of experiments had very inconclusive results as shown in Table 6.2. Applying a single filter defies the purpose of using a convolutional neural network, thus it is understandable that the accuracy is lower than the rest of the experiments. All other filter counts performed very well, achieving accuracy close to the optimum. In particular filter counts of N and N/4, where N is the size of the input, achieved the best overall accuracy and started learning the parity function immediately as illustrated in Figure

Filter count	Training Accuracy	Validation Accuracy
1	0.662	0.717
5	0.999	0.999
10	0.997	0.990
20	0.997	0.997
40	0.996	0.995

Table 6.2: Training and validation accuracies for each filter count.

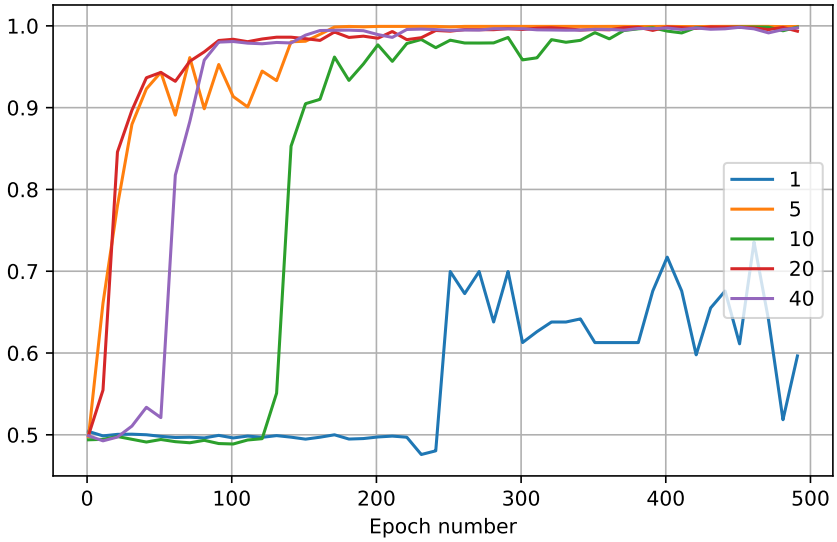

Figure 6.4: Validation accuracy during training for filter count experiments.

6.4. This could, of course, be based on pure chance of the weight initialization as the final difference in accuracy was insignificant.

Experiments with bit length of 30

This set of experiments demonstrated the inability of this architecture to learn the parity function. It seems that scaling to longer input sizes is a difficult challenge for Convolution Neural Networks when it comes down to solving the parity function. None of the settings could achieve validation accuracy above the baseline. After these disappointing results, I also tried to include a pooling layer and stack extra convolutional layers but there was no difference in the final outcome.

Epochs	N/4 Filters	N Filters	4N Filters
1000	0.494	0.494	0.494
2000	0.493	0.499	0.504
3000	0.497	0.502	0.501

Table 6.3: Validation accuracy for each filter count and epoch size on 30-bit inputs.

6.4 Discussion

In this chapter, I explored convolutional neural networks and their performance on the parity function problem. I experimented with various filter sizes, counts and depths of the network to establish which hyperparameters achieve the best results in solving this problem.

Convolutional neural networks achieved almost perfect generalization for parity functions of bit length 10 and 20. They arguably achieved better results than the fully-connected network experiments. While the difference between filter sizes was negligible, when the filter size was equal to the input length, the network achieved the highest accuracy and learned the function faster.

The number of filters in the convolutional layer also did not have a significant impact on the final results, even when using a more constricted network. The final accuracies were very promising except when using only 1 filter, which could also be considered as a fully-connected layer.

The convolution architecture had troubles learning the 30-bit parity function. Even after multiple attempts with various epoch sizes, depths, filter sizes and filter counts, the network could not achieve a validation accuracy above the baseline. Furthermore, it did not seem to be more efficient than fully-connected networks as more parameters had to be learned.

Many hyperparameters such as stride and pooling type were not considered in this set of experiments. Furthermore, the optimal activation function, initializer and optimizer for the convolutional layer might not have been the same as for the fully-connected one, which I used as a reference point. The number of examples (100,000) could have been insufficient as well but ultimately, using more examples might lead to scaling problems of different nature.

Chapter 7

Recurrent Network Experiments

Recurrent Neural Networks, as described in Section 2.3, are a type of neural network model which deals particularly well with sequential data such as languages, financial information and speech recognition. In theory, this architecture is ideal for a problem such as the N-bit parity function because of its continuous nature.

In this chapter, I explore two similar recurrent models. The first approach is specific to the parity function and requires a slightly different data generation procedure. The other one is a more general application of recurrent neural networks. Ultimately, the more conventional architecture is preferred, as no extra tweaking is required.

7.1 Architecture

The n-bit parity function can be defined as a series of XOR operations - $f(x) = x_1 \oplus x_2 \cdots \oplus x_n$. While going through the input of the function we can compute the XOR of the first two bits, then compute the XOR of that result with the third bit and continue until we have computed the final output. This is the basis for the first recurrent neural network architecture which we can call the **explicit memory** one. The input of this model is the same as before, but the output is a recursively calculated parity function at each time step. For example, if we want to know the parity of a bit string 0110 we are going to have an output:

```
0 -> 0
0 1 -> 1
0 1 1 -> 0
0 1 1 0 -> 0
-------------
0 1 0 0
```

Therefore the final parity function solution is the last bit of the output, which in this case is 0. Hopefully, the network will be able to learn to apply the XOR function between the input and the carried information from the previous output. This architecture is illustrated in Figure 7.1.

The other configuration I examined is similar, but instead of explicitly outputting every step of this operation, I just output the final answer. I call this the **implicit memory** model since we do not guide the model to learn the XOR representation. Although using such model might make the parity function more difficult to learn, it would be the default method of applying recurrent neural networks to any other similar problem. This architecture is illustrated in Figure 7.2.

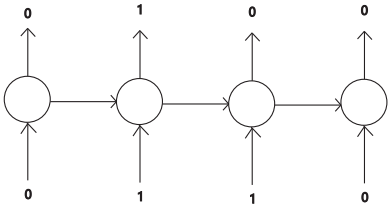

Figure 7.1: Explicit memory architecture.

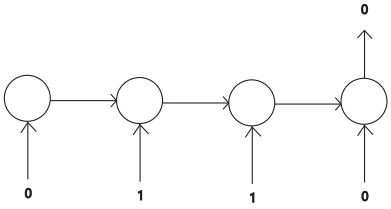

Figure 7.2: Implicit memory architecture.

7.2 Description

In this set of experiments,f I use the LSTM recurrent architecture described in Section 2.3.1. For the LSTM layer, I use the best hyperparameters from the preliminary experiments - SELU as an activation function at the hidden layer and Lecun Uniform for weight initialization. I use RMSProp as a gradient descent optimizer since its recommended for recurrent architectures. Unless otherwise stated, I use a single sigmoid output neuron, 'binary crossentropy' loss function and batch size of 100.

Explicit Memory Experiment

For this set of experiments, I use the explicit memory architecture described above and I vary the number of neurons in the LSTM layers. The values I explore are N/2, N, and 2N where N is the size of the input. I test this architecture on the parity function of size 10, 20 and 30 with epoch sizes of 500, 50 and 50 respectively.

Implicit Memory Experiment

These experiments are very similar to the explicit memory ones. I change the architecture to the implicit memory one described above and vary the number of neurons in the LSTM layers between N/2, N and 2N. I also increase the number of epoch sizes to 1000, 100, 100 for bit lengths of 10, 20 and 30 respectively.

7.3 Results

Explicit Memory Experiment

The explicit memory recurrent architecture achieved perfect generalization on all of the bit sizes provided, given enough neurons at the LSTM layer. Increasing the number of neurons speeds up learning. After experimenting with more neuron variations

I concluded that N+1 neurons are sufficient to learn the function to a 100% validation accuracy after only a few epochs. I also explored the scalability of the of the architecture and tried experiments with up to 50 bits with 100,000 examples and the network had no problem learning the function.

NEURONS	TR(10)	VAL(10)	TR(20)	VAL(20)	TR(30)	VAL(30)
N/2	0.830	0.792	0.818	0.818	0.818	0.820
N	0.996	0.995	1.0	1.0	1.0	1.0
2N	1.0	1.0	1.0	1.0	1.0	1.0

Table 7.1: Training (Tr) and validation (Val) accuracies for each layer and input size of the explicit memory architecture.

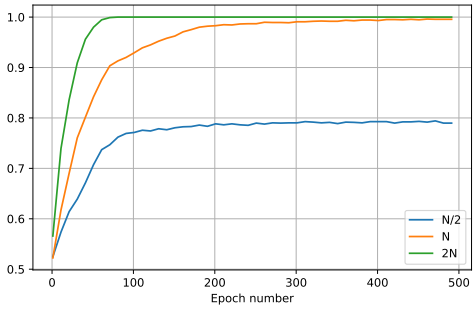

Figure 7.3: Validation accuracy during training for layer size experiments of 10-bits on the explicit memory model.

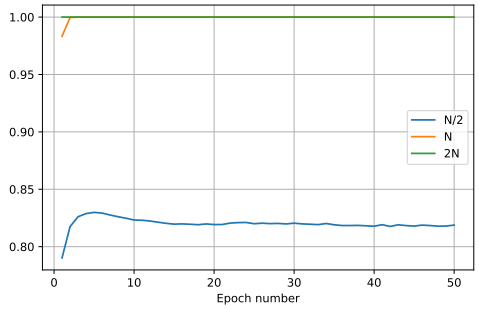

Figure 7.4: Validation accuracy during training for layer size experiments of 20-bits on the explicit memory model.

Implicit Memory Experiment

The implicit memory architecture also achieved impressive results on the 10 and 20-bit inputs but like many other architectures, it failed to learn the 30-bit parity function. Having too many neurons might lead to overfitting as shown in the 10-bit experiments in Table 7.2 but in general, increasing the number of neurons leads to better learning.

In order to try to solve the 30-bit parity function with this type of architecture, I increased the number of epochs, stacked more LSTM layers and even added extra fully-connected layers before the output but none of the configurations could achieve accuracy above the validation baseline.

NEURONS	TR(10)	VAL(10)	TR(20)	VAL(20)	TR(30)	VAL(30)
N/2	0.849	0.731	0.597	0.569	0.555	0.492
N	0.964	0.917	0.949	0.939	0.577	0.500
2N	1.0	0.839	0.993	0.993	0.601	0.498

Table 7.2: Training (Tr) and validation (Val) accuracies for each layer and input size of the implicit memory architecture.

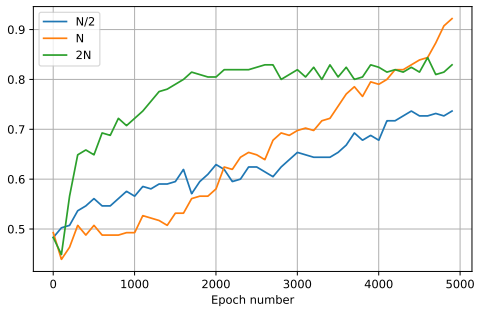

Figure 7.5: Validation accuracy during training for layer size experiments of 10-bits on the implicit memory model.

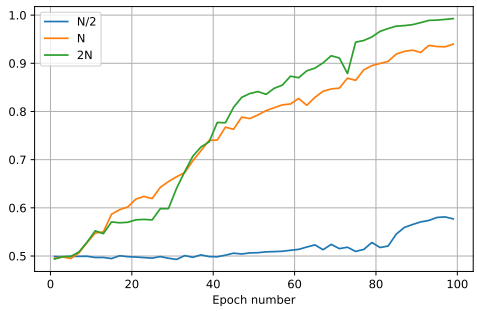

Figure 7.6: Validation accuracy during training for layer size experiments of 20-bits on the implicit memory model.

7.4 Discussion

In this chapter, I explored two recurrent neural network architectures and their performance on the parity function problem. I varied the number of neurons at the recurrent layer, but held other hyperparameters constant, since the architectures seemed to learn very well.

I started with the explicit memory architecture, which required recursively calculated output. Using only N+1 neurons at the recurrent layer causes the network to achieve perfect generalization with 100% validation accuracy after only a few training epochs. I tested this architecture with up to 50-bit parity function and the results were the same while still using only 100,000 examples. Unfortunately, this architecture is not applicable to other simple arithmetic functions. Using it is similar to crafting an overly-specific activation function that can solve the which have been done by many of the previous works on the topic.

The implicit memory architecture also yielded impressive results but ultimately could not learn the parity function of bit length 30. I am suspecting that increasing the number of examples might lead to better results, but have not empirically tested this. Nevertheless, this is a general approach that can be applied to any other arithmetics problem, as I use the same input and output as the fully-connected and convolutional structures.

Recurrent neural networks achieved the best results in solving the parity function from all configurations tested. That is expected since RNNs perform very well when dealing with sequential data such as the input and output examples of the parity function.

Chapter 8

Conclusion

In this report, I have investigated various neural network architectures in order to understand how they behave when solving the parity function. While some of the initial research questions were confirmed, many answers still remain unknown. It is very difficult to understand why certain configurations work better than others.

I empirically concluded that backpropagation has difficulty learning the correct weights for shallow neural networks using sigmoid activation function as described in (Setiono, 1997) even when using batch normalization. Changing the activation function increased the accuracy and stability of learning. The best performing ones being Softsign and SELU. Furthermore, appropriate weight initialization and gradient descent optimization produced more stable models with the best performing configuration being - SELU activation, Lecun Uniform initialization and Adadelta optimization. Despite the claims of batch normalization, I could not find any improvements in stability or accuracy during the learning of the parity function.

I confirmed that the depth of the network and the number of neurons in the hidden layers have a positive correlation with the final validation accuracy. Unfortunately, the fully-connected architecture had troubles scaling and very few experiments managed to learn the parity function of bit length 30.

I tested how one-dimensional convolutional networks handle the parity function problem. At first, convolutions seemed promising as they achieved high accuracy on 10 and 20-bit inputs when the filter size was equal to the input size. Unfortunately, this architecture also had problems scaling to longer input lengths and after many experiments, I could not find a suitable configuration that learns the 30-bit parity function. Convolutional networks seemed to start learning very suddenly and quickly reach their maximum potential.

Recurrent neural networks, as hypothesized, achieved the most promising results on the parity function problem. When using the specifically crafted explicit memory architecture, the network achieved perfect generalization while using very few neurons. It also seems to learn the parity function with few examples and epochs. The more general implicit memory architecture also achieved good results with inputs of bit length 10 and 20 but, ultimately could not solve the 30-bit parity function.

Overall, I can conclude that neural networks can learn the appropriate weights to solve the parity function using gradient descent. Some architectures had trouble scaling up and solving inputs of length 30. One suggestion would be to generate more data, although this would become computationally impossible after a certain number of bits. Recurrent neural networks might require a little bit of tweaking to get right but achieve the best accuracy without the need for many examples and computational power.

It is interesting to note that overfitting did not seem to be a problem when learning the parity function, regardless of the configuration used. Most of the times, given enough neurons and layers, the networks would either start learning at the beginning and achieve high accuracy or will not learn anything above the baseline of 50% despite the large number of epochs used.

8.1 Limitations and further work

It is worth mentioning the limitations of my experiments and to provide directions for further work. Although I tried to investigate many activation functions, initialization methods and optimizers, inevitably I have missed some standard ones that might perform well on the parity function problem. Furthermore, I did not experiment with all of the possible hyperparameters for the convolutional and recurrent networks. While overfitting was not a large issue, using regularization could aid learning in unpredictable ways and should potentially be tested.

As described in Section 2.8, there are several modern neural network architectures that claim to solve arithmetic problems with ease. It would be interesting to see how they perform when solving the parity function.

Finally, as shown in most of the experiments, there is a degree of randomness in learning the parity function. Even when running the same experiment three times as I did, it is difficult to know whether the results are not due to pure chance, especially when the differences between some hyperparameters are so small. Confirming the preliminary experiments would be a worthwhile task.

Bibliography

Abadi, Agarwal, Ashish, Barham, Paul, and Eugene... TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Arslanov, M.Z, Ashigaliev, D.U, and Ismail, E.E. N-bit parity ordered neural networks. *Neurocomputing*, 48(1):1053 – 1056, 2002. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(02)00610-0. URL http://www.sciencedirect.com/science/article/pii/S0925231202006100.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

Chollet, François et al. Keras. https://keras.io, 2015.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. Technical Report UCB/EECS-2010-24, EECS Department, University of California, Berkeley, Mar 2010. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-24.html.

Franco, L. and Cannas, S. A. Generalization properties of modular networks: implementing the parity function. *IEEE Transactions on Neural Networks*, 12(6):1306–1313, Nov 2001. ISSN 1045-9227. doi: 10.1109/72.963767.

Franco, Leonardo and Cannas, Sergio A. Solving arithmetic problems using feedforward neural networks. *Neurocomputing*, 18(1):61 – 79, 1998. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(97)00069-6. URL http://www.sciencedirect.com/science/article/pii/S0925231297000696.

Fung, Hon-Kwok and Li, Leong Kwan. Minimal feedforward parity networks using threshold gates. *Neural Computation*, 13(2):319–326, 2001. doi: 10.1162/089976601300014556. URL https://doi.org/10.1162/089976601300014556.

Glorot, Xavier and Bengio, Yoshua. Understanding the difficulty of training deep feedforward neural networks. In Teh, Yee Whye and Titterington, Mike (eds.), *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pp. 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL http://proceedings.mlr.press/v9/glorot10a.html.

Graves, Alex. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013. URL http://arxiv.org/abs/1308.0850.

Graves, Alex, Mohamed, Abdel-rahman, and Hinton, Geoffrey E. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013. URL http://arxiv.org/abs/1303.5778.

Graves, Alex, Wayne, Greg, and Danihelka, Ivo. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL http://arxiv.org/abs/1410.5401.

Grossberg, Stephen. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks*, 1(1):17 – 61, 1988. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(88)90021-4. URL http://www.sciencedirect.com/science/article/pii/0893608088900214.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Hohil, Myron E, Liu, Derong, and Smith, Stanley H. Solving the n-bit parity problem using neural networks. *Neural Networks*, 12(9):1321 – 1323, 1999. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(99)00069-6. URL http://www.sciencedirect.com/science/article/pii/S0893608099000696.

Hopfield, J J. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 79(8): 2554–2558, 1982. ISSN 0027-8424. doi: 10.1073/pnas.79.8.2554. URL http://www.pnas.org/content/79/8/2554.

Hornik, Kurt, Stinchcombe, Maxwell, and White, Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359 – 366, 1989. ISSN 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. URL http://www.sciencedirect.com/science/article/pii/0893608089900208.

Ince, Turker, Kiranyaz, Serkan, Eren, Levent, Askar, Murat, and Gabbouj, Moncef. Real-time motor fault detection by 1d convolutional neural networks. 63, 11 2016.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Jansen, B. and Nakayama, K. Neural networks following a binary approach applied to the integer prime-factorization problem. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 4, pp. 2577–2582 vol. 4, July 2005. doi: 10.1109/IJCNN.2005.1556309.

Kaiser, Lukasz and Sutskever, Ilya. Neural gpus learn algorithms. *CoRR*, abs/1511.08228, 2015. URL http://arxiv.org/abs/1511.08228.

Kiranyaz, Serkan, Ince, Turker, and Gabbouj, Moncef. Real-time patient-specific ecg classification by 1d convolutional neural networks. 63, 08 2015.

Klambauer, Günter, Unterthiner, Thomas, Mayr, Andreas, and Hochreiter, Sepp. Self-normalizing neural networks. *CoRR*, abs/1706.02515, 2017. URL http://arxiv.org/abs/1706.02515.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

Kumar, Ankit, Irsoy, Ozan, Su, Jonathan, Bradbury, James, English, Robert, Pierce, Brian, Ondruska, Peter, Gulrajani, Ishaan, and Socher, Richard. Ask me anything: Dynamic memory networks for natural language processing. *CoRR*, abs/1506.07285, 2015. URL http://arxiv.org/abs/1506.07285.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. ISSN 0018-9219. doi: 10.1109/5.726791.

LeCun, Yann, Bottou, Léon, Orr, Genevieve B., and Müller, Klaus-Robert. Efficient backprop. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pp. 9–50, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2. URL http://dl.acm.org/citation.cfm?id=645754.668382.

Liu, Derong, Hohil, Myron E., and Smith, Stanley H. N-bit parity neural networks: new solutions based on linear programming. *Neurocomputing*, 48(1):477 – 488, 2002. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(01)00612-9. URL http://www.sciencedirect.com/science/article/pii/S0925231201006129.

McKinney, Wes. pandas: a foundational python library for data analysis and statistics.

Minor, J.M. Parity with two layer feedforward nets. *Neural Networks*, 6(5):705 – 707, 1993. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(05)80114-5. URL http://www.sciencedirect.com/science/article/pii/S0893608005801145.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. URL http://dl.acm.org/citation.cfm?id=3104322.3104425.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Roadknight, Chris M., Aickelin, Uwe, Qiu, Guoping, Scholefield, John, and Durrant,

Lindy. Supervised learning and anti-learning of colorectal cancer classes and survival rates from cellular biology parameters. *CoRR*, abs/1307.1599, 2013. URL http://arxiv.org/abs/1307.1599.

Saxe, Andrew M., McClelland, James L., and Ganguli, Surya. 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.

Setiono, Rudy. On the solution of the parity problem by a single hidden layer feedforward neural network. *Neurocomputing*, 16(3):225 – 235, 1997. ISSN 0925-2312. doi: https://doi.org/10.1016/S0925-2312(97)00030-1. URL http://www.sciencedirect.com/science/article/pii/S0925231297000301.

Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, van den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore, and Hassabis, Demis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, jan 2016. ISSN 0028-0836. doi: 10.1038/nature16961.

Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex, Sutskever, Ilya, and Salakhutdinov, Ruslan. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Stork, David G. and Allen, James D. How to solve the n-bit encoder problem with just one hidden unit. *Neurocomputing*, 5(2):141 – 143, 1993. ISSN 0925-2312. doi: https://doi.org/10.1016/0925-2312(93)90033-Y. URL http://www.sciencedirect.com/science/article/pii/092523129390033Y.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014. URL http://arxiv.org/abs/1411.4555.

Walt, Stefan van der, Colbert, S. Chris, and Varoquaux, Gael. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13 (2):22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37. URL http://dx.doi.org/10.1109/MCSE.2011.37.

Wilamowski, B. M., Hunter, D., and Malinowski, A. Solving parity-n problems with feedforward neural networks. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 4, pp. 2546–2551 vol.4, July 2003. doi: 10.1109/IJCNN.2003.1223966.

Zeiler, Matthew D. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012. URL http://arxiv.org/abs/1212.5701.