Investigating the Ability of Neural Networks to Learn Simple Modular Arithmetic

Theodoros Palamas

Master of Science
School of Informatics
University of Edinburgh

2017

Abstract

Neural networks are computing systems built to learn to do certain tasks by analyzing examples that are representative of that task, and progressively improve themselves, instead of being specifically programmed for that task. They were developed during the 1960s and were inspired by the neural structure of the human brain. Initially, research was scarce, mainly due to the fact that the computers of that era did not have enough processing power to handle the training of large neural networks yet. As computer hardware got better, however, advances in neural networks were slowly made. It was only until recent years, along with with advances in parallel programming and GPU processing, that neural networks became of interest again, leading to an explosion in research activity. They have found application and have been successful in many areas, including speech recognition, natural language processing, computer vision, medical diagnosis and have even learned to play board and video games. One common characteristic of the tasks in these areas is that they cannot be easily solved by specifically programming for them, hence the success of neural networks. An interesting question then is: How can a neural network handle an already solved problem that has a specific algorithmic solution? To get a glimpse of their capabilities in such an environment we will investigate whether various neural network architectures can learn simple modular arithmetic, i.e. whether they can learn to predict the remainder of an integer modulo another integer. Our results show that neural networks struggle with this task, not being able to outperform random chance in most cases. There is also some evidence that the networks might be memorizing instead of learning.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Theodoros Palamas*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this first section we will provide context about the achievements of neural networks, discuss our motivations for choosing modular arithmetic and briefly describe the task. We will also provide relevant literature and an outline of the thesis.

1.1 The Strength Of Neural Networks

From the moment research activity begun on neural networks and until today, they have been applied to numerous areas, in which they have been very successful in learning difficult tasks, for which there is no apparent algorithmic solution.

Some of the early research involves handwriting recognition. It was shown that they were better at recognizing handwritten digits and letters than systems with hand-designed heuristics (LeCun et al., 1998) and achieving high accuracy in general (LeCun et al., 1990; Knerr et al., 1992; Martin and Pittman, 1990).

In speech recognition they were very successful in recognizing single words from set of noisy confusable words (Lang et al., 1990), continuous speech (Bourlard and Morgan, 1993; Hinton et al., 2012a) and phonemes (Waibel, 1989; Waibel et al., 1989; Mohamed et al., 2012). Their performance improved with the usage of recurrent neural networks (Graves et al., 2013).

They have also found application in natural language processing, in which they have made great progress by being able to analyze word sequences and recognize the underlying semantic and grammatical roles (Bengio et al., 2003; Collobert and Weston,

2008), as well as parse sentences (Pei et al., 2015; Chen and Manning, 2014; Durrett and Klein, 2015; Weiss et al., 2015).

In recent years, mainly due to the increase in processing power, they have exploded in the area of computer vision and image classification, where convolutional neural networks have been applied in a large degree. They have performed exceptionally well in classifying images from various datasets and especially ImageNet, which is one of the largest and most diverse ones (He et al., 2016; Szegedy et al., 2015; Krizhevsky et al., 2012; Goodfellow et al., 2013; Zeiler and Fergus, 2014; Oord et al., 2016).

Neural networks have also begun making strides in the area of medical diagnosis, in which they have managed to diagnose lung cancer (Ganesan et al., 2010), colorectal cancer (Bottaci et al., 1997) and recognize invasive cancer cells using only their shape (Lyons et al., 2016).

We should also mention that they have managed to learn to play simple video games (Mnih et al., 2013) and even have defeated the human European champion in the game of Go (Silver et al., 2016), which has been considered a board game more difficult than chess and has posed a great challenge for AI.

One of the main reasons why they have been so successful is their representational power. It has been proven that a neural network with a single hidden layer that contains a finite number of units can approximate any real continuous function (Cybenko, 1989; Hornik, 1991; Mhaskar, 1993; Hornik et al., 1989). However, these are not guarantees that they can learn these functions by themselves and even if they can it says nothing about how fast they can do it. Recently, there have been some research activity on these matters. It is shown that neural networks can learn any low-degree polynomial (Andoni et al., 2014) and that there are some cases where training specific network architectures can be achieved in polynomial time (Livni et al., 2014; Arora et al., 2014), while in general it has been shown that finding the optimal weights is NP-hard (Bartlett and Ben-David, 1999; Blum and Rivest, 1989).

1.2 Motivation

The motivation behind this thesis is of two kinds: short-term and long-term. The short term motivation is the success of neural networks itself. They have found application in many tasks that are extremely difficult to solve by specifically programming for them,

if any such solution exists at all. But this is what they were designed for. It would be very interesting to see what they can do in a task that is already algorithmically solvable and that has been easily calculated by computers for a long time. This is why we chose modular arithmetic. On might wonder why we chose it over addition or multiplication. The reason behind this is because, intuitively, these operations seem far simpler to learn. We have very specific rules for how to add or multiply two integers, which are universal and we can apply just by looking at them. However, the case is not so simple for modular arithmetic. While there are some such simple rules, like checking the last digit of an integer to find its remainder $mod 2$, these do not apply to the majority of moduli. To find whether the remainder of an integer $mod 3$ is 0 we need to sum its digits and check the result for the same condition, which does not directly solve the problem but rather transforms it into a simpler one. Still, this same rule cannot be applied to other moduli and for some of them we even need to execute a division algorithm. We would therefore like to see whether a neural network can learn such rules.

Then there is the long-term motivation, which might seem to border the impossible but it is quite intriguing. Let us assume that in the future, neural networks do indeed learn modular arithmetic and that they learn it for large integers. A possible next step, then, could be to try and teach them how to factorize large integers. If they even learn to do that, then the only thing we would need, is for the training process to be faster than the factorization algorithm. The implications of such an achievement would be tremendous for cryptography and especially RSA cryptosystems whose security guarantees are based on the fact that factorization is extremely difficult. Again, even though this might seem far-fetched, it would be wise to at least consider the possibility.

1.3 The Task

In this thesis we will investigate the capability of neural network architectures to learn modular arithmetic by treating it as a classification problem. The congruence relation $n \equiv r(mod\, p)$ can be seen as classifying n to one of p classes depending on the remainder r that their Euclidean division produces. Our datasets consist of randomly generated 128-bit integers. We will initially test several feed-forward networks for small moduli p. We will continue by trying to improve some of these results with convolutional and recurrent networks, followed by regularized feed-forward ones. Finally,

we will test some of the initial networks on modified datasets. The point of this thesis is to test whether the networks can make the correct classifications just by looking at the integers, and therefore the feature vectors of our datasets will consist only of their digits with no feature engineering.

1.4 Related Work And Contribution

The first pieces of related work come to us from the 1990s. During that period neural networks were also studied simply as circuits that could be implemented to calculate basic arithmetic operations. It was shown that feed-forward neural networks of up to five layers could compute very fast addition, multiplication, division and other operations (Siu and Bruck, 1990; Siu et al., 1993; Franco and Cannas, 1998). However, the networks in question were not trained but rather their weights were fixed and set by hand. Nevertheless, this research indicates that there are indeed weights that render the network able to compute these operations.

Another area of interest that neural networks are starting to show some progress recently, is that of learning program execution. They have been taught how to execute with high accuracy simple programs like copying, sorting and addition (Zaremba and Sutskever, 2014; Reed and De Freitas, 2015; Graves et al., 2014). While this is not exactly like learning addition just by looking at examples of integers and their result, it can be seen in a sense as such, and is much closer to what we are trying to accomplish.

The aim of this thesis is different from what we have discussed above in two ways. First of all, we are trying to teach a neural network modular arithmetic by training with examples and not setting the weights by hand. Furthermore we are trying to achieve this not by teaching it an algorithm, like Euclidean division, that has already been used to solve the problem, but rather let the network itself recognize any underlying structure that an integer has, which might determine its remainder. Something similar was done for addition (Hoshen and Peleg, 2016), where it was shown that a neural network could add 7-digit integers with high accuracy.

1.5 Thesis Outline

In chapter one we gave a brief introduction on the general standing of neural networks as of recently. We also provided motivation for choosing to investigate modular arithmetic and briefly described the task at hand, as well as presented literature that could be considered relevant to our work.

In chapter two we will describe in more detail the three core neural network architectures that will be used in our experiments: feed-forward, convolutional and recurrent. We will also discuss the training process of neural networks, from how we use the available data to how the networks are actually trained and we will also consider some of the most common issues that emerge during that process.

In chapter three we will describe in more detail the task at hand and our experimental setup. We will present our datasets, the technical details of our environment and a brief introduction on hyperparameter optimization. This will be followed by the description of the experiments, which consist of testing several neural network architectures, some regularization methods and finally, some modifications on the datasets themselves.

In chapter four we will present the results produced by the experiments described in chapter three. They show that the vast majority of the networks we trained was no better than random chance and in the cases that they seemed to be learning, evidence leads us to believe that they were instead memorizing.

In chapter five we will provide a thorough discussion on the results of our different sets of experiments, as well as some general remarks, in the context of the achievements of neural networks and relevant literature.

We will conclude in chapter five by summarizing our findings, providing critique on our experimental process and suggesting ideas for further work that might prove fruitful in the future.

Chapter 2

Background

In this section we will present some background about neural networks, which are the core elements of our experimentation. We will start with a brief introduction about their origins and then continue with the description of some widely used architectures, as well as the process with which they are trained.

2.1 Perceptron

Before neural networks evolved into the more sophisticated versions that we see today, we had the perceptron. Inspired by the workings of the neurons in the human brain, it was developed in the 1960s (Rosenblatt, 1961).

Generally speaking, a neuron receives input signals from other neurons and depending on whether the combination of those signals exceeds a certain threshold, it produces a signal that is in turn transmitted to other neurons. Similarly, the basic model of a perceptron takes a number of inputs, computes a weighted sum of them, as well as adding a bias and finally outputs 1 or 0 depending on whether the computed quantity is positive or not. Below we have the formula for the perceptron in Figure 2.1.

$$f(x_1, x_2, x_3) = \begin{cases} 1, & \text{if } w_1x_1 + w_2x_2 + w_3x_3 + b > 0 \\ 0, & \text{otherwise} \end{cases}$$

The weights w_i and bias b are the learnable parameters of the perceptron that are determined through training. The weights control the importance of each input and the

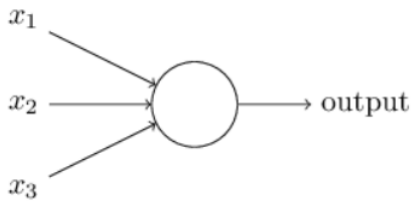

Figure 2.1: A simple perceptron.

Source: http://neuralnetworksanddeeplearning.com/

bias controls how easy it is for the perceptron to output 1. This simple mathematical model can serve as a binary linear classifier. Linear because the decision is made using a linear combination of the inputs and binary because the output is either 1 or 0.

Modern neural networks are composed of units (sometimes called neurons), that are based on the simple perceptron model mentioned above.

2.2 Architectures

Throughout the years, several neural network architectures have been proposed and have been used. Here, we will focus on feed-forward networks, which are the simplest and also the stepping stone for the development of many of the rest, as well as convolutional and recurrent networks which have been prominent recently due to their success.

2.2.1 Feed-forward

One of the earliest neural network architectures are the feed-forward networks. They are composed of a stack of layers which themselves are composed of units similar to perceptrons. Each unit within a layer has as inputs the outputs of all the units from the previous layer and its output is directed to all the units of the next layer. However, units within a layer share no connection whatsoever between them. Due to the units between layers being fully pairwise connected, this type of layer is also called a fully-connected layer. The weights and the biases of each unit in the network constitute the totality of its learnanble parameters and are commonly used to measure its size.

This structure is also the reason why they are called feed-forward. Due to the way the layers are connected, they form an acyclic graph and therefore the information flows form one end of the network to the other without any loops. For historical reasons, feed-forward networks may also rarely be called multilayer perceptrons, which is not entirely accurate since most of the time the units are not pure peceptrons.

In Figure 2.2 we can see a feed-forward network composed of five layers.

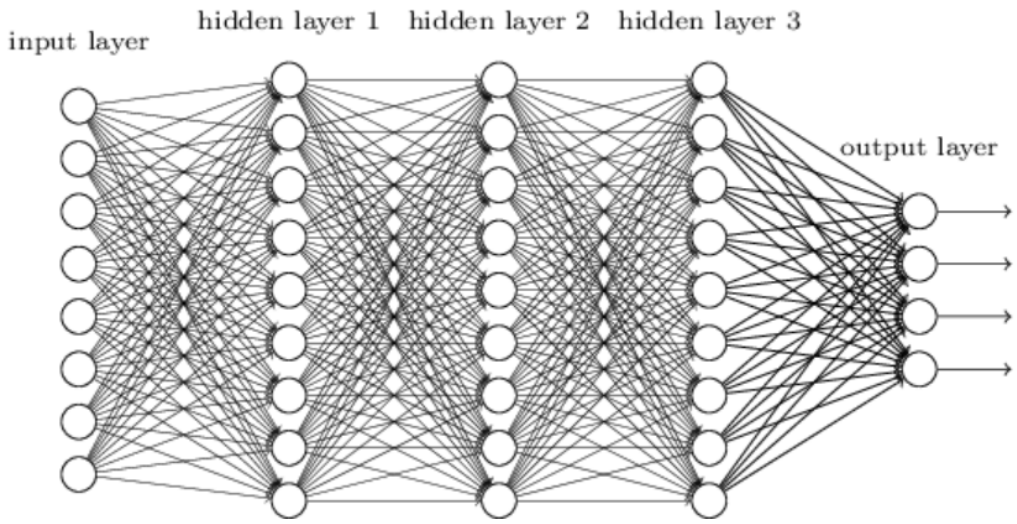

Figure 2.2: A simple feed-forward network.

Source: http://neuralnetworksanddeeplearning.com/

The leftmost layer is called the input layer. This layer does not consist of actual units but instead these are the features of our data that are fed into the next layer. In an image classification task, for example, the features are usually the pixel values of the image and in a natural language processing task they can be the words in a sentence. When we talk about the number of layers of a network we do not count the input layer.

The next three layers are called hidden layers and their number can vary depending on the task. As we mentioned earlier, they do not actually consist of perceptrons, but modified versions of them. The reason behind that is that perceptrons make the training process difficult to control. As we train the parameters, small adjustments can cause certain perceptrons to flip, which in turn can cause a cascade of sudden flips in the rest of the network. Even though these changes may cause some datapoints to be predicted correctly, it is quite possible that they have caused the network to behave in a drastically different way on other datapoints. This makes it difficult to adjust the parameters

accurately in order to smoothly reach their desired values. The way we solve this issue is by using what is called an activation function. As we saw earlier, perceptrons compute a weighted sum of their inputs and they output a binary outcome depending on whether this sum is positive or not. With activation functions, however, we don't produce a binary output. We pass the weighted sum through the function, which is smoother, and output its outcome. Historically, the most used activation function is the sigmoid ($f(x) = 1/(1+e^{-x})$).

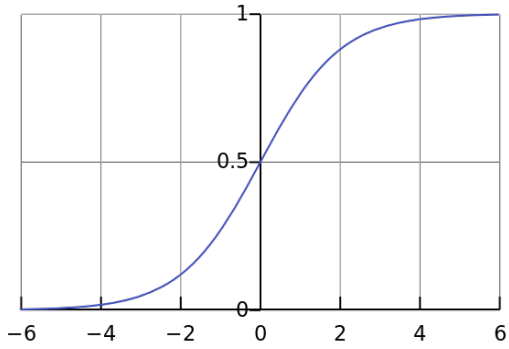

Figure 2.3: The form of the sigmoid function.

Source: https://en.wikipedia.org/wiki/Sigmoid_function

Sigmoids produce values between 0 and 1 and in contrast with perceptrons, the transition between them is smooth and gradual. There have been proposed many other activation functions, like the hyperbolic tangent ($f(x) = (e^x - e^{-x})/(e^x + e^{-x})$) which has a form similar to the sigmoid and the rectified linear unit (ReLU) ($f(x) = max(0,x)$) which has been vary popular in recent years.

Finally, we have the rightmost layer, which is called the output layer and its output is taken as a prediction made for a certain datapoint. This layer does not use any activation functions (or we can say that it uses the identity activation function $f(x) = x$) because the outputs of its units represent different things depending on the task. For a regression task this can be some real-valued target and for a classification task it can be the scores of the different classes. We should also mention that while in regression we can take the prediction of the output layer as is, in classification we often want a probability distribution over the different classes. To accomplish that we use a softmax activation function ($f_i(\mathbf{x}) = e^{\mathbf{x}_i} / \sum_{j=1}^{N} e^{\mathbf{x}_j}$ where $\mathbf{x}$ is a vector of length N) on the output layer. This function takes the class scores and transforms them to make them all positive and add up to 1, in order to be interpreted as a distribution.

2.2.2 Convolutional

Throughout the years, other neural network architectures have also been developed, that thrive in areas where simple feed-forward networks could not perform very well. One such architecture is the convolutional neural network (LeCun et al., 1998). The structure of these networks is very well-suited for the task of image classification and in recent years their use has increased considerably.

Convolutional neural networks utilize two new types of layers, the convolutional and the pooling layer. Their general structure consists of a stack of convolutional layers with pooling layers in-between some of them, followed by a few of the fully-connected layers that feed-forward networks use.

The convolutional layer is the one that makes these networks perform very well in image recognition tasks and its main difference from a fully-connected layer is structural. Since every unit in a fully-connected layer connects with every unit in the next layer, it is unable to take into account any spatial properties of the data that might be useful. In an image classification task, for example, pixels close to one another share greater dependencies that those far apart. A layer that treats all pixels the same way will have difficulties differentiating between images. The convolutional layer, however, is structured in a way that takes advantage of any spatial properties that might exist. At this point it should be better to think of the layers of the network as a square of units instead of a line as we did for feed-forward networks. Now, instead of a unit being connected to every unit in the previous layer, it is only connected to a small window of units, which is called its local receptive field. These connections are the weights associated with that unit and we also have a bias. The units of the convolutional layer behave similarly to those of a fully-connected layer, but instead of a simple weighted sum, they compute a 2-D convolution (hence the name) and then pass the result through an activation function to the next layer. In Figure 2.4 and Figure 2.5 we can see an example of this kind of connectivity, where each unit is connected to a 5x5 area of the previous layer. As we move to the next unit, the local receptive field also moves, which can be thought of as sliding a window over the previous layer. Using this process we construct the convolutional layer.

Its size is determined by two factors: the size of the local receptive field and the stride. The stride is how many units the local receptive field slides when we move to the next unit of the convolutional layer (it is 1 for both dimensions in our example). The size

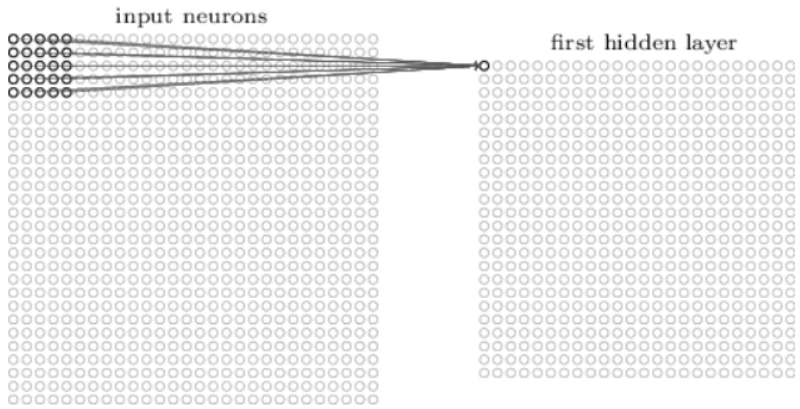

Figure 2.4: The connections of the first unit of a convolutional layer.

Source: http://neuralnetworksanddeeplearning.com/

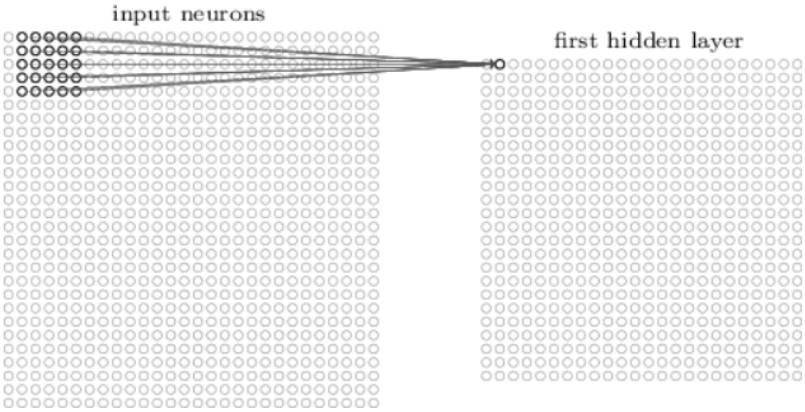

Figure 2.5: The connections of the second unit of a convolutional layer.

Source: http://neuralnetworksanddeeplearning.com/

of the convolutional layer will usually be smaller since the local receptive field cannot move outside the limits of the previous layer. In cases, however, where we want to keep the same size we can use zero padding, which means that we allow the local receptive field to go outside the limits and treat this extra area as consisting entirely of zeros.

Another important reason that convolutional layers are so good at detecting spatial properties is the fact that all the units within the layer share the same weights and bias. As a consequence all the units of a convolutional layer will detect the same feature (in the case of images it can be an edge or some other shape for example), but each unit will do that in a different position on the previous layer. This gives the convolutional layer the ability to be translation-invariant with respect to features. This group of units that share weights and bias is usually called a feature map and the shared weights and bias themselves are called a kernel or filter. In our example, the convolutional layer consists only of one feature map, but almost always we stack many different ones together in order to detect different features, as we can see in Figure 2.6.

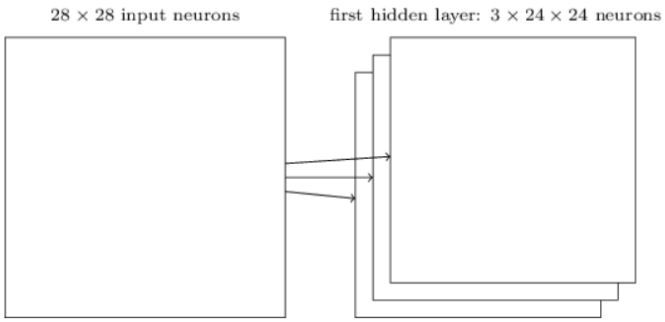

Figure 2.6: The feature maps of a convolutional layer.

Source: http://neuralnetworksanddeeplearning.com/

This seems to add a third dimension or depth to the way we think about the layout of the units in the convolutional layer. This is not a problem since we can also extend the local receptive field of the next layer with a third dimension to cover this case. It is common practice for the size of the local receptive field depth of the next layer to be equal to the number of feature maps of the previous layer. We also use no zero padding in this dimension, which means that the local receptive field will only be sliding in two dimensions. In the end, each convolutional layer scans the previous one as if it was two-dimensional but at each location takes into account all the different features that could be there, as detected by the previous feature maps. This also helps with situations where the input of the first convolutional layer is an image with three color channels.

We should also mention that due to this sharing of weights and biases, they have far less total parameters than a feed-forward network. This leads to faster training times and deeper networks that can achieve better performance.

The other type of layer used by convolutional networks is the pooling layer, which is found between some of the convolutional layers. It is a simple layer with no parameters and no activation function and its main purpose is to simplify the information of the previous layer. The most common form of pooling, max-pooling, uses a 2x2 local receptive field with a stride of 2 and no zero padding, which is applied on a feature map from the previous convolutional layer, and propagates to the next convolutional layer only the largest value in its local receptive field. This results to the feature map being halved in size as seen in Figure 2.7. This process is repeated for all the previous feature maps and the result is given as input to the next convolutional layer as seen in Figure 2.8.

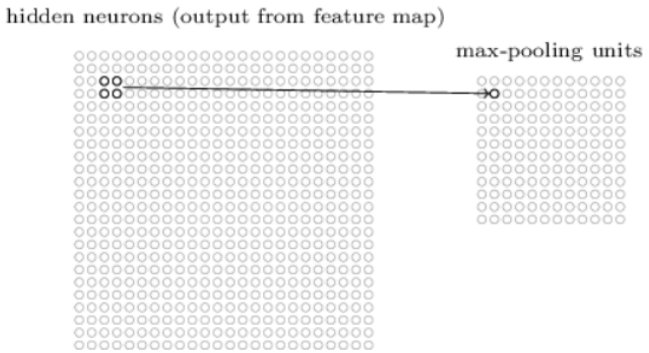

Figure 2.7: Pooling of a single feature map.

Source: http://neuralnetworksanddeeplearning.com/

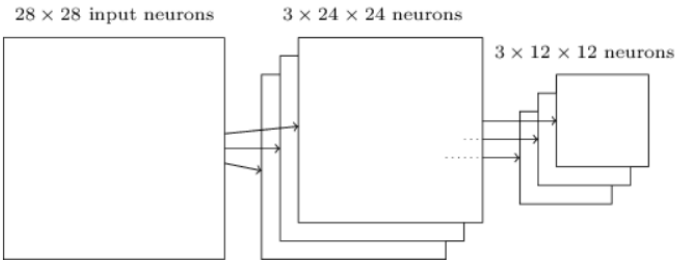

Figure 2.8: Pooling of an entire convolutional layer.

Source: http://neuralnetworksanddeeplearning.com/

Intuitively, we can think of a max-pooling layer as propagating to the next layer the rough location of a feature relative to others instead of its exact one, which is not so important. As a result, there are fewer pooled features, which helps to progressively reduce the number of parameters in later layers.

2.2.3 Recurrent

Another network architecture that has been very prominent is recurrent neural networks. It solves a core issue from which feed-forward networks suffer, that of sequential dependencies. Simple feed-forward networks treat datapoints as being completely independent from one another. This can be problematic in tasks where we are provided with sequential data that are closely related, like predicting the next word in a sentence or the next frame in a video. It is quite clear that we need a longer context than just the previous word or frame to make correct predictions. Recurrent neural networks are structured in a way that lets them take advantage of these dependencies and have been very successful in areas such as natural language processing and speech recognition.

The simplest recurrent network architectures are very similar to the feed-forward ones, with one exception. At each training step, each layer, instead of only using as input the current output of the previous layer, it also takes as input the output of itself from the previous training step.

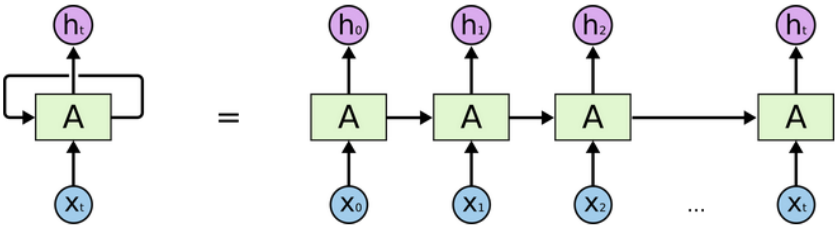

Figure 2.9: A recurrent neural network and its unrolling.

Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

On the left part of Figure 2.9 we have a layer A that takes some input x_t and output h_t at training step t. By using an arrow that starts from A and ends in A we symbolize the fact that the layer also takes input from itself at training step $t-1$. Thinking of recurrent networks in this loopy form may not make clear what their actual structure is or how far back the dependencies go. What we can do to better visualize the network is

to unroll it as seen on the right of Figure 2.9. We can also think of a recurrent network as multiple copies of the same feed-forward network, with each copy propagating past information to the next one.

This unrolling we mentioned above, is what gives a recurrent neural network its ability to form dependencies between sequential data. The more we unroll the network, the longer the context on which our prediction depends becomes. If we are trying to predict the next frame of a video we may only need a few previous frames and so we only need to unroll the network a couple of times. There are cases, however, where a longer context might be needed. We might want to predict the next word in a sentence, but the necessary information is not present in the sentence itself. It might be located a few sentences back or even a few paragraphs back. We could end up unrolling so much that the network becomes unable to learn to connect the information. It has been shown that recurrent neural networks have difficulties learning these long-term dependencies (Bengio et al., 1994).

This problem is solved by what is, in recent years, one of the most prominent types of modules in recurrent neural networks, the Long Short Term Memory or LSTM cell (Hochreiter and Schmidhuber, 1997). The LSTM recurrent neural network has the same structure as we mentioned above, but instead of a simple fully-connected layer it uses an LSTM cell. These cells are memory-like structures that are specifically designed to deal with long term dependencies. They contain information outside of the regular flow of the network and by using special structures called gates, they can regulate the state of this information. An LSTM cell contains three gates: the forget gate, the input gate and the output gate. The forget gate controls which of the information already stored in the cell is no longer relevant and should be discarded. The input gate controls which of the new information should be stored in the cell and finally, the output gate controls how much information the cell will output and allows it to keep information that is not relevant at the current training step but might be relevant in the future.

2.3 Training

Apart from all the different architectures that exist, another core part of neural networks is their training procedure. We will discuss all the important aspects, from how we use

the available data to how the parameters are optimized, as well as some common issues that emerge during training.

2.3.1 Datasets

Neural networks belong in the category of supervised learning. This means that they are trained on labeled data. Each datapoint consists of a feature vector and a label. The feature vector contains all the features of that datapoint (e.g. pixels of an image or words in a sentence) that we will use to make a prediction, and the label is the target outcome (e.g. what the image actually depicts or what the next word in the sentence is). The available data are most commonly separated in three sets, a training set, a validation set and a test set, to be used at different stages of training.

The training set contains the majority of the data and is the one used during training to adjust the parameters of the network.

The validation set is smaller and is used as a means of controlling the training. This set contains data that are never shown to the network and therefore have no influence on the adjustment of the parameters. At certain points during training we evaluate the performance of the network on this set, which is there to verify that an increase in accuracy on the training set actually yields an increase in accuracy on unseen data. Another reason to use a validation set is to help us compare the performance of different networks. Since high accuracy on the training set does not always translate to high accuracy in unseen data, the training set is not useful for these comparisons. The validation set, however, can serve this purpose since it has not been used for training by any of the networks.

Finally, the test set is used as a measure of the expected performance of a network in future data. It is used only once and only in the end of training. Unlike the validation set the network performance on it is not monitored during training. The core reason behind this is to avoid a form of quasi-training that happens on the validation set. Often the networks differ on what we call hyperparameters, like the number of layers, the number of units per layer, the activation function, etc. By comparing all these networks on the validation set we are in a sense training those hyperparameters. The test set solves this issue by being the last evaluation step and therefore a more accurate measure of future performance.

2.3.2 Optimization

Training a neural network is in essence an optimization problem. We are trying to find values for the network parameters (weights and biases) so that it approximates the training set as accurately as possible by minimizing the error between its predictions and the actual labels.

In order to calculate that error we need to define an appropriate cost function. These are non-negative functions that approach zero when the predictions of the networks approximate very well the actual labels of datapoints and grow larger when that is not the case. Mean squared error and cross-entropy are two of the most commonly used cost functions. While both perform very well in classification tasks, cross-entropy usually achieves faster training than mean squared error (Golik et al., 2013), which stems mainly from the structure of cross-entropy which allows the network to learn faster when the error is larger.

With a cost function defined, we can proceed to actually train the network by using a process called gradient descent, which is widely used in optimization problems. Gradient descent makes use of the gradients of all the network parameters with respect to the cost function. These gradients essentially show us in which direction we have to adjust the parameters in order to reach an optimal configuration that minimizes the cost function.

Assuming we have a cost function C, the gradient of parameter w of our network is the partial derivative $\frac{\partial C}{\partial w}$. We use this gradient in the update rule $w' = w - \eta \frac{\partial C}{\partial w}$ to calculate the new value w'. The same is also done for the rest of the parameters. This process is then repeated in order to minimize the error as much as possible. The gradients are calculated using the algorithm of backpropagation which was introduced in the 1970s but it was only a few years later until its efficiency for neural networks was discovered (Rumelhart et al., 1988).

In the update rule above we have introduced a new variable η which is called learning rate. This controls how much we change the parameters. If η is too large then the parameters change too much and we might miss the optimal configuration we are looking for. On the other hand if η is too small the parameters change very slowly and it might take too long to reach it.

The way we usually measure how much we have trained our network is in epochs.

An epoch is a whole pass through the training set. When using the basic form of gradient descent we just described, in order to compute the gradients we calculate the cost function based on the whole training set. That means that during an epoch each of the network parameters are only updated once. This leads to our network learning extremely slowly when we have a huge number of training datapoints. This problem is solved by a modified version of gradient descent, called stochastic gradient descent. The idea behind it is to estimate the gradients by only using a small randomly chosen sample of training datapoints (called a mini-batch) and use these gradients to update the parameters. After using a mini-batch we chose another from our training set, that contains different datapoints, and update the parameters again. This process is then repeated until we exhaust the training set. As a result, many parameter updates are happening during an epoch and training is much faster.

Another way we can modify gradient descent to speed up training is to change how we update the parameters after we have computed the gradients. There have been numerous update rules proposed throughout the years and they can generally be classified in three categories: momentum updates, annealing updates and adaptive updates.

There are cases where by using the simple update rule, instead of moving directly towards the minimum of the cost function, we oscillate slowly around it until we reach it. Momentum updates solve this problem by incorporating the concept of momentum or velocity in the update rule. As we move toward the minimum we gather momentum and therefore any gradients that makes us change direction matter less than those that do not. Nesterov momentum is one of the most used momentum updates (Nesterov, 1983).

If we use a large learning rate, then there is the danger of passing over the minimum as we get closer to it. If we do, then the next update will make us move in the opposite direction in order to reach it and we might pass over it again and as a result we might end up oscillating over it. Annealing updates do not have this problem. These updates reduce the learning rate as time passes and so we end up taking smaller and smaller steps towards the minimum. A simple way to do that is to introduce an exponential decay to the learning rate, so that at epoch t it becomes $\eta_t = \eta_0 e^{-kt}$, where η_0 is the initial learning rate and k is an adjustable hyperparameter.

In all the update rules described so far, the same learning rate is used for all the parameters. Adaptive updates, however, tune the learning rate of each parameter individually.

The most popular adaptive updates are Adagrad (Duchi et al., 2011), RMSProp (Tieleman and Hinton, 2012) and Adam (Kingma and Ba, 2014). These updates are usually the ones that work best in practice.

2.3.3 Training Issues

In the previous section we described how neural networks are trained and tried to solve some of the issues that arise. These are in a sense minor issues that either have been partially solved or mitigated in a acceptable degree. There exist, however, some other ones that pose a greater threat to neural network training. In this section we will discuss two of those issues: unstable gradients and overfitting.

The issue of unstable gradients stems from the way backpropagation works. Generally speaking, it first calculates the gradients of the parameters of the last layer and then uses these to calculate the ones of the second-to-last layer. Then these new gradients are used to calculate those of the third-to-last layer and the process is repeated until we reach the first layer. This is also the reason why it is called backpropagation. It propagates the calculation of gradients from the back of the network to the front. In the end, the gradients of earlier layers may end up containing products of terms from later layers. From this we get two versions of the unstable gradient issue: vanishing and exploding gradients.

Vanishing gradients, which is the most commonly encountered version of the issue, result from many of those terms in the product being very small. Subsequently, the gradients of earlier layers become themselves very small, which hinders their ability to learn. If our network is too deep it is quite common for early layer to get stuck during training. Very rarely we might also encounter exploding gradients. This when the exact opposite happens. The gradients of early layers get too large the later layers are the ones getting stuck. The issue of unstable gradients is most commonly resolved by controlling the depth of the network or by a well-chosen activation function. Earlier we also mentioned that recurrent neural networks have difficulties learning long-term dependencies and vanishing gradients is one of the main reasons why (Hochreiter et al., 2001).

Overfitting is another one of the major issues we encounter during the training process of a neural network. It happens when a network learns the training set in such a degree

that it can no longer generalize well to unseen data. Instead of learning the underlying general structure of the data, it ends up learning specific traits or noise that only exist in the training set. In a sense we could say that it does not learn but instead memorizes. Overfitting is more prominent in neural networks with a large number of parameters, since those are the ones with the most degrees of freedom and can approximate a larger range of phenomena.

The low-level reason for why overfitting happens is weights with extreme values. When a feature of the datapoints has an extreme weight associated with it, it becomes much more important than all the others when we make a prediction. This leads to our network depending too much on it when in reality it might be just noise or a peculiarity of the training set.

Generally, the best thing we can do to mitigate the damage of overfitting is to enrich our training set with more data. However, in most cases this is not possible due to data being expensive or difficult to acquire. Fortunately, there is another way we can solve the issue, namely regularization techniques. As the name suggests, the techniques try to control the training process in order to avoid overfitting. While there are many of them, we will discuss two of the most commonly used ones: L2 regularization and dropout.

L2 regularization tries to solve overfitting by restricting the size of the weights during training. It does that by introducing a regularization term to the cost function. Assuming we have a cost function C, we get the new regularized cost function C_{reg} by adding to C the sum of the squares of all weights in the network S, scaled by a constant λ which is called the regularization strength or penalty ($C_{reg} = C + \lambda S$). This way, if any of the weight gets too large, C_{reg} will also get larger, which is the opposite of what we want, since we are trying to minimize the cost function. As a result, our neural network prefers to learn small weights, with larger ones being allowed only if they improve the first term (C) of the cost function (C_{reg}) more than they do the regularization term. How much more is controlled by the the regularization penalty λ, with larger λ resulting in smaller weights. We should also mention that the biases are usually not included in the regularization term. This is mainly due to the fact that large biases do not make the network sensitive to specific features as much as large weights and so we do not need to worry that much about them. Small weights have the appealing property of making the network use the majority of the data features a little instead of specific features a lot. Therefore it is more difficult for it to learn noise or peculiarities controlled by a

few features.

The second method we will discuss, dropout (Hinton et al., 2012b; Srivastava et al., 2014), works in a completely different way than L2 regularization. Instead of acting on the network weights, it acts on the network structure itself. Before we calculate the gradients to make the parameter updates we randomly and temporarily delete a certain set of units from the hidden layers of the network, with each one having a certain probability p of being kept. With $p = 0.75$, for example, we will delete about a quarter of the units.

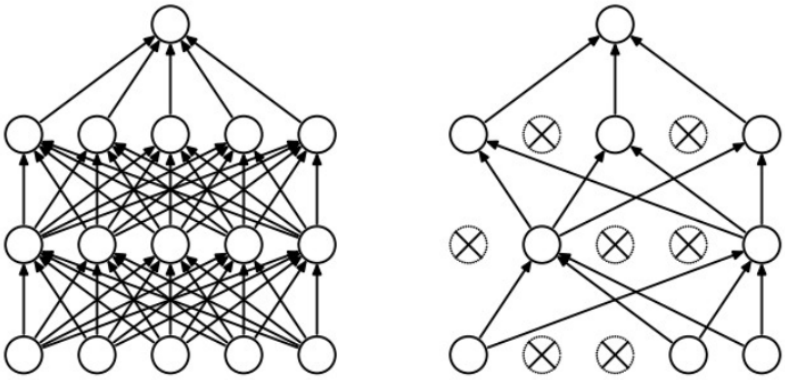

Figure 2.10: An example of dropout.

Source: Srivastava et al. (2014)

We then calculate only the gradients for those units that were kept and update only them. Then, before the next update we restore the previously deleted units and delete a different set. We repeat this process throughout the whole training session.

We can think of dropout in a different way which will also give some insight on why it helps with overfitting. In a sense we can imagine this procedure as training a different neural network at each training step, with all of them using the same data, and in the end averaging their predictions. Each of these networks is overfitting in a different way and hopefully, the net effect of the averaging will be to minimize the overfitting of the original network.

Chapter 3

Experimental Setup

In this section we will present our experimental setup, starting with a more detailed description of the task a hand. We will provide information about the datasets we will use, as well as the technical details of our environment. This will be followed by a brief discussion about hyperparameter optimization. Finally, we will describe the experiments we conducted, which are split in three categories: architectural, regularization and dataset experiments.

3.1 Task

The task at hand is to investigate whether neural networks can learn simple modular arithmetic, i.e. congruence relations of the form $n \equiv r(mod\, p)$, where r is the remainder of the Euclidean division between n and p. While we could treat this as a regression problem, we chose classification because we are dealing with integers. Regression is mainly used when we want to predict real continuous values. Classification seems much more fitting when we are dealing with distinct entities, in our case remainders. In mathematical terminology, the set of all integers n that have the same remainder when divided by a modulus p, is actually called a residue or remainder class.

We will tackle the problem in two different ways: as a multi-class and as a binary-class classification problem. A multi-class network will have to classify an integer between all possible remainders when divided by a modulus p, while a binary-class network will only need to decide between whether the remainder is equal to 0 or not. While with multi-class networks we demand complete information about the remainder of an

integer, with binary-class networks we only need information about a single remainder. This might seem less powerful but actually it is not. Knowing which integers have remainder 0 when divided by a modulus p is equivalent to knowing which integers have remainder $1, 2, ..., p-1$. This is due to the following property that congruence relations have: $n \equiv r(mod\, p) \Rightarrow n + k \equiv r + k(mod\, p)$ for any integer k. Assuming we have a network that can answer correctly whether the remainder is 0 or not, then the question of whether an integer n has remainder $1, 2, ..., p-1$ when divided by a modulus p is equal to whether $n-1, n-2, ..., n-(p-1)$ has remainder 0. For this reason, as well as the fact that trying to learn only one remainder is less demanding information-wise, we chose to also rephrase the task as a binary-class problem instead of just a multi-class one.

3.2 Datasets

For our experiments we will be using two datasets of randomly generated 128-bit integers: a training set of 1 million integers and a validation set of 100 thousand integers. These dataset sizes were mainly chosen due to time and memory constraints. Depending on the representation we choose for each network, each integer is converted to a feature vector of binary or decimal digits. The feature vector that will be given as input to the networks has a size of 128 for binary and 39 (the longest 128-bit integer) for decimal representation. Depending on the modulus, the label of each feature vector is set to the remainder that is produced when we divide the corresponding integer by that modulus.

Due to our work being investigative in its core and does not aim to improve performance or find the best hyperparameter settings, we will not have to make choices based on the validation set and therefore there is no danger of training the hyperparameters on it. For this reason, we chose not to use a test set, but rather let the validation set play that role.

We also wanted for this task to be quite challenging for our networks, which is why we chose 128-bit integers. They are large enough to ensure a huge diversity of data and small enough to deal with our memory constraints.

3.3 Metrics

During training we will monitor four quantities: the error and the accuracy on both the validation and training sets. The error is the raw value that is produced by our cost function and the accuracy is the percentage or correct classifications. The validation accuracy (usually the test accuracy but as we already mentioned we will not be using a test set) is the standard way of assessing the performance of a neural network and the error will be used to track the learning process as well as potential overfitting. Our networks will be trained for 50 epochs and these quantities are all checked at the end of each epoch.

3.4 Technical Details

For the implementation of our neural networks we used Python, which is widely used in machine learning due to its huge variety of libraries that can support such tasks. We used NumPy for its extensive mathematical capabilities and TensorFlow for building and training our networks. There are other libraries for machine learning tasks, like Keras and Theano, but we chose TensorFlow due to familiarity both with how it works and how it can be set up for GPU processing. As for the versions, we are using Python 3.5.2 and TensorFlow 0.12.1 with GPU support.

3.5 Hyperparameter Optimization

The weights and biases of a neural network constitute its parameters. These are adjusted during the training process in order to find the optimal configuration. However, there is a second set of parameters that are not optimized this way. They are called hyperparameters and consist of things like the size of the network, the learning rate, the activation function, the L2 regularization penalty and others. Tuning these hyperparameters is very frequent in neural networks and in machine learning generally, and can be a very time-consuming process. The bad news is that there is no one optimal setting (or an optimal way to find one) for these and they are usually task-dependent. This is why there has been also great interest in automating that process (Snoek et al., 2012). They also provide a great summary of the problem:

> ”The use of machine learning algorithms frequently involves careful tuning of learning parameters and model hyperparameters. Unfortunately, this tuning is often a ”black art” requiring expert experience, rules of thumb, or sometimes brute force search.” (Snoek et al., 2012)

Having said all that, we will now provide justification for some of the hyperparameters that are common to all our experiments. These values will be used in all of them unless otherwise specified. Justification for hyperparameters specific to certain experiments will be given in their corresponding sections. The choices we made for these hyperparameters are a product of preliminary experimentation, evidence from literature, previous experience in the training of neural networks and reasoning.

- **Representation**: Our networks will be tested on both decimal and binary representation of our data. Decimal is the most logical choice since in this system we already have some established rules to easily find the remainder (e.g. when dividing by 10 the remainder is equal to the last digit). However, we have decided to use a binary representation as well. Results from (Siu and Bruck, 1990; Siu et al., 1993; Franco and Cannas, 1998), where neural networks are used as circuits and are shown to be able to compute simple arithmetic operations, are all based on the binary representation of integers. Its usage in our experiments, therefore, might prove fruitful.

- **Output Layer**: The output layer of all our networks will have a size equal to the number of classes corresponding to a particular experiment. For a multi-class experiment trying to classify an integer when divided by a modulus p, its size will be equal to p (one class for each remainder) and for the same binary-class experiment its size will be 2 (one class for 0 and one for the rest of the remainders). We can think of the output layer as a vector with elements representing the score of each class. The element with the highest score is the one determining the prediction. In addition, in order to have a more interpretable result as a probability distribution over classes, we will also use a softmax activation function on the output layer.

- **Activation Function**: For our activation function we will be using ReLUs. These type of units are quite popular with neural networks and have been shown to speed up training (Krizhevsky et al., 2012) when compared to sigmoid or tanh units. An advantage ReLUs have over them, is the fact that ReLUs do not cause gradients to saturate. In contrast, when the input of a sigmoid or tanh unit be-

comes too large or too small the gradients very quickly reach values close to 0. For this reason, ReLUs can also greatly help to mitigate the issue of vanishing gradients.

- **Cost Function**: Since this is a classification task we will chose cross-entropy as our cost function instead of mean squared error. As we have already mentioned cross-entropy has been found to achieve faster training and better results (Golik et al., 2013), which have made it more popular than mean squared error in recent years. Cross-entropy is also better when used in conjunction with a softmax output layer.

- **Optimizer**: We will train our networks with stochastic gradient descent along with Adam adaptive learning rate updates. Due to their nature, adaptive updates can lead to faster and more flexible training. Adam has been shown to slightly outperform other adaptive updates (Kingma and Ba, 2014).

- **Learning Rate**: The learning rate is a hyperparameter that has to be tuned by hand most of the time. Usual values that have been shown to perform well, include negative powers of 10 and multiples of them. Preliminary experiments showed that with a learning rate of 0.001 (which is also the default in TensorFlow's Adam implementation) several networks had difficulties learning even the training set. This is likely due to the fact that we are taking too large steps when we adjust the parameters and end up missing optimal configurations. With 0.00001 the networks were learning but it was a bit slow. We finally settled with 0.0001 as a middle ground.

- **Batch Size**: We will be using a batch size of 100. Sizes of that degree are very common in literature. A small batch runs the risk of not being representative of the whole dataset, while larger batches can result in very few parameter updates and can slow down learning. Our choice is a compromise between these two and at the same time deals with our memory constraints.

- **Parameter Initialization**: We initialize the weights of our networks randomly according to (Glorot and Bengio, 2010), where it is recommended to use a normal distribution with mean equal to 0 and variance equal to $2/(n_{in}+n_{out})$ where n_{in} and n_{out} are the number of units in the previous and next layer respectively. The biases are initialized to 0.

3.6 Architectural Experiments

Here we will describe the first set of experiments we conducted. They are mainly focused on testing the capabilities of various networks with different architectural properties. We will begin with feed-forward networks and continue with convolutional and recurrent ones. All the networks we will describe in this and future sections will be used in both the multi-class and the binary-class problem.

We decided to also test convolutional and recurrent architectures because in a sense an integer can also be viewed as a 1-D image or as an ordered sequence of digits. Convolutional networks can deal with the first view and may be able to learn rules that determine the remainder based on local features, like a few neighbouring digits. Recurrent networks deal with the second view and they may be able to learn how much the remainder of an integer is dependent on each of its digits as a sequence.

3.6.1 Feed-forward

Our initial experiments, consist of a simple architectural search on feed-forward networks. We trained networks of 2 and 3 layers with 256 and 512 units per layer. The main inspiration for the number of layers was (Siu et al., 1993). There it is shown that a feed-forward network with 2 layers is able to compute the modulo operation. But as we have already said the weights in these networks were set by hand. This does not mean that networks of that depth can also learn them. Therefore, we decided to also add a third layer to increase their flexibility. Above that, preliminary experiments showed that the networks were barely learning and we also run the risk of vanishing gradients. Networks with 128 units per layer did not perform any better than those with 256 units, so we chose the latter for flexibility. The few preliminary networks we also tested with 1024 units per layer showed extreme amounts of overfitting. We decided, for now, to only go up to 512 units and revisit these networks again when we trained our regularized networks.

Each of those networks was trained for ten different moduli, from 2 to 11. Among these we have moduli for which we can decide the remainder based on a few digits (2, 4, 5, 8, 10) and moduli for which the rules are more complex (3, 6, 7, 9, 11). Therefore, we have a range wide enough for the networks to be tested.

To avoid future confusion and avoid long names, we will also introduce the naming convention that will be used throughout this thesis to indicate each network. Here we will consider only feed-forward names and any additional conventions will be discussed in their respective sections. For each of our four architectural combinations we use the following names:

- **FF28**: 2 layers / 256 units per layer
- **FF29**: 2 layers / 512 units per layer
- **FF38**: 3 layers / 256 units per layer
- **FF39**: 3 layers / 512 units per layer

The FF is used to indicate that the network if feed-forward and after this first part there will be a dash followed by the letter B or D to indicate binary or decimal representation and a number indicating the modulus. For example, a feed-forward network with 2 layers, 256 units per layer trained on binary integers and modulus 3, will be named FF28-B3.

3.6.2 Convolutional

Guided by the results of our initial feed-forward tests, which were not very encouraging, we decided to test convolutional architectures only on modulus 3 to see if the performance could be improved. The four architectures we will be using and their initial notation are as follows:

- **C6**: 2C16-P-2C32-P-2C64-P
- **C8**: 2C16-P-2C32-P-2C64-P-2C128-P
- **C9**: 3C16-P-3C32-P-3C64-P
- **C12**: 3C16-P-3C32-P-3C64-P-3C128-P

In the descriptions above nCk stands for n convolutional layers with k feature maps and P stands for pooling layer. The C in the name stands for convolutional. The name will also be followed by dash and then a number indicating the size of the local receptive field and finally we will append B3 as described in the previous section (e.g. C6-3-B3). Below we discuss the rest of hyperparameters.

- **Network Depth / Number Of Filters**: As a starting point for our architectural choices we used VGGNet (Simonyan and Zisserman, 2014), which was the runner-up in the ImageNet image recognition competition in 2014. Its input is much larger and its datasets are approximately the same size as ours, so we tried to scale accordingly and ended up with the networks presented above. Preliminary experiments showed that increasing either the number of layers or feature maps per layer did not improve performance significantly.
- **Local receptive fields / Strides / Padding**: For these we will be using some of the settings that have been found to perform well in literature. We will try both 3x3 and 5x5 local receptive fields with a stride of 1, as well as zero padding. With these local receptive fields we can start recognizing the smallest features and as we progress the network will have a wider view of the input. A stride of 1 and zero padding are mainly used in order for the convolutional layers to preserve the size of their input, so that we will only concern ourselves with reducing it in the pooling layers.
- **Pooling**: Max-pooling layers will be used between some of the convolutional layers, but not after each one. We will add max-pooling only before a layer where we increase the number of feature maps, in order to control the spatial size of the integer representation and to compensate for the increase in parameters.
- **Fully-connected Layers**: For ease of comparison we decided to use 2 fully connected layers with 256 units per layer (as in FF28) after the stack of concolutional layers.

As a final note we should mention that since we do not deal with images but with integers, we have modified the convolutional layer to use 1-D convolutions instead of 2-D, to compensate for the loss of a dimension.

3.6.3 Recurrent

As with convolutional networks, we will only test recurrent neural networks on modulus 3 to see whether there is any increase in performance. We will not be using the general architecture of a recurrent network, but rather the LSTM version. LSTM networks are among the most popular ones and as we have mentioned in Chapter 2, they do not have some of the problems that the simple architecture has.

An issue that LSTM networks usually have is that they are slow learners and their training takes more time than in other networks. For this reason and due to time constraints, in these experiments we will use a learning rate of 0.001 instead of 0.0001 and a training dataset of 100 thousand integers instead of 1 million.

For ease of comparison with our feed-forward networks we will train LSTM networks with 2 and 3 layers but for the reasons mentioned above, as well as memory constraints we will halve the units per layer to 128 and 256.

Another hyperparameter we need to configure is how much we will unroll our LSTM networks. The amount we unroll will determine how far the dependencies go within an integer. For the decimal representation we will unroll our networks 3, 13 and 39 times and in binary we will unroll 32, 64 and 128 times. These values were chosen since they divide the length of our integers exactly and in order to test both short and long dependencies.

Following the previously mentioned notation we have the following network architectures:

- **R27**: 2 layers / 128 units per layer
- **R28**: 2 layers / 256 units per layer
- **R37**: 3 layers / 128 units per layer
- **R38**: 3 layers / 256 units per layer

Similarly, the R stands for recurrent and the names above will be followed by a dash along with a number to indicate the unrolling factor. At the end we will also append B3 or D3 (e.g. R27-32-B3).

3.7 Regularization Experiments

We previously mentioned that some of our feed-forward networks showed signs of extreme overfitting. Even some of our less flexible networks had considerable generalization errors (i.e. the difference between training and validation error). In this section we will present the regularized versions of some of our feed-forward networks. We will use two regularization techniques: L2 regularization and dropout. Again we will focus on modulus 3.

We have the following regularized networks:

- **FF29-(B/D)3-L4**: 2 layers / 512 units per layer / L2 regularization penalty 0.0001
- **FF29-(B/D)3-L5**: 2 layers / 512 units per layer / L2 regularization penalty 0.00001
- **FF210-(B/D)3-L4**: 2 layers / 1024 units per layer / L2 regularization penalty 0.0001
- **FF210-(B/D)3-L5**: 2 layers / 1024 units per layer / L2 regularization penalty 0.00001
- **FF39-(B/D)3-L4**: 3 layers / 512 units per layer / L2 regularization penalty 0.0001
- **FF39-(B/D)3-L5**: 3 layers / 512 units per layer / L2 regularization penalty 0.00001
- **FF310-(B/D)3-L4**: 3 layers / 1024 units per layer / L2 regularization penalty 0.0001
- **FF310-(B/D)3-L5**: 3 layers / 1024 units per layer / L2 regularization penalty 0.00001
- **FF29-(B/D)3-D90**: 2 layers / 512 units per layer / dropout keep probability 0.9
- **FF210-(B/D)3-D75**: 2 layers / 1024 units per layer / dropout keep probability 0.75
- **FF39-(B/D)3-D90**: 3 layers / 512 units per layer / dropout keep probability 0.9
- **FF310-(B/D)3-D75**: 3 layers / 1024 units per layer / dropout keep probability 0.75

Networks FF29 and FF39 were chosen since they were showing the most overfitting among the ones tested. We also decided to include some of our preliminary networks with 1024 units per layer (FF210 and FF310), where overfitting was most prevalent. Maybe by increasing the size of our networks while we regularized them, we could take advantage of the extra flexibillity.

As for the hyperparameter selection, preliminary experiments showed that regularization penalties over 0.0001 caused the regularization term to take over and the networks

were not learning at all, while those under 0.00001 had no significant effect. For similar reasons, we chose a high dropout keep probability (0.9) for our smaller networks and lower one (0.75) for our larger networks.

The naming convention is the same, but with an extra L or D at the end for L2 regularization or dropout respectively, followed by a number indicating their hyperparameter value.

3.8 Dataset Experiments

Our last set of experiments will consist of modifications on the training set. Some of our initial feed-forward networks managed to learn to classify integers correctly when dividing by 2, 4, 5, 8 and 10 but struggled with the rest. What these five moduli all have in common is that we only need to look at a few of the last digits of the integer in order to determine the remainder. Since, however, we are using very large integers all these combinations of last digits exist multiple times in the training set. What we wanted to see is whether our networks actually learned or just memorized the combinations. In order to do that we will take one of our initial feed-forward networks, FF29, and test it on training sets with missing data, while the validation set stays unmodified.

We will use two modified datasets:

- **LAST34**: In this dataset we will replace all the integers that end in 3 or 4 with ones ending in other digits. This way we equally reduce both the set of integers that have odd remainders and the set of integers that have even remainders, without removing a whole remainder class. Incidentally, when dividing by 10 this does not hold since we end up removing these two remainder classes. We should also mention that this is only possible with a decimal representation due to it having multiple different digits. With a binary representation we can only remove integers ending in 0 or in 1, which would result in removing a whole remainder class. Therefore, we will only use this set with a decimal representation.
- **REM3**: In this dataset we will replace all integers that have remainder 3 when divided by 4 with ones that have one of the other remainders. Here, we are actually removing a whole remainder class. This dataset will be used with both binary and decimal representations and only in conjunction with modulus 4.

Having said all this, on the set LAST34 we will train the network FF29 with a decimal representation for moduli 2, 5 and 10. On the set REM3 we will train FF29 with both binary and decimal representations for modulus 4.

The few experiments we will present now only apply only to the binary-class task. In the multi-class task all classes will be equally represented in the training set. In the binary-class task, however, this is not true since for all moduli except 2, the integers that have remainder 0 are less than those who do not. For this reason we will create three more datasets, SKEW25, SKEW50 and SKEW75 that will be used to test the network FF29 with both binary and decimal representations for modulus 3. The distribution in these datasets has been skewed to only contain 25%, 50% and 75% of integers that have remainder 0 when divided by 3 respectively.

Chapter 4

Results

In this chapter we will present the results that were produced from our networks, first those of the multi-class task and then those of the binary-class task. They will be presented through tables, line graphs that track both the training and validation accuracy during training and confusion matrices. These matrices are commonly used in classification tasks to give us a better idea about correct and wrong classifications. Its lines represent the true labels of the data and its columns represent the prediction of our network. Each cell, then, shows us how many datapoints had a certain class as a true label and were classified in another class (possibly the same) by our network.

4.1 Multi-class Task

4.1.1 Initial Feed-forward Search

In Table 4.1 we have the final performances of our networks with the moduli split in four categories depending on certain properties they exhibited.

Moduli 3, 7, 9 and 11 behaved very similarly, regardless of representation. For each of them, none of our networks managed to achieve better validation accuracy than their respective random chance baseline. All oscillated around it, with changes only in the third decimal place. In addition, as the modulus increased, learning slowed down. Figures 4.1 and 4.2 show the training and validation accuracy for modulus 3. The graphs for the rest of the moduli in this category are all very similar to these.

Modulus	Binary Training Accuracy	Decimal Training Accuracy	Binary Validation Accuracy	Decimal Validation Accuracy	Random Chance Baseline
3	45%-70%	40%-50%	33%	33%	33%
7	24%-45%	20%-30%	14%	14%	14%
9	19%-35%	16%-25%	11%	11%	11%
11	16%-34%	14%-20%	9%	9%	9%
2	100%	100%	100%	100%	50%
4	100%	100%	100%	100%	25%
8	100%	100%	100%	100%	12.5%
6	45%-70%	38%-42%	33%	33%	16.6%
5	30%-54%	100%	20%	100%	20%
10	30%-54%	100%	20%	100%	10%

Table 4.1: Feed-forward accuracies at the end of training.

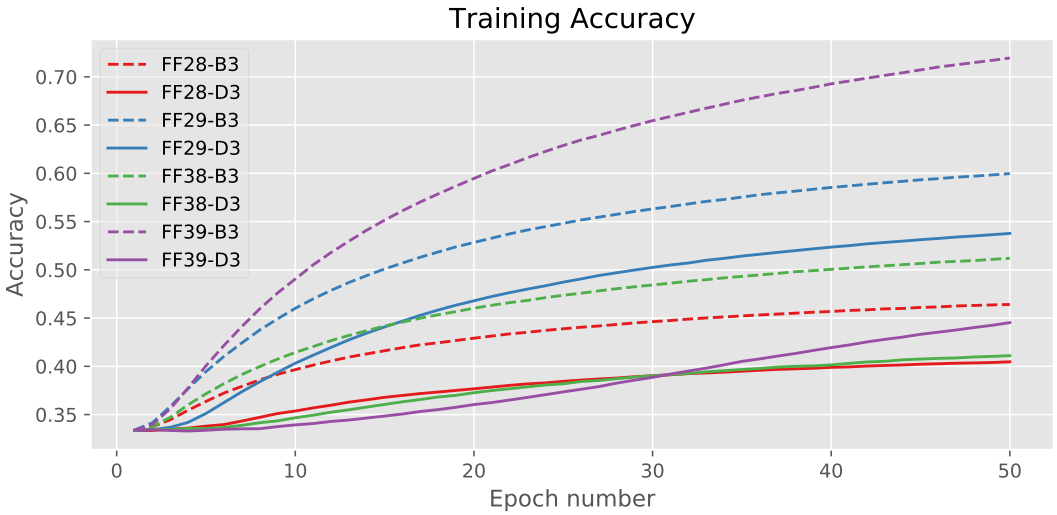

Figure 4.1: Training accuracy for networks trained on modulus 3.

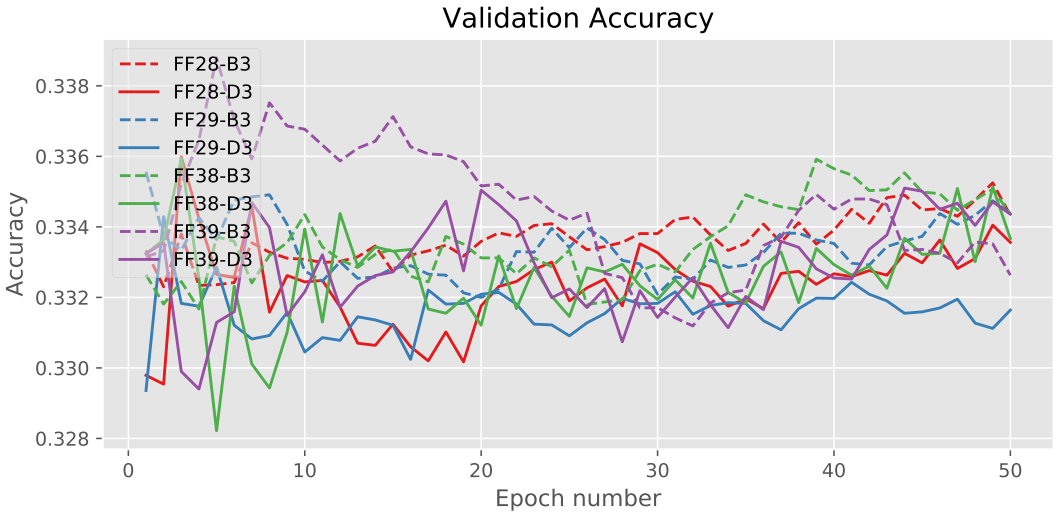

Figure 4.2: Validation accuracy for networks trained on modulus 3.

Next, we have moduli 2, 4 and 8. These are all powers of the binary system's base and can be also found in powers of the decimal system's base. All networks reached 100% validation accuracy. Networks trained with binary representation reached it almost immediately, while those with decimal representation did it after a few epochs passed, with the number of epochs approximately doubling as the modulus increased.

Modulus 6 is a special case itself. Our networks all had a validation accuracy oscillating around 33% instead of the corresponding random chance baseline which is 16%. Furthermore, a binary representation resulted in this accuracy being reached faster than a decimal representation. The validation accuracy graphs below show a similarity in the behaviour of our networks for moduli 2 and 6.

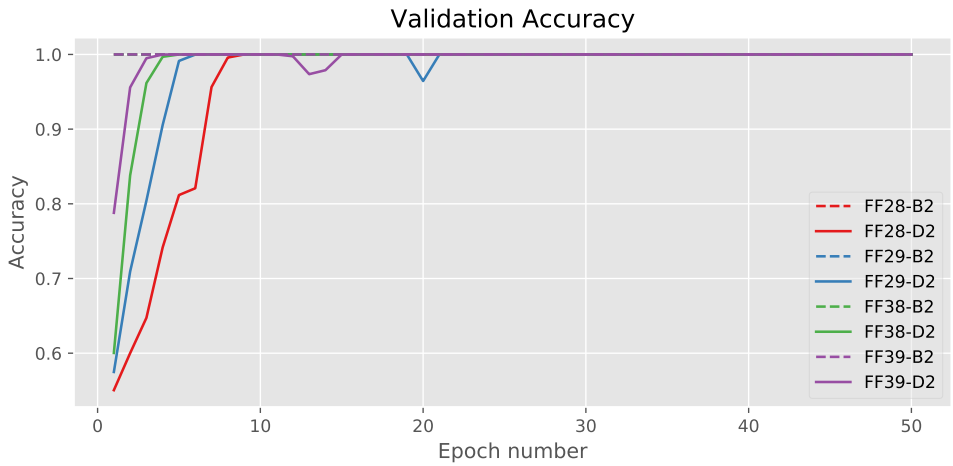

Figure 4.3: Validation accuracy for networks trained on modulus 2.

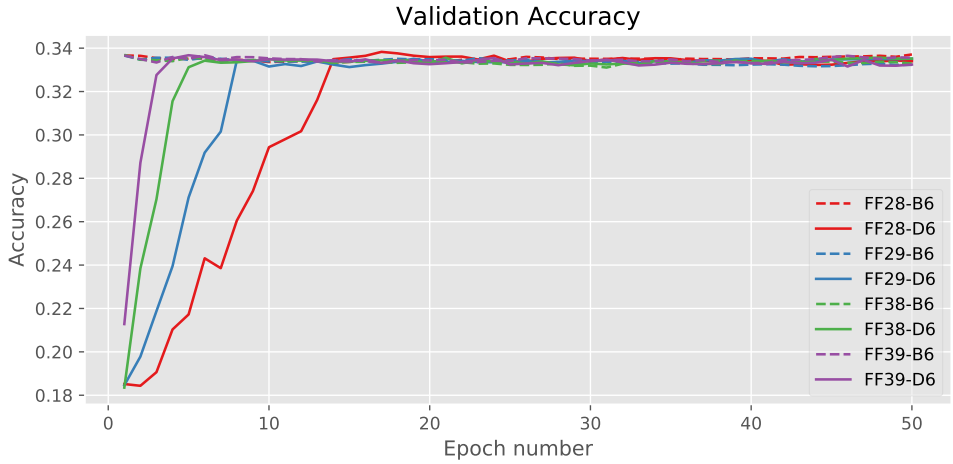

Figure 4.4: Validation accuracy for networks trained on modulus 6.

The following sample confusion matrix shows us that the networks have managed to differentiate between odd and even remainders.

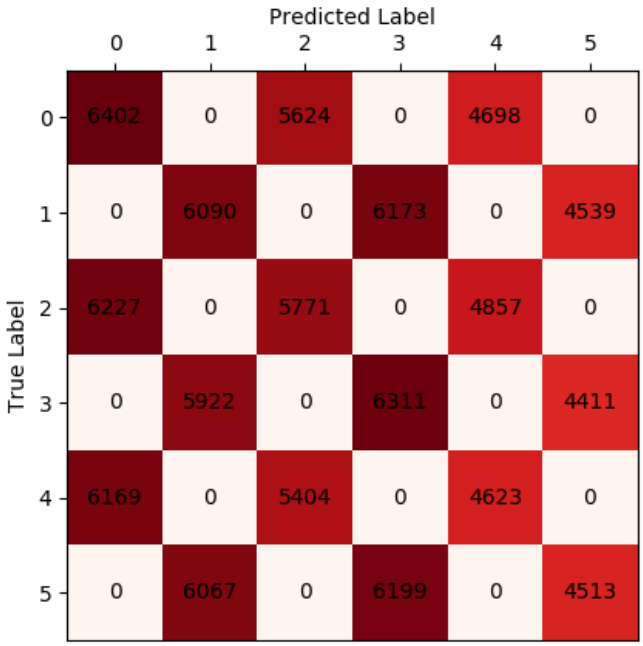

Figure 4.5: Confusion matrix for FF28-B6.

The last category consists of moduli 5 and 10. These are found in or are powers of the decimal system's base. Networks trained with decimal representation reached 100% validation accuracy on both moduli. However, this percentage fell to 20% for both moduli for binary representation. For modulus 5 this follows the pattern we saw previously, of networks not being able to perform better than random chance. But for modulus 10 this is better than its corresponding 10% random chance. Similarly to modulus 6, the network has managed to differentiate between odd and even remainders, which can be seen in Figure 4.6.

Finally, we will mention some general observations. In the vast majority of these experiments, larger networks reached higher training accuracy in the following order: FF39 $>$ FF29 $>$ FF38 $>$ FF28. Additionally, apart from moduli 5 and 10, a binary representation resulted in higher training accuracy. In the end, this increase in training accuracy did not lead to an increase in validation accuracy in most cases, which means that these networks suffered from overfitting.

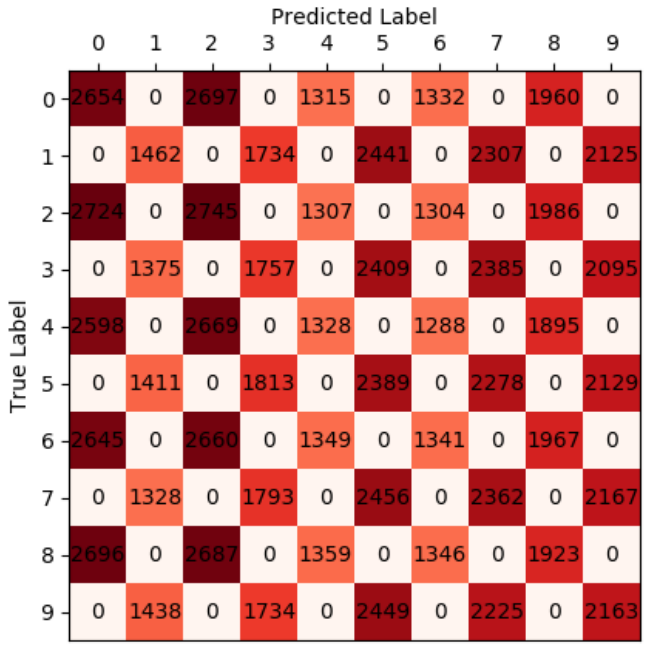

Figure 4.6: Confusion matrix for FF28-B10.

4.1.2 Specialized Architectures

The table below summarizes our findings for convolutional and LSTM networks. Neither of those architectures managed to improve the performance on modulus 3.

Architecture	Binary Training Accuracy	Decimal Training Accuracy	Binary Validation Accuracy	Decimal Validation Accuracy	Random Chance Baseline
Convolutional	52%-62%	33%-48%	33%	33%	33%
LSTM	Most around 35%	40%-80%	33%	33%	33%

Table 4.2: Specialized architecture accuracies at the end of training.

The validation accuracy of all our convolutional networks oscillated around 33% with changes only in the third decimal place, similarly to the feed-forward networks. Apart from that, the rest of our remarks are about training performance. Again, using a binary representation lead to higher training accuracy. In contrast with feed-forward networks, here the convolutional networks with the least layers had the highest training accuracy: C6 $>$ C8 $>$ C9 $>$ C12. As for the local receptive fields, there was no clear evidence of which is better.

Our LSTM networks did not perform any better. Still we have these oscillations around 33%. As for the training process, in these networks it was the decimal representation

that achieved higher training accuracy. Networks with less unrolling also had higher training accuracy than those with more unrolling. However, there was no clear evidence about which settings were generally better in terms of depth or width.

We should also mention that in order to make sure that the lack of improvement in performance was due to the modulus and not the architecture, we also ran some preliminary experiments for both convolutional and LSTM networks on modulus 2, in which they all reached 100% validation accuracy.

4.1.3 Regularization

The results from the regularized versions of the initial feed-forward networks, were not encouraging either, as seen below.

Technique	Binary Training Accuracy	Decimal Training Accuracy	Binary Validation Accuracy	Decimal Validation Accuracy	Random Chance Baseline
L2 0.0001	33%-47%	33%-42%	33%	33%	33%
L2 0.00001	60%-80%	35%-70%	33%	33%	33%
Dropout 0.75	53%	33%	33%	33%	33%
Dropout 0.9	53%	33%	33%	33%	33%

Table 4.3: Regularization technique accuracies at the end of training.

Regardless of regularization technique or representation, all the regularized networks still had a validation accuracy oscillating around 33%. The only difference was in the training performance which for the vast majority of regularized networks was worse than that of the best unregularized one. Regarding L2 regularization, a penalty of 0.0001 turned out to be more punishing and several networks that used it had only a small increase in training accuracy.

4.1.4 Data Modifications

The results from the modified are very interesting. All the networks we tested on the set LAST34 (FF29-D2, FF29-D5 and FF29-D10) reached almost immediately a training accuracy of 100% like they did in our initial experiments but now they only achieved a validation accuracy of 80%, which is the percentage of last digits (8 out of the possible 10) that remained in LAST34. This means that the integers ending in 3 or 4 were classified entirely incorrectly. If we assume that they are classified randomly then we have the following baselines: 90% (mod2), 84% (mod5) and 82% (mod10). These are

all higher than what our networks achieved. The following confusion matrices show in which classes these integers where incorrectly classified.

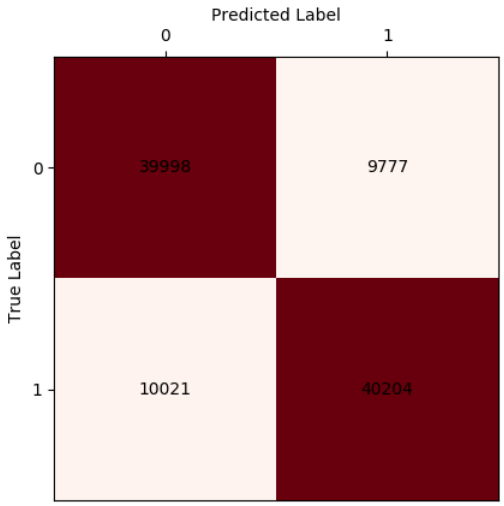

Figure 4.7: Confusion matrix for FF29-D2-LAST34.

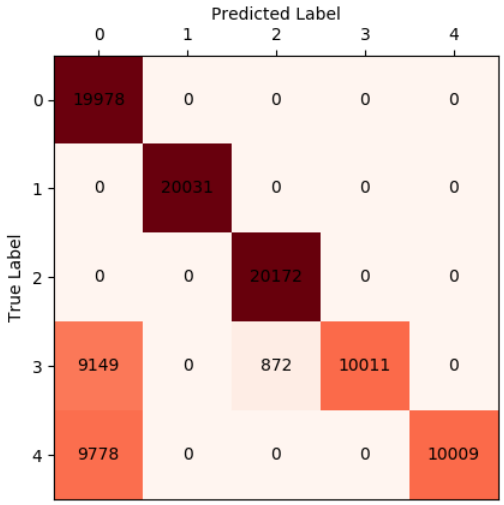

Figure 4.8: Confusion matrix for FF29-D5-LAST34.

For modulus 2, in order to have complete misclassification the integers ending in 3 need to be classified as having remainder 0 and those ending in 4 classified as having remainder 1, which is exactly what happened. We had similar results for modulus 10. For modulus 5, the majority of the unseen last digits were classified as having remainder 0. In Figure 4.8 we can also see that there were some integers classified as having remainder 3 or 4. These were the ones ending in 8 or 9 respectively.

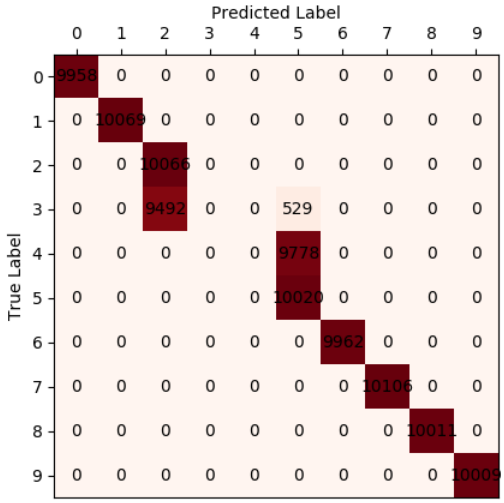

Figure 4.9: Confusion matrix for FF29-D10-LAST34.

We observed a similar behaviour in the network we tested on REM3 (FF29-D4). Both binary and decimal representations reached a training accuracy of 100% like before, but a validation accuracy of only 75%. This is the percentage of remainder classes (3 out of the possible 4) that still remained in REM3. Again, integers with a remainder of 3 were completely misclassified. Here, the baseline is 81.25%. Below we have the confusion matrix for both representations.

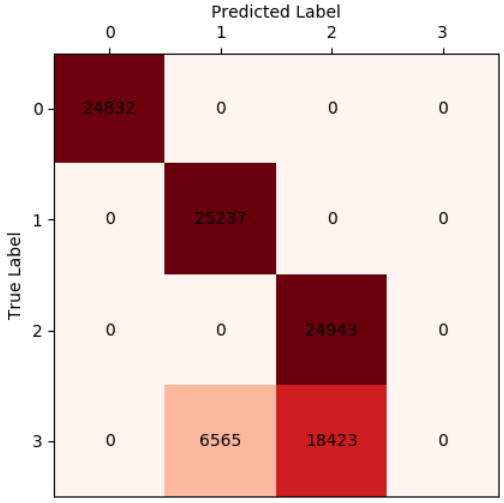

Figure 4.10: Confusion matrix for FF29-B4-REM3.

For the decimal representation the unseen integers were all classified as having a re-

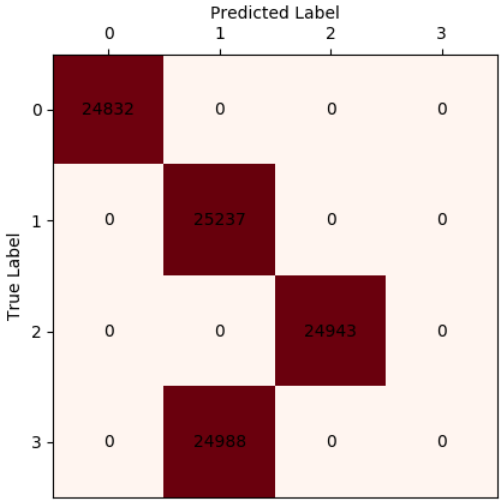

Figure 4.11: Confusion matrix for FF29-D4-REM3.

mainder of 1, which is the only other odd remainder, while for the binary representation they were split between remainders 1 and mostly 2.

4.2 Binary-class Task

4.2.1 Initial Feed-forward Search

Below we have the performances of our networks with the moduli split in three categories depending on certain properties they exhibited. Here as a baseline we will use a classifier that always classifies an integer as not having a remainder of 0.

Modulus	Binary Training Accuracy	Decimal Training Accuracy	Binary Validation Accuracy	Decimal Validation Accuracy	Simple Classifier Baseline
3	70%-91%	66%-76%	56%-60%	58%-65%	67%
6	87%-98%	84%-87%	76%-79%	79%-83%	83%
7	86%-96%	86%-89%	77%-84%	82%-85%	86%
9	89%-97%	89%-91%	79%-87%	85%-89%	89%
11	91%-98%	91%-93%	84%-90%	89%-91%	91%
2	100%	100%	100%	100%	50%
4	100%	100%	100%	100%	75%
8	100%	92%	100%	92%	87.5%
5	81%-95%	100%	69%-77%	100%	80%
10	92%-98%	100%	84%-88%	100%	90%

Table 4.4: Feed-forward accuracies at the end of training.

The first category consists of moduli 3, 6, 7, 9, and 11. Regardless of representation, all

the networks had an initial validation accuracy at the baseline which started decreasing as training progressed, in contrast with the multi-class task where their validation accuracy oscillated around the random chance baseline. As for the training accuracy, the vast majority of the networks surpassed 90%. Modulus 6 is also included in this category since it does not have the strange behaviour it exhibited in the multi-class task. Below we have the graphs for the training and validation accuracy for modulus 3. The graphs for the rest of the moduli in this category are all very similar to these.

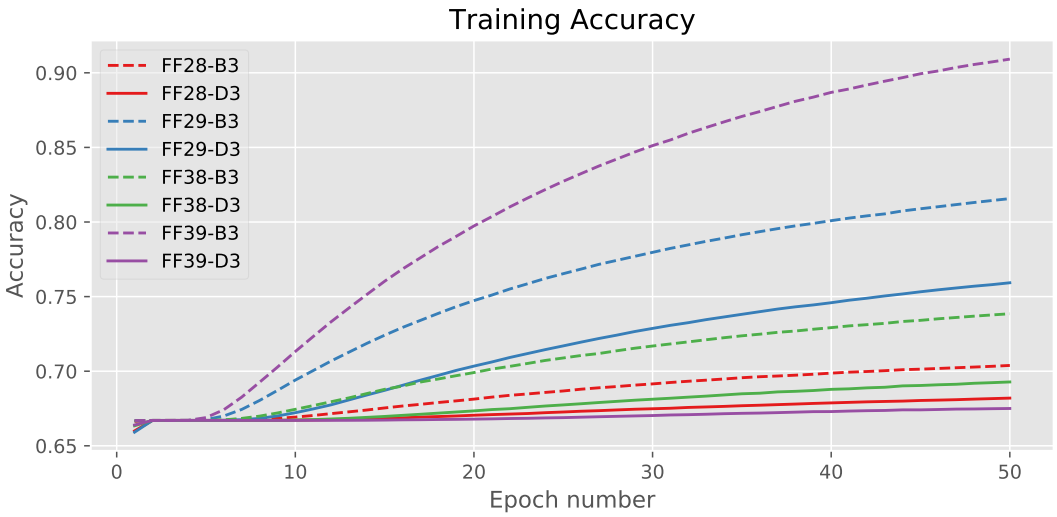

Figure 4.12: Training accuracy for networks trained on modulus 3.

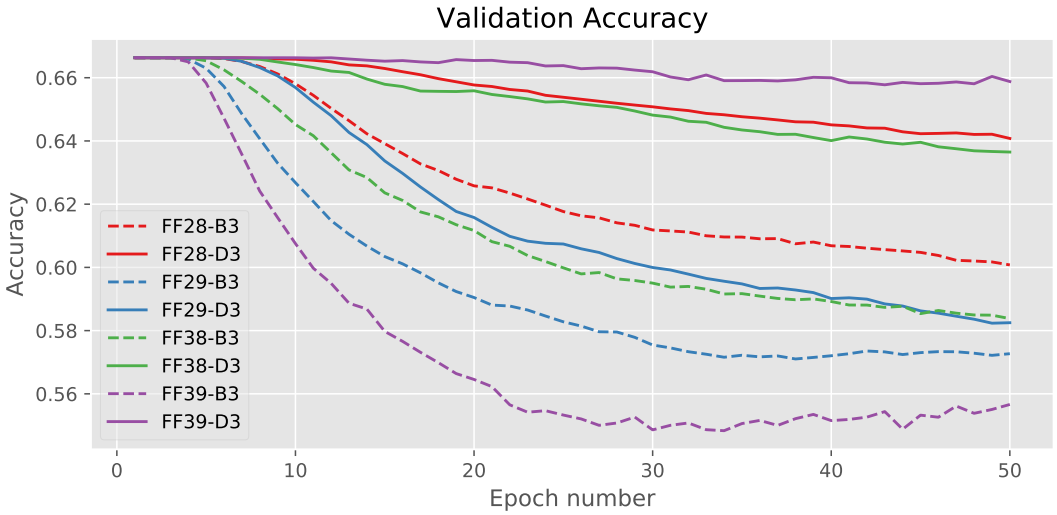

Figure 4.13: Validation accuracy for networks trained on modulus 3.

Again we have moduli 2, 4 and 8 which are grouped similarly to the multi-class task. Networks trained on modulus 2 and 4 all reached 100% validation accuracy regardless

of representation. Networks trained with a binary representation reached it almost immediately, while those with decimal representation did it after a few epochs passed. For modulus 8, however, only the binary representation resulted in 100% validation accuracy, in contrast with 92% which was the highest among networks with decimal representation.

The remaining moduli are 5 and 10. Networks with decimal representation achieved 100% accuracy, while the accuracy of those with a binary representation fell below the baseline, decreasing as training progressed. Similarly to modulus 6, modulus 10 did not exhibit any strange behaviour in the binary-class task.

The general remarks are the same as in the multi-class task. Larger networks reached higher training accuracy, as well as binary representation did for all moduli except for 5 and 10. Also, since in the majority of these networks, as the training accuracy increased the validation accuracy decreased, overfitting was more prevalent.

4.2.2 Specialized Architectures

The performance of convolutional and LSTM networks is presented below. Again, neither of those architectures was able improve the performance on modulus 3. The validation accuracy for both architectures started at the base line and then fell off similarly to the feed-forward networks.

Architecture	Binary Training Accuracy	Decimal Training Accuracy	Binary Validation Accuracy	Decimal Validation Accuracy	Simple Classifier Baseline
Convolutional	69%-82%	67%-73%	56%-63%	61%-66%	67%
LSTM	Most around 67%	67%-95%	Most around 67%	Most around 56%	67%

Table 4.5: Specialized architecture accuracies at the end of training.

Regarding the training performance of convolutional networks our remarks are similar to those in the multi-class task. Binary representation resulted in higher training accuracy, networks with less layers also had the same effect and none of the local receptive fields was strictly better.

For LSTM networks, decimal representation resulted in higher training accuracy than binary representation with less unrolling also having a similar effect. In terms of depth or width no setting was strictly better.

Finally, we also ran some preliminary experiments for both convolutional and LSTM networks on modulus 2 similarly to the multi-class task, in which they all reached

100% validation accuracy.

4.2.3 Regularization

As in the multi-class task, regularized networks did not improve performance at all.

Technique	Binary Training Accuracy	Decimal Training Accuracy	Binary Validation Accuracy	Decimal Validation Accuracy	Simple Classifier Baseline
L2 0.0001	67%-68%	67%	63%-67%	65%-67%	67%
L2 0.00001	79%-90%	79%-81%	54%-57%	59%-64%	67%
Dropout 0.75	73%	67%	45%-60%	64%-66%	67%
Dropout 0.9	73%	67%	45%-60%	64%-66%	67%

Table 4.6: Regularization technique accuracies at the end of training.

The validation accuracy, again, started at the baseline and kept decreasing as training progressed. Regarding the training performance, there was no regularized network that achieved higher training accuracy than that of the best unregularized one. Similarly to the multi-class task regularized networks, an L2 regularization penalty of 0.0001 slowed down learning much more than a penalty of 0.00001.

4.2.4 Data Modifications

Modified datesets, again, provide us with some interesting results. Similarly to the multi-class task the networks that were tested on LAST34 (FF29-D2, FF29-D5 and FF29-D10) reached a training accuracy of 100%. However, the validation accuracy for each modulo was different. FF29-D2 reached 80%, FF29-D5 reached 90% and FF29-D10 reached 100%. As a baseline we will use a simple classifier that always classifies the unseen data as not having remainder 0 , which in our case will have a validation accuracy of 90% (mod2), 100% (mod5) and 100% (mod10). Apart from FF29-D10, the other network did not reach these. The confusion matrices that show the misclassifications can be seen in Figures 4.14, 4.15 and 4.16.

For modulus 2 the results were exactly the same as in the multi-class task. For modulo 10 there were no misclassifications and for modulus 5 only the integers ending in 4 were misclassified.

In contrast with the multi-class task, the network we tested on REM3 (FF29-D4) not only reached 100% training accuracy, but also achieved a validation accuracy of 100% for both representations, despite the decimal one doing this much slower.

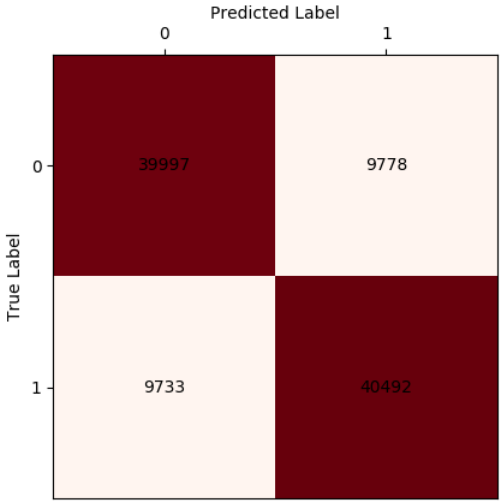

Figure 4.14: Confusion matrix for FF29-D2-LAST34.

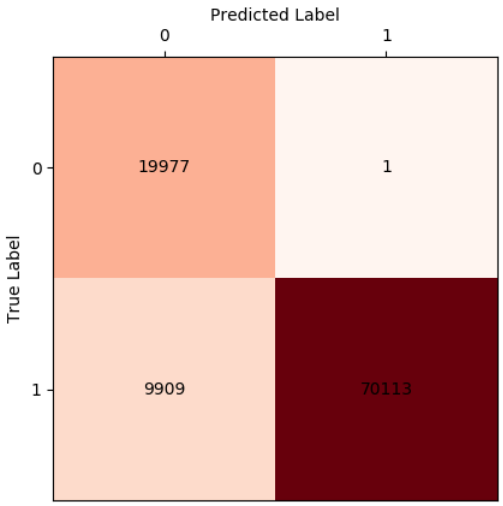

Figure 4.15: Confusion matrix for FF29-D5-LAST34.

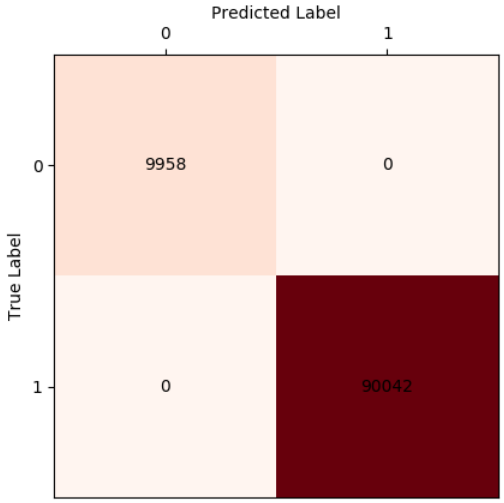

Figure 4.16: Confusion matrix for FF29-D10-LAST34.

Finally, we have the experiments on the SKEW datasets.

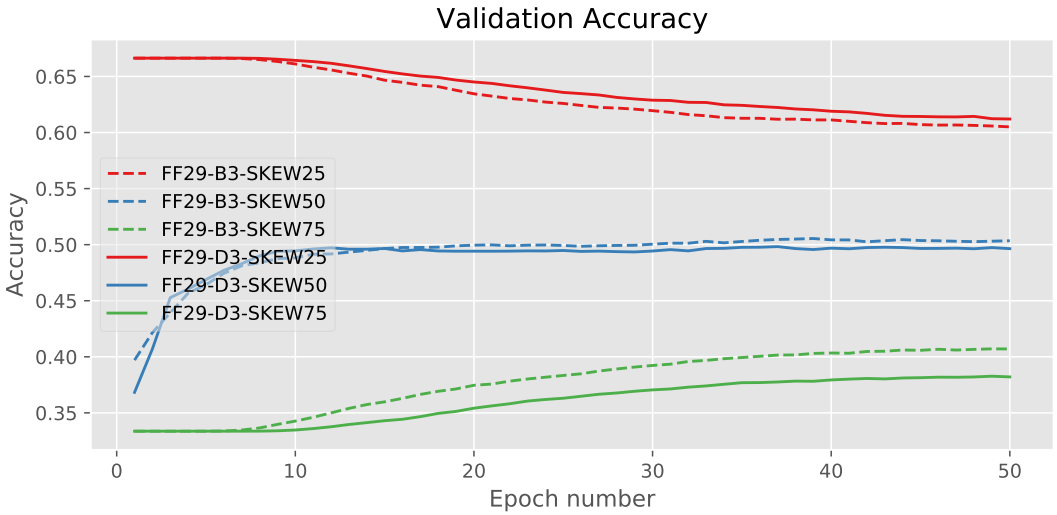

Figure 4.17: Validation accuracy for FF29-(B/D)3 trained on the SKEW datasets.

As we can see from the graph, the networks (FF29-B3 and FF29-D3) behave very differently in each of the datasets. In SKEW25 the validation accuracy starts at 66% and slowly falls down to 60%, in SKEW50 it starts at 37% but after a while it stays at 50% and in SKEW75 it starts at 34% and goes up to 40%. It seems as though networks trained on SKEW25 and SKEW75 converge towards those trained on SKEW50. Apart from that, the differences between representations are negligible.

Chapter 5

Discussion

In this section we will discuss the results that were produced from our experiments. We will do that in the context of the breakthroughs that neural networks have achieved and try to explain why our networks have not performed as well. We will offer comments separately for each batch of experiments, as well as a general discussion on the issues at hand.

5.1 Initial Feed-forward Search

With only a few exceptions that will be mentioned in a while, in both the multi-class and binary-class tasks, for moduli 3, 6, 7, 9 and 11 the networks did not manage to perform any better than random chance or a very simplistic classifier respectively. In the binary-class task especially, not only did they not improve but rather performed worse than the baseline of always classifying an integer as having a non-zero remainder.

In contrast, they were all able to learn perfectly moduli 2, 4, 8, 5 and 10. Networks trained on a decimal representation learned all of them, while those trained on a binary representation only learned 2, 4 and 8 and were no better than the baselines for the rest. Essentially, each representation learned the moduli that are found in or are powers of its base.

The exception we mentioned earlier refers to the behaviour exhibited by networks trained for modulus 6 on both representations and for modulus 10 on binary representation and was only observed in the multi-class task. In each of these cases, the

validation accuracy of the networks instead of oscillating around the random chance baseline (16% and 10%) it did that a bit higher (33% and 20%). What these cases have in common is that their prime factorization (2*3 and 2*5) contains a factor that could be learned (2) and one that could not (3 and 5). The higher validation accuracy that the networks reached in each case corresponds to the baseline of the unlearnable factors. This indicates that when a modulus contains a learnable factor the networks can at least make a partial classification. We should also mention that this behaviour was not exhibited in the binary-class task since there we are only targeting a single remainder (0).

There is a huge difference in the behaviour of the networks on each of these categories of moduli. As we have already mentioned the moduli in the second category are all found in or are powers of a numerical system's base. What they have in common is that the rule for determining the remainder is very simple. For moduli 2^n, 5^n and 10^n, we only need to look at the last n digits of the integer. It seems as though our networks are learning moduli for which the information needed to determine the remainder is localized.

5.2 Specialized Architectures

We initially chose to test convolutional and LSTM networks, with the aim of hopefully improving the performance even a little. These specialized architectures fare very well in a variety of areas that simple feed-forward networks do not, most notably image recognition and natural language processing. It could be possible that they were able to identify features of an integer in a way similar to an image or a sentence. As we saw in the previous chapter, however, they were unsuccessful.

Regarding convolutional networks, if they can even improve performance at all, then one thing that might hinder that improvement is the pooling layer. These layers are used in order to downsize the convolutional layers as the networks gets deeper. If we think for a while in the context of image classification, the effect they have is to throw out the exact positional information of a feature on an image while retaining its general location. In this case, this is beneficial, since this kind of information is not important to make a prediction and sometimes it might even lead to overfitting. In a task like ours, in contrast, where our predictions are very closely tied to all the digits of an integer,

this loss of precision is problematic.

As for LSTM networks, they did not manage to improve performance either. However, due to the specific circumstances of our experiments we cannot draw any concrete conclusions. Since LSTM networks are more difficult to train, we chose to use a smaller dataset and a larger learning rate and even then there were networks that barely learned the training set. Their inability to learn might as well be a result of those choices. While, based on our results so far, we are not optimistic, it is possible that different configurations might lead to better results. If there are such configurations, it should be expected that networks with an unrolling factor equal to the length of the integer will perform better, since they will preserve dependencies over all digits.

5.3 Regularization

In both multi-class and binary-class tasks all the networks, apart from those trained for moduli that are found in or are powers of a numerical system's base, suffered from overfitting. While the training accuracy increased, the validation accuracy stayed at the same level as the baseline and did not improve at all. In the binary-class task, particularly, it fell below the baseline with higher training accuracy leading to lower validation accuracy.

Regularization techniques, like L2, dropout (Srivastava et al., 2014) and others (Zaremba et al., 2014; Wan et al., 2013; Zou and Hastie, 2005), usually work very well in practice by improving the performance of the networks due to the reduced overfitting. However, here, this is not the case. Putting aside the networks that did not even learn the training set due to the applied regularization being too strong, the only effect it had was to slow down learning in the rest.

The question then is, what is the reason behind this behaviour. A possible explanation is that the problem stems from the nature of our task. Regularization is mostly used in order for the network to be able to learn certain general patterns that exist, for example, in an image or in the structure of sentences, without learning noise or very specific peculiarities that they may possess. In an image of a dog, we do not need to learn exactly how far apart its ears are, just their general location and in the sentence "I went to the store and bought milk" we do not need to learn that the last word is exactly milk, just that it is the object of the sentence and can be found in a store. In contrast, our

task needs this exactness. The remainder of an integer is fully determined by all its digits for the vast majority of the cases, apart from when the modulus is found in or is a power of a numerical system's base. Due to that and since our networks were not even learning in the first place, regularization did not have any positive effects. Having said this, it might still be possible for it to be of assistance if any future unregularized networks show improvement in performance but also suffer from overfitting.

5.4 Data Modifications

Previously we mentioned that our networks seem to learn moduli for which we only need the last few digits to determine the remainder. However, in our experiments we never got further than 3 last digits (mod8). The possible combinations that can appear in these are only 1000 for decimal representation and 8 for binary. The training set we used contained 1 million integers, which means that all these possible combinations will appear multiple times in it. The question then is, whether the networks are actually learning or just memorizing the combinations, which is why we also ran experiments on the modified datasets LAST34 and REM3.

In the multi-class task, both the networks tested on LAST34 and those tested on REM3 completely misclassified the unseen data, managing to perform worse even than random chance. Additionally, there was no consistence in the misclassification.

In the binary-class task, from the networks trained on LAST34, FF29-D2 performed exactly the same as in the multi-class task, which is not surprising since in this case the tasks are exactly the same. FF29-D10 had no misclassifications. We could say that it learned that only integers ending in 0 have a remainder of 0, so when it encountered the unseen data it just classified them in the other class. However, we cannot say that for sure. Maybe it just got lucky with its particular parameter configuration and had some accidental correct classifications. If we look at the results of FF29-D5, we will see that this is the more likely explanation. If we apply the same reasoning here, all the unseen data should have been correctly classified but this is not the case. The unseen integers ending in 3 were correctly classified but those ending in 4 were not. For similar reasons, we should also not say that the networks trained on REM3 were successful. In the end, it seems that our networks are quite data-dependent and seem to be memorizing which integers go to each class instead of learning.

Furthermore, we should not draw any conclusions about why these specific misclassifications happen. Unfortunately, in the parameter space of a neural network there are many minima and which one we end up with can be in part influenced by the parameter initialization. Different ones might lead to different minima and while they will all be chosen to make the correct classifications for the data in the training set, they might behave quite differently for completely unseen data. In the end, different minima might have different misclassifications or accidental correct classifications.

As for the networks trained on the SKEW datasets, we also had some interesting results. These network seem to be influenced heavily by the different distribution of integers in these datasets, as can be seen in 4.17, in which they all have different starting points. Networks trained on SKEW25 seem to assume that there mostly exist integers that do not have a remainder of 0 when divided by 3 and therefore they initially behave as a classifier that always classifies integers in that way (67% initial validation accuracy). In contrast, those trained on SKEW75 seem to assume the exact opposite and behave as a classifier that always classifies integers as having a remainder of 0 (33% initial validation accuracy). While these networks have different initial assumptions, they seem to be converging towards the validation accuracy of networks trained on SKEW50, which after a while stabilizes around 50%, a validation accuracy that incidentally follows the distribution of SKEW50. While we cannot assume that they will inevitably converge, if that happens it would mean that the networks will not be better than random chance which would have an accuracy of 50%. In part, this might also explain why for the majority of the networks in the binary-class task, their validation accuracy started at the same point as if they were behaving like the simplistic baseline classifier and then fell off towards 50%. This will also be consistent with the behaviour exhibited by the majority of the multi-class networks of not being better than random chance.

5.5 General Discussion

In the introduction of this thesis we dedicated a whole section on the numerous achievements of neural networks throughout the years. So, how can they fail in a simple and already solvable task when they have succeeded in many more algorithmically demanding tasks? In chapter 3, we mentioned VGGNet (Simonyan and Zisserman, 2014). This convolutional network took as input 224x224 images (for a total of 50176

input features) and was trained on 1.3 million images, in the end reaching a test accuracy of 93%. This shows that convolutional networks can handle many input features and large training sets. A logical assumption then, is that they should also be able to handle our 1 million integers with at most 128 input features. However, this is not what we observed.

A possible explanation for why this happens seems to be the nature of the tasks themselves. Classifying an image is not an exact task. We do not need complete information about every pixel to know whether there is a dog in it, only some general patterns. All other images of dogs will follow these patterns but again, they do not need to do it exactly. In our case, for the majority of moduli we need to know exactly all the digits in order to determine the remainder. We can see this from another perspective by comparing how strongly connected datapoints and their labels are. In an image of a dog, we can tamper even with one quarter of the pixels and a good network would still be able to classify it as a dog. This is a loose connection. In contrast, if we try to change a single digit of an integer, there is a very high chance that its label (or remainder) will change. This is a strong connection. In essence, in image classification, the datapoints within a class have small variance while in our task we have a huge variance (integers with same remainders might look completely different).

But are these networks actually learning? In the majority of our experiments, the networks learned the training set but were no better than random chance on the validation set. Similar results can be found in (Zhang et al., 2016). The authors of this paper trained several neural networks on ImageNet and CIFAR10 but in one case they randomized the labels and in another they replaced the images with random noise. All the networks managed to learn the training set perfectly with 0 error but on the test set they were no better than random chance since there was no correlation between images and labels. They then show, that large enough neural networks have the capacity to learn any labeling of the training data. This can, in a sense, be seen as them having the ability to memorize any training set without necessarily generalizing well. This would also mean that their performance is closely related with the quality of the dataset. This is partially supported by the results of our last set of experiments where the networks learned the modified training sets with 0 error but were unable to handle the unseen data.

In the end, what might actually be happening is that the convolutional networks just memorize the few general dog patterns that we provide them and then classify as dogs

any images that match one of them very closely. Given a dog pattern that is not among those learned, they could possibly have trouble classifying it, like our networks did with unseen data. If that is shown to be true, then another reason why neural networks might not be able to learn modular arithmetic is that there might be no such set of general patterns we can provide them that can be used in the same way and we would also need an exact and not approximate match.

In the introduction we also mentioned that there exist parameter configurations that make a neural network able to compute arithmetic operations (Siu and Bruck, 1990; Siu et al., 1993; Franco and Cannas, 1998). In these results, as we said however, the weights were set by hand. In a sense the networks were programmed rather than taught how to perform the tasks.

While modular arithmetic might be a difficult operation to learn, addition on the other hand is much simpler. There are already results, showing that a neural network can add 7-digit integers with high accuracy (Hoshen and Peleg, 2016). The reason why this is possible can be attributed to the fact that, in contrast with modular arithmetic, addition has very few and universal rules, which lead to a small set of patterns for the neural network to learn even exactly. They only need to learn the results of the addition between any two or three single digits and propagate the carry.

Interestingly, based on our results we can draw another parallel between neural networks and the human brain, apart from their similar structure. It seems as though they both struggle with the same tasks. Both are able to recognize an image or speech or process a sentence but given an arithmetic task that is very simple for computers to already solve, they have difficulties coping with it.

Chapter 6

Conclusion

In this chapter we will conclude, initially by summarizing our findings. We will then provide critique on our experimental process, as well as suggestions for further work.

6.1 Summary

What we wanted to accomplish with this thesis, was to investigate whether neural networks are able to learn modular arithmetic. In the end, our networks were unable to achieve this and were not even better than random chance. There were also some evidence which indicate that neural networks might be memorizing instead of learning, which was partially supported by existing literature. In our discussion we explained how our task might be different from one that neural networks already excel in, that of image recognition. We hope that our results provide an insight on the behaviour of neural networks and incentivize further research on the topic.

6.2 Critique

Due to time and memory constraints we only used 1 million integers and trained for only 50 epochs. However, in recent years, neural networks have been trained on much larger datasets. If, as we discussed in the previous chapter, the performance of a neural network is closely tied to the quality of the dataset, then by providing it with a larger variety of integers and training it for longer we might see some improvement.

6.3 Further Work Suggestions

While our approaches were not successful, we will present two others that might prove fruitful: curriculum learning and algorithmic approaches.

Curriculum learning was introduced in (Bengio et al., 2009) and is a method for progressively teaching a neural network, first by providing it with simple examples and then with more complex ones. This could be helpful for learning modular arithmetic since there is a certain degree of recursiveness in the task. Most of the divisibility rules involve performing certain operations between the digits of the integer and then checking for divisibility on the result. It could be possible that, by initially training a neural network on integers of a smaller size, we could achieve better performance.

Algorithmic approaches involve neural networks that, instead of learning the task explicitly from the data, they learn the algorithm that solves it. We have already mentioned some of those in the introduction (Zaremba and Sutskever, 2014; Reed and De Freitas, 2015; Graves et al., 2014). If modular arithmetic proves to be too difficult to explicitly learn, then the closest thing to learning we can do next, is use some of these approaches. They can be seen as a compromise between teaching and programming.

Bibliography

Andoni, A., Panigrahy, R., Valiant, G., and Zhang, L. (2014). Learning polynomials with neural networks. In *International Conference on Machine Learning*, pages 1908–1916.

Arora, S., Bhaskara, A., Ge, R., and Ma, T. (2014). Provable bounds for learning some deep representations. In *International Conference on Machine Learning*, pages 584–592.

Bartlett, P. and Ben-David, S. (1999). Hardness results for neural network approximation problems. In *Computational Learning Theory*, pages 637–637. Springer.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM.

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

Blum, A. and Rivest, R. L. (1989). Training a 3-node neural network is np-complete. In *Advances in neural information processing systems*, pages 494–501.

Bottaci, L., Drew, P. J., Hartley, J. E., Hadfield, M. B., Farouk, R., Lee, P. W., Macintyre, I. M., Duthie, G. S., and Monson, J. R. (1997). Artificial neural networks applied to outcome prediction for colorectal cancer patients in separate institutions. *The Lancet*, 350(9076):469–472.

Bourlard, H. and Morgan, N. (1993). Continuous speech recognition by connectionist statistical methods. *IEEE Transactions on Neural Networks*, 4(6):893–909.

Chen, D. and Manning, C. D. (2014). A fast and accurate dependency parser using neural networks. In *Emnlp*, pages 740–750.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314.

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Durrett, G. and Klein, D. (2015). Neural crf parsing. *arXiv preprint arXiv:1507.03641*.

Franco, L. and Cannas, S. A. (1998). Solving arithmetic problems using feed-forward neural networks. *Neurocomputing*, 18(1):61–79.

Ganesan, N., Venkatesh, K., Rama, M., and Palani, A. M. (2010). Application of neural networks in diagnosing cancer disease using demographic data. *International Journal of Computer Applications*, 1(26):76–85.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256.

Golik, P., Doetsch, P., and Ney, H. (2013). Cross-entropy vs. squared error training: a theoretical and experimental comparison. In *Interspeech*, volume 13, pages 1756–1760.

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

Graves, A., Mohamed, A.-r., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*, pages 6645–6649. IEEE.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arXiv:1410.5401*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image

recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., et al. (2012a). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97.

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

Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

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

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

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

Hoshen, Y. and Peleg, S. (2016). Visual learning of arithmetic operation. In *AAAI*, pages 3733–3739.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Knerr, S., Personnaz, L., and Dreyfus, G. (1992). Handwritten digit recognition by neural networks with single-layer training. *IEEE Transactions on neural networks*, 3(6):962–968.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Lang, K. J., Waibel, A. H., and Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural networks*, 3(1):23–43.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E.,

and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Livni, R., Shalev-Shwartz, S., and Shamir, O. (2014). On the computational efficiency of training neural networks. In *Advances in Neural Information Processing Systems*, pages 855–863.

Lyons, S. M., Alizadeh, E., Mannheimer, J., Schuamberg, K., Castle, J., Schroder, B., Turk, P., Thamm, D., and Prasad, A. (2016). Changes in cell shape are correlated with metastatic potential in murine and human osteosarcomas. *Biology open*, 5(3):289–299.

Martin, G. and Pittman, J. A. (1990). Recognizing hand-printed letters and digits. In *Advances in neural information processing systems*, pages 405–414.

Mhaskar, H. N. (1993). Approximation properties of a multilayered feedforward artificial neural network. *Advances in Computational Mathematics*, 1(1):61–80.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mohamed, A.-r., Dahl, G. E., and Hinton, G. (2012). Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech, and Language Processing*, 20(1):14–22.

Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady an SSSR*, volume 269, pages 543–547.

Oord, A. v. d., Kalchbrenner, N., and Kavukcuoglu, K. (2016). Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*.

Pei, W., Ge, T., and Chang, B. (2015). An effective neural network model for graph-based dependency parsing. In *ACL (1)*, pages 313–322.

Reed, S. and De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Rosenblatt, F. (1961). Principles of neurodynamics. perceptrons and the theory of brain

mechanisms. Technical report, CORNELL AERONAUTICAL LAB INC BUFFALO NY.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Siu, K.-Y. and Bruck, J. (1990). Neural computation of arithmetic functions. *Proceedings of the IEEE*, 78(10):1669–1675.

Siu, K.-Y., Bruck, J., Kailath, T., and Hofmeister, T. (1993). Depth efficient neural networks for division and related problems. *IEEE Transactions on information theory*, 39(3):946–956.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31.

Waibel, A. (1989). Modular construction of time-delay neural networks for speech recognition. *Neural computation*, 1(1):39–46.

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., and Lang, K. J. (1989). Phoneme

recognition using time-delay neural networks. *IEEE transactions on acoustics, speech, and signal processing*, 37(3):328–339.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1058–1066.

Weiss, D., Alberti, C., Collins, M., and Petrov, S. (2015). Structured training for neural network transition-based parsing. *arXiv preprint arXiv:1506.06158*.

Zaremba, W. and Sutskever, I. (2014). Learning to execute. *arXiv preprint arXiv:1410.4615*.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*.

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer.

Zhang, C., Bengio, S., Hardt, M., Recht, B., and Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*.

Zou, H. and Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320.