Regularizing ANNs - a Review

G14SML Statistical Machine Learning

Group project report

2019/20

School of Mathematical Sciences

University of Nottingham

Group I:

Isaac Oldwood

Martin Sanner

Ioanna Papanikolaou

Jun Guo

Stepan Romanov

Abstract

In this paper we will introduce an area of machine learning known as artificial neural networks. This paper then uses a convolutional neural network to perform a numerical experiment on different methods of regularization to avoid overfitting of the machine learning algorithms.

Contents

1 Introduction

Machine learning is a field of computer science where algorithm parameters are learned through experience rather than being explicitly programmed. More specifically, machine learning is an approach to data analysis that involves building and adapting models, which allow programs to "learn" through experience. Machine learning involves the construction of algorithms that adapt their models to improve their ability to make predictions [1].

Artificial neural networks (ANNs) are a specific type of machine learning model inspired by the structure of the brain.

The first case of an ANN was in 1943, when neurophysiologist Warren McCulloch and mathematician Walter Pitts wrote a paper about neurons, and how they work. They decided to create a model of this using an electrical circuit, and therefore the ANN was born. In 1952, the world saw the first computer program which could learn as it ran. It was a game which played checkers, created by Arthur Samuel [2].

Recently machine learning and specifically ANNs have started growing in popularity. This can be attributed to many factors, the increase in computing power is one such factor, as it has allowed us to produce more complicated models than before. ANNs are able to effectively model highly complex and non-linear relationships which naturally means they are useful in many real life examples. They are highly adaptable, able to be used for many types of tasks such as regression and classification, and can be as simple or as complicated as we desire. Exactly for this reason however ANNs tend to overfit data, hence many techniques have been developed in order to combat this. This paper well give an in depth look into theory behind artificial neural networks, specifically convolutional ones, and introduce 3 common reguralization methods used to improve them.

Specifically, we will focus on convolutional neural networks, a type of ANNs frequently used in image classification, 3 common reguralization methods which can be applied to them, and explore their effectiveness on 2 standard datasets.

2 ANNs

2.1 Notation and Topology

Artificial Neural Networks are connected collections of individual units, modelled after a neuron. Thus, to properly examine the functionality of the ANN, the function of the approximated neuron first needs to be understood.

Throughout this paper, we shall be referring to the input as $\underline{x}$, which has d dimensions.

When this input is passed into a typical neuron, it either fires or not. A sample decision rule for this is

$$\underline{w} \cdot \underline{x} + b > 0 \tag{2.1}$$

where $\underline{w}$ are weights associated with each input and b is a bias, kept constant, for each neuron. In other words, neurons encapsulate the behaviour of linear functions, firing when they are in the positive regime.

In principle, one might therefore consider such a decision rule to be arbitrary, defined for the neuron, writing $\phi^{(k)}(\underline{w}, \underline{x}, b)$ as the activation, which shall be further discussed later. Here, k defines the kth neuron activation in a collection of many neurons. Typically, every layer can be assumed to have the same activation function. We can thus write superscripts (k) as the kth layer being considered.

From the general form of 2.1, we can see that it is trivial to extend the form of $\underline{w}$,$\underline{b}$ such that they contain the scalar addition, at which point the rule can be written in terms of a dot product between the weights including bias, $\underline{\theta}$ and the design vector, $\underline{x'}$. Furthermore, this design vector can be extended into the design matrix

$$\underline{\underline{Z}} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & ... & x_d^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & ... & x_d^{(2)} \\ ... & ... & ... & ... & ... \\ ... & ... & ... & ... & ... \\ 1 & x_1^{(n)} & x_2^{(n)} & ... & x_d^{(n)} \end{bmatrix} \tag{2.2}$$

which incorporates all inputs to the system including the bias terms. Each row would be considered its' own design vector. Therefore, if Y is the output of the neuron, we can write

$$Y = \underline{\Theta}^T \cdot \underline{\underline{Z}} > 0 \tag{2.3}$$

for the case of equation 2.1. This can be promoted for a collection of neurons via $Y \to Y_k$. As stated before, in principle this can be any function $\phi(\underline{\Theta}, \underline{\underline{Z}})$. Note however that this decision rule outputs a single scalar.This output can be turned into a new design vector, passing it on to a different neuron, which will have its' own decision rule based on the new input. In the scenario of many of these neurons concatenated, the superscript $^{[j]}$ refers to the jth collection of neurons. If we furthermore consider this new neuron to be capable of adapting many different inputs, like the last one, it is logical to consider what happens when instead of one neuron for a certain input, multiple neurons act concurrently. This structure is then known as a layer of neurons, described by the weight matrix, $\underline{\underline{\Theta}}$. Each column in this weight matrix describes the entirety of the prior neuron. As such, it has dimensionality $(d + 1, m)$, where m is the number of neurons in a layer.

In Principle, any of these neurons can have different activation functions. However, in most modern methods, the same decision rule is applied over the output of the layer, as will be done in the network discussed later in this paper.

Noticeably, it is important that the activation function has certain properties, which we shall discuss now.

Neural Networks tend to be used in supervised environments, in other words, environments that have a known true value to test against. The question of how correct the network result is can then be measured by the Loss function, $L(\hat{Y}, Y)$. This will be further explained in section 2.3.

2.2 Forward Propagation, Activation Functions

Forward propagation is the process of feeding input data forward through the neural network. At each layer an input is received, the layer performs some set calculations on the input (governed by the activation function) and passes the output onto the next layer. This process is repeated until the final layer which produces the final output.

Let the output of real numbers from one layer of neurons be a vector a. Then the vector of the output from that layer is:

$$\phi(\underline{W} \cdot \underline{a} + \underline{b}) \tag{2.4}$$

Where ϕ is some activation function, W is the weight matrix for that layer, a is the input vector and b is the vector of biases. This equation can be chained together to generate an equation for the whole neural network. Let x be the input data then the output of the neural network $F(x)$ can be written as:

$$F(x) = \phi(W^{[L]}\phi(W^{[L-1]}...\phi(W^{[2]}x + b^{[2]})... + b^{[L-1]}) + b^{[L]}) \tag{2.5}$$

This the general form of the equation that explains forward propagation.

Activation functions are just "a function that defines the output of a node (or neuron) given an input"[3]. If the activation functions are omitted in a neural network, then each output would just be a linear function. So, a neural network without them is just a linear regression model [4]. Hence, to introduce more complexity and allow our neural networks to solve non-trivial problems a non-linear element must be introduced. This is where activation functions are used. Another key characteristic of activation functions is that they should be differentiable, this so to allow the process of backpropagation which is explained later, and preferably non-linear. There are many functions that have these properties but some activation functions are better than others. Some take longer to train the network but are more efficient computationally. Whereas some functions take more computing power but take less time to train the neural network. The most common activation functions are described below.

2.2.1 Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$

The sigmoid function is usually thought of as a smoothed step function. It is easy to understand and apply since the equation is simple with an equally simple derivative, having the form:

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$

The sigmoid function is not as popular as it used to be since it has some problems. Its output is not centred around 0, it has slow convergence and it suffers from the vanishing gradient problem which we shall now discuss.

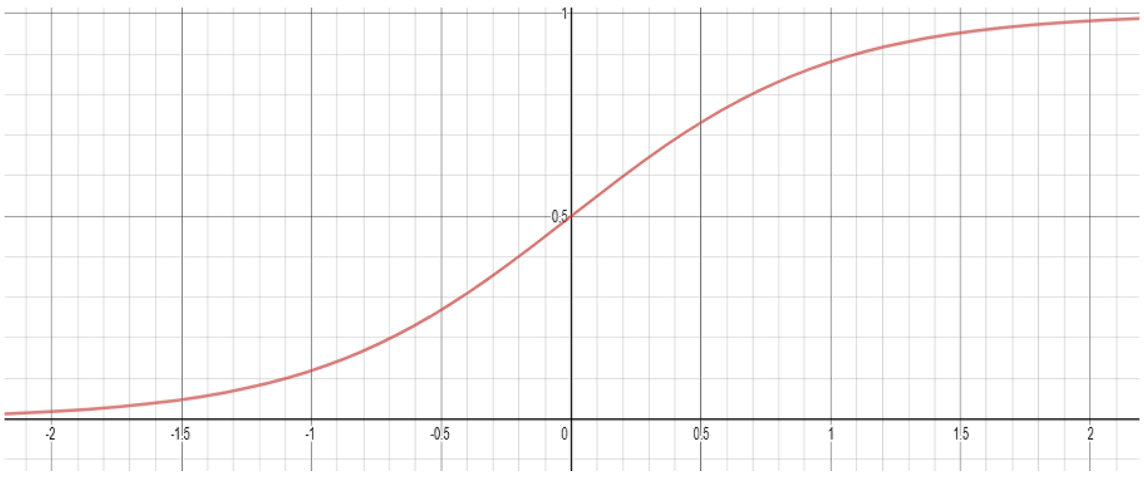

Figure 1: Sigmoid Graph

In machine learning the vanishing gradient problem occurs when using gradient-based training methods and back propagation 2.5 [3]. If the activation function has a portion where the gradient is very small then the change in the weights vanishes and prevents further training. To fix this issue, different types of algorithms have been developed(e.g. Adagrad).

2.2.2 Tanh function $f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$

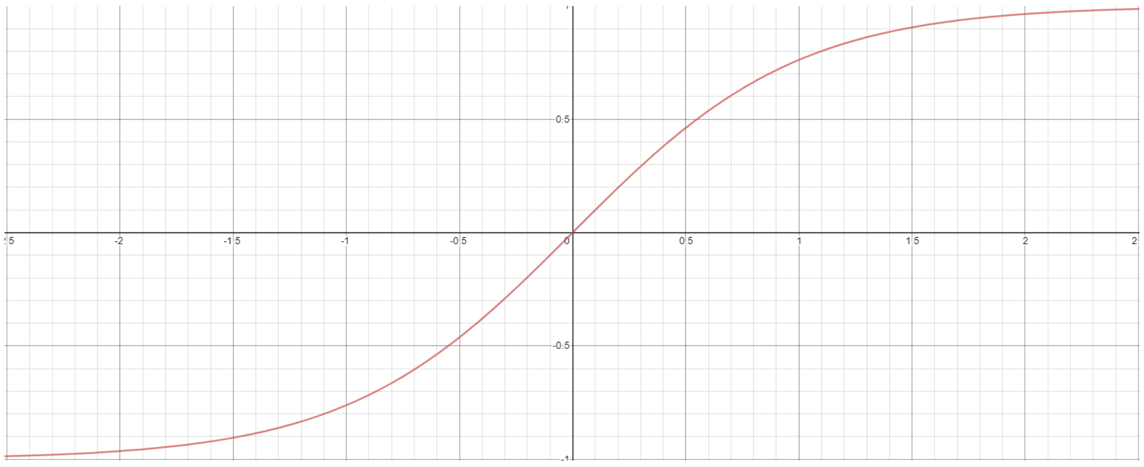

Figure 2: Tanh Graph

This function is an improvement on the sigmoid function since it is centred around 0. However due to its shape it still suffers from the vanishing gradient problem.

2.2.3 Rectified Linear units (ReLU) $R(x) = max(0, x)$

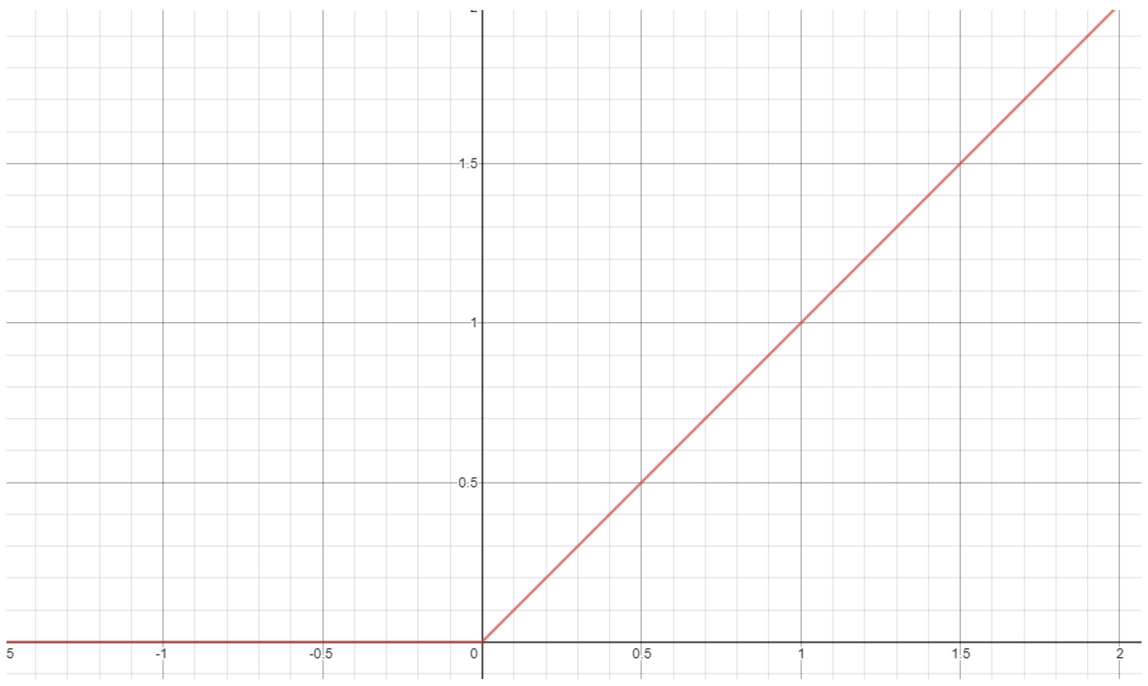

Figure 3: ReLU Graph

Recently this function has gained a lot of popularity and it has been proven to converge six times faster than the Tanh function. Furthermore it is popular because it does not fall prey to the vanishing gradient problem. Its main limitation is that it can only be used within the hidden layers.

2.2.4 SoftMax $P(x, j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$

Finally, we shall look at the SoftMax function. It is often used in classification, as it yields the probability of an input x belonging to some output class j for any amount of classes.

This direct relation to the probabilities makes SoftMax a great fit for linking it to the cross-entropy measure found in thermal physics and information theory, which we shall now discuss further via their use in loss functions.

2.3 The Loss Function

Neural networks are trained using an optimisation process that requires a function to calculate the model error. The loss function calculates the model error for a single training example, whereas the cost function is the average of the loss functions of the entire training set. The Cost function is a differentiable function, which takes the predicted outputs of a model and the corresponding real values and calculates how wrong the model was in it's prediction, returning a single value (the cost). The goal optimising a neural network is to minimise the cost by shifting the weights and biases in a way that the correct parameters of the probability distribution Y are given at the output, for a given set of inputs X. This is achieved by using an iterative optimization algorithm, the Stochastic Gradient Descent, which will be further explained in section 2.4.

The choice of a loss function is related to the activation function used in the output layer of our neural network. Depending upon the type of evaluation model - i.e. regression or classification - the loss functions can also be divided into two main categories: the Mean Squared Error (MSE) and the Cross-entropy.[5]

MSE, is one of the simplest possible losses to use for regression problems. A regression predictive model implies predicting a real-valued quantity. MSE is calculated taking the average of the squared differences between the predicted and actual values. Hence, regardless of the sign of the predicted and actual values the result is always positive and a perfect value is 0. Larger mistakes result in more error than smaller mistakes because of the squaring and so the model is punished for making larger mistakes.[6] The formula of this loss function is given by:

$$L(\hat{Y}, Y) = \mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2, \tag{2.6}$$

where n is the number of output nodes, Y_i is the desired neural network output and $\hat{Y}_i$ is the predicted output.

Cross-entropy is widely used as a loss function when optimising classification models. Classification problems are those that involve one or more input variables and the prediction of a class label. Classification tasks are separated to binary and multi-class classification problems depending on the number of classes at the output.[7] A model can estimate the probability of an example belonging to each class label, taking a value in the set $\{0, 1\}$.

Cross-entropy calculates a score that summarizes the average difference between the actual and predicted probability distributions for all the classes in a model. Then, the score is minimized and a perfect Cross-entropy value is as close to 0 as possible.[6] Cross-entropy has the ability to punish confident wrong predictions increasing the cost to infinity, when the estimated probability is very far from the actual probability distribution. Therefore, Cross-entropy is a good choice for classification problems. The formula of this loss function for multi-class classification problems is given by:

$$L(\hat{Y}, Y) = \text{CE} = -\sum_{i=1}^{C} Y_i \text{log} \hat{Y}_i, \tag{2.7}$$

whereas the formula for binary classification problems is given by:

$$L(\hat{Y}, Y) = \text{CE} = -\sum_{i=1}^{2} Y_i \text{log} \hat{Y}_i = -Y_1 \text{log} \hat{Y}_1 - (1 - Y_1)\text{log}(1 - \hat{Y}_1), \tag{2.8}$$

where C is the number of classes, Y_i is the desired neural network output and $\hat{Y}_i$ is the predicted output.

2.4 Gradient descent

When the neural network is initially constructed its weights are initialised randomly or from a simple distribution and biases to 0. We can then forward propagate a certain sample through the neural network as in 2.5, to calculate the corresponding predicted output. However, since the weight parameters of the network were initialised randomly there will likely be a sizeable difference between the initial predictions and labels, which can be described by a large **cost function**. So in order to optimise our predicted outputs, the neural network needs a way to optimise the weight and bias parameters in order to minimise the cost function. **Gradient descent** is a common algorithm used to help the neural network do exactly that by iteratively updating the weights to reduce the cost function.[3]

2.4.1 Minimising the cost function

As discussed previously, the **loss function** computes the error for a single training example while the cost function is the average of the **loss functions** for all the training examples. If the number of training examples is N, the **cost function** would be the total squared error.[8]

$$\mathrm{L(w)} = \frac{1}{N}\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2 \tag{2.9}$$

The principle behind the **gradient descent** is just like climbing down a hill. Consider that you are walking along the graph below and are aimed at reaching the minimum 'red' dot when you are in the position of the 'green' dot. To achieve this, you need to think about two questions:

1. Your direction of travel.

2. The length of your steps, avoiding longer downhill times due to smaller steps or missing the lowest point due to longer steps.

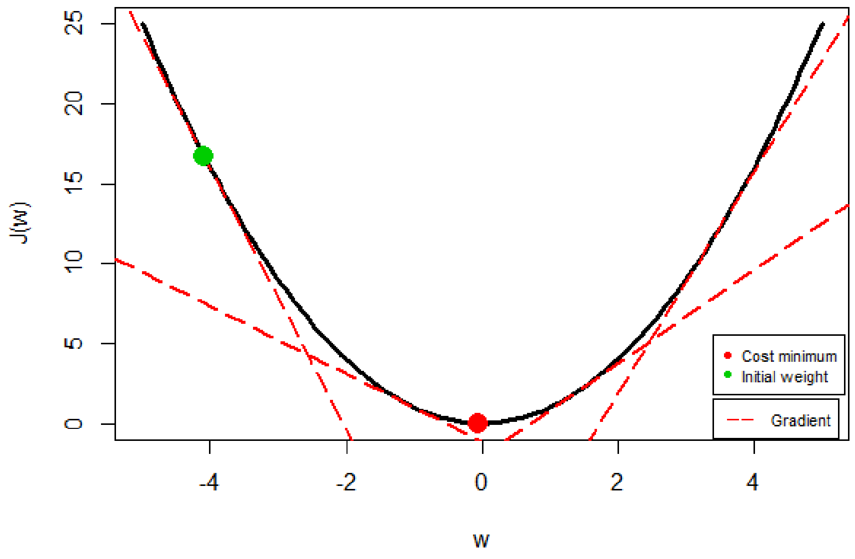

Figure 4: Gradient Descent

Similarly, **gradient descent** determines its own direction of travel by the negative gradient which is the result of the partial derivative of the cost function for each weight. In the image, this result is represented by the slope $\nabla\mathrm{L(w)}$ corresponding to the tangent of each point. When the slope $\nabla\mathrm{L(w)}$ is negative, the point is adjusted to the right, and the point is adjusted to the left when the slope $\nabla\mathrm{L(w)}$ is positive. After determining the direction, a learning rate η should be chosen to determine the step size $\Delta\mathrm{w}$. If the step size Δw is proportional to the slope ∇L(w), this will make the gradient step closer to the minimum value, and the smaller the step size Δw, so as to avoid overshooting.

The learning rate η plays a vital role in answering question 2. Values which are too low can mean extremely slow convergence meanwhile values which are too high might mean the gradient might never "settle" at the minimum.

So in the gradient descent algorithm, the way to update the weight is to add the initial weight to its corresponding step size

$$w = w + \Delta w \tag{2.10}$$

where the size of step is the product of negative gradient and learning rate.

$$\Delta w = -\eta \nabla \mathrm{L(w)} \qquad \text{where} \qquad \eta > 0 \tag{2.11}$$

After the parameter update, further training samples are forward propagated through the network to calculate and evaluate new predictions. Based on them the algorithm is repeated again.

2.4.2 Batches

In neural networks, a **batch** refers to a sample set of the training set, which is propagated through the network before the weights are updated. It follows that the batch size refers to the number of samples in a **batch**. A batch with a batch size greater than 1 and less than the training set is called a **mini-batch**.

Batches tie directly to the GD algorithm. If we use batch size of 1, the algorithm updates the weights after every single datapoint which can results in noisy updates. This process is otherwise known as **stochastic gradient descent** (SGD). On other hand we can use the whole training set to update the weights, also known as **batch gradient descent**. Doing it this way can be computationally expensive as we need to store the whole dataset during training. [8] GD can also take batch sizes outside of the 2 extremes using mini-batches which defines mini-batch gradient descent.

2.5 Back Propagation

So far we have discussed network topology, different attributes of neural networks and the gradient descent algorithm for updating the weights. We have not mentioned how to actually compute the gradient of the loss function. As it turns out, this is a complicated problem. Loss functions have a highly non-linear dependence on weights and hence finding a solution analytically is far too difficult. Instead, we resort to the use of a iterative numerical solution which we call **backpropagation**.

The first step of backpropagation is called **forward propagation**, also known as the forward network activation. This process has the given data push through the network to activate the neurons of the network and produce an output at the output layer. We start by initialising the input $x_i^n = z_i^{[1]}$ for each $i \in D_1$ dimension of the feature space. Now propagate the given data point through the network using the current weights and biases for all $l \in [1 : m]$ as follows,

$$a_j^{[l]} = \sum_{i=0}^{D_{l-1}} z_i^{[l-1]} w_{ji}^{[l]} \tag{2.12}$$

$$z_j^{[l]} = \phi(a_j^{[l]}) \tag{2.13}$$

where $z_j^{[l]}$ denotes the jth output of the activation function $\phi(x)$ at layer l and $w_{ji}^{[l]}$ denotes the weight from the ith unit in layer $l-1$ to the jth unit in layer l.

The process is then repeated until we reach the output layer m and the final value is set as $z^{[m]} = \hat{Y}$ the predicted output of our neural network. The output is then evaluated using the chosen loss function denoted as $L_n(\hat{Y}, Y)$, the error of the nth data point. Note that if we choose to use a higher batch size, say N, then the loss function would become,

$$L(\hat{Y}, Y) = \sum_{n=0}^{N} L_n(\hat{Y}, Y). \tag{2.14}$$

Next our aim is to calculate $\frac{\partial L_n}{\partial w_{ji}^{[l]}}$ for every i, j and l which can then be used to calculate ∇L required for the gradient descent algorithm. Using product rule we can rewrite,

$$\frac{\partial L_n}{\partial w_{ji}^{[l]}} = \frac{\partial L_n}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial w_{ji}^{[l]}} \tag{2.15}$$

and setting

$$\delta_j^{[l]} = \frac{\partial L_n}{\partial a_j^{[l]}} \tag{2.16}$$

denoted as the **error** term. Note that using 2.12 we can evaluate,

$$\frac{\partial a_j^{[l]}}{\partial w_{ji}^{[l]}} = \frac{\partial}{\partial w_{ji}^{[l]}}(a_j^{[l]}) = \frac{\partial}{\partial w_{ji}^{[l]}}(\sum_{i=0}^{D_{l-1}} z_i^{[l-1]} w_{ji}^{[l]}) = z_i^{[l-1]}. \tag{2.17}$$

Hence we can write equation 2.15 succinctly as,

$$\frac{\partial L_n}{\partial w_{ji}^{[l]}} = \delta_j^{[l]} z_i^{[l-1]}. \tag{2.18}$$

Since we already know the values of $z_i^{[l]}$ after forward propagation we just need to find the values of the error terms. Start with $\delta^{[m]}$ which are the errors for the output layer. We can find these explicitly by differentiating the loss function w.r.t to $\hat{Y}$,

$$\delta^{[m]} = \phi'(a^{[m]}) L'(\hat{Y}, Y). \tag{2.19}$$

Now propagate the error backwards layer by layer. Using chain rule,

$$\delta_j^{[l]} = \frac{\partial L_n}{\partial a_j^{[l]}} = \sum_{k=0}^{D_{l+1}} \frac{\partial L_n}{\partial a_k^{[l+1]}} \frac{\partial a_k^{[l+1]}}{\partial a_j^{[l]}} \tag{2.20}$$

where we take the sum over all units k, in the next layer to which unit j, in the previous layer, connects to. This sum allows us to rewrite the equation for the error of unit j in terms of errors of units k in the layer ahead. Using equations 2.12 and 2.13, and the Chain rule we can evaluate.

$$\frac{\partial a_k^{[l+1]}}{\partial a_j^{[l]}} = \frac{\partial}{\partial a_j^{[l]}}(\sum_{j=0}^{D_l} \phi(a_j^{[l]}) w_{kj}^{[l+1]}) = \phi'(a_j^{[l]}) w_{kj}^{[l+1]} \tag{2.21}$$

which gives us,

$$\delta_j^{[l]} = \phi'(a_j^{[l]}) \sum_{k=0}^{D_{l+1}} w_{kj}^{[l+1]} \delta_k^{[l+1]} \tag{2.22}$$

where ϕ is the activation function for the layer. This finally gives us the desired result,

$$\frac{\partial L_n}{\partial w_{ji}^{[l]}} = z_i^{[l-1]} \phi'(a_j^{[l]}) \sum_{k=0}^{D_{i+1}} w_{kj}^{[l+1]} \delta_k^{[l+1]}. \tag{2.23}$$

Now we have all the essential components needed in order to use the gradient descent algorithm and update the weights of the network accordingly. The whole algorithm can be summarised as,

1. Propagate the data point forward whilst storing the values of each of the $z_i^{[l]}$ until the final layer.
2. Using the loss function evaluate the error $\delta^{[m]}$ at the output layer.
3. Backpropagate the errors to find the $\delta^{[l]}$ for every hidden layer.
4. Use formula (2.23) to evaluate the derivatives of the loss function.

[9]

3 Convolutional Neural Networks

3.1 Introduction and Advantages over linear networks

As we have seen above, Linear Layers of Neurons take vectors as input information. This means that if one wants to pass a high-dimensional set of data (e.g. $2, 3, 4...$ dimensions), information encoded in these dimensions is lost, as a transformation into $1d$ space has to be made.

For instance, if one analyses the MNIST [10] or CIFAR100 [11], alot of information is lost when applying a linear network to it.

Convolutional Neural Networks (CNNs for short) however are resistant to this, as they always work on the data surrounding a given point in the set, as defined through its convolutional kernel. In General, Convolution of a function f with some kernel g is defined through

$$H(\underline{x'}) = f * g = \int_{\mathbb{R}^d} f(\underline{x})g(\underline{x} - \underline{x'})dx^d \tag{3.1}$$

[3] In case of a convolutional layer, the kernel g is given through the kernel matrix, with f representing an image, and thus, the equation for e.g. images are written as:

$$H[idx_x, idx_y] = \sum_{j=-g_1}^{g_1} \sum_{k=-g_2}^{g_2} f[idx_x - j, idx_y - k] * g[j, k] \tag{3.2}$$

where the kernel shape is (g_1, g_2) and idx_x, idx_y are the pixels in H, the convolved image. Typically, $g_1 = g_2 = g_c$, so the kernel matrix is square. This convolution is performed on each layer of the input data. [3]

Note that it might be the case that not all pixels are assumed to convey relevant data. If for instance the data to be convolved is very sparse, skipping unnecessary steps might become relevant. This is encoded into a convolution parameter known as the **stride** s_c, which essentially sets the step size inbetween convolutions when scanning over the data.

Because of this structure, convolution is capable of picking out information related to different shapes when transformed, in other words, a convolutional layer when properly trained should be able to detect a given object in an image regardless of it being reflected, at an angle or otherwise transformed. [3, 12] At this point, it is important to note that if the kernel is not of shape $(0, 0)$, then clearly pixels around the edge of the image f, are going to be undefined. There two most common ways of tackling this issue.

First off there is zero-padding. Because the convolution is an integral, padding appropriately (such that it is impossible to reach an undefined value) by implanting 0 means that no effect will be visible from the faked new data. This is useful in scenarios where no periodicity can be assumed.[3] In some scenarios however, this might lead to a whitening out of the edges, as they are slowly reduced to 0.

If however it is possible to assume so, for instance when considering MRI images, periodic boundary conditions can also be used. In this scenario, each side section is treated by stitching on appropriate amounts of the corresponding opposite side, such that the periodicity is directly implanted in the image. Notably, this will need to be cropped out by the end. In either case, the padding is designated p_c.

The convolution returns a different amount of information as prior, as e.g. due to the padding additional information was taken into account. The output size can be calculated per dimension as:

$$n_{out} = \frac{n_{in} - g_c + 2p_c + s_c}{s_c} \tag{3.3}$$

where n_{in} is the input size in that dimension, so in case of an image either width or height. [3]

3.2 Convolutional Network structure

Based on the output of the convolution, it might be possible to extract further information. For instance, the convolution could filter for the existence of an object (by having the filter pass objects with the correct shape, discarding other potential matches). At that point, it might still be possible to extract further information, or immediately go into either classification or regression.

Therefore, typical convolutional networks tend to combine multiple convolutional layers (as each one defines only one type of filter), followed by either Pooling (see section 3.3) or normalizing filters (see section 4.4). The outputs of all the convolutions are then modified into a vector format, being passed into a linear layer or network for the task at hand. [3, 5]

In some cases, Dropout layers (see section 4.5) might even be applied, though these tend to work quite a bit differently on convolutional layers when compared to linear ones, as shall be shown in Appendix A.

3.3 Convolutional Pooling

Pooling is an important part of any convolutional neural network. It is generally added to a network as a standalone layer with the purpose of down-sampling the input images. Despite being a layer it remains static throughout the learning process and does not change as more data is being passed through the network. Instead it takes images as input and outputs a new image, one with significantly reduced dimension which still preserves the maximum amount of features and information.

The image is first partitioned into subsections, where each subsection will subsequently be summarised by a single number and put back together preserving the spatial structure. The new digit is produced according to the pooling formula which is selected as a hyper parameter before training starts. Out of the notable pooling formulas there is the **Max-pool**,

$$a_i = max(x_{ij}) \tag{3.4}$$

where x_{ij} represents the jth point of the ith partition and a_i is the output representing the jth partition. Another one is the **Average pool**, which simply calculates the mean of each partition,

$$a_i = 1/J \sum x_{ij} \tag{3.5}$$

where J is the number of digits in the jth partition. Max pooling is traditionally more preferred as it preserves the sharpest features of the given image and generally gives the best performance.Other types of pooling formulas are called the L_p pooling formulas,

$$a_i = (1/J \sum x_{ij}^p)^{1/p} \tag{3.6}$$

for any $p \in [1, \infty)$ but these are used less often than the previously mentioned ones. [3, 5]

Pooling layers can generally be placed before other convolutional layers or before traditional layers or both. The reduction in dimension greatly benefits the network in terms of computational costs and reduces overfitting.

4 Regularization

One of the main properties of a neural network is its generalization capability. Generalization refers to the model's ability to perform well on previously unseen data (i.e. test set) and is determined by two important factors: the model's complexity and the size of the training data.

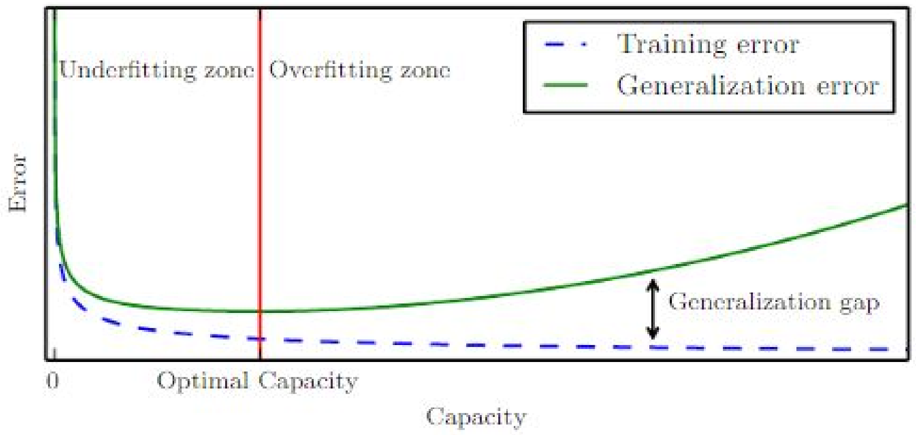

Figure 5: Relationship between error and model's capacity [5]

As the model complexity increases, the training error generally decreases, but the generalization (or test) error seems to have a "U" shape. Initially, it is high for models of low complexity, it follows a downward trend until the model's complexity matches the unknown data distribution and then it starts becoming high again for models of higher complexity.

Models of low complexity tend to underfit the data, which means that they are not flexible enough to properly fit the data. Hence, the model is unable to make accurate predictions with both training and test set. Underfitting can be fixed either by adding more data or increasing the complexity of the model.

On the other hand, models of high complexity tend to overfit the data. This type of model is so flexible, since it allows the model to fit not only to broad patterns, but also to various types of noise in training data. Under overfitting, the model has high prediction accuracy on the training set, but low on the test set and so, it does not generalize well. This can be avoided by reducing the model complexity. Doing so however, may jeopardise the model flexibility and its ability to capture important trend in data.

In general, the risk of overfitting is getting higher in an over-dimensioned neural network (with too many weights) at a limited dataset. For instance, too many neurons in the hidden layers of a neural network may result in overfitting. This occurs since the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers.[13]

Solving the issue of bias and variance is actually about dealing with overfitting and underfitting. High variance causes overfitting, while high bias causes underfitting. As more and more parameters are added to a model, the complexity of the model increases and variance, which is also increasing, becomes our primary concern while bias tends to decrease.[14]

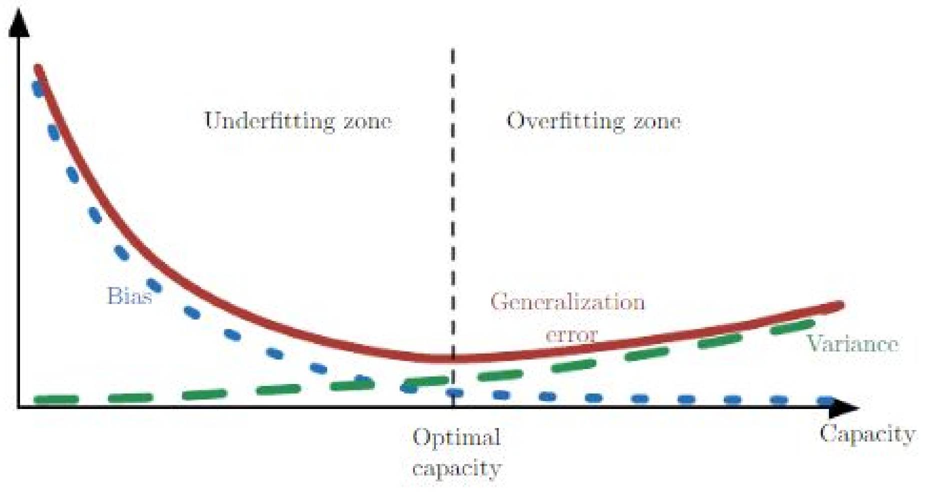

Figure 6: A graph for bias and variance trade-off. [5]

Bias and variance are two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter and is given by:

$$\mathrm{Bias}(\hat{f}(x)) = \mathrm{E}(\hat{f}(x)) - f(x) \tag{4.1}$$

Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause. Variance is given by:

$$\mathrm{Var}(\hat{f}(x)) = \mathrm{E}(\hat{f}^2(x)) - \mathrm{E}(\hat{f}(x))^2 = \mathrm{E}(\hat{f}(x) - \mathrm{E}(\hat{f}(x)))^2 \tag{4.2}$$

The most common way to negotiate this trade-off is to use cross-validation. Alternatively, the MSE of the estimates can be used, which is given by:

$$\mathrm{MSE} = \mathrm{E}(y - \hat{f}(x))^2 = \mathrm{Bias}(\hat{f}(x))^2 + \mathrm{Var}(\hat{f}(x)) + \sigma_\epsilon^2 \tag{4.3}$$

where σ^2 is the irreducible error (the noise that cannot be reduced by algorithms).[5]

The easiest way to reduce overfitting is to essentially limit the capacity of a model. Model complexity is one of the hyperparameters of the neural network. In contrast with the parameters of a neural network (e.g. weights and biases), the hyperparameters (e.g. number of hidden layers, complexity) are set independently, before training and are used in order to help estimate the model parameters.

There are several methods for improving the generalization capability of neural networks in order to avoid overfitting. These mostly split in two main categories: restricting the number of weights - or equivalently the number of neurons - in the network, or restricting the magnitude of the weights.[15] The most common technique for this purpose is regularization. This is achieved directly through a penalty term (also called regularizer) in the cost function or indirectly by early stopping.[14] In general, regularization can be defined as any modification we make to a learning algorithm, which aims to reduce its generalization error but not its training error.[5]

Early stopping is the simplest method to control the complexity of a model. In general, the model performance on the validation set improves only at the first stage of training, it reaches an optimum at some point and then it gets worse with further iterations. By this method, the training procedure stops when the error on the validation set first starts to increase (before the weights have converged).[9]

A different and more explicit method for regularization is called weight regularization, which is actually adding a parameter norm penalty $\Omega(\theta)$ to the cost function L in order to penalise large weights. The regularized cost function is given by:

$$\widetilde{\mathrm{L}} = \mathrm{L} + \lambda\,\Omega(\theta) \tag{4.4}$$

where $\lambda \geq 0$ is the hyperparameter that weights the relative contribution of the norm penalty Ω, relative to the cost function L (defining the strength of regularization). A $\lambda = 0$ means that no measures have been taken against the possibility of overfitting. If λ is too large, then the model will prioritize keeping θ as small as possible over trying to find the parameter values that perform well on the training set. As a result, choosing λ is a very important task and can require some trial and error.[16]

Under this constraint, when the training algorithm minimizes the regularized cost function, it is actually minimizes both the initial cost function and the penalty term. More details about the different regularization types will be discussed in the following sections.

4.1 L_p Norm

The L_p norm is a method of regularization that aims to reduce the variance of the weights whilst not changing the biases. Depending on the value of the p the L_p norm can be used in different ways. For example when $p = 0$ the norm is a measure of the sparsity of the layer, the L_0 norm cannot be used for updating weights due to the vanishing gradient problem. As long as $p > 1$, the L_p norm can be used for optimisation.

The L_p norms are additions such that

$$L \to J = L + \sum_{i}^{n} \alpha_i \|w\|^i$$

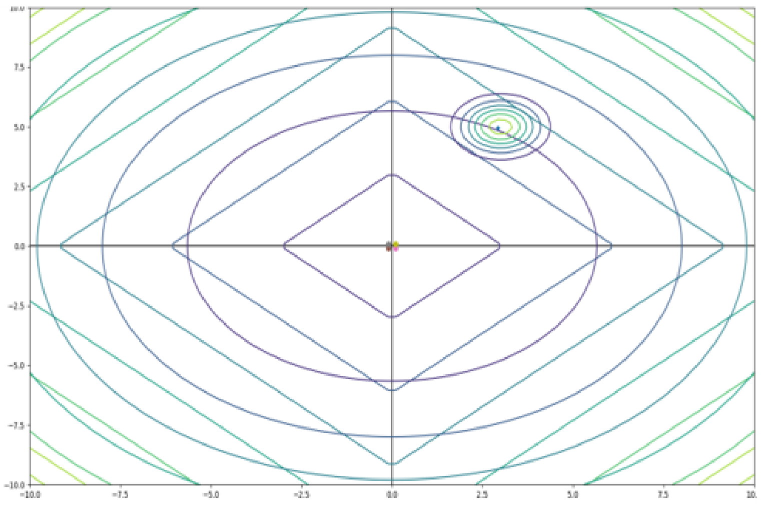

Figure 7: A graph of L_1 and L_2 norm

4.2 L_1 Norm

The effect of the L_1 norm is increasing sparsity by encouraging values on the diagonals of the weight matrices. The mathematical form is

$$L \to J = L + \alpha_1 \sum_{layers} \|w\|$$

And the method updates the weights by

$$w := w - (\lambda \nabla_w L + \alpha_1 sgn(w))$$

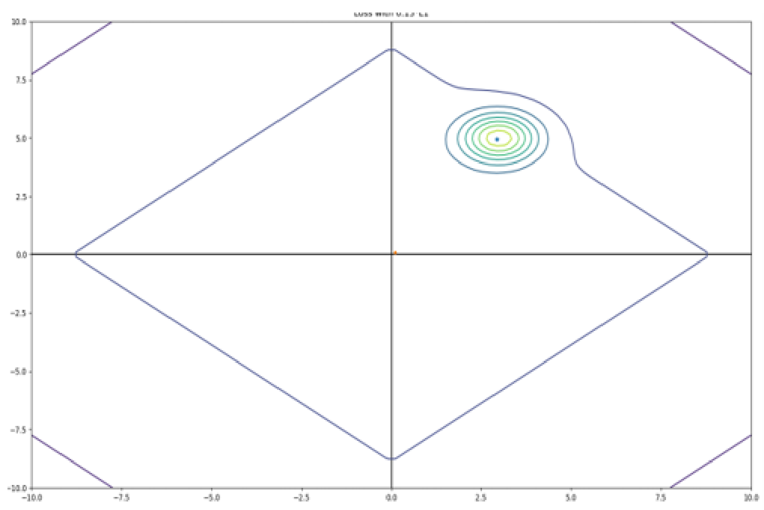

Figure 8: A graph of L_1 norm

This graph also highlights another issue with the L_1 norm and the reason why the L_1 norm is rarely used. On the axes there is a point where the function is undiferrentiable. The derivative is undefined at every axes and so there is little point using the L_1 norm for optimisation.

4.3 L_2 Norm

The effect of the L_2 norm is to rescale the weights, limit their size. Due to this the variance is reduced. The mathematical form is

$$L \to J = L + \frac{\alpha_2}{2} \sum_{layers} \|w\|^2$$

And this form of the method adjusts the weights by

$$w := w - (\lambda \nabla_w L + \alpha_2 sgn(w))$$

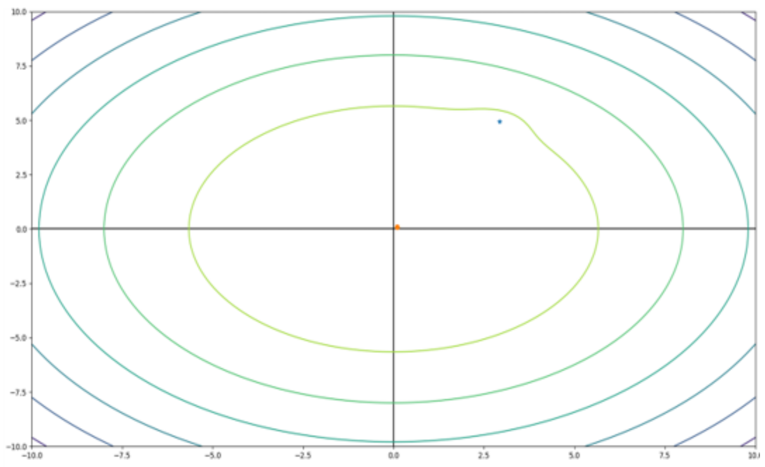

Figure 9: A graph of L_2 norm

4.4 Batch Normalisation

4.4.1 Reasons for using batch normalisation

We know that during the learning process of the neural network, it will continuously adjust the parameters to reduce the cost function, which in turn causes the distribution of the activation function to constantly change during the learning process. And all of these will give rise to a problem called the **internal covariance shift**.[17] Specifically, we can use 2.4 to help us understand this concept. In the gradient descent process, the parameters w and b of each layer will be updated, which will cause the entire activation function to change and so that for the change of its distribution. Also, the result of this activation function is the input of the next layer, which will require next layer to continuously adapt to the new distribution during the training process, and this eventually leads to a significant decrease in learning speed of the network. In order to solve this problem, **batch normalisation** was proposed.

4.4.2 The principle of batch normalisation

Generally, **batch normalisation** is based on **batch** and it needs to be applied before the activation function. Batch normalisation normalises the output of the previous activation layer by subtracting the standard deviation of the batch mean, so that this processed output is limited to a distribution with a mean of 0 and a standard deviation of 1.

However, after that, this normalised result is not conducive to using the gradient descent method to find optimal weights of the next layer due to the deficiency of information caused by normalisation. So **batch normalisation** needs to multiply this normalised result by a “standard deviation” parameter γ and add a “mean” parameter β to recover the information. Consequently, the determinants of the distribution of activation function are changed from the original weights and bias of the previous layer to the “standard deviation” parameter γ and “mean” parameter β , which increases the stability of the neural network[18]. The next layer will adapt to the new distribution faster so that the learning time of the neural network will be shrunk.

4.4.3 Batch Normalisation in training processing

We will use superscript (i) to refer to the i-th sample data in one mini-batch, superscript[l] to refer to the l-th layer and the subscript j to refer to the j-th node of one layer.

Now,we input one **mini-batch** into the neural network. For the j-th node of one layer, the algorithm is as follows:

$$Z_j^{[l](i)} = w^{[l]}a^{[l-1](i)} + b^{[l]} \tag{4.5}$$

$$u_j^{[l]} = \frac{1}{m}\sum_{i=1}^{m} Z_j^{[l](i)} \tag{4.6}$$

$$\delta 2_j^{[l]} = \frac{1}{m}\sum_{i=1}^{m}(Z_j^{[l](i)} - u_j^{[l]})^2 \tag{4.7}$$

$$\hat{\mathrm{Z}}_j^{[l](i)} = \frac{Z_j^{[l](i)} - u_j^{[l]}}{\sqrt{\delta 2_j^{[l]} + \epsilon}} \tag{4.8}$$

$$\widetilde{\mathrm{Z}}_j^{[l](i)} = \gamma_j^{[l]}\,\hat{\mathrm{Z}}_j^{[l](i)} + \beta_j^{[l]} \tag{4.9}$$

$$a_j^{[l](i)} = \phi^{[l]}(\widetilde{\mathrm{Z}}_j^{[l](i)}) \tag{4.10}$$

Then, we apply this algorithm to the first node of the second layer to help us understand the process of **batch normalisation**, which means $j = 1$ and $l = 2$. But It should be noted that when $l = 1$, $a^{[l-1](i)}$ represents the characteristic values of the i-th sample data, but when $l > 1$, $a^{[l-1](i)}$ represents the activation values of previous layer created by the i-th sample data.

For 4.5, if we assume $i = 1$, then $Z_1^{[2](1)}$represents the result of linear calculation of the activation values of the first layer created by the first sample data. Specifically, $w^{[2]}$ and $b^{[2]}$ respectively represent the weights and bias of the second layer corresponding to this node. Then we could calculate each $Z_1^{[2](i)}$ for all the samples in this **mini-batch**.

Now, we use 4.6 to get the mean $u_1^{[l]}$ using all $Z_1^{[2](i)}$ in this **mini-batch**. After that, we use the results of 4.5 and 4.6 to calculate the variance $\delta 2_1^{[2]}$ of $Z_1^{[2](i)}$ through 4.7. Also, we could find that for one **mini-batch**, there is just one mean and one variance for one node.

On the basis of the first three functions, the result of 4.8 is obtained by normalising the $Z_1^{[2](i)}$. Note that, ϵ in this equation is to prevent invalid settlements with a variance of 0.

In 4.9, $\gamma_1^{[2]}$ refers to the "standard deviation" parameter and $\beta_1^{[2]}$ refers to "mean" parameter for the first node of second layer. And we could offset the distribution of $Z_1^{[2](i)}$ by changing just the two parameters to recover the information, which means we could change just two parameters instead of a set of weights and bias to control the distribution of last layer. Interestingly, we could also find that $\widetilde{Z}_j^{[l](i)}$ will equal to $Z_j^{[l](i)}$, if $\gamma_j^{[l]}$ equals to $\delta 2_j^{[l]}$ and $\beta_j^{[l]}$ equals to $u_j^{[l]}$.

Finally, we bring all $\widetilde{Z}_1^{[2](i)}$ into the activation function $\phi^{[2]}$ of the second layer to get the activation value corresponding to the first node of the second layer.

By analogy, we can use this algorithm to find the activation values of other nodes in the hidden layers. All of these steps could also be applied to the other **mini-batches** during the training process.

4.4.4 Batch Normalisation in test processing

In the testing phase, we may only need to test one sample or a small number of samples, which will lead to biased estimates of the mean and variance. Therefore, after training the model with the **batch normalisation** algorithm, we need to remain the means and variances of each **mini-batch** at each node (u_{batch} and δ_{batch}^2 respectively). In the test phase, the following functions are used to make unbiased estimates of the mean and variance of each node:

$$u_{test} = \mathrm{E}(u_{batch}) \tag{4.11}$$

$$\delta^2{}_{test} = \frac{m}{m-1}\mathrm{E}(\delta^2{}_{batch}) \tag{4.12}$$

Then according to the 4.5 function, calculate each Z_{test} for one node, and then use the following equation to batch normalise Z_{test}:

$$\widetilde{\mathrm{Z}}_{test} = \gamma\frac{\mathrm{Z}_{test} - \mathrm{u}_{test}}{\sqrt{\delta^2{}_{test} + \epsilon}} + \beta \tag{4.13}$$

Finally, bring $\widetilde{\mathrm{Z}}_{test}$ into the activation function and find the activation value of this node.

$$a_{test} = \phi^{[l]}(\widetilde{\mathrm{Z}}_{test}) \tag{4.14}$$

Similarly, we can find the activation values of the other nodes in the first layer. Then we can use them as the input of the next layer, find the activation values of all nodes in the second layer, and so on, until the final layer gets its activation values and the corresponding output results.

4.5 Dropout Regularization

4.5.1 Reasons for using Dropout

For complex neural networks with a many parameters but few training samples, a **co-adaptation phenomenon** often occurs during the learning process. This phenomenon means that during the iterative training of the neural network, some strong connections will be continuously strengthened, while weaker links will be gradually ignored, thus making some links more predictive than others.[19]

Unfortunately, the existence of this phenomenon can exacerbate the problem of overfitting the model. The problem of overfitting is mainly as follows: the **cost function** of the model fitted by the training data with the neural network is small and the accuracy of the prediction is high, but the **cost function** is relatively large on the test data, and the accuracy of the prediction is relatively low. So a model with the overfitting problem is almost unusable. Fortunately, **dropout** can prevent overfitting problems to some extent.

4.5.2 The principle of Dropout

Dropout is a method to solve the problem of neural network overfitting by setting a certain probability at each layer of the neural network and randomly deleting some nodes in the neural network according to this probability.

Suppose we need to train the following neural network, where the input is x and the output is y.

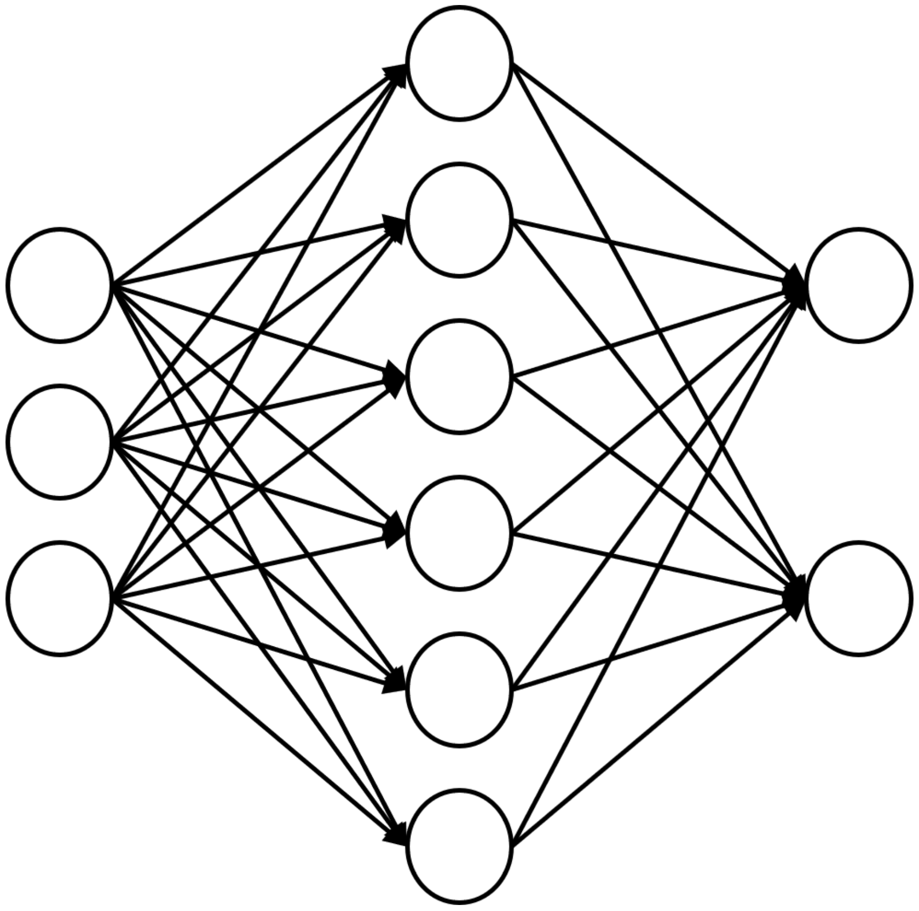

Figure 10: Neural Network

In a common neural network, after inputting x, we will get the predicted value y through **forward propagation**, then use the comparison result between the predicted value and the real value to evaluate the **cost function**. Finally, **backpropagation** will be used to update the parameters in the network according to the cost function. The parameter update process is the learning process of the neural network.

After using the dropout method, the neural network becomes the following workflow:

1. The input and output layers unchanged, some neurons and all their corresponding

connections in the hidden layer are randomly deleted according to a certain probability. If we randomly delete half of the neurons in a layer with a probability of 0.5, the result is shown in the following figure:

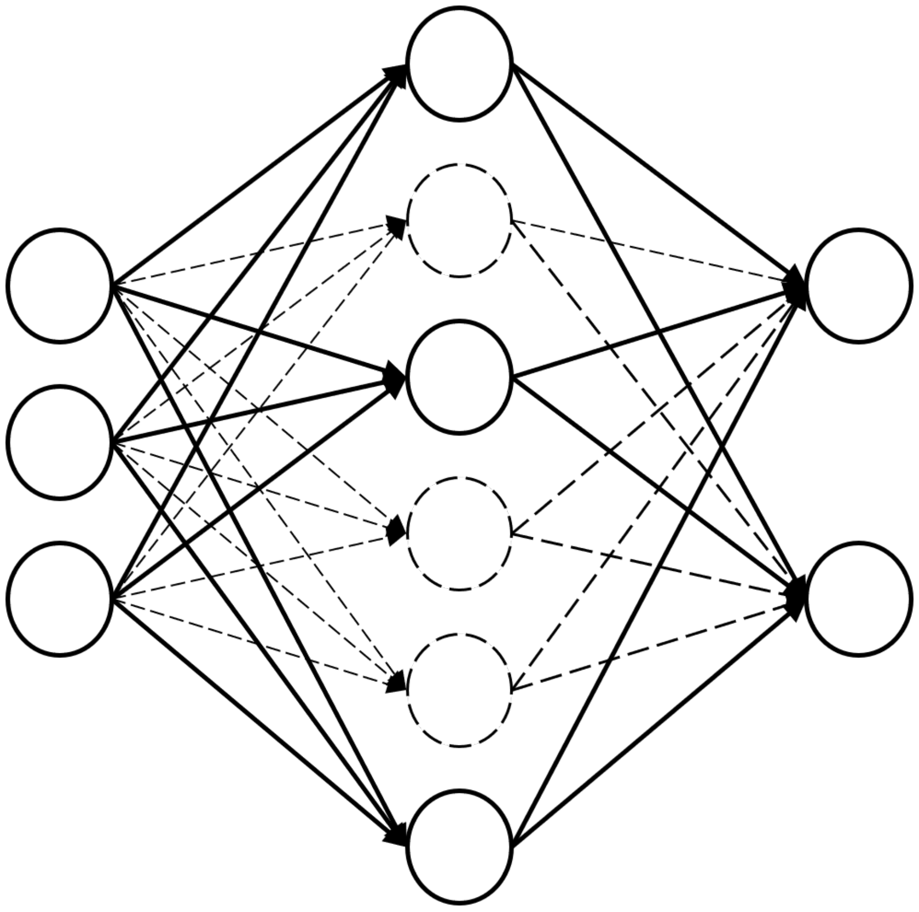

Figure 11: Neural Network with dropout

2. Then input sample data x to the newly generated neural network and get y through **forward propagation**. After that, the relevant parameters of the neurons that have not been deleted will be updated through **backpropagation** and by using **gradient descent**.

3. Recover eliminated neurons and their corresponding connections .

4. Repeat the above three steps

The key point in the neural network obtained through the above first three steps, is that the deleted neurons and corresponding parameters remain as they are, whilst the remaining neurons and parameters are updated.

4.5.3 Dropout in training processing

First, set a parameter for each layer whose value comes from the Bernoulli distribution with a probability p.

$$\theta_j^{[l]} \sim \text{Bernoulli(p)} \tag{4.15}$$

Then to begin randomly deleting neurons, we need to multiply $\theta^{[l]}$ obtained in the previous step by the activation values of the corresponding layer.

$$\widetilde{a}^{[l]} = \theta^{[l]} a^{[l]} \tag{4.16}$$

After that, the activation values of this layer are processed linearly to get the media values $\mathrm{z}_i^{[l+1]}$ of next layer.

$$\mathrm{z}_i^{[l+1]} = \mathrm{w}^{[l+1]}\mathrm{a}^{[l]} + \mathrm{b}^{[l+1]} \tag{4.17}$$

Finally, bring this linearly processed result into the activation function to get the activation values of next layer.

$$a_i^{[l+1]} = \phi(\mathrm{z}_i^{[l+1]}) \tag{4.18}$$

4.5.4 Dropout in test processing

In the test phase, we no longer need to delete neurons, we just multiply the weight parameters of each neuron by the probability p used to generate the Bernoulli distribution, which means that the neuron will stop working with a probability of $\mathrm{p}^{[l]}$.

$$\mathrm{w}_{test}^{[l]} = \mathrm{p}^{[l]}\mathrm{w}^{[l]} \tag{4.19}$$

5 Experimental Approach and Implementation

5.1 Network Structure

To elaborate on the effects of the different regularizers, we shall be tackling the MNIST and CIFAR100 datasets, two very well known sets typically used as an introduction into machine learning using convolutional neural networks.

Both sets are relatively simple sets of images, either with only one channel or three, alongside a sequence of labels.

The labels for MNIST are numericals between 0-9, encoding which digit the image shows. CIFAR10's labels are general classes for the image, specifying whether the image is of a vehicle, food etc. This is generalised in CIFAR100, where we find a difference between these classes, now dubbed superclass(of which there are 20), and subclasses, which is what the classifier is trained to predict. As the name suggests, there are 100 subclasses. The higher dimensionality of the task(increase in channels) allows for a valid comparison of the methods between the different sets [10, 11]

5.1.1 MNIST

The MNIST network is one of the most commonly used datasets for machine learning. It consists of 60000 training samples and 10000 test samples of handwritten digits 0 to 9 taken from the larger NIST dataset. The datapoints consist of 28x28 pixel images with associated labels. The first part of the MNIST code has the data loaded and transformed to a usable format. The test and train samples are combined into a single set since cross-validation will be performed to evaluate the performance of the network.

The CNN structure is based on a working example found online [20], with relevant regularization methods commented in/out. The network has 2 convolutional layers with the ReLU functions followed by a MaxPool layer. A single standard layer follows this and the output is parsed through the softmax function to receive a probability distribution. Other relevant hyperparamets such as bacth size and number of epochs are selected given the networks performance. The best ones are chosen and then fixed for all of the resulting networks.

The network is then trained over 5 folds, the test and validation accuracies are plotted over epochs for each fold. These results can be views in the following chapter.

5.1.2 CIFAR

For the CIFAR network, it consists of $32 * 32$ RGB images (3 input channels). We define the network in terms of convolutional blocks, combining convolutions, dropout and batch normalization where applicable. All networks consist of this structure with sections being left out if the test does not concern the regularization method. Each block contains more than one convolution, as every convolution only learns one type of filter. The activations associated with the filters are the ReLU function, and the final output is parsed through a softmax activation, yielding the probability of an input image belonging to some subclass. The secondary convolution is passed through a maxpool, only retaining the maximum outlying information in every 2x2 section of the image.

The input images can be assumed to belong to some distribution of all classes. Thus, the batch normalization is located at the end of a subsector, meaning that the input of the next set is batchnormed.

The complete network structure consists of two such blocks followed by a linear layer, activated with a softmax function to classify the data.

This structure is based on samples found online, slightly modified by running cross-validation, using 10% of test data, on the structure(e.g. the dropout density, number of filters). The networks are using the best values found in this approach.

The reguralization parameters used are described below where they are relevant. For combinations of methods the reguralization parameters are kept the same as in the networks with single method.

6 Results

The results of the combination of regularizers are outlined below, starting with the results on MNIST.

6.1 MNIST

We start with the base unregularized case which will help us demonstrate the effectiveness of each type of regularization. Relevant discussion and explanations are found in the following section.

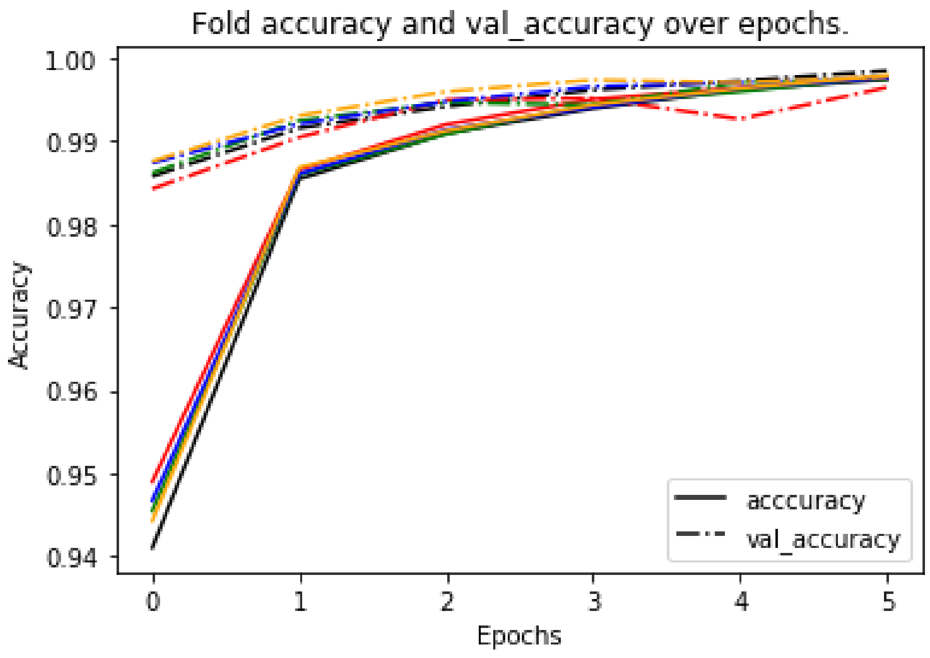

Figure 12: Accuracy of the unregularized system. Avg. validation accuracy = 0.9898285746574, average loss = 0.039244905777740206

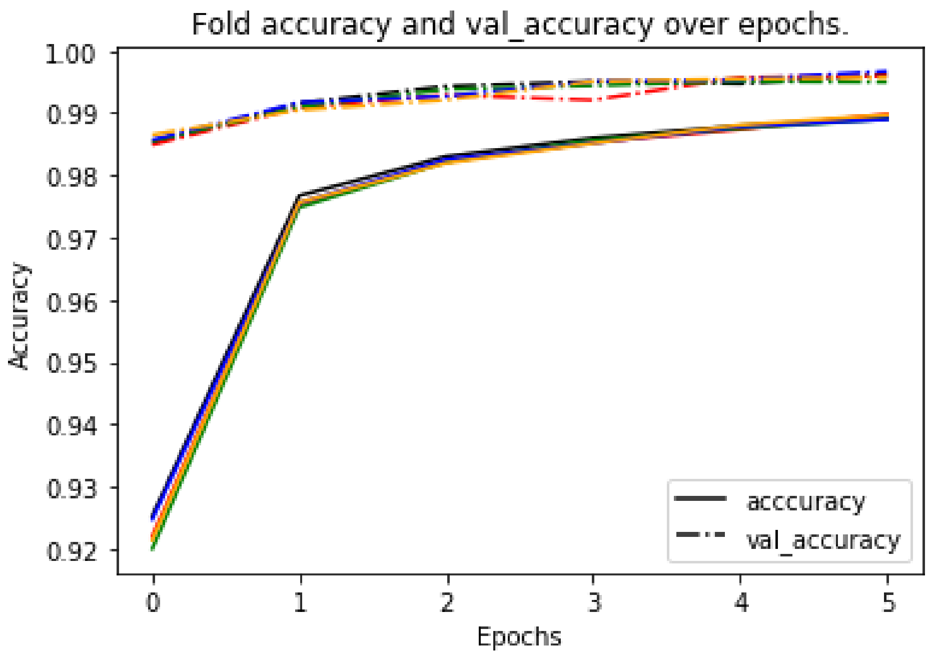

Figure 13: Accuracy for the system with dropout 0.5 at the final layer, avg. validation accuracy = 0.9896285653114318, average loss = 0.03532647787571628

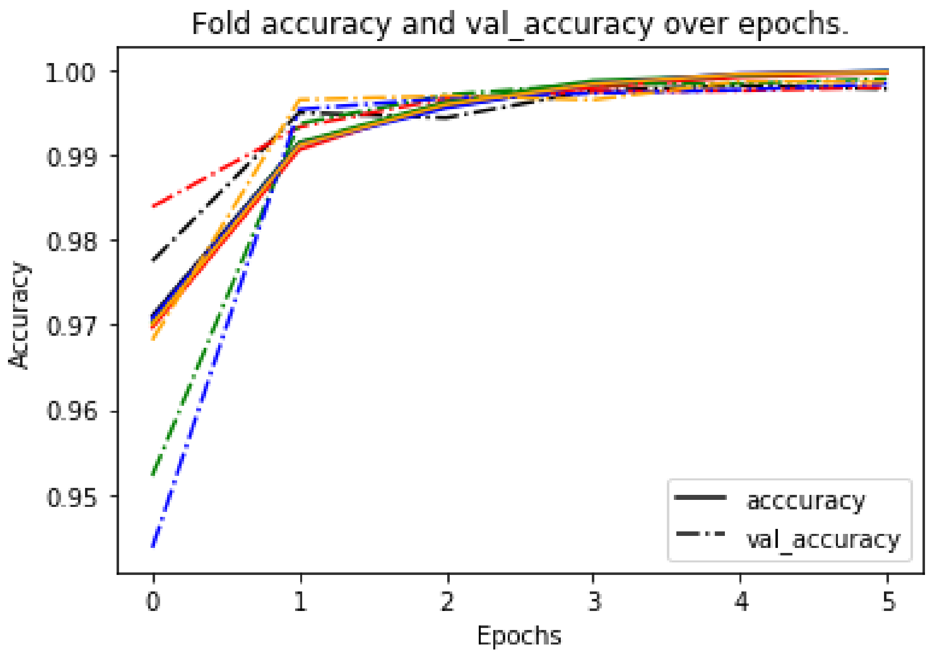

Figure 14: Accuracy of the system with Batchnorm only at every layer.. Avg. validation accuracy = 0.9910571455955506, average loss = 0.031274502889939326

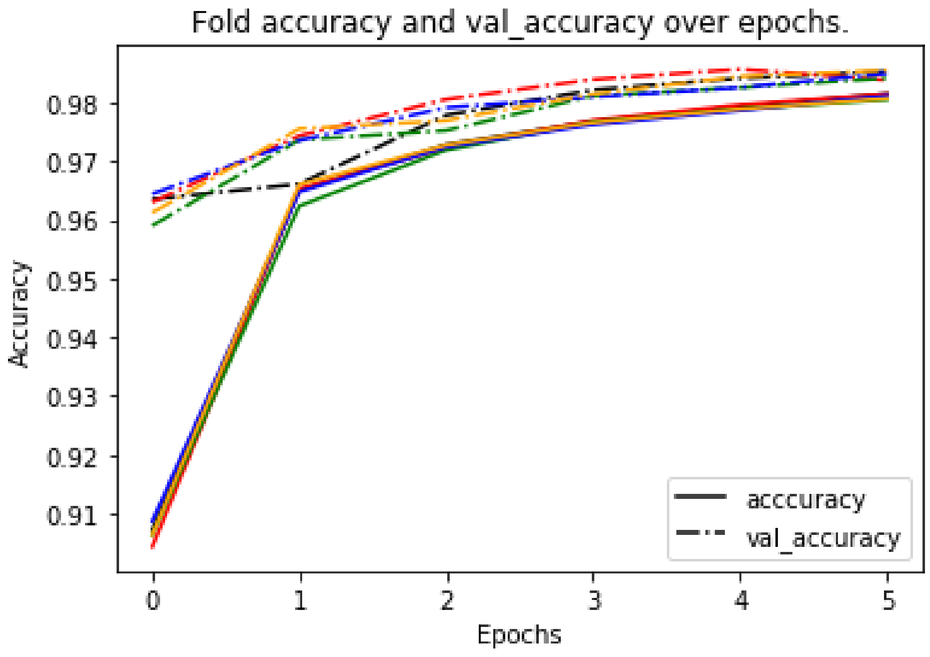

Figure 15: Accuracy of the system with L_2 penalty $\lambda = 0.001$ at every layer. Avg. validation accuracy = 0.9799285769462586, average loss = 0.0861870028095586

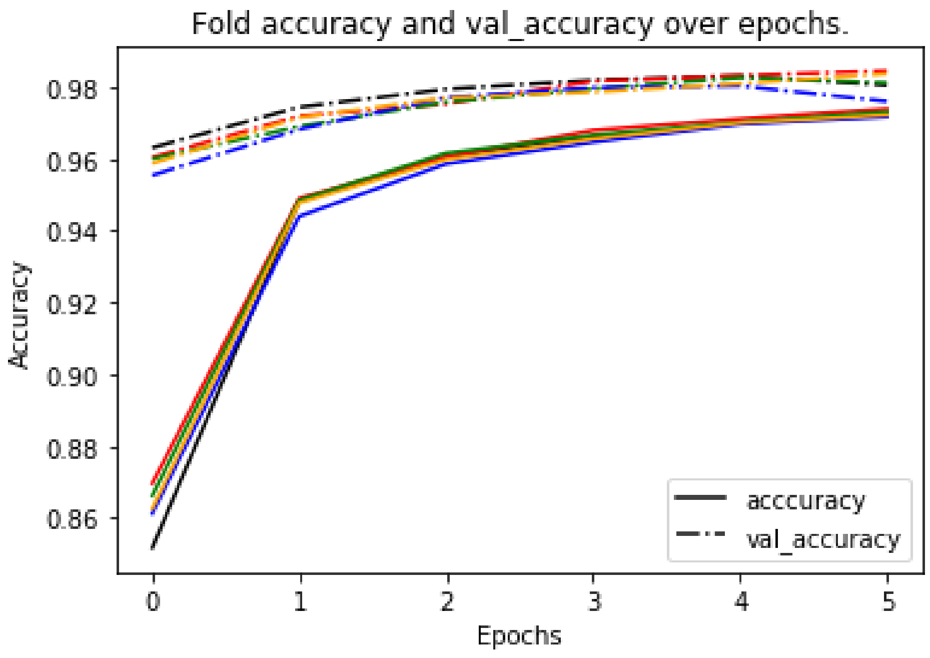

Figure 16: Accuracy of the combination of L_2 and dropout. Avg. validation accuracy = 0.9830571413040161, average loss = 0.10726005648459706

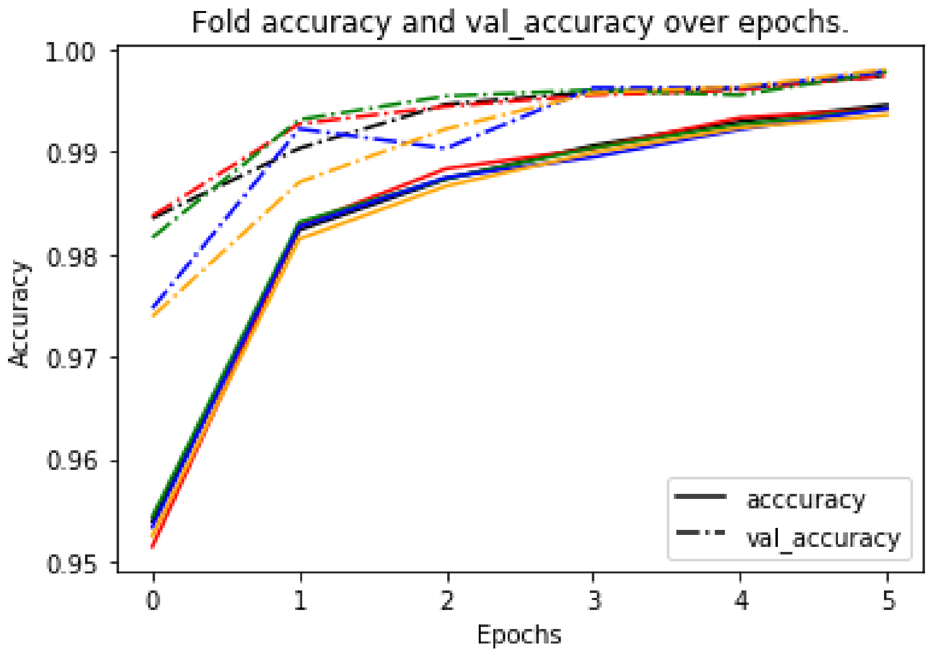

Figure 17: Accuracy of the combination of BatchNorm and dropout. Avg. validation accuracy = 0.9900285720825195, average loss = 0.03531015274427835

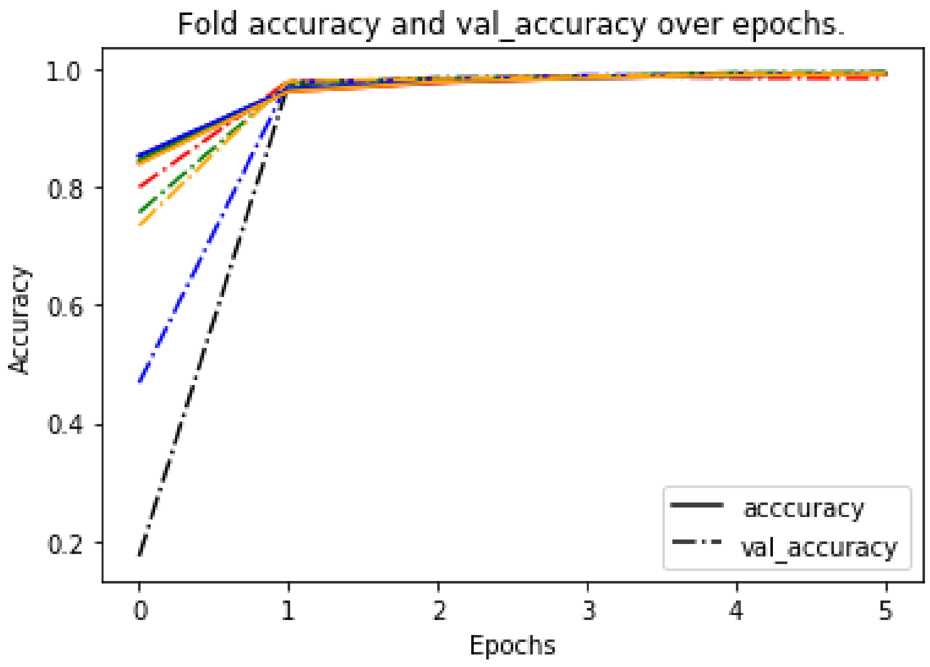

Figure 18: Accuracy of the combination of L_2 and BatchNorm. Avg. validation accuracy = 0.9782571434974671, average loss = 0.08466720829292068

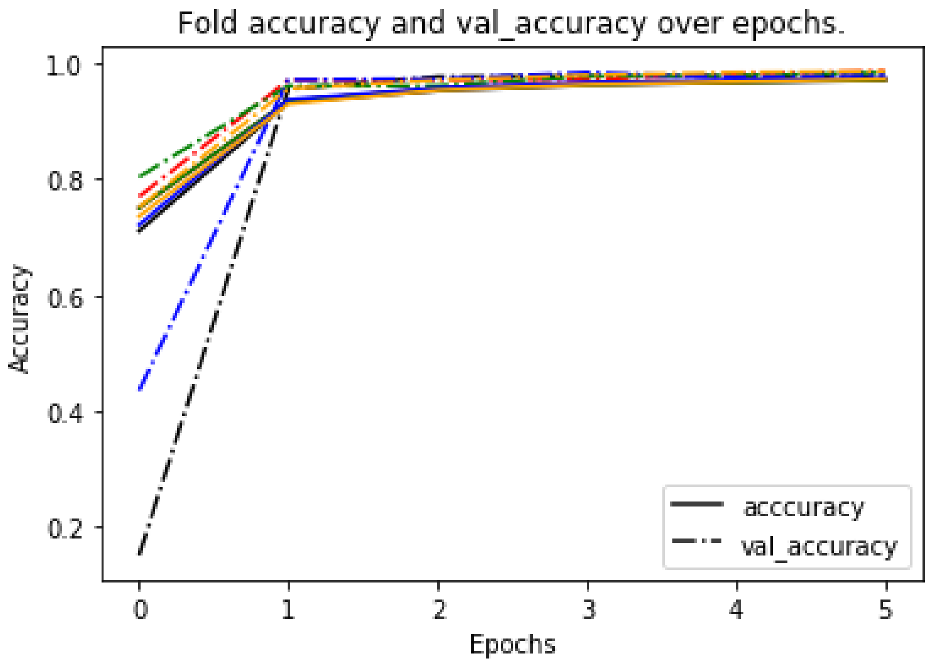

Figure 19: Accuracy of the combination of all our tested regulizers. Avg. validation accuracy = 0.974314296245575, average loss = 0.09298402957618236

6.2 CIFAR

To start with, we investigated the behaviour of the outlined network without the use of regularisers, ie $\lambda = 0$, without Batch Normalisation and without dropout.

This resulted in the following loss and accuracy plots:

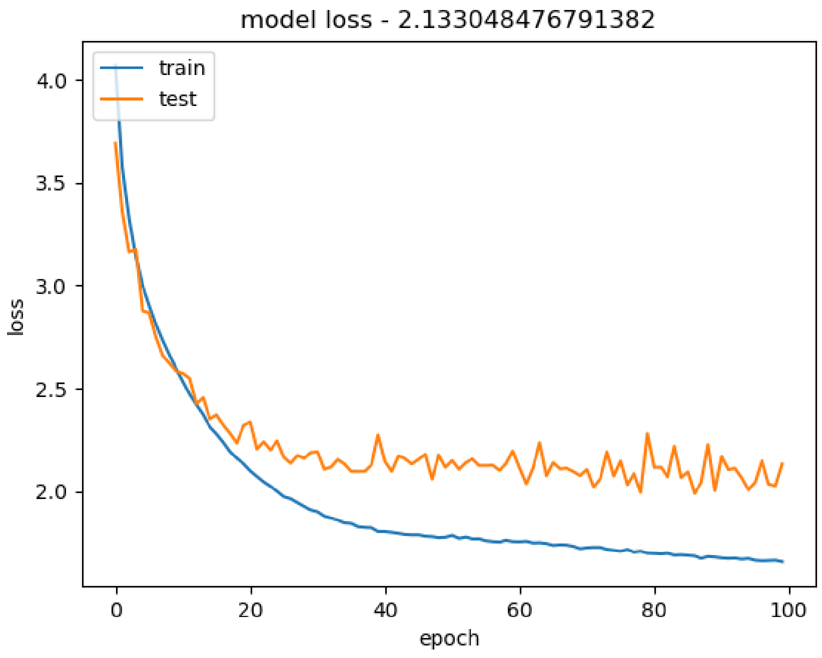

Figure 20: Loss for the unregularized system

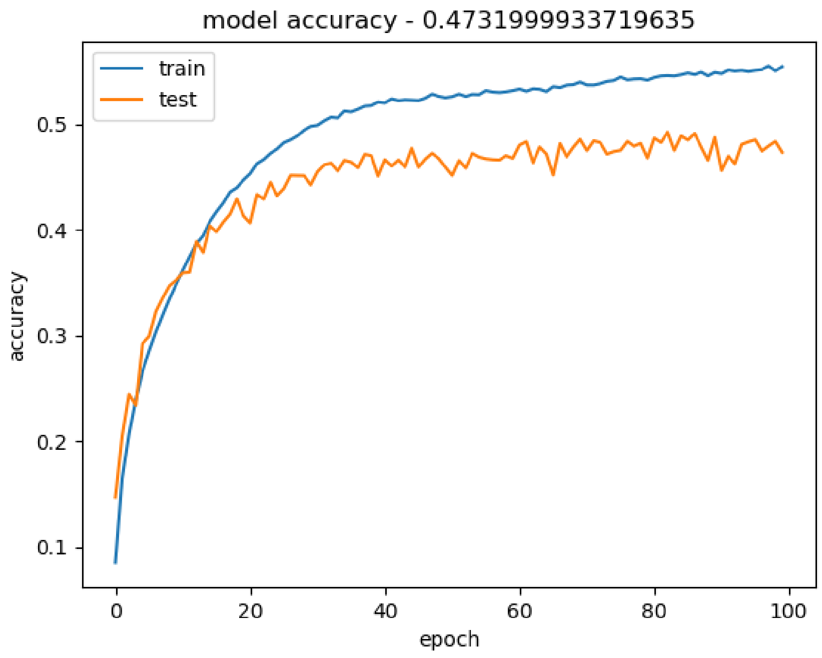

Figure 21: Accuracy of the unregularized system

Training the same model with the L_2 regularizer and $\lambda = 10^{-6}$ resulted in the following model history:

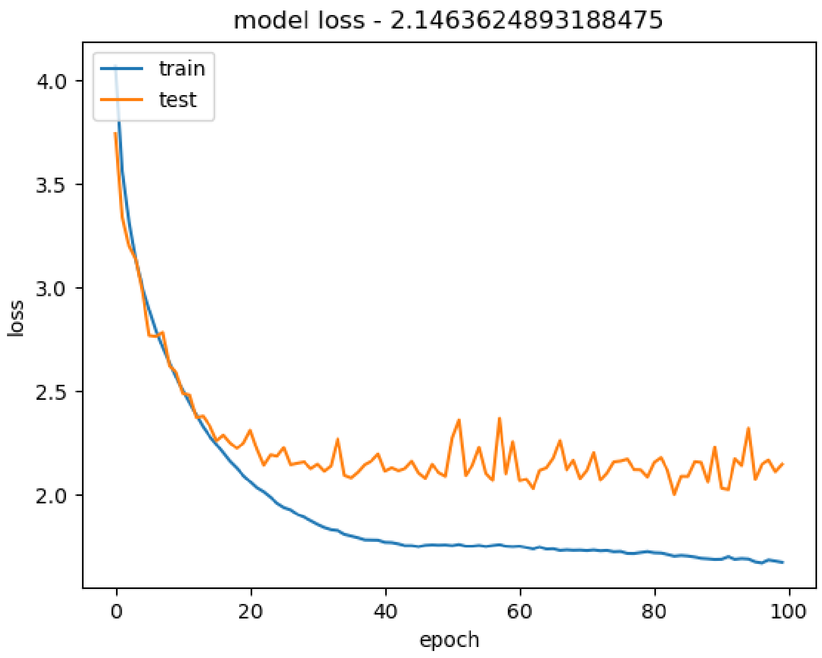

Figure 22: Loss for the system with $\lambda = 10^{-6}$

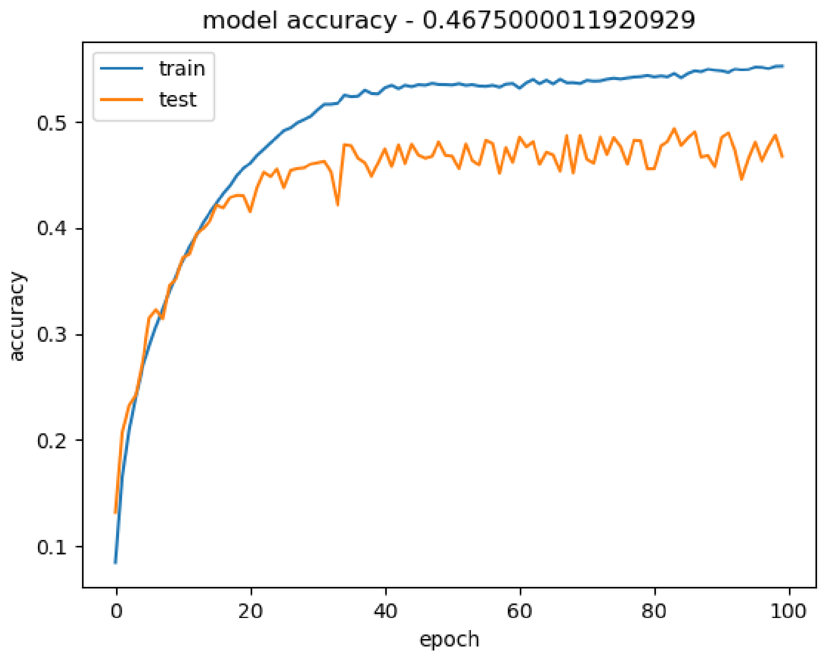

Figure 23: Accuracy for the system with $\lambda = 10^{-6}$

As we can see, the improvement in stability to the model is minor so far. If Dropout is instead performed on the unregularized system, the results are:

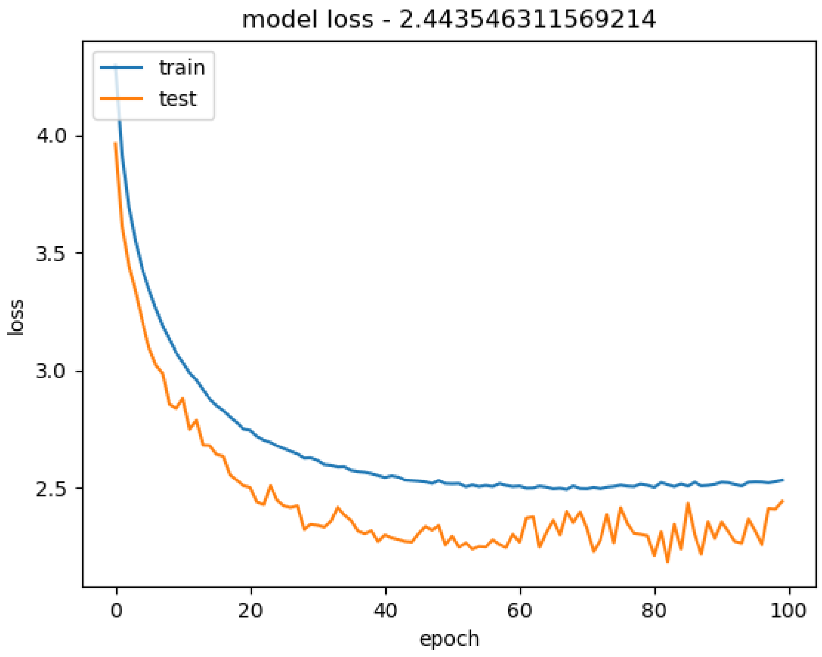

Figure 24: Loss for the system with Dropout only

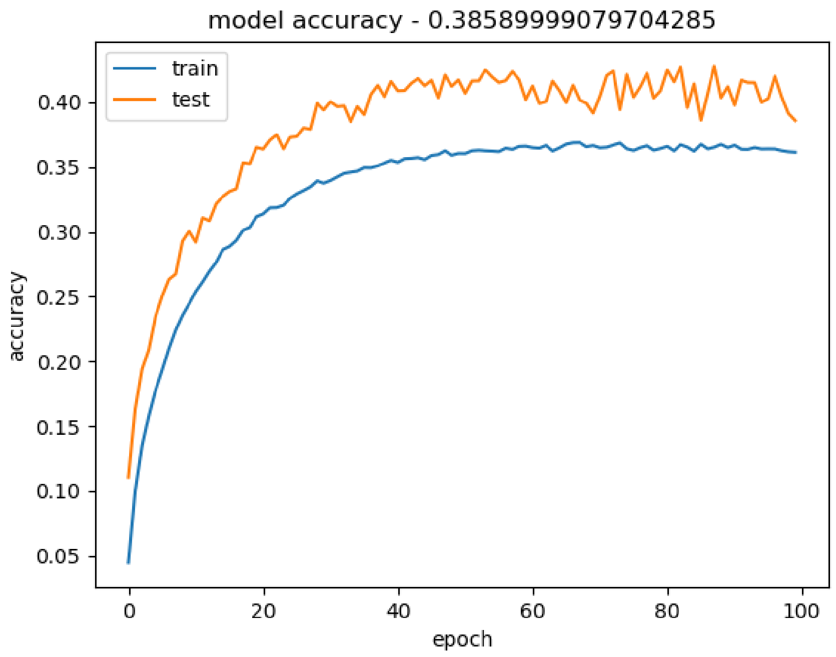

Figure 25: Accuracy for the system with Dropout only

For which we immediately notice a lower risk of overfitting, which is assumed to mostly occur in the final layer as outlined in appendix A. If Batch Normalization is instead performed, the results are

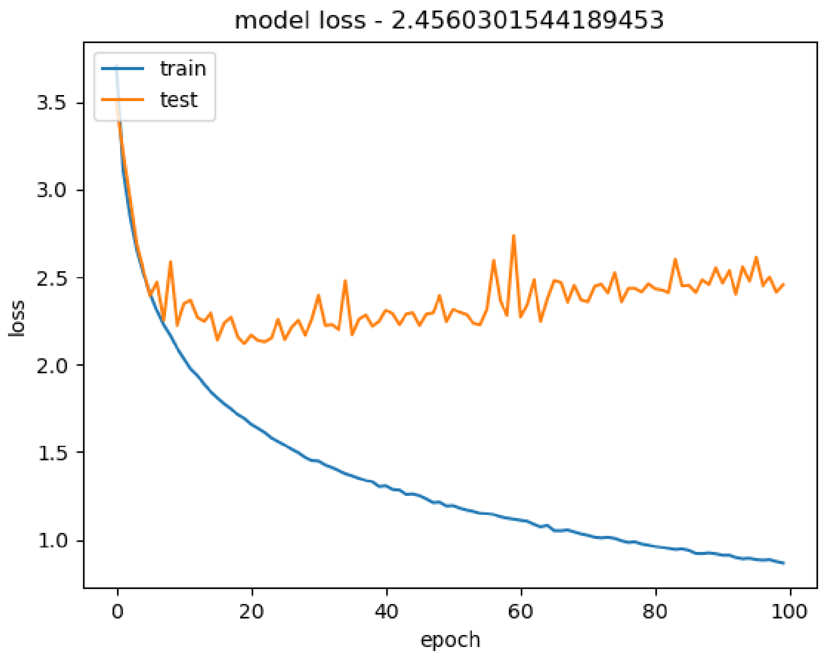

Figure 26: Loss for the system with Batch Normalization

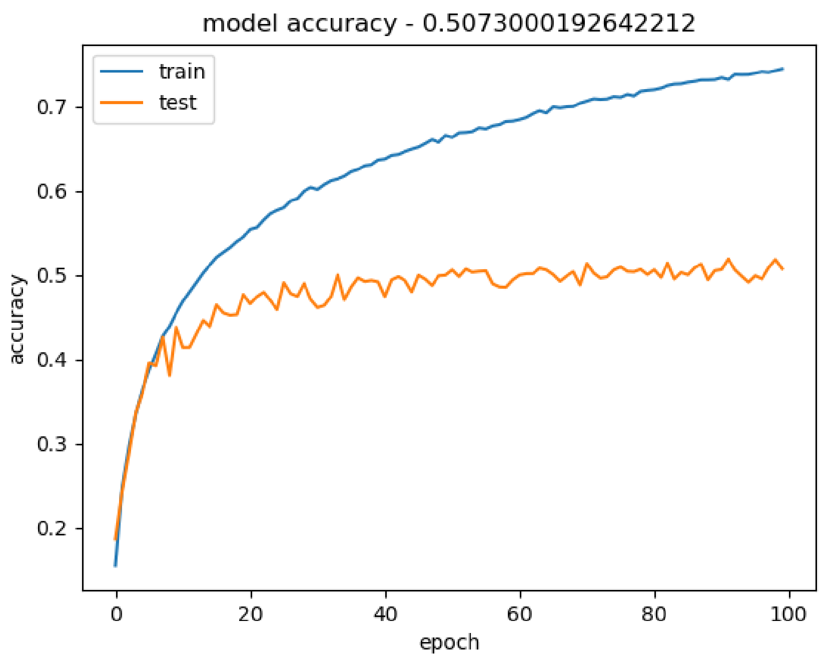

Figure 27: Accuracy for the system with Batch Normalization

Onto combining the different regularizations, we start by combining dropout with Batch Normalization:

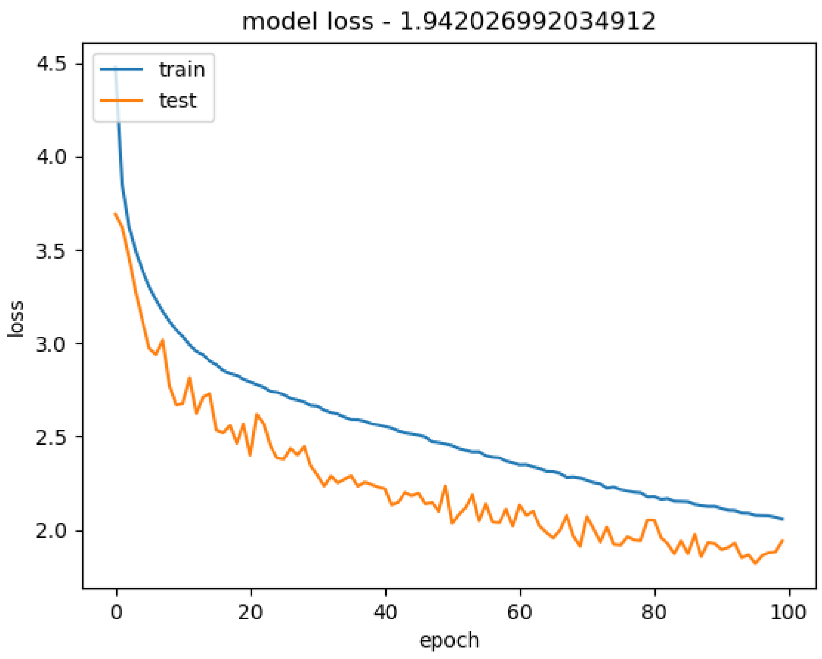

Figure 28: Loss for the system with Dropout and Batch Normalization

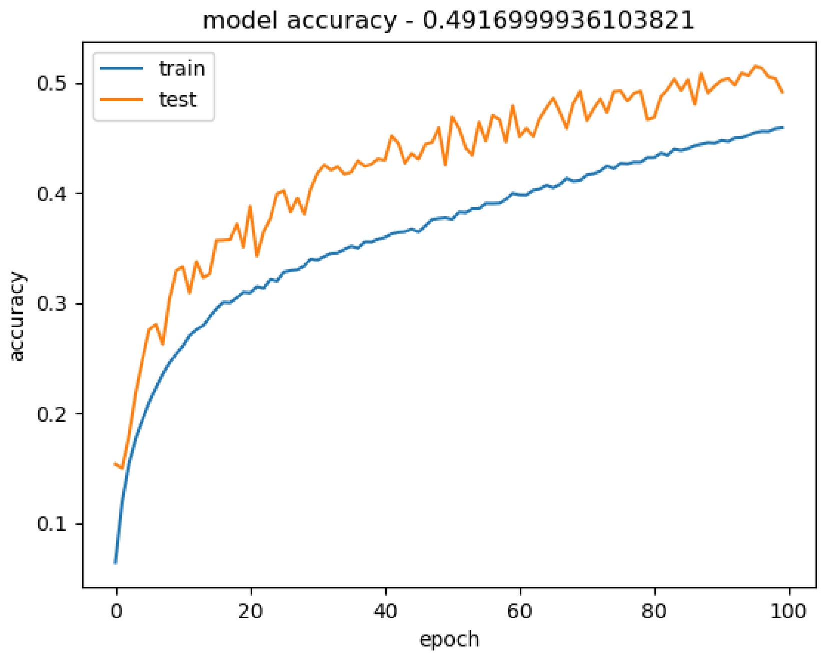

Figure 29: Accuracy for the system with Dropout and Batch Normalization

Similarly applying Dropout and L_2, we find:

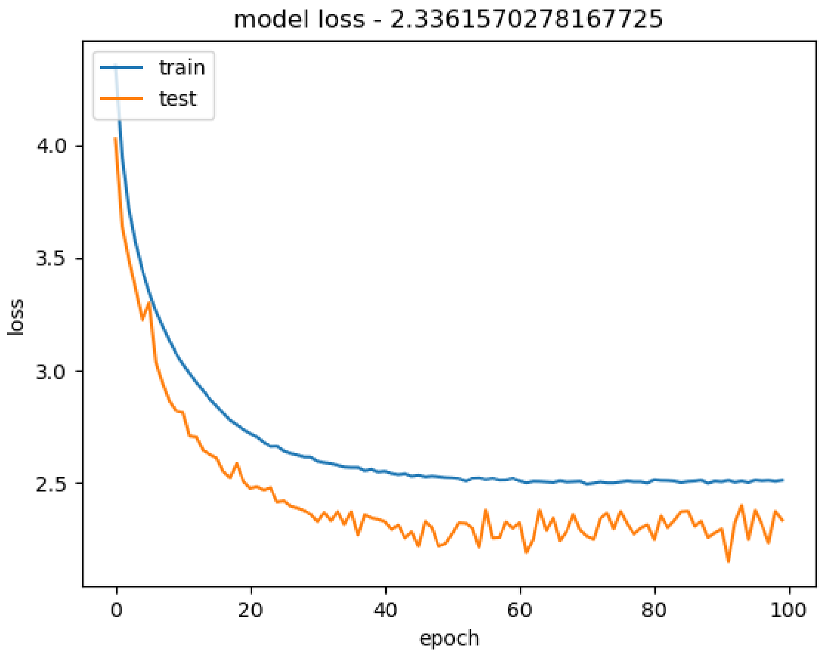

Figure 30: Loss for the system with Dropout and L_2

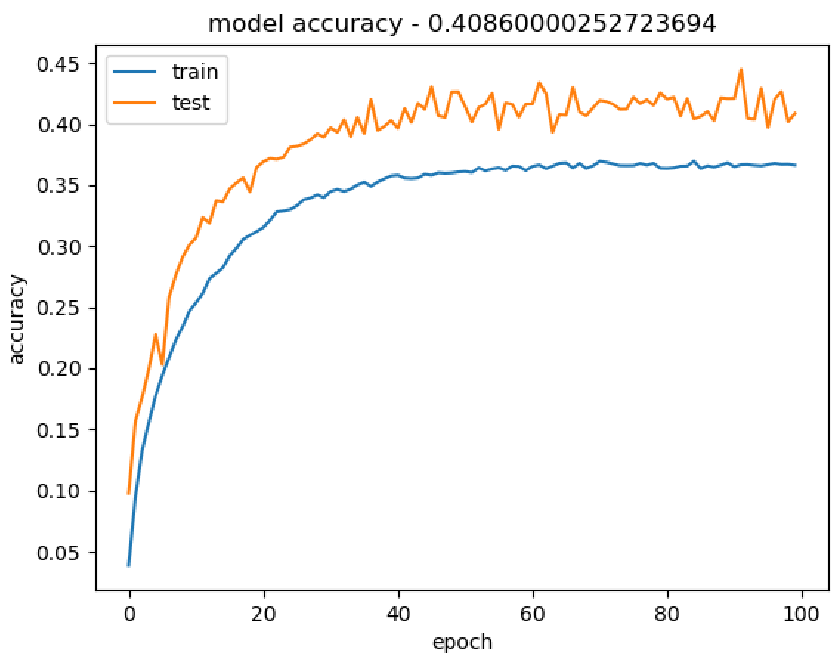

Figure 31: Accuracy for the system with Dropout and L_2

If instead we apply L_2 and Batch Normalization, the results are as shown

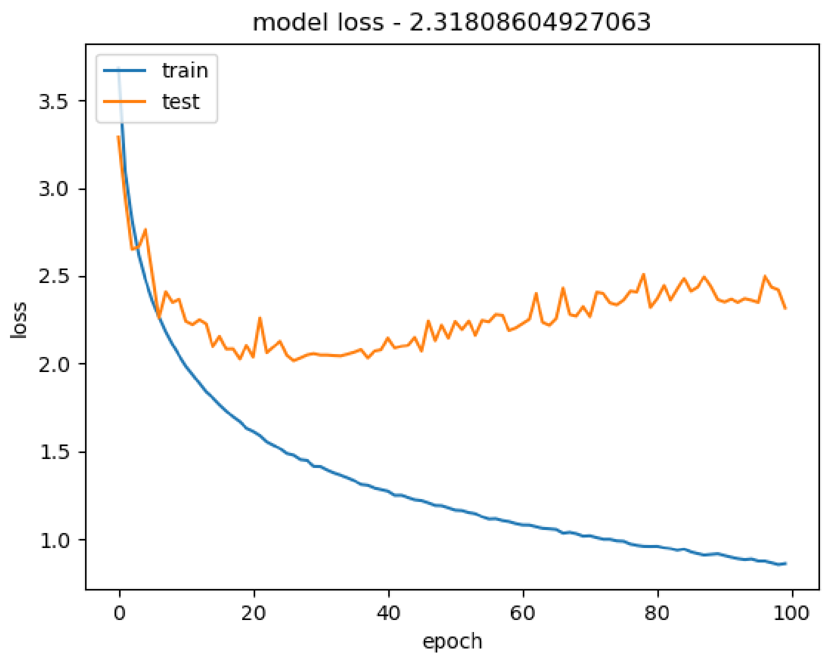

Figure 32: Loss for the system with L_2 and Batch Normalization

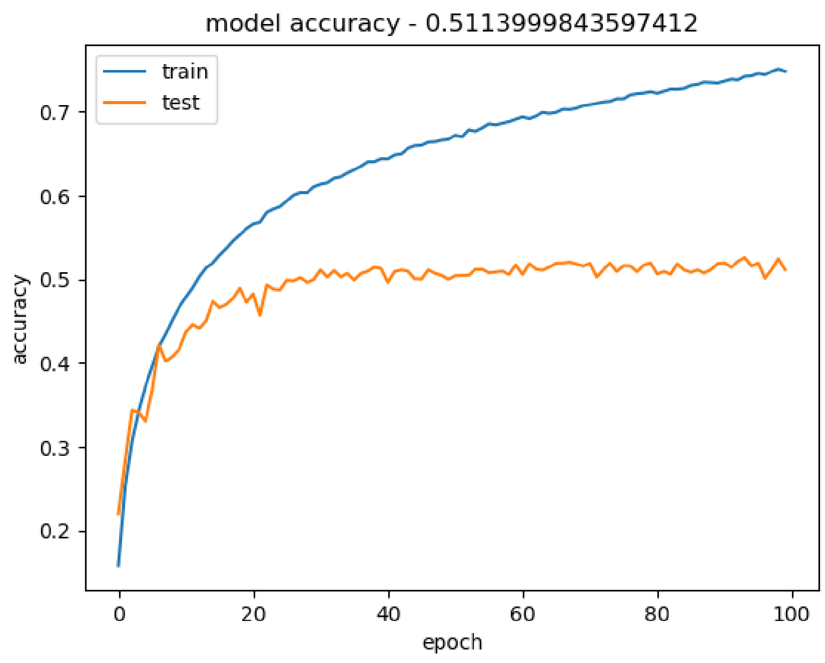

Figure 33: Accuracy for the system with L_2 and Batch Normalization

Finally, including all regularizers, we find:

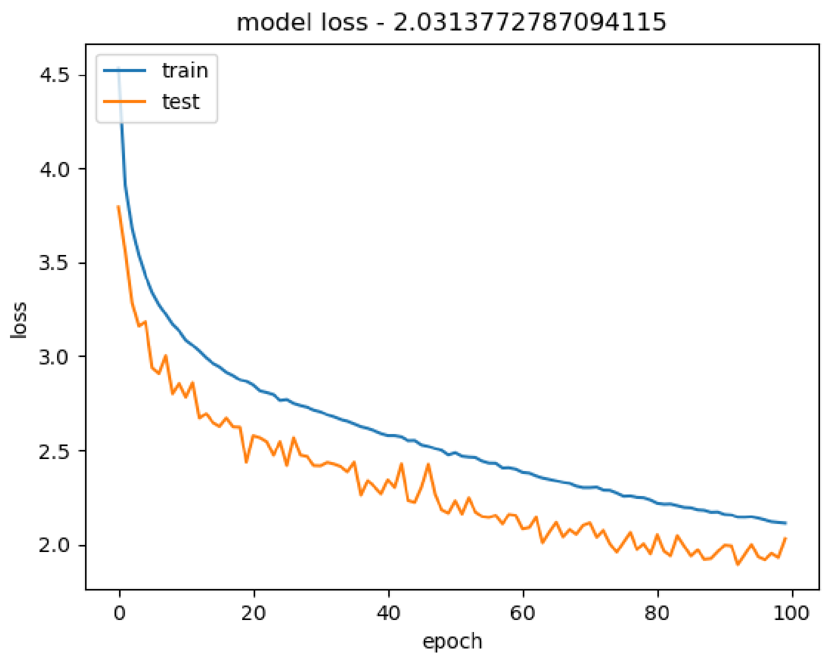

Figure 34: Loss for the system with Dropout,L_2 and Batch Normalization

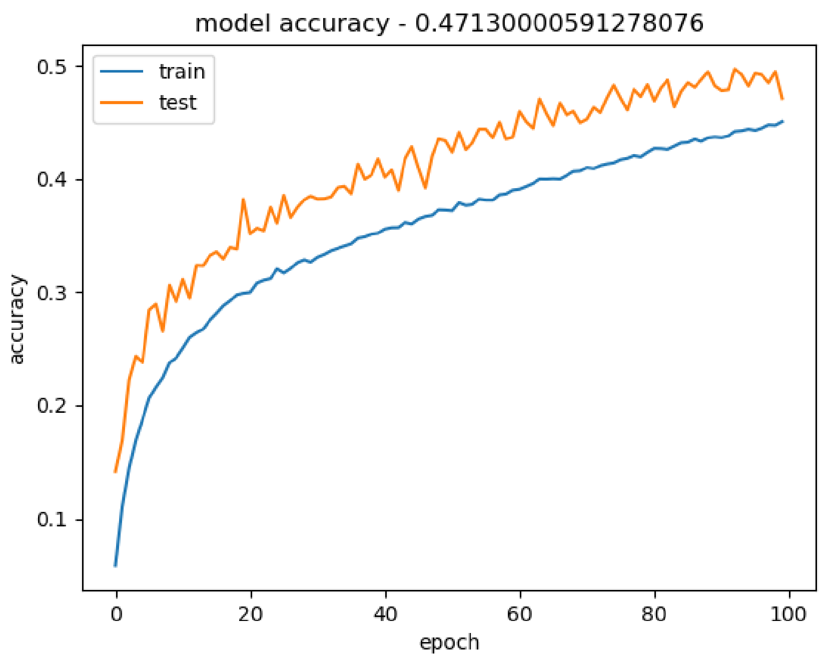

Figure 35: Accuracy for the system with Dropout,L_2 and Batch Normalization

7 Discussion

7.1 MNIST

The immediate fact which stands out from our data is how well the model performed overall with a very high validation accuracy even without any reguralization applied. This can be attributed to how "nice" the MNIST data set really is. It has 10 very distinct classes hence does not require a complicated model to make good predictions. Despite this, from Figure 12 we see the validation and train accuracies converging to a point which indicates potential overfitting.

The first method applied to the network is Dropout performed on a single non-convolutional layer. From Figure 13 we see an immediate improvement, train accuracy was reduced whilst keeping the validation accuracy the same as in the non-reguralized model, to the point where they are now clearly separable. A similar trend can be seen in all models with dropout and another reguralization method. This suggests dropout being beneficial and working as intended reducing risk of overfitting. More discussion about

Dropout is found in the next session.

Batchnorm on the other hand did not seem to have a great effect for reasons mentioned below.

Finally L_2 weight decay will affect the network by keeping the weights at each layer low, which is indeed the case. It has a small but certainly noticeable impact with improving generalization as can be seen from Figure 15 due to a slightly larger value of λ then the one used in the CIFAR models.

Overall, for the MNIST data set, dropout has certainly performed the best. In this particular case it is recommended to keep the dropout on the specifyed layer to improve the function and potentially add L_2 reguralization.

7.2 CIFAR100

On the CIFAR100 dataset, we have shown that dropout has the largest effect, L_2 having minimal effects due to its small parameter (although checking the weights by hand shows that it indeed managed to keep everything close to 0) and batchnorm not performing as could initially be expected if the data were completely different.

We shall now go over the expected results for each of the methods to discuss why the found results match with expectations.

Starting with L_2 regularization, we expect all weights(including the convolutional layer) to be converging to zero. This is not expected to have a massive effect to over-fitting, merely modifying the variances involved by keeping it small, manoeuvring the bias-variance-tradeoff. As stated above, it is found that the weights are of order 10^{-2}, much closer to 0 than a counter example without L_2. As such, L_2 is proposed to behave as intended, helping to reduce the overall loss function, lessening the effect the different number of training to testing examples might have.

Considering Batch Normalization, as stated in section 4.4, it is intended to normalize the distribution such that all input data shall be on the same set. For CIFAR100, it is important to realize that the classes are indeed distinct. Therefore, normalizing inputs brings the superclasses closer together. As CIFAR100 measures subclasses however, it would be expected that batchnormalization does not have that large of an effect overall, which is supported by the data wherein almost no improvement was seen between samples with and without the normalization.

Finally, Dropout provides an interesting issue, as it is performed on both the convolutional blocks and the linear layer (see appendix C). As pointed out in appendix A, Dropout on convolutional layers is extremely unlikely to be capable of reducing overfitting by coadaptation, which is its main purpose on linear layers. Nonetheless, we found a massive increase in model stability when including it, as well as a subsequent rise in overall accuracy. To compare whether this was solely due to the overfitting on the convolutional or linear layers, it is relevant to take a look at appendix B, where we see that the MNIST network applied does not perform dropout on the convolutions, but the results show the same effect. Therefore, we can indeed see that the assumption in Appendix A is correct, and Dropout does not improve convolutional layers, but works very well against the massive amount of data tackled by the linear discrimination layer.

8 Conclusions

In Conclusion, we find that Dropout and L_2 both work very well as regularizers on the CIFAR100 and MNIST datasets, however Batchnorm appears to not work as well on the same sets.

A Dropout on Convolutional Layers

In this appendix, we shall show why Dropout does not perform the same way on convolutional layers as it does on linear ones.

To start with, consider the dropout acting on a linear module. The new layer shall have nodes

$$\underline{\underline{W'}} = \underline{\underline{P}} \odot \underline{W} \tag{A.1}$$

where P contains Bernoulli iid variables. Thus, certain elements in the system are muted. In the case of convolutional networks however, one does not have a weight matrix/vector, but rather a convolution kernel, $\underline{\underline{C}}$, with dimensions (g_1, g_2). Consider now an input image I, of shape (I_1, I_2). The convolution defined in equation 3.2 can be rewritten into a matrix-matrix multiplication as follows.

First off, the corrected matrix shape needs to be calculated, given by $(g_1 + I_1 - 1, g_2 + I_2 - 1)$. The kernel is then zero-padded to this dimension.

For each row l in this padded filter a toeplitz matrix is then generated. A toeplitz matrix is defined such that only constant values appear alongside diagonals, that is, for matrix $\underline{\underline{A}}, \underline{\underline{A}}_{i,j} = \underline{\underline{A}}_{i+1,j+1}$. Once this is done for all rows, the a double blocked form is generated from the matrices, $\underline{\underline{A_l}}$, that is:

$$\underline{\underline{Conv}} = \begin{bmatrix} \underline{\underline{A_0}} & \underline{\underline{0}} \\ \underline{\underline{A_1}} & \underline{\underline{A_0}} \\ \underline{\underline{A_2}} & \underline{\underline{A_1}} \\ \ldots & \ldots a \\ \underline{\underline{A_{n-1}}} & \underline{\underline{A_n}} \end{bmatrix} \tag{A.2}$$

To perform the convolution, the input data needs to be vectorized such that each row is a later section of the vector(ie the first row become the first I_2 elements of the vector. The convolution then takes the form of a vector-matrix multiplication, which can be reconstructed into an output image through the transpose of each portion of the output.

Considering the convolution matrix outlined above, it shall be apparent that, under dropout, not all portions are necessarily muted, that is, because each value of the original filter appears more than once, it cannot be guaranteed that that portion of the filter will be ignored as it would be for a linear layer.

The effect of the dropout layer on a convolution then must be the introduction of errors. Therefore, while dropout tends to be used correct against a surplus of neurons, this can't be guaranteed in the convolutional scenario(where the amount of weights is far lower either way).

The dropout layer can still help in the training of a CNN however, because the drop of a filter portion on some section effectively generates errors in the sample. If then, like outlined in section 3.2, the convolution is followed by a linear layer, it is the data fed into this sequence is erroneous by default. If it is still possible to train this layer, the layer will thus become more robust to errors.

B MNIST code

The code is an adaptation taken from [20] with relevant reguralization methods added. As discussed in 5.1.1, further modification to the code was done by combining the test and train sets, adding cross-validation and optimizing some hyperparameters.

```
import numpy as np
import logging
logging.basicConfig(level=logging.DEBUG)
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout,
    BatchNormalization
from keras.datasets import mnist
from keras import backend
from keras import regularizers
import keras



# input image dimensions
img_rows, img_cols = 28, 28
#Number of classes
classes = 10
#Load the MNIST dataset.
(x_train, y_train), (x_test, y_test) = mnist.load_data()

#Tranform the data.
if backend.image_data_format() == 'channels_first':
    x_train = x_train.reshape(x_train.shape[0], 1, img_rows, img_cols)
    x_test = x_test.reshape(x_test.shape[0], 1, img_rows, img_cols)
    input_shape = (1, img_rows, img_cols)
else:
    x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, 1)
    x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, 1)
    input_shape = (img_rows, img_cols, 1)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
x_whole = np.concatenate((x_train,x_test))

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, classes)
y_test = keras.utils.to_categorical(y_test, classes)
y_whole = np.concatenate((y_train,y_test))

#Number of folds.
n_splits = 5
#Initialise the cross validaion.
kfold = KFold(n_splits, shuffle=True)
#Used to store the scores over the folds.
cvscores= []
#Used to store accuracies over epochs.
histories = []
fold_number = 1
for train, test in kfold.split(x_whole):
    #Setup the model.
    print('Fold :' +str(fold_number))
    fold_number += 1
    #Basic model parameters without any regularization. Appropriate
       reguralizers can be commented in.
    cnn = Sequential()
    cnn.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
        input_shape=input_shape,))
# ==============================================================================
#     cnn.add(Conv2D(32, kernel_size=(3, 3),activation='relu',
#
    input_shape=input_shape,activity_regularizer=regularizers.l2(0.001)))
# ==============================================================================
    #cnn.add(BatchNormalization(axis = 1))

    #cnn.add(Conv2D(64, (3, 3),
        activation='relu',activity_regularizer=regularizers.l2(0.001)))
    cnn.add(Conv2D(64, (3, 3), activation='relu'))
    #cnn.add(BatchNormalization(axis = 1))
    cnn.add(MaxPooling2D(pool_size=(2, 2)))
    cnn.add(Flatten())

    cnn.add(Dense(128,
        activation='relu',activity_regularizer=regularizers.l2(0.001)))
    #cnn.add(Dense(128, activation='relu'))
    #cnn.add(BatchNormalization())
    #cnn.add(Dropout(0.5))

    cnn.add(Dense(classes, activation='softmax'))
    cnn.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])
    #Fit the model using specifyed hyper parameters.
    history = cnn.fit(x_whole[train], y_whole[train],
            batch_size=100,
            epochs=6,
            verbose=1,
            validation_data=(x_test, y_test))
    #Evaluate the models's performance.
    score = cnn.evaluate(x_whole[test], y_whole[test], verbose=0)
    print('Test loss:', score[0])
    print('Test accuracy:', score[1])
    cvscores.append(score)
    histories.append(history)
print(cvscores)
```

C CIFAR Network

The following code has been taken from [12], where it has been listed to tackle the CIFAR10 [11] dataset, which only contains 10 classes. It has been adapted to fit the 100 classes, and cross validation for 10% of the testset has been performed to select hyper-parameters. Running this code will result in the generation of the final CIFAR network, running all regularizers with their proposed values.

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 30 20:26:36 2019

@author: Originally the Keras Team - taken from
    https://keras.io/examples/cifar10_cnn/ and adapted by Martin Sanner
"""

from __future__ import print_function
import tensorflow.keras as keras
from tensorflow.keras.datasets import cifar100
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D,BatchNormalization
import os
import matplotlib.pyplot as plt
import numpy as np

batch_size = 32
num_classes = 100
epochs = 100
data_augmentation = True
num_predictions = 20
save_dir = os.path.join(os.getcwd(), 'saved_models')

# The data, split between train and test sets:
(x_train, y_train), (x_test, y_test) = cifar100.load_data()
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)

model = Sequential()
model.add(Conv2D(32, (3, 3), padding='same',
               input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(BatchNormalization(axis = 3))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))

model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(BatchNormalization(axis = 3))

model.add(Dropout(0.25))


model.add(Flatten())

model.add(Dense(512))

model.add(Activation('relu'))

model.add(Dropout(0.5))

model.add(Dense(num_classes))

model.add(Activation('softmax'))


# initiate RMSprop optimizer

opt = keras.optimizers.RMSprop(learning_rate=1e-4, decay=1e-6)


model_name =

    'keras_cifar100_droupout_BN_trained_{}_{}_model.h5'.format(-np.log10(1e-4),-np.log10(1e-


# Let's train the model using RMSprop

model.compile(loss='categorical_crossentropy',

              optimizer=opt,

              metrics=['accuracy'])


x_train = x_train.astype('float32')

x_test = x_test.astype('float32')

x_train /= 255

x_test /= 255


if not data_augmentation:

    print('Not using data augmentation.')

    model.fit(x_train, y_train,

              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False, # set input mean to 0 over the dataset
        samplewise_center=False, # set each sample mean to 0
        featurewise_std_normalization=False, # divide inputs by std of the
            dataset
        samplewise_std_normalization=False, # divide each input by its std
        zca_whitening=False, # apply ZCA whitening
        zca_epsilon=1e-06, # epsilon for ZCA whitening
        rotation_range=0, # randomly rotate images in the range (degrees, 0 to
            180)
        # randomly shift images horizontally (fraction of total width)
        width_shift_range=0.1,
        # randomly shift images vertically (fraction of total height)
        height_shift_range=0.1,
        shear_range=0., # set range for random shear
        zoom_range=0., # set range for random zoom
        channel_shift_range=0., # set range for random channel shifts
        # set mode for filling points outside the input boundaries
        fill_mode='nearest',
        cval=0., # value used for fill_mode = "constant"
        horizontal_flip=True, # randomly flip images
        vertical_flip=False, # randomly flip images
        # set rescaling factor (applied before any other transformation)
        rescale=None,
        # set function that will be applied on each input
        preprocessing_function=None,
        # image data format, either "channels_first" or "channels_last"
        data_format=None,
        # fraction of images reserved for validation (strictly between 0 and 1)
        validation_split=0.0)

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    hist = model.fit_generator(datagen.flow(x_train, y_train,
                                     batch_size=batch_size),
                        epochs=epochs,
                        validation_data=(x_test, y_test),
                        workers=1)

# Save model and weights
if not os.path.isdir(save_dir):
    os.makedirs(save_dir)
mode$L_p$ath = os.path.join(save_dir, model_name)
model.save(mode$L_p$ath)
print('Saved trained model at %s ' % mode$L_p$ath)

# Score trained model.
scores = model.evaluate(x_test, y_test, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])




print(hist.history.keys())
# summarize history for accuracy
```

```
plt.figure()
plt.plot(hist.history['acc'])
plt.plot(hist.history['val_acc'])
plt.title('model accuracy - {}'.format(scores[1]))
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(model_name[:-3]+"_accuracy.png")
plt.show()
# summarize history for loss
plt.figure()
plt.plot(hist.history['loss'])
plt.plot(hist.history['val_loss'])
plt.title('model loss - {}'.format(scores[0]) )
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig(model_name[:-3]+"_loss.png")
plt.show()
```

References

[1] Deepai defition of machine learning. `https://deepai.org/machine-learning-glossary-and-terms/machine-learning`. Accessed: 2019-12-06.

[2] History of machine learning. `https://www.doc.ic.ac.uk/~jce317/history-machine-learning.html`. Accessed: 2019-12-06.

[3] Sebastian Raschka. *Python Machine Learning*. Packt Publishing, 2015.

[4] Activation functions in neural networks. `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`. Accessed: 2019-12-06.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] J. Brownlee. *Better Deep Learning: Train Faster, Reduce Overfitting, and Make Better Predictions*. Machine Learning Mastery, 2018.

[7] J. Brownlee. *Probability for Machine Learning: Discover How To Harness Uncertainty With Python*. Machine Learning Mastery, 2019.

[8] Understanding the mathematics behind gradient descent. `https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5`
Accessed: 2019-12-08.

[9] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2006.

[10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[11] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[12] Train a simple deep cnn on the cifar10 small images dataset. `https://keras.io/examples/cifar10_cnn/`. Accessed: 2019-12-06.

[13] Jeff Heaton. *Introduction to Neural Networks for Java, 2nd Edition.* Heaton Research, Inc., 2nd edition, 2008.

[14] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. Springer New York Inc., 2001.

[15] M.T. Hagan, H.B. Demuth, M.H. Beale, and De Jes. *Neural Network Design.* Martin Hagan, 2014.

[16] N. Buduma and N. Locascio. *Fundamentals of Deep Learning: Designing Next-generation Machine Intelligence Algorithms.* O'Reilly Media, 2017.

[17] Batch normalization: theory and how to use it with tensorflow. `https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad`. Accessed: 2019-12-08.

[18] Batch normalization in neural networks. `https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c`. Accessed: 2019-12-09.

[19] Understanding dropout with the simplified math behind it. `https://towardsdatascience.com/simplified-math-behind-dropout-in-deep-learning-6d50f3f47275`. Accessed: 2019-12-08.

[20] Train a simple convnet on mnist. `https://keras.io/examples/mnist_cnn/`. Accessed: 2019-12-06.