THE UNIVERSITY OF SYDNEY

Assignment 1

COMP5329

Assignment 1

Niruth Bogahawatta	480316176
Michael Podbury	440173010
Billy Dodds	480380144

Contents

1 Introduction

The field of artificial intelligence has exploded in recent decades due largely to the invention of the artificial neural network (NN). The idea of implementing biological neural networks like the brain into a computational model was first proposed in 1943. After several hype cycles, the idea saw stagnation plunging the field into the "AI winter" of the 1970s and 1980s. A combination of algorithmic innovation and increases in computational power led to the field's resurgence in the late 1990s, and now NNs form the cornerstone of modern AI.

In this paper, we aim to investigate how many of the innovations in NNs impact a model's ability to learn and perform classification problems. To achieve this, we will first implement a multi-layer perceptron (MLP) NN with several algorithms and components from the deep learning literature. The components we implement include different weight initialization approaches, ReLu and leaky ReLu activations, softmax and cross-entropy, stochastic gradient descent (SGD) with and without momentum, learning rate schedule, Adam optimization, mini-batch training, batch normalisation, weight decay, and dropout. This self-implementation approach will give us an intimate understanding of the components from which our models will be built. Finally, we will build several models leveraging various combinations of components and test their performance on a basic multi-class classification problem. Undertaking this study is important for gaining a deeper understanding of current NN algorithms and their effects on model training and performance, which can ultimately lead to important future research paths in the field.

2 Methods

2.1 Preprocessing

The provided dataset for the multi-class classification problem consists of 60,000 observations, each with 128 features and a class label from 0 to 9. This dataset was split into a training set of 40,000, a validation set of 10,000, and a test set of 10,000. There were four main preprocessing techniques explored in our modelling: normalisation, standardisation, principal component analysis (PCA), and one-hot encoding.

2.2 Standardisation and Normalisation

Standardisation and normalisation are two preprocessing techniques that aim to reduce bias present in particular features by scaling them to similar ranges. If some features have a larger scale than others, the model may become biassed towards these features. Standardisation is one approach to addressing this, and involves scaling each feature across the dataset to have unit variance and zero mean by taking each observation from a feature, subtracting the mean across all observations for that feature, and dividing by the standard deviation across all observations of that feature (Equation1).

Another approach is max-min normalisation which scales the input data to a range of [0,1] by taking each observation and subtracting the minimum value across all observations for that feature and dividing by the difference between the max and min values across all observations for that feature (Equation 1). In our experimentation, we investigate how both scaling operations impact the model performance.

$$x_{new} = \frac{x - \mu}{\theta} \tag{1}$$

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2}$$

2.3 PCA

Another preprocessing technique we investigated was PCA, which proposes a method of feature reduction. PCA first performs an eigen decomposition on the covariance matrix of the mean-centred data, obtaining a set of eigenvalues and eigenvectors. Each eigenvector represents a different linearly independent direction in which the dataset varies the most, with the amount of variation dictated by the size of the associated eigenvalue. To generate the principal components of the dataset, the dataset is projected onto the eigenvector space. The principal component associated with the largest eigenvalue is the first principal component, capturing the most variance in the dataset, and the second largest is the second principal component, and so on. Taking the first 'n' components is a way of capturing a large portion of the variance in the dataset in a minimal amount of features, hence reducing the feature set. This can be advantageous in models that have many features. Our dataset contains 128 features, which is quite small compared to modern deep learning standards, but we have still investigated whether our models will benefit from this feature reduction approach.

2.3.1 One Hot Encoding

A final preprocessing technique crucial for multi-class classification is one-hot encoding. The model we aim to train will output a vector of probabilities corresponding to the likelihood of that observation belonging to each class (explained further in the softmax function). Because of this, the training examples need to be prepared in this same probability vector format. To do this, we assign a probability of 1 to the true label for that particular observation and a probability of 0 to all other labels. In our case, these probability vectors are of length 10 meaning a label of 2 will be encoded as a vector of [0,0,1,0,0,0,0,0,0,0]. Unlike the previous preprocessing methods which can be tried and adopted only if performance improves, one-hot encoding is an essential step for all of our model training.

3 Modules

3.1 Architecture

3.1.1 Multi-Layer Perceptron (MLP)

The basic architecture for our NN will be the multi-layer perceptron (MLP). This is a feedforward network with multiple hidden layers between the input and output nodes. One of the most powerful theorems relating to NNs is the universality theorem (Hecht-Nielsen, 1992) which states that given enough nodes, a single hidden layer can approximate any function. However, this is purely an existence theorem and may require the NN to consist of an infinite number of hidden nodes and fed an infinite amount of data. In practice, this often requires training a network with many more parameters than training observations, increasing the training time and chance of overfitting. AI researchers have had much more success in increasing the depth of the NN, hence birthing the field of deep learning.

The basic architecture for our NN will be the multi-layer perceptron (MLP). This is a feedforward network with multiple hidden layers between the input and output nodes. Each hidden layer is fully connected to the layer before and after it. Hidden layers allow a NN to learn complex, non-linear mappings that can assist in complicated classification tasks that other models like support vector machines (SVM) and decision trees may struggle to learn.

While additional hidden layers enhance the model's capability to learn complex mappings, having too many layers can make the NN prone to overfitting. Overfitting occurs when the model overly tunes its parameters to capture variations in the training set, increasing the training accuracy but reducing the model's ability to generalise to unseen data. More layers also increases the number of parameters in the training process, which can increase the training time. As a result, the ideal NN should have more than one hidden layer, but cannot be too deep.

3.1.2 Weight initialization

The MLP architecture described above is only able to learn complex functions through incremental adjustments of its weight and bias parameters. Each node has a vector of weights and biases corresponding to each incoming connection from the layer before it. The input from each prior layer node is weighted by these parameters, and an activation function is applied to the sum of these weighted signals to create the output fed to the next layer. It is in these weights that information is contained. The training algorithm responsible for incrementally adjusting these weights is called backpropagation. This algorithm works by first computing the gradient of the loss function with respect to the weight parameters of the NN one layer at a time, starting from the final output layer and iterating backward. The weights are then updated in the negative direction of the gradient, the direction that minimises the loss. This is called gradient descent. The size of the update is dictated by the learning rate.

An important consideration in minimising the time of convergence and reaching a low local minimum is choosing smart initial weights. Conventionally, the weights are initialised to zero or random values. These initialisation techniques have been shown to produce poor performance on most deep learning tasks, increasing the risk of vanishing and exploding gradients. Xavier and Kaiming are two initialisation methods implemented in our modelling, which claim to be effective for hyperbolic tangent (tanh) and rectified linear unit (ReLu) activation functions respectively. In addition to these two initialisation techniques, we have also included a constant initialisation, set to either zero or a small positive decimal. We anticipate that either Xavier or Kaiming will outperform the more naive constant approach.

3.1.3 ReLu activation and Leaky ReLu

An essential component of every node in a neural network is an activation function. This function dictates how a node translates its inputs from the prior layer into its output to the next layer. The input from each prior layer node is weighted by its weight and bias parameters, and the activation function is applied to the sum of these weighted signals. The distribution of the resultant output is dependent on the activation function, and thus the choice of activation function constitutes a key decision in the design of an NNs architecture.

An activation function should have the following properties. Firstly, it should be nonlinear to facilitate the learning of non-linear mappings. Secondly, the activation functions should be continuous and differentiable almost everywhere so that gradient descent can be performed. Finally, the activation function should be monotonic in either the increasing or decreasing direction. If the activation function isn't monotonic then increasing the neuron's weight might cause it to have less influence resulting in chaotic behaviour during training, hindering convergence.

The most commonly used activation functions for MLPs include the sigmoid function (Fig. 1a), the tanh function (Fig. 1b), and the rectified linear unit or ReLu (Fig. 1c). The sigmoid function transforms the input into values between 0 and 1, and tanh outputs values between -1 and 1. These two functions have found great success in the field, but have the drawback of being prone to vanishing gradients. This comes from them having near-zero gradients at extreme input values, causing the calculated gradient to approach zero for large activation inputs, inhibiting the weights from changing their value. This stifles the learning process.

One method of addressing this potential issue is the ReLu activation function. ReLu outputs zero if the activation input is zero or negative and outputs the unmodified activation input when it is greater than zero. This fixes the potential for vanishing gradients in the positive direction, but does still leave it prone to the dying ReLu problem in the negative direction and to exploding gradients in the positive direction. Dying ReLu is similar to vanishing gradients and occurs when negative activations result in a zero gradient causing a node to become inactive. This problem can be mitigated by adopting a modified ReLu function called leaky ReLu (Fig. 1d), which outputs a small linear component of the input at negative values, rather than just zero. However, even leaky ReLu is still susceptible to exploding gradients, which occurs when large error gradients accumulate in the positive direction causing large updates and unstable convergence. Despite this limitation, an additional benefit of ReLu-based activation functions is that they are more computationally efficient than sigmoid and tanh.

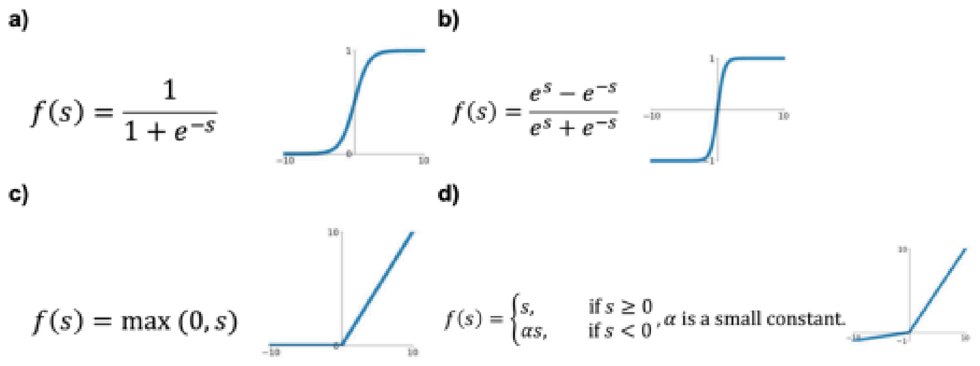


Figure 1: Common activation functions used in neural network construction: a) sigmoid function; b) tanh function; c) rectified linear unit (ReLU); d) leaky rectified linear unit (leaky ReLU). Figures reprinted from Dr. Chang Xu (2021) Multilayer Neural Networks. COMP5329 Lecture 2. University of Sydney.

3.1.4 Softmax with cross-entropy

Softmax functions are used for multi-class classification problems, working to transform the output of a NN into a set of conditional probabilities. These probabilities represent the likelihood of an observation belonging to a particular class(Equation 3). Since the output is a k-dimensional vector (where k is the number of classes), regular loss functions such as mean squared error (MSE) can not be used. Therefore multiclass NN classifiers using softmax are generally paired with cross-entropy loss. Here the softmax functions give the probability distribution for the observed values and ground truth and the cross-entropy transforms these two probability distributions into a distance measure, quantifying how close the output distribution is to the actual distribution (Equation 4). The gradient of this cross-entropy loss is what is leveraged by the backpropagation algorithm to update the weights and train the network.

$$\hat{P}(class_k|x) = \frac{e^{net}k}{\sum e^{net}k} \tag{3}$$

$$CrossEntropy(t, z) = -\sum t_i log t_i + \sum t_i log \frac{t_i}{z_i} \tag{4}$$

3.2 Optimisation

The next set of modules explored in our model training process are algorithms and techniques that assist in the optimisation process.

3.2.1 Learning rate scheduling

The first technique implemented is that of learning rate scheduling. The learning rate is the hyperparameter that determines how much the weights will update in the direction of the negative gradient. If the learning rate is too large the steps can overshoot the optimal gradient descent path, and if it is too small, the time of convergence will increase to unfeasible levels, if it converges at all. To mitigate this, learning rate schedulers were introduced to adapt the learning rate as training progresses. This allows the algorithm to make larger changes at the beginning of the training procedure (when the gradients are supposedly quite far from a minimum), and gradually reduce the changes as the training progresses (and the gradients are supposedly closer to a minimum).

There are three main types of learning rate schedules. Time-based decay, step decay, and exponential decay. Time-based decay decreases the learning rate by a time step factor (Equation 5). Step decay decreases the learning rate systematically and drops the learning rate at specific times during training (Equation 6). Finally, exponential decay decreases the learning rate on an exponential decay curve (Equation 7). Each function works to achieve a similar means.

$$lr = lr * \frac{1}{1 + k * epoch} \tag{5}$$

$$lr = lr_0 * pow(k, floor(\frac{epoch}{step})) \tag{6}$$

$$lr = lr_0 * e^{-k*epoch} \tag{7}$$

Each of these learning rate schedules allow us to change the learning rate during training phases, however it achieves this in a predefined way which may or may not be beneficial to the specific problem. Further, it still uses the same learning rate across all parameters which is not always appropriate. Other approaches like the Adam optimiser address this limitation.

3.2.2 SGD and SGD with momentum

As we have touched on before, a NN learns by optimising and updating its weight parameters using gradient descent (Equation 8). Gradient descent can be performed in multiple ways. Batch gradient descent updates the weight and biases for each node only after all training examples have been evaluated. Performing gradient descent for each weight and bias in each node before updating might result in a more accurate gradient, but is very computationally inefficient. To increase efficiency, stochastic gradient descent (SGD) can be used. Here, the weight and biases are updated after performing gradient descent on one training observation at a time (Equation 9). Despite often resulting in a less-accurate gradient, there are large efficiency gains that come from the faster computation.

Despite being faster, SGD has high variance and tends to fluctuate significantly while converging. At times this can be an advantage allowing it to escape local minima and move towards global minima, but can lead to oscillations that slow or even prevent it from converging to a minima. In particular, SGD has difficulty navigating ravines or areas in which the parameter space surface curves much steeper in one dimension than in others. To mitigate this, learning rate scheduling or other means of reducing the learning rate could assist.

Another method to address oscillations in SGD involves introducing what is called a momentum term. This momentum term helps to accelerate convergence by damping oscillations. The momentum term consists of a weighted summarisation of historical gradients and is added to the regular gradient descent update (Equation 10). The momentum parameter will determine the influence of the previous time step in the current update step. A disadvantage of momentum-based SGD is that it can overshoot a minima and is not effective at handling saddle points. The former can be mitigated by Nesterov Accelerated gradients (NAG) which introduces a correction term to account for overshooting, however we did not explore NAG in this study.

$$\Delta\theta = \eta\frac{\delta J}{\delta\theta} \tag{8}$$

$$\Delta\theta = \eta\frac{\delta J(\theta; x^{(i)}; y^{(i)})}{\delta\theta} \tag{9}$$

$$v_t = \gamma v_{t-1} + \eta\Delta_\theta J(\theta), \theta_t = \theta_{t-1} - V_t \tag{10}$$

3.2.3 Mini batch training

Having explored both batch and SGD, it is obvious that there are benefits and drawbacks to each. Mini batch training can be considered a middle ground between batch and SGD. As mentioned earlier batch gradient descent tends to be inefficient, and stochastic gradient descent can be volatile and overshoot the minima. Mini batch training aims to find a compromise between the two by updating the weights and biases after performing gradient descent on a subset or "mini batch" of n observations (Equation 11). In doing so, mini batch training tends to be more stable (figure 2) and computationally inexpensive(Fig. 2).

$$\Delta\theta = -\eta\frac{\delta J(\theta; x^{(i)}; y^{(i)})}{\delta\theta} \tag{11}$$

3.2.4 Batch normalisation

Another optimisation technique is that of batch normalisation. We have already explored how feature scaling techniques like normalisation can be beneficial in reducing bias towards larger features. Here we scale the input data, but there is often a need to perform normalisation on the input

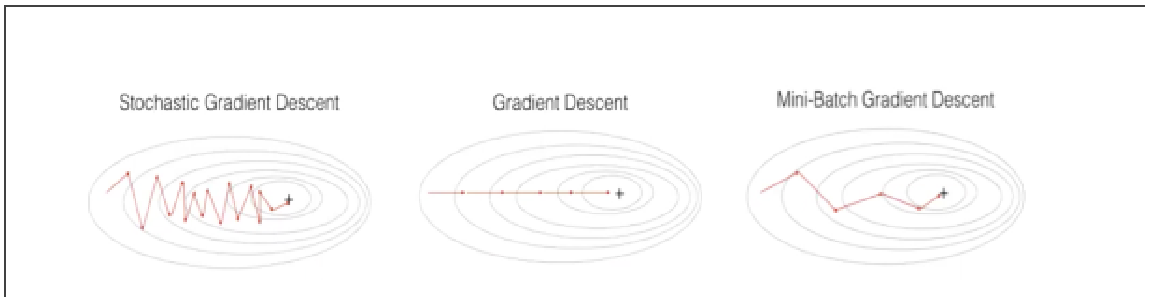


Figure 2: A comparison of stochastic gradient descent (left); [batch] gradient descent (middle); and mini-batch gradient descent (right).

to hidden layers as well. This is the fundamental idea of batch normalisation, and often works to successfully improve performance due to addressing the phenomenon of internal covariance shift.

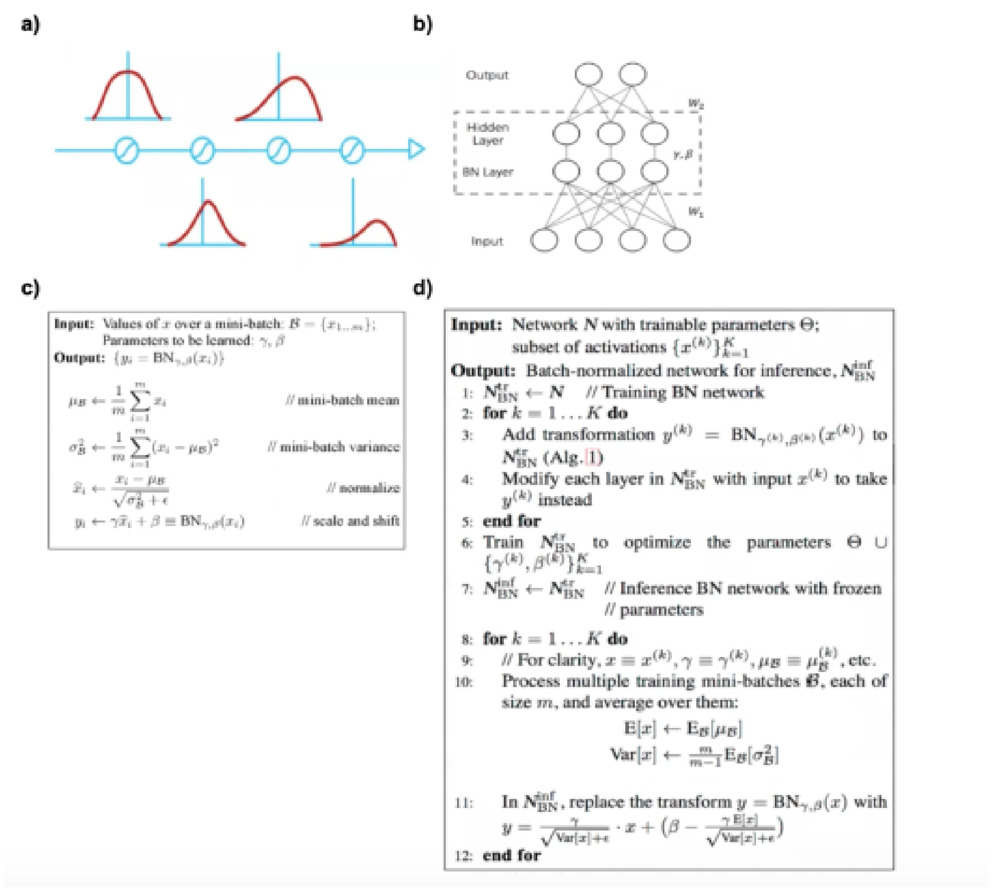
As seen in figure 3a, internal covariance shift happens when the output of a particular hidden layer deviates in distribution from the original input and cascades that deviation on subsequent layers until the final distribution is very different from the initial inputs. This is undesirable as it forces each layer to adapt to different distributions and leads to inefficient, difficult to implement, and inaccurate models. Further, internal covariance shifts could result in exploding or vanishing gradients as it cascades and amplifies changes through the layers of a NN.

Batch normalisation works to overcome internal covariance shift by applying a normalisation step between layers such that the output nodes within the layer are normalised before being fed into the next layers. During the normalisation step, a scale () and shift () parameter are defined for each layer to bring the distribution back to its original spread and location. Figure 3d illustrates the mathematical description of the batch norm with mini batch, showing how these parameters are estimated from the training data.

There are many advantages to batch normalisation. The first is the reduction in training time, as less internal covariance shift provides stability for higher learning rates. Additionally, batch normalisation makes the hidden layer in NN more resilient to parameter scales by stabilising parameter growth. As the normalisation prevents input values to layer from getting stuck in saturating high or low ranges it can also aid in training NN with activation functions such as the sigmoid and tanh mitigating against vanishing gradients. The main disadvantage of batch normalisation is that it cannot handle small batch sizes due to noisy estimations of the means and variances. This also can limit its performance on the test set if these estimations aren't reflective of the test set distribution.

3.2.5 Adam optimizer

The final optimisation algorithm that we explored was Adam. Up until this point, all of our optimisation algorithms such as mini-batch, SGD assume an equal learning rate across all features.



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$      // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$                    // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$            // scale and shift
```

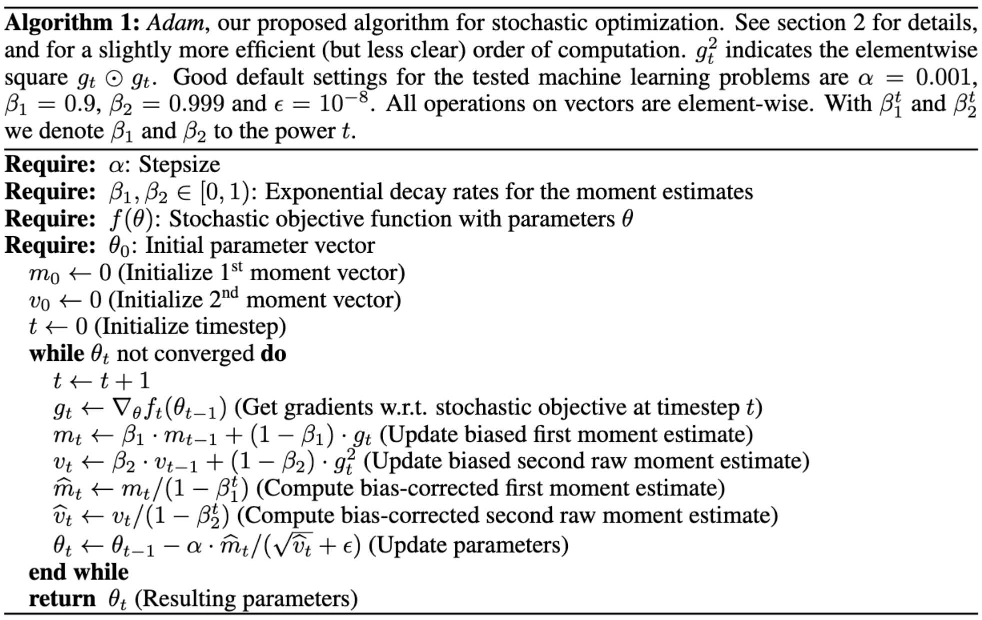
```
Input: Network $N$ with trainable parameters $\Theta$;
       subset of activations $\{x^{(k)}\}_{k=1}^{K}$
Output: Batch-normalized network for inference, $N^{\mathrm{inf}}_{\mathrm{BN}}$
1: $N^{\mathrm{tr}}_{\mathrm{BN}} \leftarrow N$    // Training BN network
2: for $k = 1 \dots K$ do
3:     Add transformation $y^{(k)} = \mathrm{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to
       $N^{\mathrm{tr}}_{\mathrm{BN}}$ (Alg. 1)
4:     Modify each layer in $N^{\mathrm{tr}}_{\mathrm{BN}}$ with input $x^{(k)}$ to take
       $y^{(k)}$ instead
5: end for
6: Train $N^{\mathrm{tr}}_{\mathrm{BN}}$ to optimize the parameters $\Theta \cup$
   $\{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N^{\mathrm{inf}}_{\mathrm{BN}} \leftarrow N^{\mathrm{tr}}_{\mathrm{BN}}$    // Inference BN network with frozen
                    // parameters
8: for $k = 1 \dots K$ do
9:     // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10:    Process multiple training mini-batches $\mathcal{B}$, each of
       size $m$, and average over them:
             $\mathrm{E}[x] \leftarrow \mathrm{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$
             $\mathrm{Var}[x] \leftarrow \frac{m}{m-1}\mathrm{E}_{\mathcal{B}}[\sigma^2_{\mathcal{B}}]$
11:    In $N^{\mathrm{inf}}_{\mathrm{BN}}$, replace the transform $y = \mathrm{BN}_{\gamma,\beta}(x)$ with
       $y = \frac{\gamma}{\sqrt{\mathrm{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\mathrm{E}[x]}{\sqrt{\mathrm{Var}[x]+\epsilon}}\right)$
12: end for
```

Figure 3: Internal covariate shift: circles represent nodes and curves represent distribution of outputs moving from left to right - in this figure the output means are shifting to the right as they are fed into subsequent nodes; b) batch normalisation layer integrated into NN: gamma and beta represent scale and shift parameters; c) input, output, and equations used in batch normalisation 2 is the mini- using mini-batch gradient descent: μ_B is the mini-batch mean, σ_B batch variance, x_i are the input values, $\hat{x}_i$i are the normalised input values, ϵ is a small constant added to the mini-batch variance for numerical st

This might not be appropriate as features will vary in their importance; each feature dimension will most likely have a different slope. Adam is an algorithm that aims to address this shortcoming, implementing a feature-dependent learning rate, known as an 'adaptive learning rate'. Other algorithms in this category include Adagrad, Adadelta, and RMSprop. The first three of these optimizers do not include momentum in their calculations hence we focused on utilising the Adam optimizer which considers both momentum and adaptive learning rates.

Like SGD, Adam is a stochastic optimisation algorithm, bringing with it the benefits of minimal memory requirements. Adaptive learning rates are calculated for each feature from that feature's first and second gradient moments. "Adaptive moment" is where the name Adam is derived from.

Formally, the algorithm calculates the exponentially decaying moving average of the gradient and square of the gradient and uses this to estimate the first and second moments for each feature. Two hyperparameters, 1 and 2 control the decay rate used in the first and second moment calculations respectively. The result is that each weight is updated individually, allowing more important features to take larger learning steps than less important ones. Figure 4 shows the pseudocode demonstrating this.



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Figure 4

3.3 Regularisation

The final set of modules explored in our study relate to regularisation. Regularisation can be broadly defined as any technique that works to reduce overfitting in the training process.

3.3.1 Weight decay

The first regularisation technique explored in our report is weight decay. Weight decay controls model complexity by penalising larger weights. Weight decay can be divided into two forms: the researcher's methods and the engineer's methods.

The research method is a theoretically proven approach that utilises L2 regularisation (Equation 13). L2 regularisation minimises the training error concerning model parameters by adding a squared term to penalise parameter magnitudes. This can be observed in equation 12 where the regular mean squared error has an additional term that is a fraction of the sum of the squared weight values. Mathematically, this leads to a change in the weight gradients, which in turn reduces the value of each weight on each training iteration. Though this is theoretically sound, this method of weight decay is quite difficult to implement and requires more intensive computing power to calculate.

The engineer's methods of weight decay alleviates this problem by updating the weights themself at each iteration (Equation 14). This makeshift method has no theoretical foundation but has shown to be empirically effective. Further, it's more computationally efficient to implement.

$$min\hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\alpha}{2}||\theta||_2^2 \tag{12}$$

$$min\hat{L}_R(\theta) = \frac{1}{n}\sum l(\theta, x_i, y_i) + \lambda * ||\theta||_2^2 \tag{13}$$

$$\theta \leftarrow \theta(1-\alpha) \text{ where } \alpha \text{ is a small constant } (0.02)) \tag{14}$$

3.3.2 Dropout

In addition to weight decay, dropout is another regularisation technique that can be used to prevent the overfitting of a model. Each iteration, some number of nodes are randomly dropped from the model architecture with some probability p. In doing this, the network can not become over-reliant on certain nodes, helping to prevent overfitting.

Once the model is fully trained, all the nodes of the NN are restored and the weights are downscaled by the dropout probability (ie. if p was 0.5 for a particular layer then all the weights for nodes in that layer are multiplied by 0.50). Since we do not drop any nodes when testing the NN, downscaling accounts for the reduced number of nodes in training.

An advantage of dropout is that it is scale-free, therefore unlike weight decay it does not penalise based on feature size/variance. This makes it invariant to parameter scaling of the weights. In addition, dropouts stimulate sparse activation for a given layer, which in turn encourages the networks to learn a sparse representation of the data. DropConnect (Fig 5) is an alternative to dropout which drops connections instead of nodes, which has the advantage of increasing the number of possible sub-networks trainable, as there are almost always more connections than nodes.

Dropout works well in practice and can replace the need for weight decay, however, it is worth noting that regularisation techniques do not have to be used in mutual exclusion with each other. Both dropout and weight decay can be implemented in the same model.

3.4 Design of the best model

After extensive experimentation with the modules outlined in the section above, our best model comprised of the following modules outlined in table 1

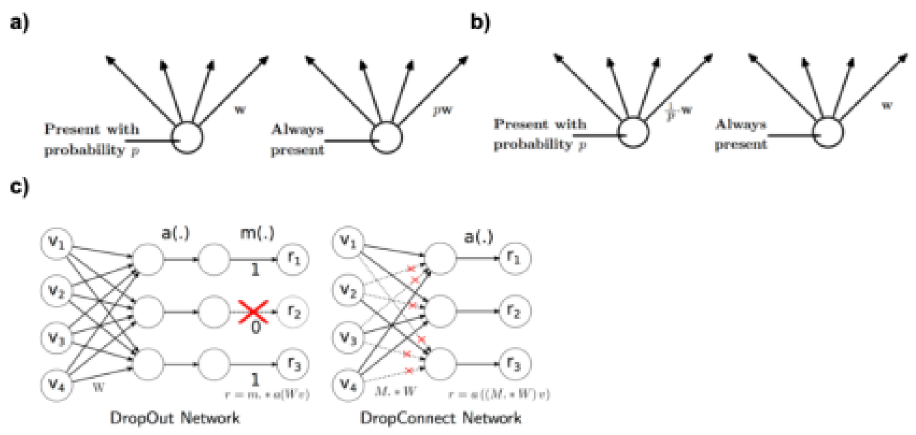


Figure 5: a) Dropout regularisation method; b) inverted dropout regularisation method; and c) comparison between dropout and DropConnect regularisation methods. Figures reprinted from Dr. Chang Xu (2021) Multilayer Neural Net- works. COMP5329 Lecture 2. University of Sydney

4 Experiments and results

4.1 hyperparameter analysis and ablation studies

Since we had a large number of hyperparameter to train it was infeasible to run a systematic grid search to find the optimum parameters (due to limited computational resources). Therefore we approached the problem as a heuristic search problem where we tried a combination of parameters based on theory. Some of the results can be seen in table 2. From those results, we took the best model and analyzed the effect of each parameter that contributed to the model's performance.

Our best model was as described in section 3.4 and will be referred to as the base model from here on. Its performance can be seen in figure 6

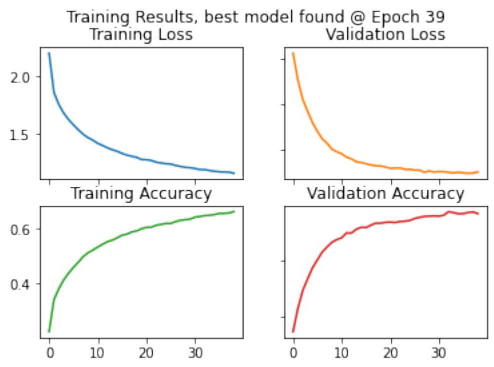
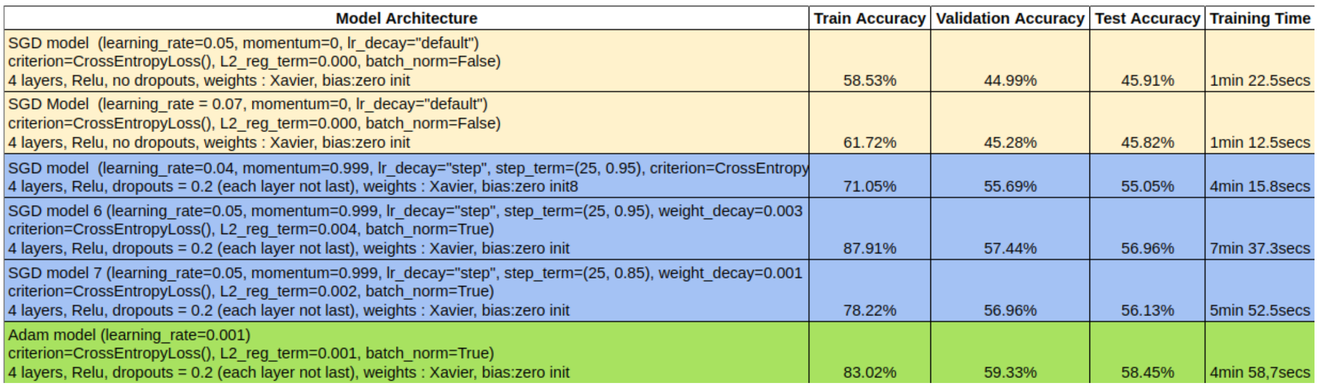


Figure 6: NN training and validation accuracy and loss for base model

To optimize the model and fine-tune its performance we conducted an analysis on the following

Nodes per layer	128, 1024, 512, 64, 16, 10
Activation function	ReLu
Loss function	Cross Entropy
Batch norm	True
Mini batch	Batch size = 500
Weight decay	L2 regularisation, =0.001
Dropout probabilities (p) per layer	(0.4, 0.2, 0.2, 0.2, 0)
Learning rate	0.01
Weight initialisation	Xavier

Table 1: Configuration of base model



Model Architecture	Train Accuracy	Validation Accuracy	Test Accuracy	Training Time
SGD model (learning_rate=0.05, momentum=0, lr_decay="default") criterion=CrossEntropyLoss(), L2_reg_term=0.000, batch_norm=False) 4 layers, Relu, no dropouts, weights : Xavier, bias:zero init	58.53%	44.99%	45.91%	1min 22.5secs
SGD Model (learning_rate = 0.07, momentum=0, lr_decay="default") criterion=CrossEntropyLoss(), L2_reg_term=0.000, batch_norm=False) 4 layers, Relu, no dropouts, weights : Xavier, bias:zero init	61.72%	45.28%	45.82%	1min 12.5secs
SGD model (learning_rate=0.04, momentum=0.999, lr_decay="step", step_term=(25, 0.95), criterion=CrossEntropy 4 layers, Relu, dropouts = 0.2 (each layer not last), weights : Xavier, bias:zero init8	71.05%	55.69%	55.05%	4min 15.8secs
SGD model 6 (learning_rate=0.05, momentum=0.999, lr_decay="step", step_term=(25, 0.95), weight_decay=0.003 criterion=CrossEntropyLoss(), L2_reg_term=0.004, batch_norm=True) 4 layers, Relu, dropouts = 0.2 (each layer not last), weights : Xavier, bias:zero init	87.91%	57.44%	56.96%	7min 37.3secs
SGD model 7 (learning_rate=0.05, momentum=0.999, lr_decay="step", step_term=(25, 0.85), weight_decay=0.001 criterion=CrossEntropyLoss(), L2_reg_term=0.002, batch_norm=True) 4 layers, Relu, dropouts = 0.2 (each layer not last), weights : Xavier, bias:zero init	78.22%	56.96%	56.13%	5min 52.5secs
Adam model (learning_rate=0.001) criterion=CrossEntropyLoss(), L2_reg_term=0.001, batch_norm=True) 4 layers, Relu, dropouts = 0.2 (each layer not last), weights : Xavier, bias:zero init	83.02%	59.33%	58.45%	4min 58,7secs

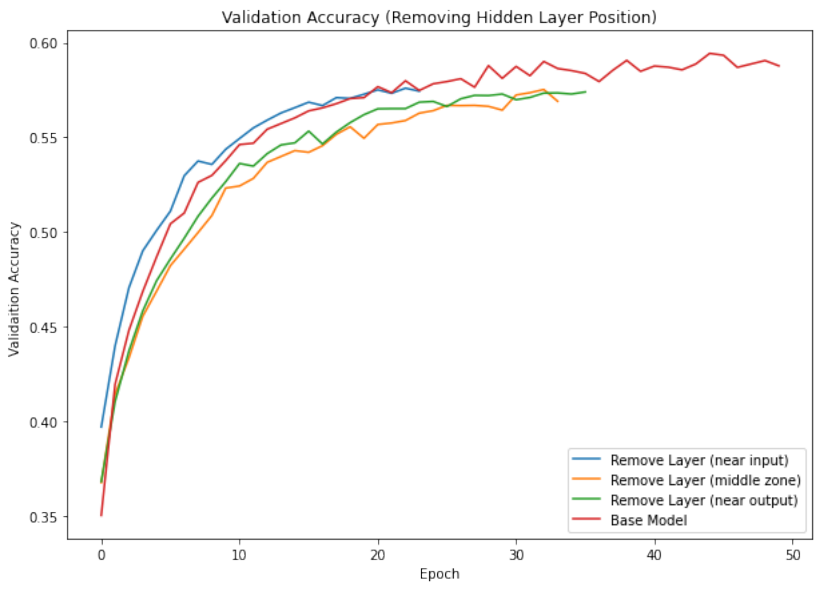
Table 2: NN designed through iterative selections(training, validation and test accuracy and training time)

hyperparameters, the number of hidden layers, weight decay, learning rate, and batch size.

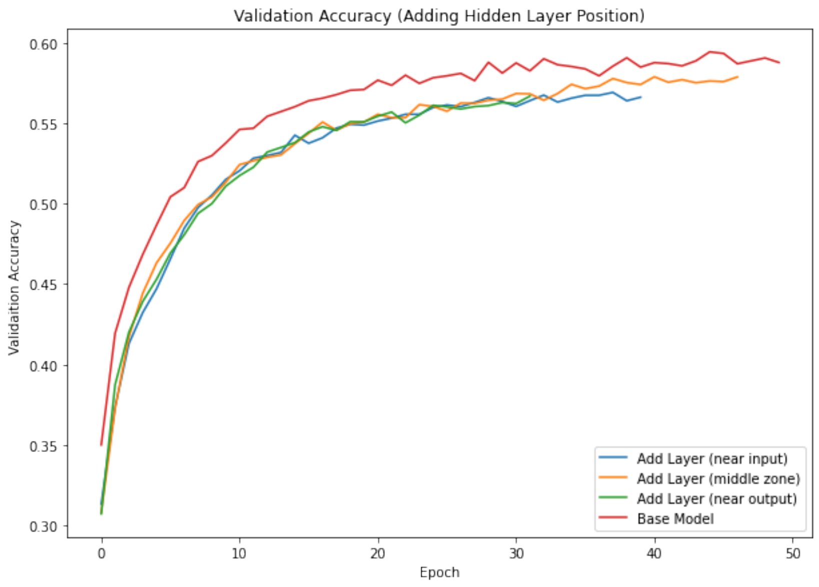
As we increased the number of layers we saw an increase in the time taken to train the entire network. Intuitively, training time increases when the number of neurons (size) of the network grows. As opposed to being a function of depth, time is more likely to increase as a function of width (for each neuron in each layer).

When we removed layers(Fig. 7a) from our model, we saw a significant reduction in time (half), whereby time decreased when removing layers closer to the input. Layers closer to the input are likelier of having higher dimensions for processing higher dimensional input feature maps.

Interestingly, adding layers (Fig. 7b) did not increase computation time as much as we expected.



(a) NN training and validation accuracy and loss when removing layers



(b) NN training and validation accuracy and loss when adding layers

Figure 7: Analysis of the number of layers

This is most likely due to layers inserted between input and output layers, being of size not exceeding the dimensions of the input feature map. Evidently, we know this to be the case as we chose the number of neurons of an inserted layer to be the average of the size of a respective input and subsequent layer.

Although we did not see an increase in processing time w.r.t the base case - we did see an increase in time for layers that were inserted closer to the input. Regardless, no performance benefit was observed from adding or removing a layer.

Consideration should be made towards how our convergence function decides termination of training, and mention should be made that added layers may have learned too slowly w.r.t the objective criteria of each subsequent model's training loss being less than the previous. This is evidenced by the training loss objective failing (learning too slowly) in all 3 models, as opposed to not achieving a lower validation loss within the next 10 epochs.

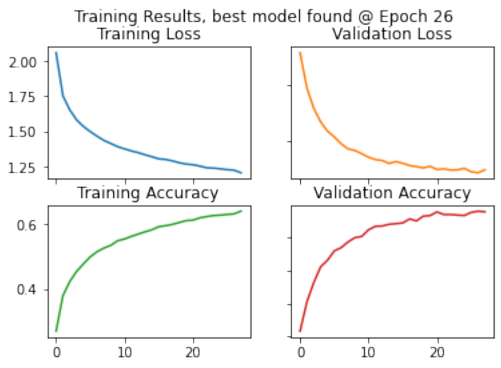
Increasing the batch size(Fig. 8) yielded a decreased performance of the model in terms of accuracy. Changing the learning rate from 0.0001 up to 0.01 yielded an increase in accuracy and then a decrease in accuracy at 0.1(further discussed in justifications). Notably increasing the weight decay function(Fig. 9) resulted in lower accuracy as was on par with that of not using any weight decay at all.

We performed ablation studies by taking our baseline model, intruding all modules(with previously specified hyperparameters), and dropping certain modules at a time (dropout and batch norm, weight decay) to observe its effects.

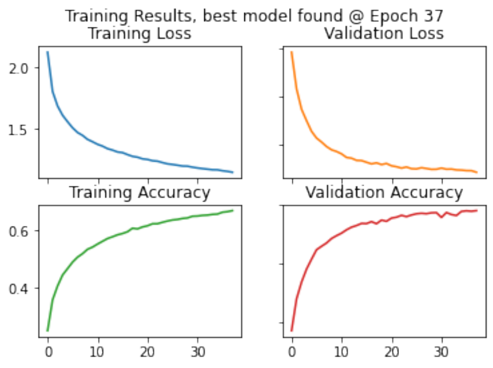
As seen from figure 10, by removing both the dropout layers, weight decay, and batch norm we can see a significant decrease in performance.

Figure 10b showcases how the validation error decrease as training progresses when the drop out layer is removed, This is intuitively explained by the fact that the lack of dropouts causes the model to overfit to the training samples.

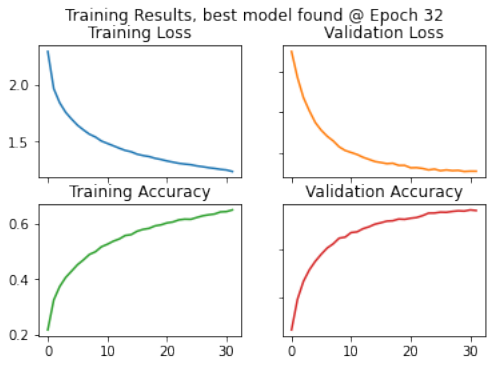
In addition, it was interesting to see that increasing the weight decay term had yielded similar results to when not using weight decay(Fig. 10c)! An explanation could be that gradients calculated from training had become very small which could present the possibility of the dead ReLU problem. From this source, an initially large gradient may deactivate a ReLU neuron such that gradient flow for this neuron in a backward pass is diminished. However, when we apply batch normalization(Fig. 10a), a substantial 5% increase in accuracy results provide weight to the theory that batch-normalization of the gradient dy in the backward pass, may possibly scale any large gradients down, which thus minimizes the chances of the dead ReLU problem.



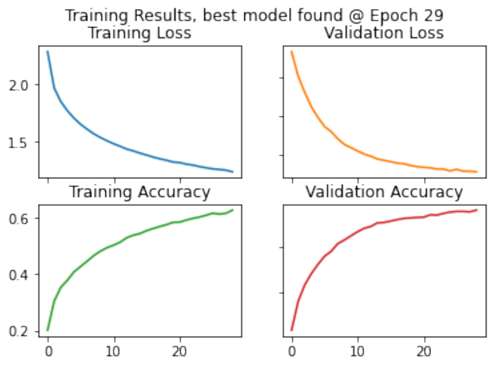
(a) NN training and validation accuracy and loss with a batch size of 100



(b) NN training and validation accuracy and loss with with a batch size of 200

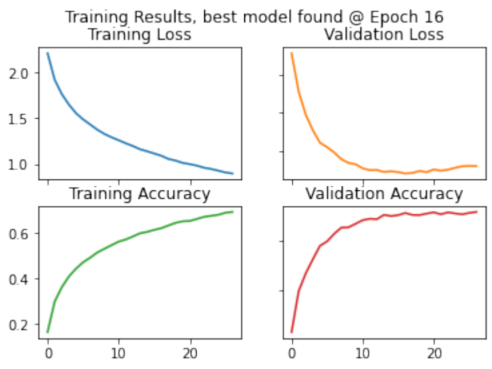


(c) NN training and validation accuracy and loss with a batch size of 700

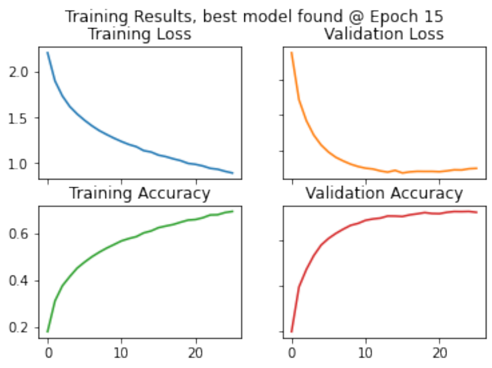


(d) NN training and validation accuracy and loss with a batch size of 1000

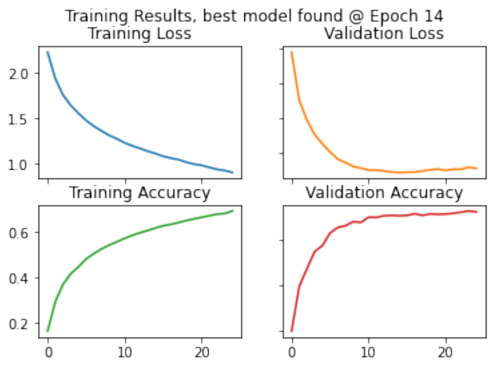
Figure 8: Optimizing Batch Size



(a) NN training and validation accuracy and loss for weight decay(L2) 0.002

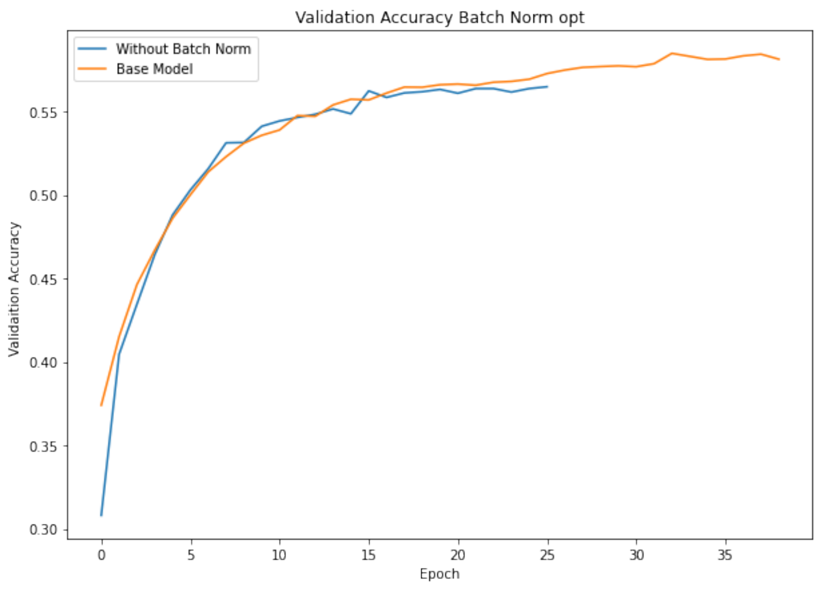


(b) NN training and validation accuracy and loss for weight decay(L2) 0.004

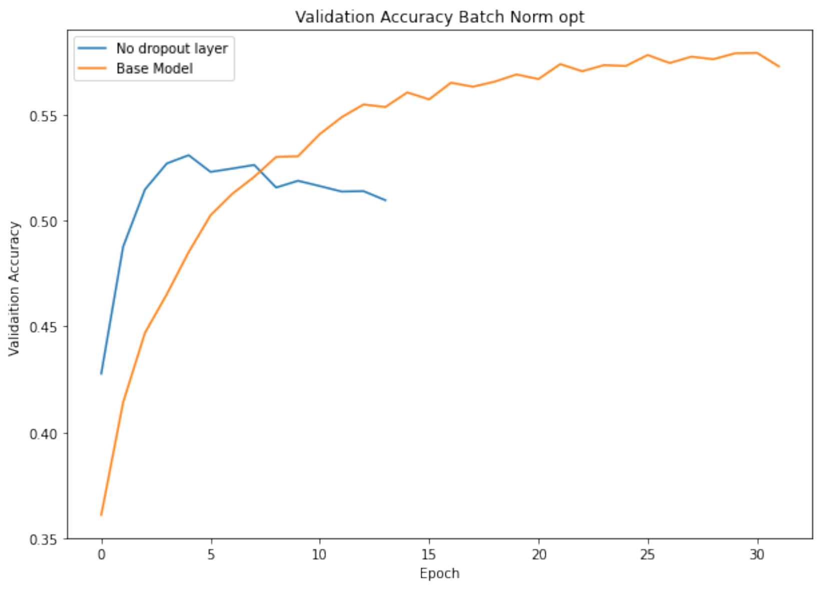


(c) NN training and validation accuracy and loss with no weight decay

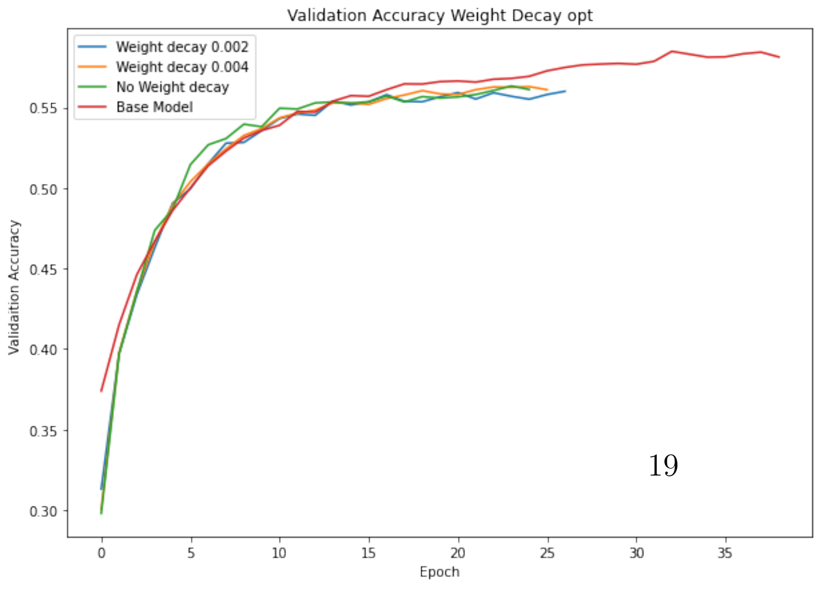
Figure 9: Optimizing weight decay



(a) NN training and validation accuracy with and without batch normalization



(b) NN training and validation accuracy with and without dropuout layers



4.2 Best model and Justification

After performing a hyper-parametric analysis of the model which revealed two parameters shown to produce better accuracy results on test data after some reconfiguring; a decision was made as to what was to be used. The parameters that saw improvement in accuracy over the base case model include - a learning rate of 0.01 and a smaller batch size of 100. Even though the parameters identified individually improve the performance of the model on the test data, as has been observed in hyper-parametric adjustments in the past, when the parameter updates are applied together - we do not see a positive effect on performance.

Conversely, some parameters may negatively affect performance when applied individually (as is noted in the 'examples/iterative-experimentation-SGD.ipynb). However, when applied in an ensemble (e.g. SGD with batch-normalisation, dropout, and L2 regularisation in the aforementioned notebooks), these parameters had actually boosted increases in accuracy of 10% as opposed to actually reducing accuracy when applied individually. In this case, applying the learning rate update and a change in batch size together had produced a lesser result in comparison to the performance of the base model.

To explain why the effect of these two hyperparameters produces a negative result when applied together, we understand that each hyperparameter does not only affect the model, but also affects the effectiveness of each other, and thus are also sensitive to each other (as well as the data that it is modeling). Given that we had found an optimal learning rate based upon a mini batch set to 500, perhaps it is indicative that this learning rate factors in some effect of the batch size it is evaluated on to perform well. Hence, rather than select a new batch size of 100, which is likely to have an effect on other selected parameters - we select the new learning rate of 0.01.

Although this value is larger than the default value of 0.001 that is used for the learning rate by Adam optimizers in popular neural network libraries, it has provided an increase in accuracy of almost 1% on the test data(Fig. 11). Looking at the results and adjusting the learning rate, we see 0.0001 and 0.1 produce accuracies of approximately 0.53, and 0.001 and 0.01 produce accuracies of 57.85 and 58.81, respectively. When the learning rate is too small (0.0001), we are unable to reach a lower loss value due to takings steps that are too small; when the learning rate is too large (0.1), we are also unable to descend a minima due to step size being too large - we may instead be oscillating back and forth. In the case of 0.001 and 0.01, we are able to descend lower, whereby 0.01 wins out by a whole percent.

The reason why a larger learning rate may be preferred is often to avoid getting stuck in any local minima and thus take larger steps and descend to a global minima (ideally). The fact 0.01 works better here may be a result of the cost function being very hilly - and this is evidenced by many SGD models prematurely converging into local minimas. The adaptive learning rate of the Adam optimizer that computes in the network as opposed to being umbrella values for all weights and bias.

5 Discussion and conclusion

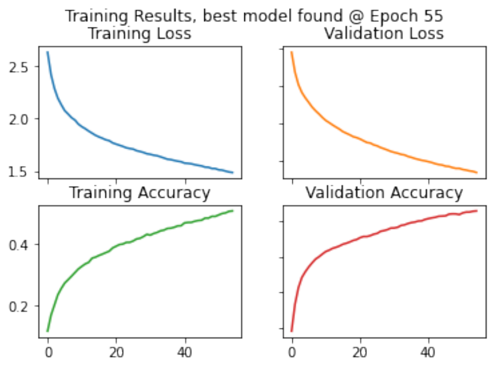
In undertaking this study, we successfully trained a high-performing model on a complicated multi-class classification problem. This enabled us to achieve our aim of deepening our understanding of modern NN innovations. This understanding was developed by successfully implementing over 11 different techniques and algorithms from the deep learning literature ourselves, before testing their performance on a multi-class classification problem. Comparing the theoretical underpinnings of each module with the actual performance both enhanced our understanding and tempered our expectations, as sometimes theory becomes obscured by the specifics of the problem at hand.

In our experimental investigation of various model architectural decisions, we found that our problem was suited towards an MLP of depth 4 with no substantial change arising from increasing this to 5 other than increasing training time. When running our experiments on optimization algorithms for this specific problem, it became evident that the weight space is much more difficult to traverse for SGD even with momentum. Conversely, Adam proved significantly better in both time to convergence and its overall convergence minima. This showcased that our problem benefitted from the flexible learning rate implemented by Adam, agreeing with the literature. In investigating mini batch, accuracy did improve over SGD, however, there was little difference between the accuracies of different batch sizes. We decided on a batch size of 500, which had comparable accuracy but took less time to train. A further investigation showed that applying batch normalization increased the performance of the model by around 5%, which was one of the most significant findings of the paper. A final interesting finding was arriving at an optimal learning rate of 0.01, which strays from the recommended value in the literature of 0.001 for Adam. This was somewhat surprising given Adam should be able to support lower learning rates with its adaptive schedule, but our finding showed that this specific problem worked better with a higher initialized learning rate. Although no formal experiment was conducted for other design decisions such as Xavier weight initialisation, ReLu activation, dropout, and L2 regularisation, these all appeared to result in better performance than alternative initialisations, and so were carried through to our final model. By combining each of our experimental findings, we were able to land at a model with an accuracy of 58.81% and an F-score of 0.58. Ultimately, this study proved successful in its aims and can help to inform future research paths in the field of deep learning.

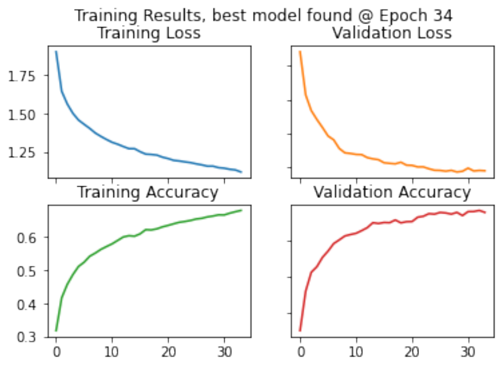
6 Other (Extra Marks)

We believe that our study contains these impressive characteristics that are deserving of extra marks:

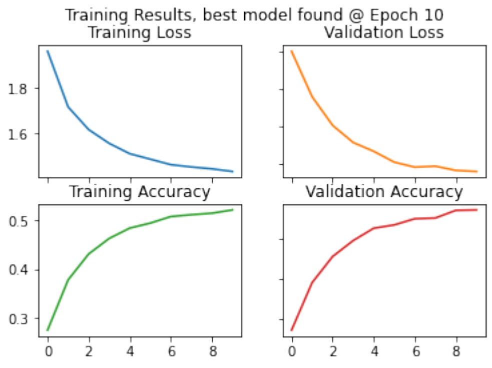
- MLPLibrary Code:
  - Weights and bias initialization methods(Xavier and Kaiming)



(a) NN training and validation accuracy and loss with a learning rate of 0.001



(b) NN training and validation accuracy and loss with a learning rate of 0.001



(c) NN training and validation accuracy and loss with a learning rate of 0.001

Figure 11: Optimizing Learning rate

  - LeakyReLU activation function
  - Adam Optimization
  - Learning Rate Schedulers (step,time and exponential decay)
  - Modularised object-oriented code
- Documentation
  - Sphinx auto-generated documentation
  - Readme files highlighting clear instruction on how to run the code
- Experiments:
  - Notebook experiments(choosing best performing models) using MLPLibrary
- Report
  - We have computed our report in Latex

7 Instructions to run the code

Installing MLPLibrary

To install MLPLibrary for use locally anywhere on your filesystem, navigate to the main root folder MLPLibrary/ and execute the command in terminal:

```
pip install .
```

This will install all required dependencies, and allow use of our library anywhere on your device. To uninstall simply type pip uninstall MLPLibrary

Running our Code

For practical demos of our code, navigate to examples/ directory. Here you will find python notebook experiments that execute MLPLibrary code to build, train and evaluate MLP models. To access our best model for COMP5329 Assignment 1, refer to Best-Model.ipynb, which the contains code of our chosen that was decided upon through numerous experiments within adjacent notebooks in this directory. Our best model is saved (pickled) in the examples/model sub-directory. For code more instructions that demo building, training and evaluation of models open Demo-MLP.ipynb. This notebook walks through using MLPLibrary functions to construct 2 MLP models, i.e. one using SGD and another using the Adam optimizer.