COMP4329/5329 Assignment 1

Lab 9 – Thursday, 7pm		
Name	Caleb Jia Le The-Tjoean	Sirui Chen
SID	490429716	530210018

LINK TO CODE:

https://github.com/rabbliiton/COMP5329-Deep-Learning.git

Introduction

Aim

The aim of this study is:

(1) to design and build a multilayer perceptron (MLP);

(2) to apply the MLP to a multiclass classification task with 10 labelled classes, under the context of supervised learning;

(3) to perform experiments analysing performance in terms of different evaluation metrics (e.g. precision, recall, F1 score), also taking into account hyperparameter analysis, ablation studies, and comparison methods;

(4) and to determine which is the best model (and its configurations) we could come up with for the task and justify why.

Importance

The study is important because it:

(1) shows the structure and implementation of basic building blocks of modern neural networks, i.e. loss and activation functions, gradient descent and backpropagation, regularisation and optimisation methods, etc.;

(2) solves a common supervised learning problem in a much simplified environment: instead of a large dataset like ImageNet (more than 14 million images), we can focus on a mere 128 features for 50,000 samples;

(3) proves quantitatively the effectiveness of the MLP when applied to a practical problem, with the help of tables summarising information like performance on evaluation metrics;

(4) provides some insight into good combination choices of hyperparameters and other settings, e.g. learning rate, dropout probabilities, weight decay, Momentum and RMSProp, number of epochs, loss and activation functions.

Methods

Pre-processing

i. Normalization

We applied a method of data pre-processing, that is to say min-max normalisation, to the feature matrix. This involved computing the maximum and minimum of the feature matrix

$$X = [[x_1^{(1)}, x_2^{(1)}, \dots, x_n^{(1)}], \dots, [x_1^{(m)}, x_2^{(m)}, \dots, x_n^{(m)}]],$$

where $x_i^{(j)}$ refers to the *i*th feature of the *j*th sample. Let *n* be the number of classes and *m* the number of samples. Then the minimum is given by

$$\min(X) = \min\{x_1^{(1)}, \dots, x_n^{(1)}, \dots, x_1^{(m)}, \dots, x_n^{(m)}\}$$

and similarly, the maximum by

$$\max(X) = \max\{x_1^{(1)}, \dots, x_n^{(1)}, \dots, x_1^{(m)}, \dots, x_n^{(m)}\}$$

These equations then give rise to a normalized *X* with its values in the range $(0, 10]$:

$$X_{norm} = \frac{X - \min(X) + \varepsilon}{0.1(\max(X) - \min(X))},$$

where $\varepsilon > 0$ is added for numerical stability.

ii. One-hot encoding

The second pre-processing step we performed involved converting the label vector to its one-hot encoding. If the label vector is given by

$$y = \left[y^{(1)}, y^{(2)} \dots, y^{(m)}\right]^T,$$

with *T* denoting the transpose, then we may define a matrix $y_{hot} \in \mathbb{R}^{m \times n}$ as follows:

$$y_{hot}{}_j^{(i)} = \begin{cases} 1, \text{ where } j = y^{(i)} \\ 0 \text{ elsewhere} \end{cases}$$

for all $1 \leq i \leq m$. This y_{hot} matrix will enable us to perform the calculations implicit in the cross-entropy loss, since its dimensions must be the same as that of the propagated output which goes through a softmax layer (i.e. m x n).

iii. Train/val/test split

We aim to split the dataset up into three subsets: training set, validation set, and test set. Originally, we received two datasets, i.e. the training set containing 50,000 samples and the test set containing 10,000 samples. So we further split the test set into half, the first half which is the new test set (5,000 samples) and the second half (the other 5,000 samples) which is the validation set. So the ratio of the split was

$$train: val: test = 0.83: 0.08: 0.08.$$

Of course, these ratios add up to 1. The distinction between the validation set and the test set is that the former is used for the purposes of experimenting with a variety of models (with a different network architecture and/or hyperparameters) and determining which one is the best with reference to evaluation metrics like precision, recall, and F1 score, while the latter is used to evaluate how well the best model performs on new data it has never seen before, that is, data from a new data distribution.

Modules

i. Network architecture

a. Layer sizes

Firstly, the neural network we are talking about here is the "raw model" in Figure 8, which is the baseline model for all our experiments. So, our neural network is a multilayer perceptron (MLP) that is comprised of: an input layer of size 128; 8 hidden layers with sizes [128, 64, 64, 32, 32, 16, 16, 10] respectively, and an output layer of size 10. The input feature matrix may be vectorised over *m* samples.

b. Activation functions

The activation functions used for the hidden layers are all leaky rectified linear units (LeakyReLU). The equation is

$$LeakyReLU(X) = \begin{cases} X_j^{(i)}, \text{ if } X_j^{(i)} \geq 0 \\ aX_j^{(i)}, \text{ if } X_j^{(i)} < 0 \end{cases},$$

where, again, $X_j^{(i)}$ refers to the *j*th feature in the *i*th example (or the *j*th column in the *i*th row). The coefficient *a* can be set to any positive value; we set it to $a = 0.5$. The reason for our incorporating the leaky rectifiers is that it helps avoid the "dying ReLU problem", which is when learning is impeded because of a proliferation of zero gradients (that non-positive inputs lead to), so that the ordinary rectifier cannot backpropagate anything but zero gradients.

c. Softmax layer

As for the output layer, we use a softmax classifier since our task is a multiclass classification one, i.e. we want a prediction of the probabilities that a specific sample belongs to different classes. The equation is:

$$softmax\left(X^{(i)}\right)_j = \frac{\exp\left(X_j^{(i)}\right)}{\sum_{k=1}^{n} \exp\left(X_k^{(i)}\right)},$$

where *n* is the number of classes or categories and $1 \leq j \leq n$.

d. Weight initialisation

We use He initialisation[1] to avoid problems associated with large or small weights causing the network to learn too slowly due to the "exploding gradient problem" or the "vanishing gradient problem". This method ensures that the weights in each layer have mean 0 and variance $\sqrt{\frac{2}{n_{in}}}$, where n_{in} denotes the number of input units for the layer.

ii. Optimisation

a. Loss function

The loss function is cross-entropy loss, which measures the degree to which two probability distributions $\{y\}$ and $\{\hat{y}\}$ differ. Note that y and $\hat{y}$ are both matrices in $\mathbb{R}^{m \times n}$, having changed y to a one-hot matrix in the normalisation step. So the cross-entropy loss over a given batch $\mathcal{B}$ is

$$CELoss_{\mathcal{B}} = -\frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{m_{\mathcal{B}}} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right),$$

where $m_{\mathcal{B}}$ is the number of samples in $\mathcal{B}$.

b. Adam optimiser

We use the Adaptive Moment optimisation algorithm[2] (Adam) which combines the benefits of Stochastic Gradient Descent (SGD) with Momentum and RMSProp. The advantages of

[1] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.

[2] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

using Adam are that we can make the gradient descent step much more smooth and heterogeneous, while still reaping the benefits of the parallelisation in batch gradient descent. Now the exponentially decaying average of past gradients $m(t)$ is given by

$$m(t) = \beta_1 m(t-1) + (1-\beta_1)\frac{\partial \mathcal{L}}{\partial w(t)},$$

and so the bias-corrected version is

$$\widehat{m}(t) = \frac{m(t)}{1-\beta_1^t}.$$

Similarly, the exponentially decaying average of past squared gradients $v(t)$ is given by

$$v(t) = \beta_2 v(t-1) + (1-\beta_2)(\frac{\partial \mathcal{L}}{\partial w(t)})^2,$$

and so the bias-corrected version is

$$\hat{v}(t) = \frac{v(t)}{1-\beta_2^t}.$$

The learned hyperparameters β_1 and β_2 are the first and second moments in the exponentially running averages, and the gradient of the loss function with respect to the weight is computed during backpropagation. The learning rule is thus modified to

$$w(t+1) = w(t) - \frac{\eta}{\sqrt{\hat{v}(t)+\varepsilon}}\widehat{m}(t)$$

for each time step $t \geq 0$ and some small $\varepsilon > 0$. Note that t may be interpreted to be the current epoch number and so $\frac{\partial \mathcal{L}}{\partial w(0)} = 0$. This learning rule was applied to all weights and biases in the network.

c. Minibatches

The entire training dataset contains 50,000 samples, so the optimiser would process each sample too slowly. Processing in (mini)batches, then, affords a rapid boost in speed every epoch and smooths out the learning curve, so that we descend in the direction of the averaged steepest descent on every sample/iteration. The number of minibatches is therefore given by

$$\#minibatches = \left\lceil \frac{\#samples}{minibatch\ size} \right\rceil.$$

Note that we apply the ceiling function just in case the dataset does not neatly divide into minibatches. In our case, we had $\left\lceil \frac{50000}{1024} \right\rceil \approx \lceil 48.83 \rceil = 49$ minibatches. The first 48 minibatches had 1024 samples while the last one had $50000 - 48 * 1024 = 848$ samples. The new dataset then becomes

$$X_{new} = \{X_1, X_2, \dots, X_{49}\}.$$

On each epoch of training, we went through the entire dataset, batch by batch, feeding the MLP one batch at a time.

d. Learning rate scheduler

We added a learning rate scheduler to help gradient descent converge to a good local minimum in the loss function space. This is because if the learning rate is too large, the loss function will oscillate around the local minima but never fully converge. The scheduling method we used involves decreasing the learning rate exponentially as the number of epochs increases, as follows:

$$\alpha(t+1) = \alpha(t) r^{\left\lfloor \frac{t}{steps} \right\rfloor},$$

where *r* is the decay ratio, which should satisfy $0 < r < 1$, and $t \geq 0$ is the current epoch number. Now the number of epochs that α is scheduled to stay the same, is given by

$$steps = \left\lfloor \frac{\#epochs}{j} \right\rfloor,$$

where *j* is the number of jumps we want to have distributed over all the epochs. In particular, we set $r = 0.5$ and $j = 6$.

iii. Regularisation

a) Batch norm

We implemented batch normalisation[3] to help reduce training time due to decreasing "covariate shift", a phenomenon that occurs when the distribution of input data (or activations) to subsequent layers in the network changes. The method makes each neuron conform to a Gaussian distribution (i.e. mean of 0 and standard deviation of 1), thereby reducing the effects of exploding and vanishing gradients. Let *n* be the number of neurons in a layer *l*. Then the input activations $z \in \mathbb{R}^{m \times n}$ for layer *l* are normalised with mean $\mu_X \in \mathbb{R}^n$ such that

$$(\mu_X)_i = \frac{1}{m} \sum_{i=1}^{m} z_n^{(i)}$$

for all $1 \leq i \leq n$ and variance $\sigma_X^2 \in \mathbb{R}^n$ such that

$$(\sigma_X^2)_i = \frac{1}{m} \sum_{i=1}^{m} (z_n^{(i)} - (\mu_X)_n)^2.$$

for all $1 \leq i \leq n$. So normalising yields

$$z_{norm}^{(i)} = \frac{z^{(i)} - \mu_X}{\sqrt{\sigma_X^2 + \varepsilon}}$$

for some small $\varepsilon > 0$ (added for numerical stability). Lastly, we have

$$z_{batch\_norm} = \gamma z_{norm} + \beta$$

for hyperparameters $\gamma \in \mathbb{R}^n$ and $\beta \in \mathbb{R}^n$, which are learned via backpropagation. There are 8 hidden layers, and we added batch norm layers after the linearities of the 2nd, 4th, 5th, 6th, and 8th hidden layers – these were added right before the non-linearities were computed (e.g. LeakyReLU or Softmax).

b) Dropout

In dropout, we had neurons in the hidden layers (not the input or output layers) are "turned off", or their activations silenced, with a dropout probability of p. It has a regularising effect due to making the network not overfit to training data; in effect, it combines the predictions of 2^u units at inference time, where *u* is the number of units in the network. We decided to add only two dropout layers to the 3rd and 7th hidden layers, right before the activations were applied (similarly to batch norm). This can be done by multiplying the pre-activations $z \in \mathbb{R}^{m \times n}$ by a Boolean mask (containing 1s and 0s), i.e.

[3] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In International conference on machine learning, pp. 448-456. pmlr, 2015.

$$mask = \frac{random \; > \; p}{1 - p},$$

where *random* denoting a matrix of random floats with the same size as *z*. We set $p = 0.2$. Note that we are dividing by $1 - p$ since the weights have been scaled up by a factor of $\frac{1}{p}$ due to dropping out units during training – this practice is called "inverted dropout". Lastly, we did not apply inverted dropout at test time, but only for training and validation.

c) Weight decay

We used weight decay, in particular the L2 norm, to a small extent to help give a regularising effect to our network. It involves adding a constant term to the criterion to discourage weights from being too large. Let *w* be the set of all weights in the network. Then the L2 norm is

$$\|w\|_2 = \sqrt{\sum_{l=1}^{L} \sum_{j} \sum_{i} |w_{i,j,l}|^2}$$

$$\mathcal{L}(w) = CELoss_{\mathcal{B}} + \frac{\lambda}{2} \|w\|_2^2$$

$$= CELoss_{\mathcal{B}} + \frac{\lambda}{2} \sum_{l=1}^{L} \sum_{j} \sum_{i} |w_{i,j,l}|^2,$$

where λ is the regularisation hyperparameter, which we set to 1e-5 in the 5th hidden layer and 0 elsewhere (i.e. no regularisation), and L is the number of hidden and output layers in the network. Note that *i* and *j* range over the row and column indices of the particular weight matrix (or bias vector, if we want to regularise that too) associated with the layer *l*. This, of course, changes the backpropagated gradients of the loss function with respect to the weights, which we must account for in the backwards step of the training process. The derivative is simply

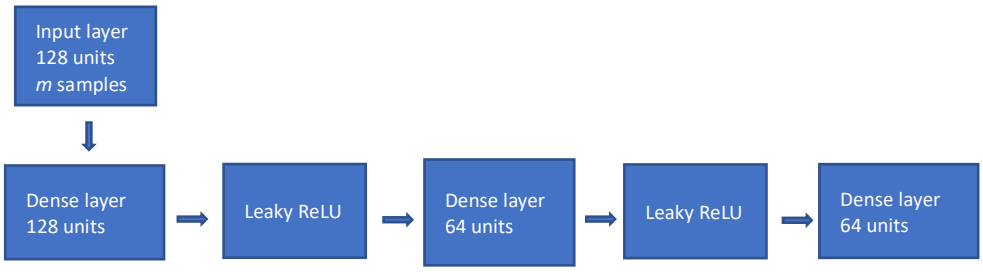
$$\frac{\partial \mathcal{L}(w)}{\partial w_{i,j,l}} = \frac{\partial CELoss_{\mathcal{B}}}{\partial w_{i,j,l}} + \lambda |w_{i,j,l}|.$$

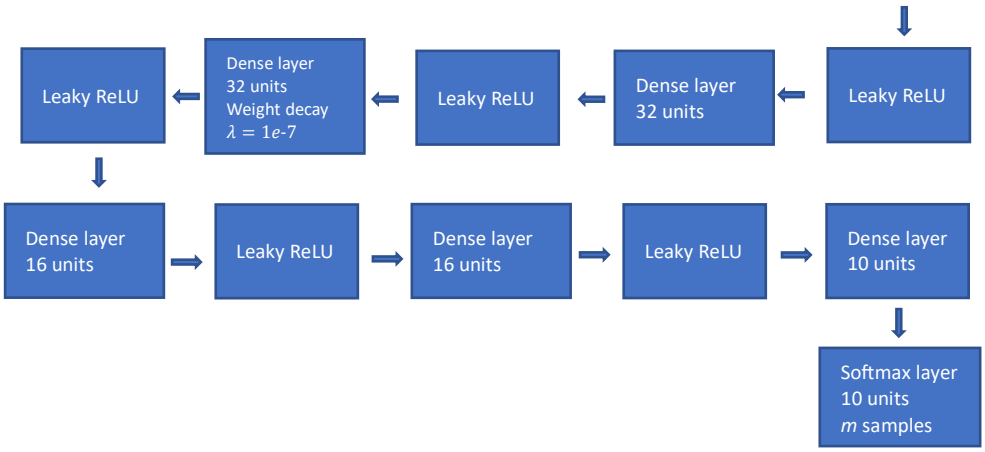
d) Early stopping

Early stopping is the technique of stopping the training process prematurely (i.e. in an earlier amount of epochs) when there appears to be an initial sign of divergence between the training loss and the validation loss. It indicates exactly when the model begins to overfit to the training set and underperform on the validation set (meaning that the bias is high). In our case, we stopped the model when it reached 150 epochs.

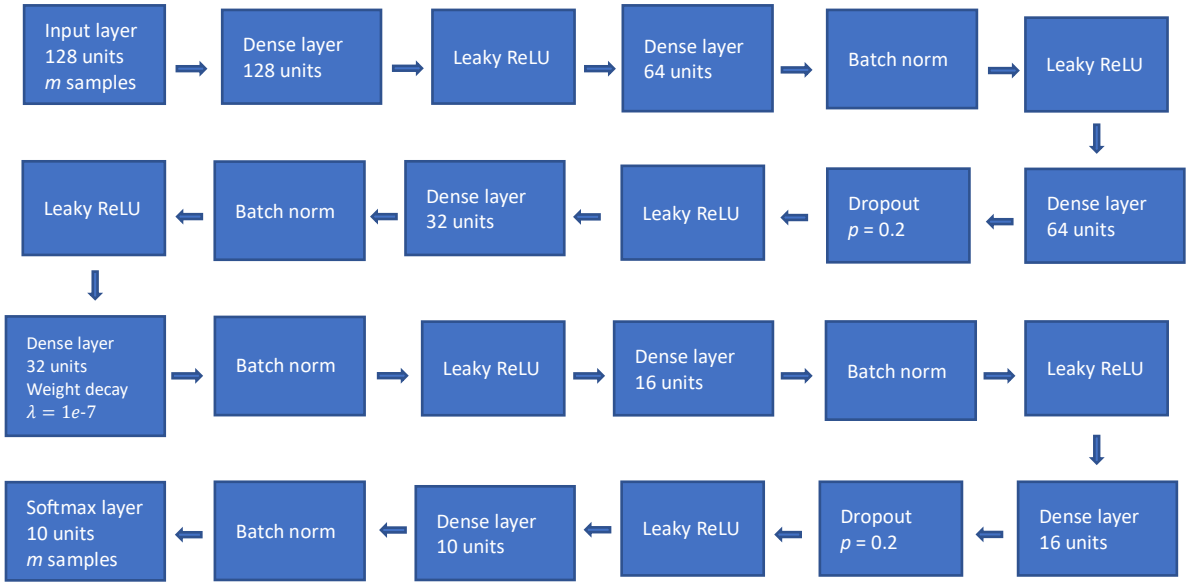
Best model

Here is a diagram of the model we were able to achieve the best results with:





It is a simplified model of the "raw model" (model 0 in the next section), shown below. That is, we omitted the regularisation methods of dropout and batch normalisation.



The configurations and hyperparameters for our best model are:

- Number of epochs: 150
- Batch size: 2048
- He (weight) initialisation
- Cross-entropy loss
  - Weight decay: $\lambda = 1e\text{-}7$
- Adam optimiser
  - Learning rate: $\alpha = 1e\text{-}3$
  - Momentum: $\beta_1 = 0.8$
  - RMSProp: $\beta_2 = 0.8$
- Learning rate scheduler
  - Decay ratio: $r = 0.5$
  - Number of jumps: $j = 6$
  - Number of epochs: 200

Experiments and results

Performance and analysis

i. Evaluation metrics

Having trained our models on the training set, we then proceeded to evaluate the performance of our models on the validation set, with reference for key evaluation metrics as loss, accuracy, precision, recall, and F1 score. Let us define the confusion matrix

$$M = \begin{matrix} True\ Positive\ (TP) & False\ Negative\ (FN) \\ False\ Positive\ (FP) & True\ Negative\ (TN) \end{matrix}',$$

where the first column denotes a positive prediction value and the second denotes a negative prediction value, while the first row denotes a positive actual value and the second a negative actual value. Then we may further define 4 evaluation metrics as follows:

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}$$

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\ score = \frac{2(Precision \times Recall)}{Precision + Recall}.$$

Accuracy measures the percentage of times we made a correct positive prediction out of all the predictions we make. Precision measures the ratio of correct positive predictions to all positive predictions, while recall measures the extent to which we missed/failed to make positive predictions. There is usually what is called a "precision/recall trade-off": if you increase one, you will necessarily decrease the other, vice versa. So one must decide what is important for one's particular task, whether it is always making the correct prediction – in which case one should try to increase the precision, or returning more results that could possibly be correct predictions – so increase the recall. As a way to resolve this dilemma, F1 score combines the benefits (and drawbacks) of these latter two metrics by taking their harmonic mean. Lastly, we also have the metric

$$Loss = \mathcal{L}(\hat{y}, y),$$

where $\hat{y} \in \mathbb{R}^m$ are the predictions made and $y \in \mathbb{R}^m$ are the actual labels. These 5 metrics will form the basis of our experiment evaluations. There We also take into consideration the time taken to run the evaluation code from beginning to end.

ii. Hyperparameter analysis

The first hyperparameter we looked at was the learning rate α. Both machine learning frameworks Google's TensorFlow (v2.12.0) and Meta's PyTorch (v2.0) use a default learning rate of 1e-3 for the Adam optimisation algorithm. So we decided to test this out, also taking into consideration learning rates on either side of this at a logarithmic scale – for models 0 and 2. However, our hypothesis was confirmed and the best results on all 5 evaluation metrics as well as the time, was reached with model 1.

Index	*Hyperparameter*	*Loss*	*Accuracy*	*Precision*	*Recall*	*F1-Score*	*Time (min)*
0	learning rate = 1e-2	2.293945689	0.1138	0.044986837	0.114034146	0.064520234	2.44254693
1	learning rate = 1e-3	**1.829896147**	**0.362**	**0.400620053**	**0.361791933**	**0.380217274**	**2.271011815**
2	learning rate = 1e-4	2.046657731	0.2848	0.306606034	0.284434529	0.295104425	3.065780252

Figure 1

The default moment for momentum β_1 in Adam for both PyTorch and TensorFlow is 0.9. Contrary to our expectations, we found that lowering β_1 by 0.1 produced more effective results on every single metric except for precision – which is beaten by model 2. But this difference in precision is negligible (less than 0.002 or 0.2%) compared to the rapid gain in accuracy (0.0232 or 2.32%), so we conclude that model 1 is still best. Lowering β_1 too much, however, produced less than ideal results, as shown in model 0.

Index	*Hyperparameter*	*Loss*	*Accuracy*	*Precision*	*Recall*	*F1-Score*	*Time (min)*
0	momentum = 0.3	2.172454131	0.1708	0.065250709	0.16900981	0.094151673	3.039137564
1	momentum = 0.8	**1.829896147**	**0.362**	0.400620053	**0.361791933**	**0.380217274**	**2.271011815**
2	momentum = 0.9	1.892326425	0.3388	**0.402056487**	0.338677707	0.367655686	2.313936625

Figure 2

Similarly, PyTorch and TensorFlow set the moment for RMSProp β_2 in Adam as 0.99 by default. When we tried this out in model 2, we saw that it did not attain as good as results in every metric as model 1 which has $\beta_2 = 0.8$ – except in the aspect of time, being less than 0.03 seconds faster, another negligible difference. The gains in F1-score when using model 1 where too great to not use. Again, lowering β_2 by too much in model 0 led to model underfitting.

Index	*Hyperparameter*	*Loss*	*Accuracy*	*Precision*	*Recall*	*F1-Score*	*Time (min)*
0	rmsprop = 0.3	2.218365764	0.1722	0.061861038	0.174773847	0.091378679	3.097103389
1	rmsprop = 0.8	**1.829896147**	**0.362**	**0.400620053**	**0.361791933**	**0.380217274**	2.271011815
2	rmsprop = 0.99	1.899407414	0.3122	0.316697216	0.31011639	0.313372257	**2.244891141**

Figure 3

In summary, it seems that the best hyperparameters to use in our study are:

$$\alpha = 1e\text{-}3, \beta_1 = 0.8, \beta_2 = 0.8.$$

We suspect that the reason $\beta_1 = 0.9$ and $\beta_2 = 0.999$ did not perform as optimally is that these hyperparameters must be suited to the particular learning problem, and may not do so well on certain tasks like the one here. Using these hyperparameters yields a loss plot as follows:

This plot shows clearly that

$$loss_{train} < loss_{val}$$

for all epochs. This means that our model is overfitting to the training set (i.e. the variance is high), and so regularisation methods should be applied to the model. This is generally the case for all of our validation models, although some have

$$loss_{train} \approx loss_{val}$$

for the later epochs. We shall further explore the effects of regularisation methods through ablation studies in the next section.

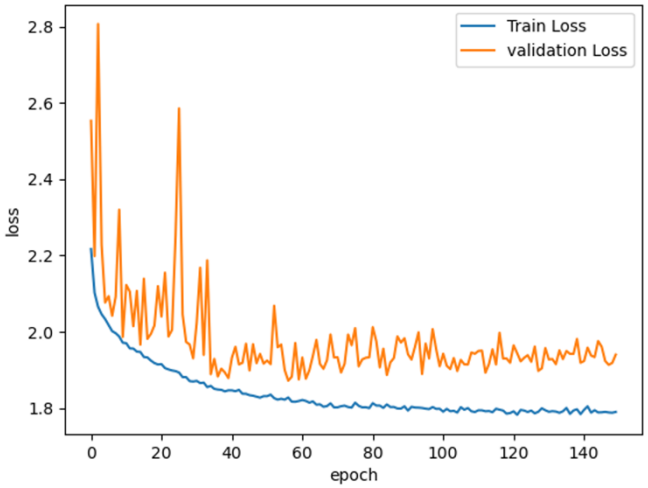


Figure 4

iii. Ablation and comparative studies

The following table summarises all the experiments we conducted with regard to ablations of features in the network, including: regularisation, optimisation, and weight initialisation methods, network architecture, methods for data pre-processing, loss or activation functions used, the training loop, etc. We will discuss groups of models at a time, so that we can compare their advantages and disadvantages with reference to evaluation metrics. We will also provide the loss plots for both training and validation sets so that we can track the training/learning progress of models that we find interesting. Finally, the "raw model" (model 0) is the baseline model that we use to compare models in all such groups with.

Index	*Ablated Feature*	*Loss*	*Accuracy*	*Precision*	*Recall*	*F1-Score*	*Time (min)*
0	raw model	1.829896147	0.362	0.400620053	0.361791933	0.380217274	2.271011815
1	change batch size from 2048 to 512	1.847862975	0.349	0.394629015	0.348688996	0.370239367	2.063256318
2	no minibatches (i.e. now batch GD)	1.937695147	0.3598	0.351526415	0.359792681	0.355611517	2.768429271
3	change all activation functions from LeakyReLU to ReLU	1.932445429	0.3342	0.430826248	0.334365146	0.376515686	2.377438197
4	change loss function from CE to MSE	**0.087406426**	0.1982	0.119980575	0.196199271	0.148903237	3.395557976
5	constant weight initialisation	2.302584093	0.1242	0.087718153	0.12390961	0.102719246	2.484425055
6	Gaussian weight initialisation	2.410476004	0.1246	0.064035526	0.125228911	0.08473963	2.294620771
7	Xavier weight initialisation	1.867077886	0.3598	0.394989337	0.359219123	0.376255984	2.740350469
8	change #epochs from 150 to 100	1.843911463	0.3634	0.340994226	0.363215859	0.351754436	1.961304078
9	change #epochs from 150 to 200	1.966676621	0.3088	0.362238659	0.30785836	0.332841951	2.428199643
10	no dropout	1.821117905	0.362	**0.43344938**	0.36106583	0.393960388	2.539794512
11	no dropout and remove 2 hidden layers	1.820335908	0.3528	0.395519535	0.351555227	0.372243762	2.222143763
12	no weight decay	1.887391538	0.3262	0.33924352	0.32544279	0.332199884	2.29910144
13	no batch norm	1.776733268	0.3666	0.357336578	0.367000882	0.362104258	1.758119526
14	no batch norm and change #epochs from 150 to 200	1.792729844	0.3618	0.359443769	0.362014781	0.360724694	1.73974205
15	no learning rate scheduler	2.300329528	0.1278	0.037298984	0.128622635	0.057828433	2.462461063
16	best model	**1.74068069**	**0.3992**	0.39843175	**0.39942544**	**0.39892798**	**1.6241932**

Figure 5

a. Batch size: models 0, 1, 2

Model 0 with batch size $2048 = 2^{11}$ achieves the best F1-score by a few percent due to already having the best precision and recall. Furthermore, the accuracy beats the other two models; the only drawback is taking about 0.2 minutes longer than model 1 to run – this may be the case because a batch size of $512 = 2^9$ may mean that we are able to simultaneously exploit the parallelism implicit in vectorisation yet also process a greater number of batches more quickly, that is to say it is a good trade-off between the whole

batch (model 2) and too large of a batch (model 0). Refer to Figure 4 for the loss plot of model 0; here it is for model 1 for comparison. We see that model 1 has a much more stochastic and variable learning curve (which makes sense since the batch size is smaller), with a few descents in the wrong gradient in the middle of training, as opposed to very few in model 0, which converges nicely.

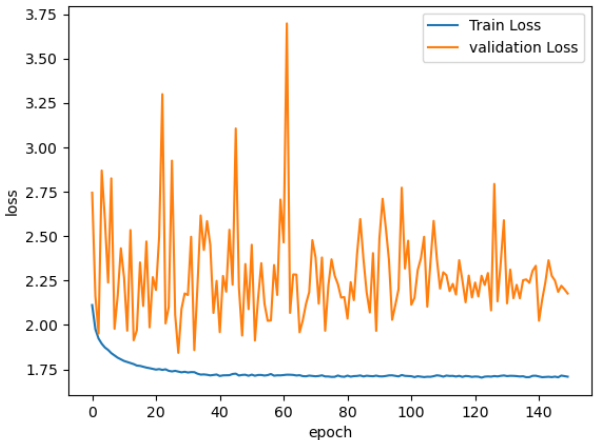


Figure 6

b. *Activation functions: models 0, 3*

We see a precision/recall trade-off since model 3 has a higher precision while model 0 has a higher recall, although their F1-scores are virtually the same. The real reason for preferring the LeakyReLU activation function in model 0 is due to its giving a 0.0278 (or 2.78%) increase in accuracy – a quite noticeable improvement. Here the loss plot for model 3 shows us that the initial learning curve for the training set is much less steep; or in mathematical terms, the second derivative of the train loss with respect to weights for model 3 is significantly greater than that for model 0.

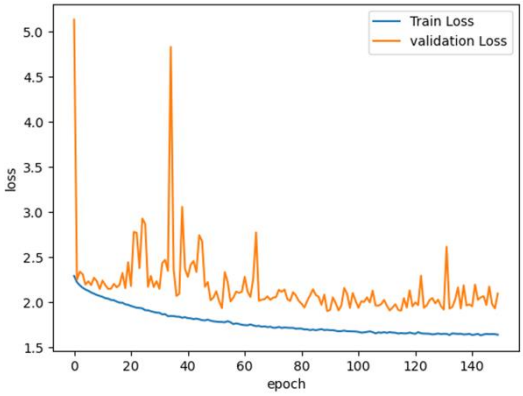


Figure 7

c. *Loss functions: models 0, 4, 13*

Model 4 achieves the lowest loss by far since it has a unique loss function (mean squared error loss) among models 0-16 (cross-entropy loss), but it fails to also achieve first place in any of the other metrics. In fact, it achieves an extremely low accuracy and F1-score, and takes the longest out of any of the models 0-16 to run. Notice how we highlight in bold two entries in the “Loss” column, one for model 4 and one for model 16.

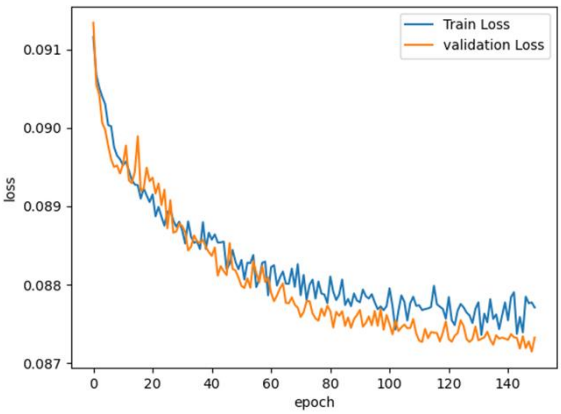


Figure 8

d. *Weight initialisation methods: models 0, 5, 6, 7*

Initialising the weights with a simply normal/Gaussian distribution as in model 6 or even multiplying it by a small constant value (i.e. 0.01) as in model 5 leads to very low accuracy – almost as if the models were making random guesses that are correct 1/10 of the time. Instead we found that He initialisation (model 0) worked well as well as Xavier (aka Glorot) initialisation (model 7). They have extremely similar losses, accuracies, and F1 scores; the difference lies in the time, for which model 0 is faster by about 0.47 minutes. The disadvantage of using 0, however, is that in practice its learning is not always stable: on some training sessions the accuracy would be good, and on other training sessions the accuracy would not be so good. Hence, using Xavier/Glorot initialisation may be preferable to using He. Here is the loss plot of model 7: It shows that the loss is much less stochastic than Xavier, and has much less jumps in the initial stage of training. For this reason also, Xavier may be more preferable.

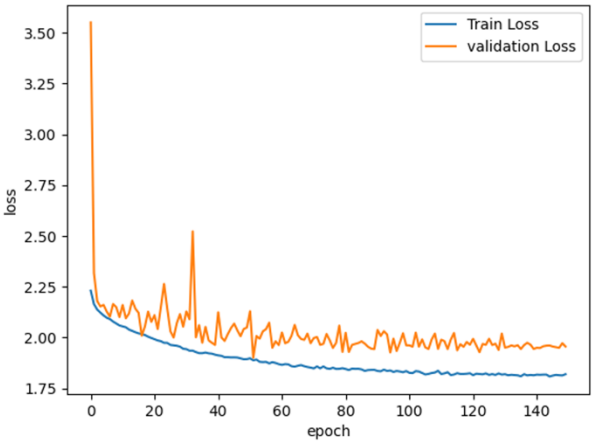


Figure 9

e. Epochs: models 0, 8, 9

We see another instance of a trade-off between precision and recall where training for 100 epochs in model 8 leads to higher recall while training for 200 epochs in model 9 leads to higher precision. Interestingly, the accuracy improves when you train for a fewer number of epochs: this is probably due to the fact that model 9 overfits to the training set and performs not so well on the validation set, while model 8 is an example of early stopping. Now model 0 also stops early, but only after 150 epochs: hence, it reaches the best balance between precision and recall, on the one hand, and bias and variance, on the other hand, with an increase in F1-score by about 0.0285 (or 2.85%). Alternatively, training for too long may actually be counterproductive, leading to significant drops in accuracies (as shown in model 9). So this justifies the use of early stopping with a suitable number of epochs to stop after. We also see that there is about 0.31 minutes saved when model 8 is used over model 0. The loss plot for model 8 is different to that for model 0 in so far as the weight initialisations produce random numbers.

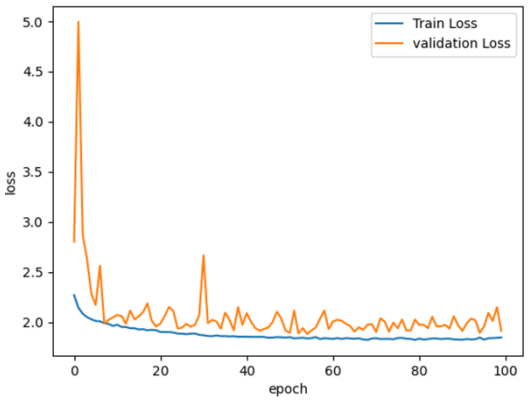


Figure 10

f. Regularisation methods: models 0, 10, 11, 12, 13, 14

Model 10 attains the highest precision, which means that we can rely on it well enough to be confident in its positive predictions. It also has the highest F1-score by far compared to the other models – the drawback is its longer inference time. This means that using dropout leads to performance decreases due to its strong regularising effect, especially if high probability ratios of dropping out are used. Model 11 is similar to model 10 except that it discards two hidden layers: the results in every metric show that it is inferior, however. This

shows that it is useful to have more hidden layers to be able to be capable of discovering more useful internal representations of the data and being more complex. Model 12 shows that not using weight decay will lead to big dips in accuracy and slight dips in F1-score, which is why we keep it in our best model. Model 13 achieves a marginally better accuracy than model 10, but most importantly takes significantly less time to execute – 0.51 minutes less than model 0. Similarly, if we train for 50 more epochs without batch norm, as in model 14, we get slightly worse results, which also justifies training for just 150 epochs. The two most important insights we got were from models 10 and 13, whose ideas we can combine to create a model that does not use batch norm or dropout, which actually led to our best model, as discussed and justified in the next section. On the left is the loss plot for models 10 and on the right model 13, which convinced us that we should come up with another experiment:

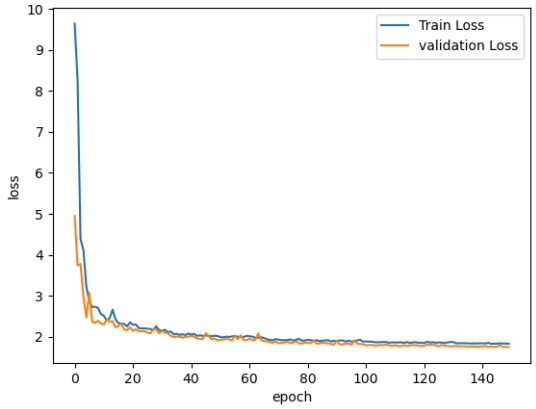


Figure 11

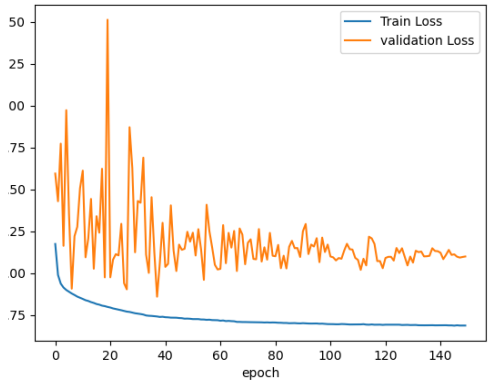


Figure 12

g. Optimisation methods: models 0, 15

We discussed already the implications of using Momentum and RMSProp and changing the hyperparameters α, β_1, and β_2 accordingly in the previous section. As for using a learning rate scheduler, when we do not use one, our accuracy drops dramatically to a few percent above random guesses (12.78%), so that it is demonstrated that with our architecture and configurations, we definitely need our learning rate to decay over the course of many epochs. Here is the loss plot for model 15. What it shows is that without decaying the learning rate with a scheduler, the training and validation losses during learning plateau at a loss of about 2. Learning is effectively prohibited, most likely due to the learning rate being too large and so causing gradient descent to oscillate right over the local minima of the loss function.

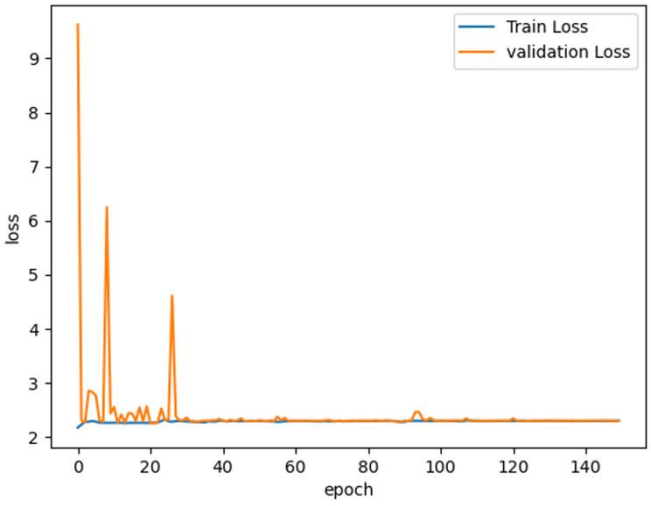


Figure 13

Justification of best model

i. Difference with raw model

In Figure 5, we can see that the "best model" gives us the best loss, accuracy, recall, F1-score, and even time. It is the model as described in pg. 7, which has no dropout nor batch normalisation, as we want to regularise our model as little as possible (since our model still performs quite well on the validation set). Furthermore, since our model contains not that many layers (only 7), does not have a high learning rate, and receives plenty of data (50,000 samples) to train on, we do not have to worry about regularising our model. Therefore, our best model will apply these two changes (i.e. no dropout and no batch norm) to the raw model, so that we can get the highest accuracy and F1-score as possible. Lastly, its inference time is very fast as it does not have to propagate and backpropagate through dropout or batch norm layers.

ii. Network structure

The network structure, as shown on pg. 7, is simple and only has 1 input layer, 7 hidden layers, and 1 output layer. The input layer expects 128 features and may be vectorised over *m* samples, while the output layer combines a linear function with a softmax classifier. The hidden layers are comprised of a linear function and non-linear one (i.e. leaky rectifier). This is as simple a structure we could think of that does not sacrifice depth of layers or too much information passed through the layers. We also made sure to halve the number of input units to each layers, so that we have layers accepting 128, 64, 32, 16 units. This is opposed to using just one hidden layer with, say, 64 units, which may not be able to approximate the function $\hat{f}(X) = y$ that well, where X is the training data and y the training labels, or using too many hidden layers, like 20, where we would have to use residual layers to avoid the vanishing gradient problem.

iii. Hyperparameters and other settings

We had 150 epochs since this was the sweet spot between 200 and 100 epochs, so we could take full advantage of training for as long as possible to improve the training as well as validation accuracy and also of the benefits of early stopping. We set the batch size to 2048 since this takes advantage of parallelising matrix operations while also not being too large for the computer's CPU/GPU to process (like batch GD would necessitate). We used He initialisation since it was a relatively stable option to use that would ensure small enough weights to learn without having to worry about the exploding/vanishing gradient problem. Cross-entropy loss was the best one for us to use given our multiclass classification task so that the distribution of our predicted values more closely matched that of the actual values, where we used a weight decay of $1e$-7 for a slight regularising effect. The Adam optimiser with a learning rate of $1e$-3 is typical, but we set both the moments of Momentum and RMSProp to 0.8 since this seemed to give rise to better slightly better F1-scores compared to the typical values of 0.9 and 0.999 respectively. We also created our "jump-down-style" learning rate scheduler with a decay ratio of 0.5 because it made the learning rate sufficiently small so that gradient descent could surely converge on a local minima in the loss function landscape; and set the number of jumps to be 6 and epochs to be 200 so that we could control the step size directly.

iv. Evaluation metrics

It is clear according to Figure 5 that our model performs best on every single metric except precision, in which it is beaten by model model 10, which also is very similar to our best model. As can be seen in the loss plot in Figure 14, the model does spectacularly well on both training *and* validation sets, so that it seems there is no overfitting at all. Similarly in Figure 15, we see that the validation accuracy very closely approximates the train accuracy at about 0.3992 or 39.92%.

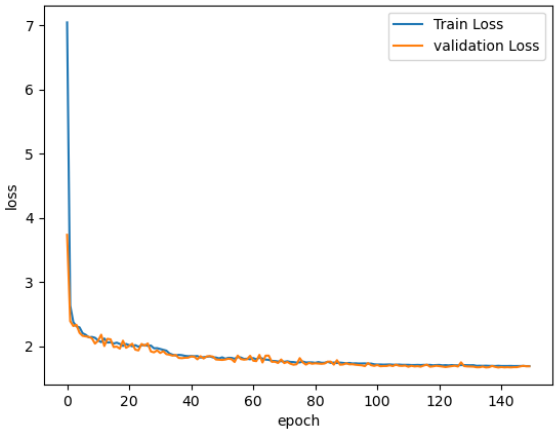


Figure 14

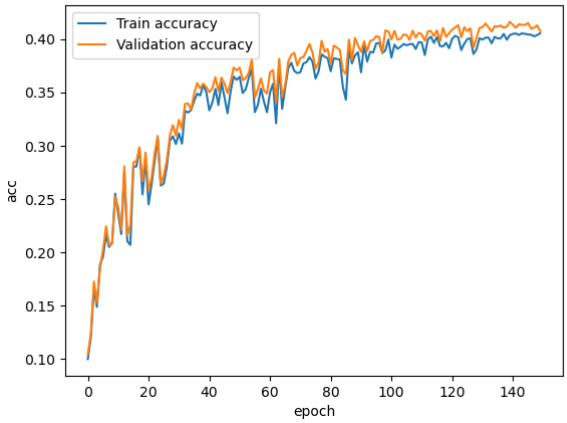


Figure 15

Discussion and conclusion

i. Problem and solution

To conclude, we had a multiclass classification task with 10 classes or categories. We were given a dataset of 50,000 training samples and 10,000 test samples. We split the 10,000 test samples into half for validation set (5,000) and half for a new test set (5,000). To classify these, we designed a multilayer perceptron with 7 hidden layers and various regularisation and optimisation methods, loss and activation functions, and other settings. In a series of experiments, we altered the network architectures or one or more of these aforementioned settings, to try to determine what were the best combination of settings, i.e. the best model. We used 5 evaluation metrics (loss accuracy, precision, recall, and F1-score) as well as time to help us discard certain models and show that other models were quite good at the classification task. This we did for analysing hyperparameters (learning rate, moment for momentum, and moment for RMSProp) as well as for ablation studies, which involved getting rid of, or changing, certain features of our network to try and obtain an even better result with reference to the evaluation metrics. For example, if we get rid of dropout layers in our network, how well or badly does the network perform?

ii. Learnings and findings

As a result of our experiments, we discovered that the best model was actually the simplest one, that did not have need of using any regularisation methods due to the sheer large scale of the dataset. We also found that we needed to have a very specific combination of settings in the network in order for it to learn as optimally as we could make it, for example, using Adam instead of SGD, minibatches instead of single examples or batches, leaky ReLU instead of ReLU or tanh, and a learning rate scheduler. If any one of these were changed in a certain way or omitted from our network, the accuracy or F1-score could potentially drop very dramatically – an undesirable result. The main takeaways from this task were twofold: firstly,

we learned every single building block of a multilayer perceptron (MLP) which are also used in more advanved networks like convolutional networks, and secondly, we learned the long and arduous process of experimenting with different hyperparameters and configurations for the network, including its architecture, to achieve the highest results with regard to common evaluation metrics.

Specifications

i. Computer
   a. 11th Gen Intel® Core(TM) i7-1165G7 CPU @ 2.80GHz
   b. 16GB RAM

ii. Packages
   a. VS Code (for IDE) – version 1.77.1
      i. Extension: Python
      ii. Extension: Jupyter
   b. Python – version 3.9.16
   c. numpy – version 1.23.5
   d. pandas – version 1.5.3
   e. matplotlib – version 3.7.0

Bibliography

- He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." In *Proceedings of the IEEE international conference on computer vision*, pp. 1026-1034. 2015.
- Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).
- Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." In International conference on machine learning, pp. 448-456. pmlr, 2015.