# COMP5329 - Deep learning - Assignment 1 Report

Group member:

SID:____460467566_______ NAME:__Fangzhou_Shi_____
SID:____470503528_______ NAME:__Haoyu_He________
SID:____470540420_______ NAME:__Jun_Xiong________

# Multilayer Neural Network - A Multi-class Classification

Table of contents

## 1. Introduction: What's the aim of the study?

Deep Learning is an artificial intelligence function that imitates the workings of the human brain in processing data and creating models for decision making. Deep Learning is a subset of Machine Learning in Artificial Intelligence(AI) that has networks capable of learning unsupervised from data that is unstructured and unlabeled. Deep Learning is also known as Deep Neural Learning or Deep Neural Network.
The study aims to establish a multi-class classification on a provided dataset. After we built the neural network, the training dataset is used to train the classification model to increase the accuracy. There are several modules that we used to improve the classifications. The classification has two hidden layers, and the activation is Relu. We have two parameterisation methods, the Weight Normalization and the Batch Normalization. The first normalization aims to normalize the weight in a neural network, and the purpose of the second normalization is to standardize the input data in each one layer. The dropout is to make the classification more efficient and prevent overfitting. There are other modules will be showed in the next part of this report. The purpose of these modules is to make the classification more efficient and increase the accuracy of prediction.

## 2. Introduction: Why is the study important?

Deep Neural Network can learn features in an unsupervised manner. With enough data and a good network architecture, neurons in a deep network can learn abstract features by themselves. Deep Learning can be applicated in classification task and be more reliable than other technical. The study is important because it provides students opportunities to have a better understanding of the Deep Neural Network and use the network to solve practical problems. After students do the assignment, we do know how to build a neural network with more than one hidden layer and using activation function in each layer. There are some other modules helps us to understand better about the whole operation, the two parameterization methods, the momentum in SGD and cross-entropy loss lead to right directions with suitable parameters. The mini-batch training increases the efficiency of the classification. In a short conclusion, the purpose of this study is to helps students to know the principle of Deep Learning and how to practice in real life.

## 3. Methods: Pre-processing

When pre-processing each epoch, we shuffle all the data in our mini-batch training set. This act aims to make sure that every mini-batch are representative of the entire training set and in this way, we make the sequence of the mini-batches more likely to be different and these mini-batches in all epochs are more likely to be unique.

## 4. Methods: The principle of different modules

### 1) More than one hidden layer

The simple perceptrons are just a linear classifier which cannot make sense for those non-linear problems. The neural network appears and does very good to solve the non-linear problems.
In this assignment, we use a four-layer network, including two hidden layers, one input layer and one output layer. Each hidden layer has 70 neurons, which can produce the best performance after the experiment. Besides, after the output layer, there is a Softmax layer which can transfer the outputs to make sense of probability.

### 2) Relu Activation

This part is implemented according the Relu's formula:f(x) = max(0,x) , and it's derivation is $f(x)' = \begin{cases} 1, & x > 0 \\ 0, & x <= 0 \end{cases}$.

### 3) Momentum in SGD

Since normal SGD has trouble in navigating areas where the surface curves much more steeply in one dimension than in another, we add Momentum to get rid of it. Momentum can accelerate SGD in the relevant direction and dampen oscillations. (i.e.comparing to SGD, the momentum term increase for dimensions whose gradients point in the same directions and reduce updates for dimensions whose gradients changes directions.)
In the process of MSGD, we need to define several parameters, the learning rate, is 0.001, the initial velocity Vt set to zeros and the velocity decay is 0.9. The formula of MSGD is below:

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - v_t$$

The $\nabla_\theta J(\theta)$ is the gradient of objective function, are parameters.
After the calculate, we upgrade the weights and biases. The MSGD can be used in Backpropagation module.

## 4) Adam Algorithm

Adam is an awesome adaptive learning rate optimise to improve gradient descent efficiency for our multilayer neural network, with low memory consumption. It is like a combination of momentum and RMSProp, which uses both first order momentum and second order momentum to help update parameters. The learning rate is no longer fixed as normal SGD, which will become larger when customising infrequently update dimension parameters and smaller when customising frequently update parameters.

$\alpha$: learning rate, which controls update proportion of weights.
$\beta_1$: first order moment estimation decay rate (set 0.9 in our assignment)
$\beta_2$: second order moment estimation decay rate (set 0.999 in our assignment)
$\epsilon$: a very small number, just in order to prevent zero denominator.
The implementation of Adam algorithm is as follow:
Set $t$ as the $t$th iteration, $\theta_t$ as the parameters value in t th iteration:
Get original SGD gradients at last iteration:

$$g_t = f'(\theta_{t-1})$$

Update biased first moment estimate:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

Update biased second moment estimate:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

Then we need to get the bias-corrected:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Now we can update the parameters:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{(\sqrt{\hat{v}_t + \epsilon})}$$

The result turns out to be that using Adam is much faster in convergence.

## 5) Dropout

The Dropout module can be used to alleviate overfitting in training phase efficiently. The Dropout function have two parameters, the input data and the probability of each units is retained. The binomial function used to output a vector whose dimensions is the same as the input data, the vector of 0 to 1 distribution. The Dropout function can be used in the processing of training and in each neural network. For this assignment, we assign 0.2 for the drop out probability of the first hidden layer and 0.5 for the second hidden layer.

$$r_i^{(l)} \sim \mathrm{Bernoulli}(p)$$
$$\boxed{\widetilde{\boldsymbol{y}}^{(l)} = \boldsymbol{r}^{(l)} * \boldsymbol{y}^{(l)}}$$
$$z_i^{(l+1)} = \boldsymbol{w}_i^{(l+1)} \widetilde{\boldsymbol{y}}^{(l)} + b_i^{(l+1)}$$
$$y_i^{(l+1)} = f\left(z_i^{(l+1)}\right)$$

**6) Softmax and cross-entropy loss**

Softmax is a function that very suitable for multi-class classification problem, which takes an N-dimensional vector numbers and transforms them into another N-dimensional vector whose numbers are all in range of (0,1). If the sum the new vector of numbers goes up, you would find the result is 1. The formula of Softmax is: $P_j = \frac{e^{output_j}}{\sum_{k=1}^{N} eoutputk}$, which can calculate the probability for this sample in class j. The derivation for Softmax is as follow:

If i = j,

$$\frac{\partial p_i}{\partial output_j} = \frac{\partial \frac{e^{output_i}}{\sum_{k=1}^{N} eoutputk}}{\partial output_j} = \frac{e^{output_i} \sum_{k=1}^{N} e^{output_k} - e^{ouput_i} e^{output_j}}{(\sum_{k=1}^{N} e^{output_k})^2}$$

$$= \frac{e^{output_i} (\sum_{k=1}^{N} e^{output_k} - e^{output_j})}{(\sum_{k=1}^{N} e^{output_k})^2}$$

$$= \frac{e^{output_i}}{\sum_{k=1}^{N} eoutputk} \times \frac{(\sum_{k=1}^{N} e^{output_k} - e^{output_j})}{\sum_{k=1}^{N} eoutputk}$$

$$= p_i(1 - p_j)$$

If i != j:

$$\frac{\partial p_i}{\partial output_j} = \frac{\partial \frac{e^{output_i}}{\sum_{k=1}^{N} eoutputk}}{\partial output_j} = \frac{0\cdot\sum_{k=1}^{N} e^{output_k} - e^{output_i}\cdot e^{output_j}}{(\sum_{k=1}^{N} e^{output_k})^2}$$

$$= -\frac{e^{output_i}}{\sum_{k=1}^{N} eoutputk} \times \frac{e^{output_j}}{\sum_{k=1}^{N} eoutputk}$$

$$= -p_i \cdot p_j$$

The Cross-entropy loss needs to be implemented to provide a loss function for the neural network to optimise. This kind of loss function indicates the distance between what the original label of the sample and what the neural network model prediction.

The formula for Cross-entropy loss is: $(label, p) = -\sum_{i}^{N} label_i \cdot \log(p_i)$. Note that ρrepresent for the outputs of the SoftMax function. From the formula, we can see that L will decrease if p is close to the actual label.

When executing back propagation, we need to calculate the δ for the output layer, which should be:

$$\delta_i = \frac{\partial L}{\partial output_i}$$

$$= -\sum_{k}^{N} label_k \frac{\partial log(p_k)}{\partial output_i}$$

$$= -\sum_{k}^{N} label_k \frac{\partial log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial output_i}$$

$$= -\sum_{k}^{N} label_k \frac{1}{p_k} \times \frac{\partial p_k}{\partial output_i}$$

$$= -label_i(1-p_i) + \sum_{k\neq i}^{\square} label_k \cdot \frac{1}{p_k} \cdot (p_k p_i)$$

$$= p_i(label_i + \sum_{k\neq i}^{\square} label_k) - label_i$$

Label is a vector who has only one element is 1 and others are 0s, hence $(label_i + \sum_{k\neq i}^{\square} label_k)$ is 1. Amazing we get a very simple version formula of the δwhich is $\delta_i = p_i - label_i$. This make us much easier to write the code.

**7) Forward Propagation**

In this part, we input the samples to our neural network to let them go through each hidden layer and finally come to the output layer. The outputs for is a prediction for this iteration, we need to calculate the distance between predictions for each iteration and the actual label using Cross-entropy, which has been discussed earlier. For each layer, the network does some calculations as follow:

$net_k = \sum_{j=1}^{n} output_j \cdot w_{kj} + b_k$ (Note that the output for the input layer is a vector of sample features.) $output_j = f(net_j)$.

When it comes to the output layer, the outputs vector will be passed to a Softmax function to calculate the probability for each class. After that, the outputs from the SoftMax layer will be used to calculate the Cross-entropy loss.

## 8) Back Propagation:

After we come up with the Cross-entropy loss in each mini-batch iteration, we should do something to adjust the weights and biases to minimise the loss error. The method we implemented here to achieve this is called Stochastic Gradient Descent with momentum (please see the momentum module above to get detail information), which use Back Propagation algorithm to calculate the gradients for every weight and then update weights and biases by subtracting their gradients. The main principles for SGD are the gradients for each weight and biases can lead the loss function to a local or global minimum status.
Back Propagation algorithm for our implementation is as follow:
For hidden-to-output weight $w_{kj}$:

$$net_k = \sum_{j=1}^{n} output_j \cdot w_{kj} + b_k$$
$$L(w) = \sum_{k=1}^{N} label_k \cdot log(p_k)$$
$$f(net_k) = Relu(net_k)$$

$$\frac{\partial L}{\partial net_k} = \frac{\partial L}{\partial output_k} \cdot \frac{\partial output_k}{\partial net_k} = (p_k - label_k) \cdot f'(net_k)$$

(This part has been mentioned in “Cross-entropy” derivation.)
Hence:

$$\frac{\partial L}{\partial w_{kj}} = \frac{\partial L}{\partial net_k} \cdot \frac{\partial net_k}{\partial w_{kj}} = (p_k - label_k) \cdot f'(net_k) \cdot output_j$$

$$\frac{\partial L}{\partial b_k} = \frac{\partial L}{\partial net_k} \cdot \frac{\partial net_k}{\partial b_k} = (p_k - label_k) \cdot f'(net_k)$$

Using Momentum Gradient Descent method to update weight and biases. (Please recording the Momentum module above to see MSGD in details.) For hidden-to-hidden weight $w_{ji}$:

$$\begin{aligned}\frac{\partial L}{\partial w_{ji}} &= \frac{\partial L}{\partial output_j} \cdot \frac{\partial output_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}} \\ &= -\sum_{m=1}^{n} \delta_m w_{mj} \cdot \frac{\partial output_j}{\partial net_j} \cdot \frac{\partial net_j}{\partial w_{ji}} \\ &= -\sum_{m=1}^{n} \delta_m w_{mj} \cdot f'(net_j) \cdot output_i \\ &= \delta_j \cdot output_i\end{aligned}$$

$$\frac{\partial L}{\partial b_j} = \delta_j$$

Using Momentum Gradient Descent method to update weights and biases (Please recording the Momentum module above to see MSGD in details.)

The input-to-hidden weights gradient calculation is the same as hidden-to-hidden, which keep in mind that the outputs of the input layer are the vector of sample's features

**9) Mini-batch training**

Instead of doing MSGD for each sample or all samples in the training set per epoch, we divide the training set into mini-batches and doing MSGD and backpropagation after going through each mini-batch. If all mini-batches have been passed, one epoch will end. For this assignment, our mini-batch size is 64, which seems can lead to a best performance after the experiment. This approach makes the estimated gradient at each iteration is more reliable than One-example based MSGD, and faster than Batching training.

**10) Batch Normalization**

Batch normalisation is useful in two ways. Firstly, instead of only normalize the data in the input layer, Batch normalization normalize data in each layer to have small difference, which accelerates the training and gives the overall higher accuracy. Secondly, BN enables the neural network to use a higher learning rate which makes converging even faster.

Forward Feed:
In our implementation we put the batch normalization layers before the activation function in every hidden layer.
Given the input of the values of x over a mini-batch before the activation function: $\mathcal{B} = \{x_{1 \ldots m}\}$ and two parameters ( $\gamma$; $\beta$ ) to be learnt, we wish to output $y_i = \mathrm{BN}_{\gamma,\beta}(x_i)$. In the first step we get the mean $\mu = \frac{1}{m}\sum_{1}^{m} xi$ and variance $\sigma^2 = \frac{1}{m}\sum_{1}^{m}(xi - \mu)^2$ of the mini-batch. Then our normalized outcome $\widehat{xi}$ = $\frac{xi - \mu}{\sqrt{\sigma^2 + \epsilon}}$. And finally, we scale and shift our output to be $\widehat{yi}$ = $\gamma\widehat{x}_i + \beta$.

Backward Propagation:

With our architecture for neural network, we get the input: {dout1…doutm} from the ReLU activation function's outcome derivative. We can get $d\beta$ = $\sum_{1}^{m} douti$, and then get the derivative for $\gamma$, $d\gamma = \sum_{1}^{m} douti * \widehat{xi}$. After that, we can get $d\widehat{xi} = dout * \gamma$.. With that, we change our notation for dout to be dl, we can get the derivative of xi, $\beta$ and $\sigma^2$ respectively to be

$$\frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i-\mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$ ①

$$\frac{\partial \ell}{\partial \mu_{\mathcal{B}}} = \left( \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}} \right) + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i-\mu_{\mathcal{B}})}{m}$$

, ② and

$$\frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial \ell}{\partial \widehat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$ ③

and finally, we can get the output of our backpropagation for BN to be

$$\frac{\partial \ell}{\partial x_i} = \frac{\partial \ell}{\partial \widehat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}} + \frac{\partial \ell}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i-\mu_{\mathcal{B}})}{m} + \frac{\partial \ell}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$.

**11) Weight decay**

Weight decay aims to add an additional L2 penalty when updating the weights basing on the weight updating rule as a regularization to avoid overfitting.

Forward Feed:
In an effort to achieve this purpose, we add the original cost function with another term. Suppose that C is the cost, C0 is the cost without weight decay technology, n is the length of the size of our mini-batch training set and wi is the corresponding weight for the m features.

$$C = C0 + \frac{\lambda}{2*n} \sum_{i}^{m} wi^2$$

Backward Propagation:

With our modification in the Forward Feed, we can easily get that the updating rule for weight wi has been modified as well from:

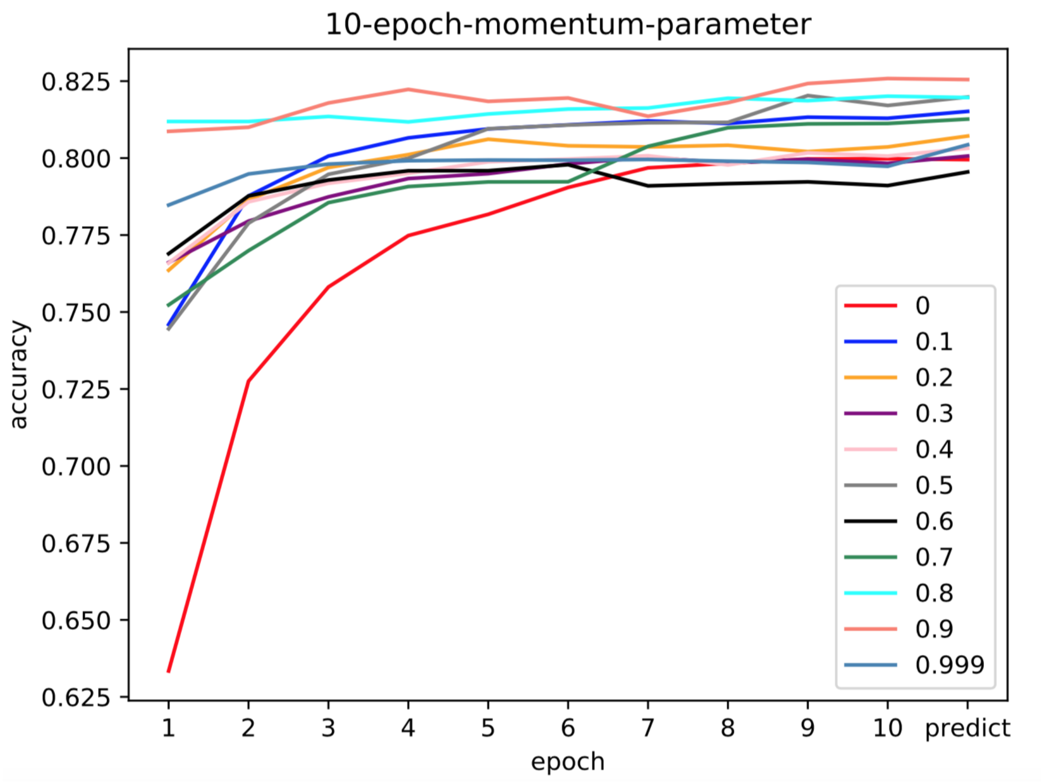
$wi = wi - \eta \frac{dC}{dwi}$ to $wi = wi - \eta \frac{dC}{dwi} - \eta * \lambda * wi$ in this way the large weights gets more penalty and the cost function get regularized.

## 5. Experiments and Results: Accuracy

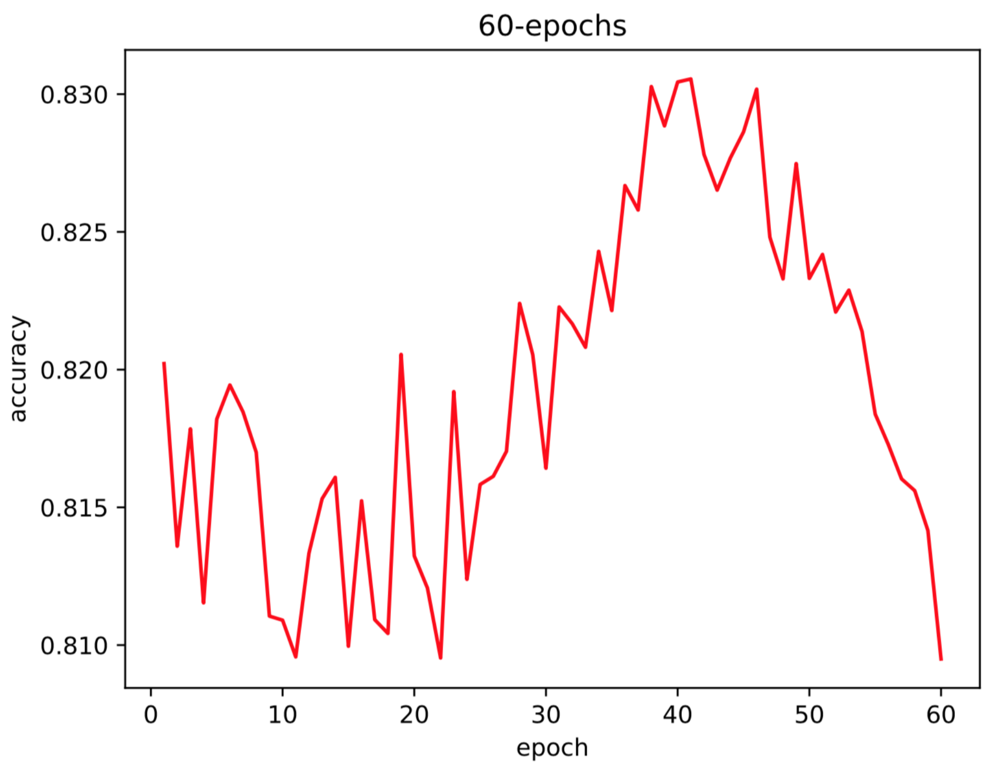
The accuracy of this assignment is around 84%, and we will explain the result detailed in next part of this report.

## 6. Experiments and Results: Extensive Analysis

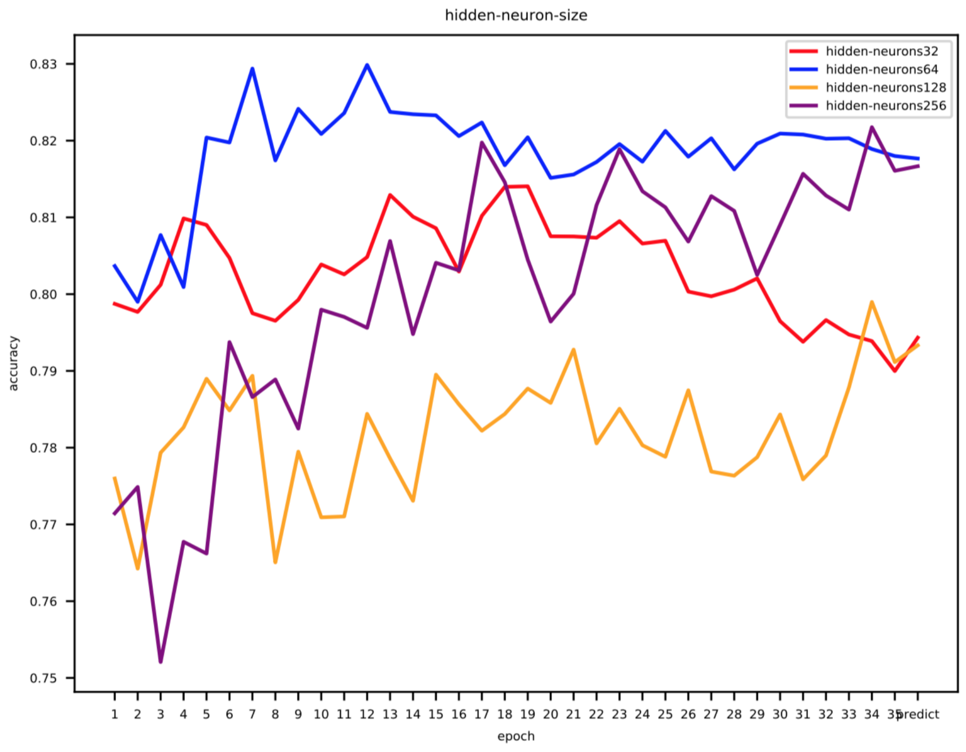
When we are analyzing our data, we are doing a 10-fold validation: we divide the given data set into training set (54000, 9/10 of the total dataset) and testing set (6000, 1/10 of the total dataset). When get the training result, we utilize the leant parameter to test our result for training which is labelled in the pictures as “predict”.
Firstly, we try to decide on the parameter $\gamma$which contributes to the momentum term. With other parameters unchanged, we experiment $\gamma$ from 0 to 0.999, we can get the result shown in the picture below. It’s obvious that when $\gamma$ is around 0.8 to 0.9 the converging is quicker than the others, and accuracy also comes to the peak. Basing on the figure, we choose $\gamma$to be 0.9.



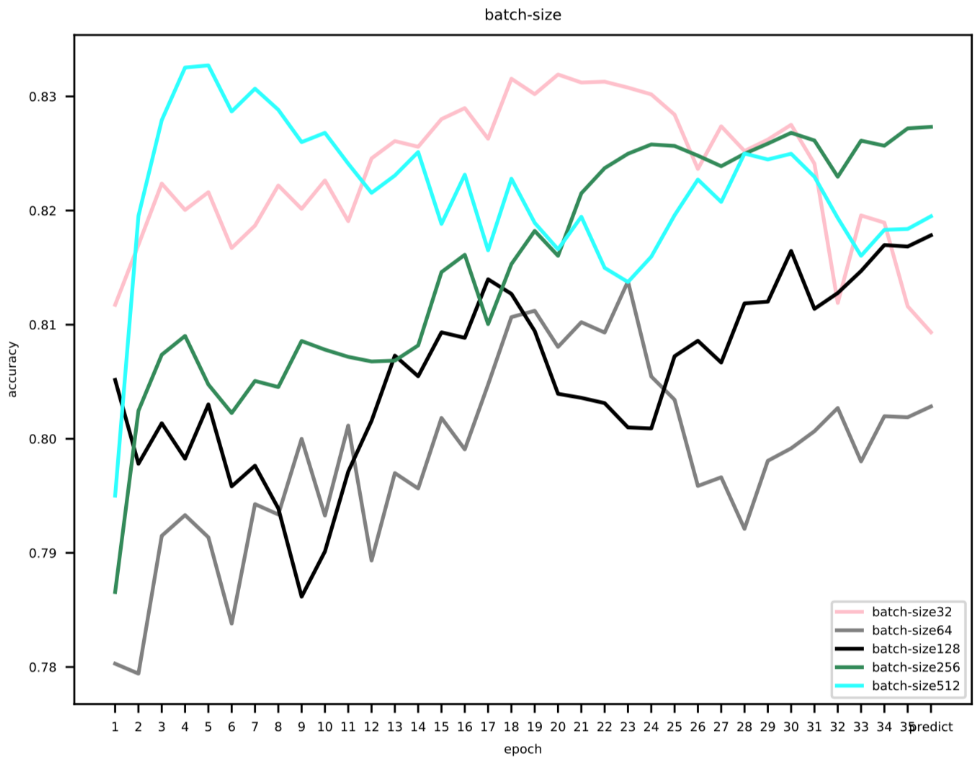
Then testing for how many epochs we need for our training. By generating the picture below, we know that after 35 epochs the accuracy has no more upward trend. So, we set the training epoch to be 35.



When trying to get the proper number for hidden neurons in each hidden layer, we get the result shown by the picture below. In this case, it shows that when having too many hidden neurons, overfitting or other side-effects can damage our accuracy and the amount around 64 gets the best accuracy.



After setting the neurons in each hidden layer to be 70, we try to find the best batch size for each mini-batch and it is shown that with the batch size of 64 gets the best accuracy of the five sizes.



## 7. Discussion: Meaningful and relevant person reflection

After gathering all the results and going through some visualizing methods, we find out that these modules on some level all helps increasing the result of the training in our deep training neural network.

Having more than one the hidden layer enables the neural network to learn more complicated distribution, then the activation function after each linear transformation function allows non-linear features be learnt. After that, with momentum term's smoothing and trying-to-escaping-local minimum feature, the converging can become a lot faster and the accuracy has been raised. Though our accuracy does not benefit a lot from the drop-out term or the weight decay function (the former deals overfitting by killing some of the neurons in hidden layer and the latter tries to avoid the same issue by adding a L2 regularization) since we might don't have to handle overfitting in our neural network with small number of neurons in hidden layers, we get the idea of how they perform by giving small parameters for them.

## 8. Conclusion: Meaningful conclusions based on results

In conclusion, the purpose of this assignment is to establish a multi-class classification on a provided dataset, we need to define many functions that help to improve the accuracy of the main function. We use a four-layer network, including one input layer and one output layer. Each hidden layer has 64 neutrons, after the output layer, there is a Softmax layer which can transfer the outputs to make sense of probability. The mini-batch training is used in the process of training, this approach makes the estimated gradient at each iteration is more reliable than One-example based MSGD, and faster than Batching training. Batch normalization is useful in two ways. Firstly, Batch normalization normalize data in each layer to have small difference in order to accelerates the training and gives the overall higher accuracy. Secondly, BN enables the neural network to use a higher learning rate which makes converging even faster. The dropout module used to prevent overfitting and the activation in each layer is Relu activation. Module of Weight decay aims to avoid overfitting by adding an additional L2 penalty as a regularization. Finally, the output is the accuracy of this classification by predicting test set. We set the momentum term $\gamma$ to 0.9, the training epoch to 35, the hidden-neurons to 64 and the 64-batch size. We use these functions and modules below to improve the accuracy result to around 84%.

## 9. Others

We use LATEX to write the formula and use matplotlib to visualization.

## 10. Instruction for running code:

Before running our code, please look at the header of our code, you can see a “beginMode” variable, then select beginMode "predict" or "validation". If "predict", the test data file will be loaded and print classification results. If "validation", the test data will be selected from the training set and print accuracy for each epoch.
After that, you can just smoothly run the code by “python3 NeuralNetwork.py”.