# REPORT FOR ARTIFICIAL NEURAL NETWORK ASSIGNMENT 1

**Xiang Zhang**
Department of Computer Science and Technology
Tsinghua University
xiang-zh17@mails.tsinghua.edu.cn

September 24, 2019

**ABSTRACT**

In this assignment, a variant of forward and backward computation of multi-layer perceptron (MLP) network is implemented, along with activation function (ReLU, Sigmoid) and loss function (Mean Squared Error, Softmax Cross Entropy) as required. The evaluation of this implementation is based on the classification of MNIST dataset, where model performances are examined and compared with regard to the choice of activation functions, loss functions and/or number of hidden layers in MLP network, yielding a brief analysis afterwards.

## 1 Preliminaries

In order to perform gradient descent, derivatives of loss function ($\mathcal{J}$) with respect to parameters of the network ($\theta$) is demanded. Here some basic mathematical forms, which directly correspond to the implementation code, are presented for the sake of better comprehension.

### 1.1 Network Formulation

In this implementation, activation functions are treated as independent non-trainable layers, allowing a uniformed representation of gradients.

#### 1.1.1 MLP Network

Consider one layer of MLP network, which has nodes fully connected to these of their previous layer (can either be input layer or output of some other network). The layer can be formulated as a linear function mapping input to its output

$$\boldsymbol{x}^{(l)} = \boldsymbol{f}(\boldsymbol{x}^{(l-1)}) = \boldsymbol{x}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)} \tag{1}$$

Where $\boldsymbol{x}^{(l-1)}$ is the input of previous layer with dimension $N$(Batch size) $\times\, n_{l-1}$(Dimension of input vector), whereas $\boldsymbol{x}^{(l)}$ is $N \times n_l$ and $\boldsymbol{W}^{(l)}$ is a $n_{l-1} \times n_l$ weight matrix.

#### 1.1.2 Activation Function

Activation function $\boldsymbol{f} : \mathbf{C} \mapsto \mathbf{C}$ ($\mathbf{C}$ denotes a certain set) used by vectors is defined through its scalar counterpart $f : \mathbb{R} \mapsto \mathbb{R}$

$$\boldsymbol{f}(\boldsymbol{x}_{(m\times n)}) = (f(x_{ij}))_{(m\times n)} \tag{2}$$

$f$ can be ReLU or Sigmoid, which are defined as

$$ReLU(x) = \max(x, 0) \tag{3}$$

$$Sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{4}$$

### 1.1.3 Loss Function

Loss function $\mathcal{J} : \mathbf{M} \mapsto \mathbb{R}$ takes in a vector or matrix and maps it to a real number denoting loss. There are two types of them implemented, which are as follows

$$\text{Mean Squared Error: } MSE(\boldsymbol{x}_{(N\times m)}) = \frac{1}{2N}\sum_{n=1}^{N}\left|\left|\boldsymbol{x}^{(n)} - \boldsymbol{t}^{(n)}\right|\right|_2^2 \tag{5}$$

$$\text{Softmax Cross Entropy: } CE(\boldsymbol{x}_{(N\times m)}) = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{m} t_k^{(n)} \ln h_k^{(n)} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{m} t_k^{(n)} \ln \frac{\exp\left(x_k^{(n)}\right)}{\sum_{j=1}^{K}\exp\left(x_j^{(n)}\right)} \tag{6}$$

Where $N$ is batch size, $m$ is output dimension and $\boldsymbol{x}$, $\boldsymbol{t}$ for predictions and ground truth respectively.

## 1.2 Gradient Propagation

A $K$-layered MLP network can be generalized as a composite function, mapping input to output in an end-to-end fashion

$$\boldsymbol{x}^{(l)} = \boldsymbol{f}^{(l)}(\boldsymbol{x}^{(l-1)}\boldsymbol{W}^{(l)} + \boldsymbol{b}^{(l)}),\ l = 1, 2, \cdots, K \tag{7}$$

$\boldsymbol{f}^{(l)}$ is the activation function and can be ReLU, Sigmoid or Identity($\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{x}$). $\boldsymbol{x}^{(0)}$ and $\boldsymbol{x}^{(K)}$ are input and output respectively. The loss is computed through $\mathcal{J}(\boldsymbol{x}^{(K)})$.

One can easily derive the following equation with the aid of chain rule

$$\begin{aligned}
&\frac{\partial \mathcal{J}}{\partial \boldsymbol{W}^{(p)}} \\
&= \frac{\partial \mathcal{J}}{\partial \boldsymbol{x}^{(K)}} \odot \frac{\partial \boldsymbol{x}^{(K)}}{\partial\left(\boldsymbol{x}^{(K-1)}\boldsymbol{W}^{(K)} + \boldsymbol{b}^{(K)}\right)} \cdot \frac{\partial\left(\boldsymbol{x}^{(K-1)}\boldsymbol{W}^{(K)} + \boldsymbol{b}^{(K)}\right)}{\partial \boldsymbol{x}^{(K-1)}} \odot \cdots \odot \frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)} + \boldsymbol{b}^{(p)}\right)}{\partial \boldsymbol{W}^{(p)}} \\
&= \frac{\partial \mathcal{J}}{\partial \boldsymbol{x}^{(K)}} \odot \boldsymbol{f}'^{(K)}\left(\boldsymbol{x}^{(K-1)}\boldsymbol{W}^{(K)} + \boldsymbol{b}^{(K)}\right) \cdot (\boldsymbol{W}^{(K)})^{\mathrm{T}} \odot \cdots \odot \frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)}\right)}{\partial \boldsymbol{W}^{(p)}}
\end{aligned} \tag{8}$$

Where $\odot$ denotes element-wise multiplication and $\cdot$ is normal matrix multiplication. $\frac{\partial \mathcal{J}}{\partial \boldsymbol{x}^{(K)}}$ is a $N \times n_K$(output dimension) matrix since batch size is included here. $\boldsymbol{f}'^{(K)}\left(\boldsymbol{x}^{(K-1)}\boldsymbol{W}^{(K)} + \boldsymbol{b}^{(K)}\right)$ is computed by taking $f'^{(K)}$ through all elements of $\boldsymbol{x}^{(K-1)}\boldsymbol{W}^{(K)} + \boldsymbol{b}^{(K)}$, yielding an $N \times n_K$ matrix. By multiplying values left of $(\boldsymbol{W}^{(K)})^{\mathrm{T}}$ with it, the gradient is passed from layer $K$ to layer $K-1$, and the dimension of gradient vector changes from $N \times n_K$ to $N \times n_{K-1}$.

### 1.2.1 Gradient of Current Layer

The notation $\frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)}\right)}{\partial \boldsymbol{W}^{(p)}}$ here is a symbolic representation of the gradient within current layer, which is a tensor defined as

$$\left(\frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)}\right)}{\partial \boldsymbol{W}^{(p)}}\right)_{ij} = \frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)}\right)}{\partial w_{ij}^{(p)}} = \begin{bmatrix} 0 & \dots & x_{i1}^{(p-1)} & \dots & 0 \\ 0 & \dots & x_{i2}^{(p-1)} & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & x_{in_{K-1}}^{(p-1)} & \dots & 0 \end{bmatrix} \quad (0 < i \le n_{K-1},\ 0 < j \le n_K) \tag{9}$$

where $i$-th column of $\boldsymbol{W}^{(p)}$ resides on the $j$-th column of the resulting matrix.

Note in (8), left side of $\frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)}\right)}{\partial \boldsymbol{W}^{(p)}}$(denoted as $\boldsymbol{G}^{(K)}$) is applied to it in a manner of element-wise multiplication and summation to form a normal matrix in $\mathcal{M}_{n_{K-1},n_K}$ (the summation is owing to the requirements of multivariate chain rule), yielding $\frac{\partial \mathcal{J}}{\partial w_{ij}^{(p)}} = \sum_{m,n}\left(\left(\boldsymbol{G}^{(K)}\right)_{ij} \odot \frac{\partial\left(\boldsymbol{x}^{(p-1)}\boldsymbol{W}^{(p)}\right)}{\partial w_{ij}^{(p)}}\right)_{mn}$. The equivalent form is as follows

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{W}^{P}} = (\boldsymbol{x}^{(p-1)})^{\mathrm{T}}\boldsymbol{G}^{(K)} \tag{10}$$

Similarly, we have

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{b}^{P}} = (\boldsymbol{E}_{(N\times n_K)})^{\mathrm{T}}\boldsymbol{G}^{(K)} \quad \text{where } E_{ij}=1,\ \forall i,j \tag{11}$$

This is equivalent to taking the sum of all the rows in $\boldsymbol{G}^{(K)}$ to form a vector $\boldsymbol{V}$, and then construct a matrix whose rows are entirely $\boldsymbol{V}$.

### 1.2.2 Gradient of Loss Functions

It is trivial to prove that for $MSE$ loss, the gradient takes the form

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{x}^{(K)}} = \frac{1}{N}(\boldsymbol{x}^{(K)} - \boldsymbol{t}^{(K)}) \tag{12}$$

For $CE$ loss, let $\mathbf{M} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{x}^{(K)}}$, we have

$$\mathbf{M}_{ij} = \frac{\partial \mathcal{J}}{\partial x_j^{(i)}} = \sum_{a=1}^{n_K}\frac{\partial \mathcal{J}}{\partial h_a^{(i)}}\frac{\partial h_a^{(i)}}{\partial x_j^{(i)}} = -\frac{1}{N}\sum_{a=1}^{n_K}\frac{t_a^{(i)}}{h_a^{(i)}}\cdot h_a^{(i)}(\delta_{aj}-h_j^{(i)}) = -\frac{1}{N}\sum_{a=1}^{n_K} t_a^{(i)}(\delta_{aj}-h_j^{(i)}) \tag{13}$$

where $\delta_{ij}$ is Kronecker delta. We must let $a=j$ in order to get $\delta_{aj}=1$, therefore $\mathbf{M}_{ij} = \frac{1}{N}(h_j^{(i)}\sum_{a=1}^{n_K} t_a^{(i)} - t_j^{(i)})$, indicating

$$\mathbf{M} = \frac{1}{N}(\mathbf{h}' - \mathbf{t}) \tag{14}$$

where $\mathbf{h}'$ can be obtained by taking the $h$-value of each element in output matrix and scale it with the sum of the corresponding row in $t$-matrix. When labels are one-hot encoded, this scale factor reduces to identity.

In computing the loss gradient with respect to network parameters, $\boldsymbol{G}^{(K)}$ can be reused and multiplied by $\boldsymbol{W}^{(K)}$ to be propagated to previous layer, thus the entire gradients can be calculated within a single backward routine.

## 2 Experiments

### 2.1 Network Architecture

In my implementation, `run_mlp.py` supports reading network architecture from command line arguments, rendering it convenient to change network structure without the hassles of source code modification. Here are three major types of networks implemented, with different number of hidden layers (0, 1 and 2).

- Affine network(AffNet): No hidden layer, mapping input directly to output.
- MLP with Single Hidden Layer(MLP1)

Table 1: Single Hidden Layer Network (Activation can be either ReLU or Sigmoid)

| Layer | Type | In Dim | Out Dim |
|---|---|---|---|
| 0 | Linear | 784 | 256 |
| 1 | Activation | 256 | 256 |
| 2 | Linear | 256 | 10 |
| 3 | Activation | 10 | 10 |

- MLP with Two Hidden Layers(MLP2)

Table 2: MLP with Two Hidden Layers

| Layer | Type | In Dim | Out Dim |
|---|---|---|---|
| 0 | Linear | 784 | 500 |
| 1 | Activation | 500 | 500 |
| 2 | Linear | 500 | 256 |
| 3 | Activation | 256 | 256 |
| 4 | Linear | 256 | 10 |
| 5 | Activation | 10 | 10 |

Several pre-adjustment procedures have been carried out to optimize hyperparameters of training, and finally we came up with the following set:

`learning_rate=1e-2, weight_decay=5e-4, momentum=0.9, batch_size=100, max_epoch=50`

Note here some activation function may be equal to identity, if that is the case, the output of `Linear` layer will be adopted directly, thus the following `Activation` layer is removed.

The concrete network architecture is represented by commandline parameters used in training, which are designed to be self-explanatory and directly connected with the abbreviated names of the network. For example, `MLP1_ReLU_CE` indicates that the network has one hidden layer with **ReLU** as activation of hidden layer, **Identity** as activation of output layer, and **Cross Entropy** as loss function, whereas `MLP1_Sigmoid_ReLU_MSE` means that it is a two-layered network with **Sigmoid** as activation of hidden layer, **ReLU** for output layer, and **Mean Squared Error** as loss. See **Appendix A** for detailed information.

## 2.2 Results

The loss and accuracy during training process is presented in **Fig. 1, 2, 3**, whereas the numerical value of them during training and testing along with training time can be found in **Table 3**.

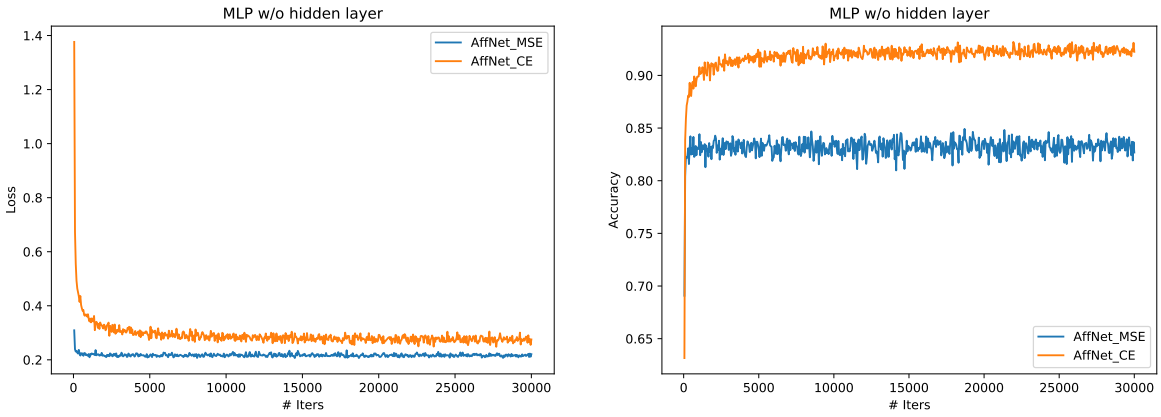


Figure 1: AffNet without hidden layer

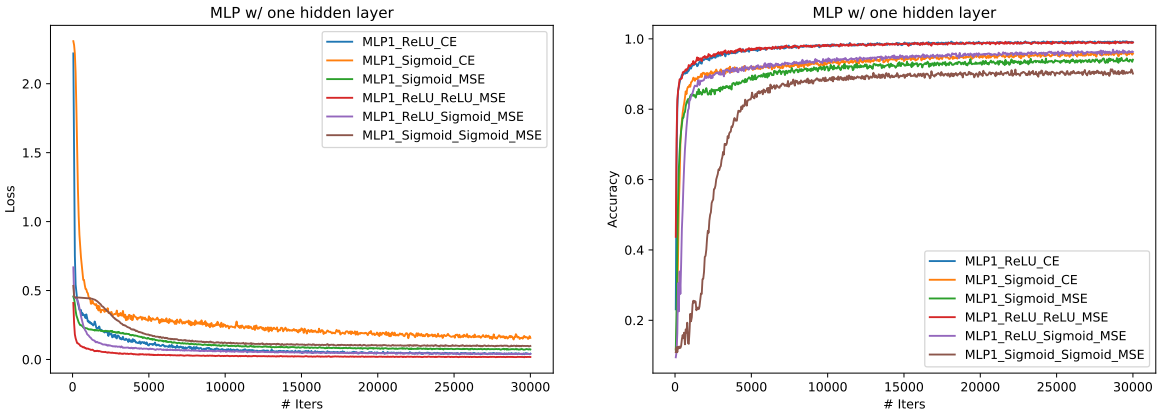


Figure 2: MLP1 with one hidden layer

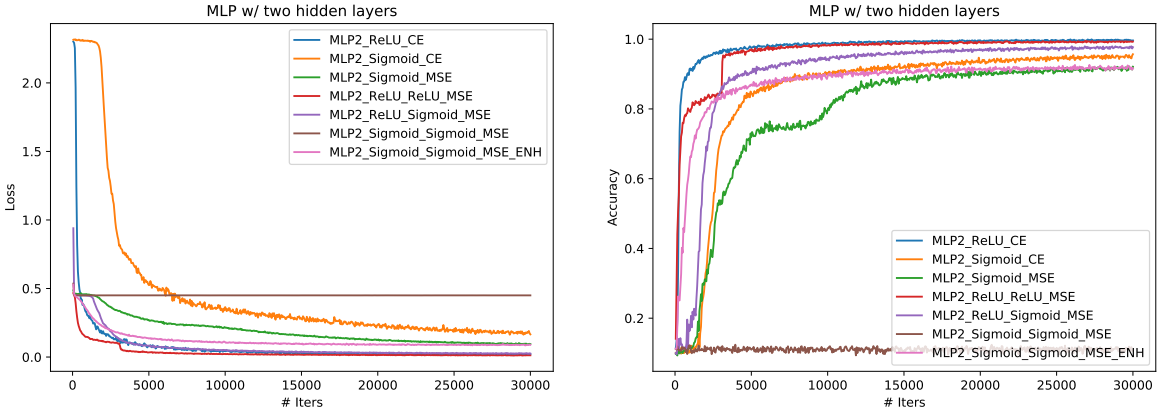


Figure 3: MLP2 with two hidden layers. Note here `MLP2_Sigmoid_Sigmoid_MSE` failed to converge, the loss of which remained flatly low, and we will discuss this later

Table 3: Training / Test loss / accuracy after 30000 iterations

| Metric | AffNet_MSE | AffNet_CE | MLP1_ReLU_CE | MLP1_Sigmoid_CE | MLP1_Sigmoid_MSE | MLP1_ReLU_ReLU_MSE | MLP1_ReLU_Sig_MSE | MLP1_Sig_Sig_MSE |
|---|---|---|---|---|---|---|---|---|
| Train Loss | 0.21670 | 0.27423 | 0.04095 | 0.15963 | 0.07384 | 0.01790 | 0.04090 | 0.09750 |
| Train Acc | 83.178% | 92.313% | 99.102% | 95.775% | 93.860% | 98.963% | 96.258% | 90.537% |
| Test Loss | 0.21354 | 0.27873 | 0.06716 | 0.15917 | 0.07351 | 0.02401 | 0.04127 | 0.09337 |
| Test Acc | 84.110% | 92.270% | 98.050% | 95.610% | 94.200% | 98.090% | 96.170% | 90.900% |
| Train Time | **20.405s** | 22.550s | 111.166s | 131.832s | 131.916s | 106.118s | 112.154s | 128.236s |

| Metric | MLP2_ReLU_CE | MLP2_Sig_CE | MLP2_Sig_MSE | MLP2_ReLU_ReLU_MSE | MLP2_ReLU_Sig_MSE | MLP2_Sig_Sig_MSE* | MLP2_Sig_Sig_MSE_ENH |
|---|---|---|---|---|---|---|---|
| Train Loss | 0.01714 | 0.17617 | 0.09550 | **0.01250** | 0.02632 | 0.44994 | 0.08873 |
| Train Acc | **99.725%** | 95.008% | 91.577% | 99.275% | 97.590% | 11.138% | 91.778% |
| Test Loss | 0.05685 | 0.18310 | 0.09118 | **0.01878** | 0.02854 | 0.44985 | 0.08580 |
| Test Acc | 98.160% | 94.730% | 91.900% | **98.280%** | 97.160% | 11.350% | 92.110% |
| Train Time | 247.885s | 298.611s | 288.587s | 251.921s | 236.015s | 296.137s | 294.468s |

# 3 Analysis

## 3.1 Choices of Activation Functions

From the results in **Table 3**, it is obvious that given identical conditions, i.e. number of layers, loss function and hyperparameters, etc., ReLU has unmatched performance in terms of loss, accuracy and training time, compared with Sigmoid. ReLU's fast training time can be attributed to the simplicity of its calculation, which only requires linear

operation instead of complex arithmetic operations needed by Sigmoid. Adding Sigmoid to output layer can hardly hamper the performance in occasions where output dimension is quite small.

Plus, it is rather an awful choice to employ Sigmoid as the activation function of hidden layers. Since the output value of Sigmoid is constrained to $[0,\ 1]$, there is high tendency that information can be compressed multiple times before it can reach the output layer, thus yielding much poorer performance, which can be corroborated by the comparison between `MLP1_Sigmoid_Sigmoid_MSE` and `MLP1_ReLU_Sigmoid_MSE`. As for the output layer, it would be reasonable to use Sigmoid in some cases where you need to regularize the output value to a 0-1 range, though in this experiment, Sigmoid could not parallel ReLU even if it resides on the output layer. Worse still, one can simply find that the gradient of Sigmoid is in $[0,\ 0.25]$, indicating that in deep MLP networks with Sigmoid as activation of hidden layers, gradient vanishing is likely to occur. (See 3.4 for details) Moreover, using ReLU as activation functions can drastically accelerate the gradient descent process as well, which can be observed from loss/accuracy-iteration figures, where ReLU produces a much steeper slope. This can be explained by the absolute value of gradient as well, since ReLU has a constant derivative 1 for $x > 0$, a value much bigger than that of Sigmoid.

### 3.2 Loss Functions

In this implementation, the Cross Entropy loss is a variant of Softmax Cross Entropy, which is equivalent to a Softmax activation function with Cross Entropy loss. It is for this reason that `AffNet_CE` has an accuracy much higher than that of `AffNet_MSE`. For networks with hidden layers, output activation is set to Identity when Cross Entropy is present to avoid conflicts with Softmax. From the comparison of `MLP_ReLU_ReLU_MSE` and `MLP_ReLU_CE`, we can find that they resemble each other in performance regardless of numbers of hidden layers, though Mean Squared Error can have a minor advantage ($\sim$ 0.04%-0.12%).

### 3.3 Number of Layers

Increasing the number of hidden layers do contribute to better performance (`MLP2_ReLU_Sigmoid_MSE` vs `MLP1_ReLU_Sigmoid_MSE`, `MLP2_ReLU_ReLU_MSE` vs `MLP1_ReLU_ReLU_MSE`) as expected, yet this happens only when the network is properly structured. In networks with Sigmoid as hidden-layer activation function, the performance is dramatically impaired when they get deeper. This is related to the gradient problem present in Sigmoid, and in extreme cases inability to perform gradient descent may occur (just as what happened in `MLP2_Sigmoid_Sigmoid_MSE`).

### 3.4 Sigmoid: the Vanishing Gradient

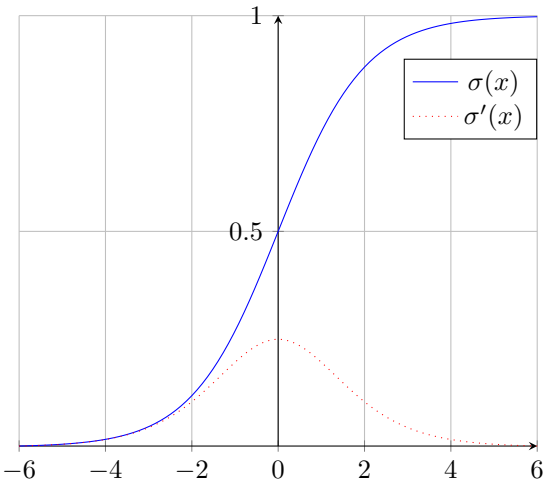


Figure 4: Sigmoid function and its derivative

As is shown in **Fig. 4**, Sigmoid has the beneficial property that its value is bounded within a range valid for probability. Yet this good property comes at a price-the gradient of Sigmoid is always smaller than 1, and only peaks when $x = 0$. This can cause problems when MLP network get deeper, in which case, the gradient can decay exponentially during the process of back propagation, meaning the closer you are to the input layer, the smaller gradient you will obtain. Finally, the layers with very small gradients can hardly learn, which will behave as the network is not learning at all.

In our experiment, the loss and accuracy of `MLP2_Sigmoid_Sigmoid_MSE` is almost a flat line, indicating the network has failed to learn. Adding some lines of codes, we are able to output the L2-norm of the gradient matrix during batch update process, and here is some output

```
Training iter 350, batch loss 0.4503, batch acc 0.1022
Linear: fc, (?,784,500): L2-norm of gradient is 0.00022083940633880848
Linear: fc, (?,500,256): L2-norm of gradient is 0.0025717604780565602
Linear: fc, (?,256,10): L2-norm of gradient is 0.04024818961584222
```

The gradient has already dropped to $2 \times 10^{-4}$ in the first hidden layer, which combining with learning rate $0.01$ will generate an extremely tiny $\Delta \boldsymbol{W}$, thus explaining why the loss will never drop.

### 3.4.1 Mitigation

One way to combat gradient vanishing is to tune parameters related to the initial weights, i.e. initial standard deviation. Since all the initial weights are Gaussian distributed, increasing variance could yield more weight values that are farther from zero, thus increasing overall gradient during back propagation phase. In `MLP2_Sigmoid_Sigmoid_MSE_ENH`, the variance of initial weights is adjusted for each layer, in which case, the layer closer to input will have a higher variance, and the variance decays exponentially as we step into next layer. After optimization, this network can achieve an accuracy of $92.110\%$, a bit higher than `MLP1_Sigmoid_Sigmoid_MSE`, which corroborates the fact that increasing the number of layers can better the performance. Simultaneously, the learning curve in **Fig. 3** becomes steeper, indicating the network converges quicker in this case.

## 4 Conclusion

In this assignment, the author implemented a multi-layer perceptron network equipped with activation function (ReLU, Sigmoid) and loss function (Mean Squared Error, Softmax Cross Entropy). Through the comparison experiments, it is found that using ReLU as the activation function and increasing the number of hidden layers can better the performance of the model. And the implementation can achieve a maximum accuracy of $98.280\%$ in MNIST digit classification task. Plus, the gradient vanishing issue of Sigmoid occurred during the experiment is analyzed, warning practitioners to be prudent when trying to use Sigmoid in hidden layers, in which case, the fine-tuning of initial variance of weights can be of vital importance.

# A Commandline Parameters for Networks Used in Evaluation

These layer configurations are passed to the training script via `-layer` option. Basically, the linear layer is represented as `linear, layer_name, input_dimension, output_dimension, initialization_stdev`

- AffNet_CE / AffNet_MSE: `linear,fc,784,10,0.01`
- MLP1_ReLU_CE:
  `linear,fc1,784,256,0.01; relu,relu1; linear,fc2,256,10,0.01`
- MLP1_Sigmoid_MSE / MLP1_Sigmoid_CE:
  `linear,fc1,784,256,0.01; sigmoid,sg1; linear,fc2,256,10,0.01`
- MLP1_ReLU_ReLU_MSE:
  `linear,fc1,784,256,0.01; relu,relu1; linear,fc2,256,10,0.01; relu,relu2`
- MLP1_ReLU_Sigmoid_MSE:
  `linear,fc1,784,256,0.01; relu,relu1; linear,fc2,256,10,0.01; sigmoid,sg1`
- MLP1_Sigmoid_Sigmoid_MSE:
  `linear,fc1,784,256,0.01; sigmoid,sg1; linear,fc2,256,10,0.01; sigmoid,sg2`
- MLP1_Sigmoid_MSE / MLP1_Sigmoid_CE:
  `linear,fc1,784,256,0.01; sigmoid,sg1; linear,fc2,256,10,0.01`
- MLP2_ReLU_CE:
  `linear,fc1,784,500,0.01; relu,relu1; linear,fc2,500,256,0.01; relu,relu2; linear,fc3,256,10,0.01`
- MLP2_Sigmoid_MSE / MLP1_Sigmoid_CE:
  `linear,fc1,784,500,0.01; sigmoid,sg1; linear,fc2,500,256,0.01; sigmoid,sg2; linear,fc3,256,10,0.01`
- MLP2_ReLU_ReLU_MSE:
  `linear,fc1,784,500,0.01; relu,relu1; linear,fc2,500,256,0.01; relu,relu2; linear,fc3,256,10,0.01; relu,relu3`
- MLP2_ReLU_Sigmoid_MSE:
  `linear,fc1,784,500,0.01; relu,relu1; linear,fc2,500,256,0.01; relu,relu2; linear,fc3,256,10,0.01; sigmoid, sg1`
- MLP2_Sigmoid_Sigmoid_MSE:
  `linear,fc1,784,500,0.01; sigmoid,sg1; linear,fc2,500,256,0.01; sigmoid,sg2; linear,fc3,256,10,0.01; sigmoid,sg3`
- MLP2_Sigmoid_Sigmoid_MSE_ENH
  `linear,fc1,784,500,1; sigmoid,sg1; linear,fc2,500,256,0.1; sigmoid,sg2; linear,fc3,256,10,0.01; sigmoid,sg3`
- MLP2_Sigmoid_MSE / MLP2_Sigmoid_CE:
  `linear,fc1,784,500,0.01; sigmoid,sg1; linear,fc2,500,256,0.01; sigmoid,sg2; linear,fc3,256,10,0.01`