Multi-class Classification

Wavelix

1 Tasks

1. Construct a Multilayer Perceptron (MLP) model using Numpy.
2. Train the MLP model using the training dataset `optdigits.tra`.
3. Evaluate the model's performance using the testing dataset `optdigits.tes`.
4. Calculate and report the classification accuracy of the model on the test dataset.

2 Model Structure

The MLP model implements MBGD method, and initially will accept parameters including

- `layers`, a list indicates the size of MLP. For example, `[64, 128, 64, 10]` indicates that the MLP has a input layer with 64 neurons, 2 hidden layers with 128 neurons and 64 neurons in each, and a output layer with 10 neurons.
- `activation`, indicates the type of activation function used in the hidden layers, which can be chosen from **ReLU**, **tanh** and **sigmoid**.
- `lr`, the learning rate.
- `epochs`
- `batch_size`

The weights are initalized by randomly Gaussian distribution, and the bias are initalized as zero vectors.

Forward Propagation: In the hidden layers, we compute:

$$z^i = a^{i-1} \cdot W^i + b^i$$

$$a^i = f(z^i)$$

z^i and a^i are saved for backward propagation. For the output layer, we use **softmax** function:

$$\textbf{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Backward Propagation: We apply cross-entropy loss function:

$$\mathcal{L} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_{ij} \log(\hat{y}_{ik})$$

where m is the number of samples, and K is the number of classes. y_{ik} represents the one-of-K encoding for sample i, while $\hat{y}_{ik}$ represents the probability which estimates whether or not sample i belongs to class K.

For the output layer,

$$\delta^L = \hat{y} - y$$

$$\frac{\partial \mathcal{L}}{\partial W^L} = a^{(L-1)^T} \cdot \delta^L$$

$$\frac{\partial \mathcal{L}}{\partial b^L} = \mathbf{sum}(\delta^L)$$

For the hidden layers,

$$\delta^l = (\delta^{l+1} \cdot W^{(l+1)^T}) \cdot f'(z^l)$$
$$\frac{\partial \mathcal{L}}{\partial W^l} = a^{(l-1)^T} \cdot \delta^l$$
$$\frac{\partial \mathcal{L}}{\partial b^l} = \mathbf{sum}(\delta^l)$$

Update the parameters by:

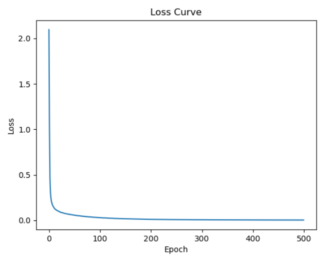
$$W^l \leftarrow W^l - \eta \cdot \frac{\partial \mathcal{L}}{\partial W^l}$$
$$b^l \leftarrow b^l - \eta \cdot \frac{\partial \mathcal{L}}{\partial b^l}$$

where η is the learning rate.

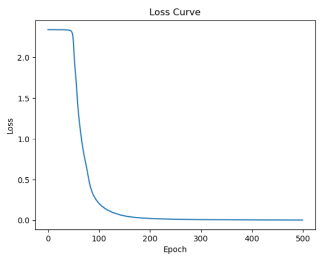
During the training, we apply MBGD method. The samples in training set are initially randomly shuffled, then splited to mini batches. For each batch, we process forward and backward propagation, then update the weights and bias. The avarage loss is computed, stored and printed in each epoch.

As for the prediction, we apply forward propagation on testing set, and chose the class with the largest probability as the result. Accuracy is used to evaluate the model's performance.

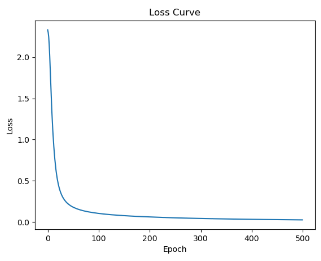
3 Training Process and Test Results



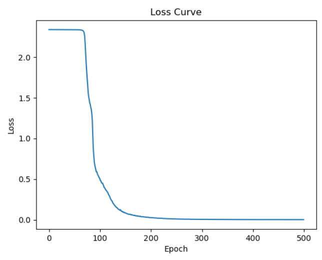
(a) Accuracy=0.9599



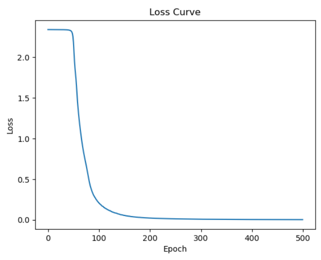
(b) Accuracy=0.9633



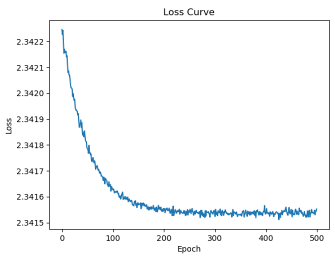
(c) Accuracy=0.9622



(d) Accuracy=0.9466



(e) Accuracy=0.9393



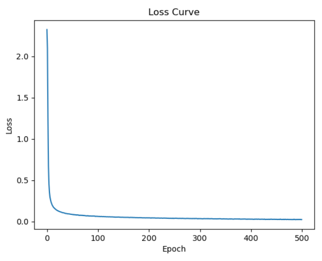
(f) Accuracy=0.1008

Figure 1: Results 1

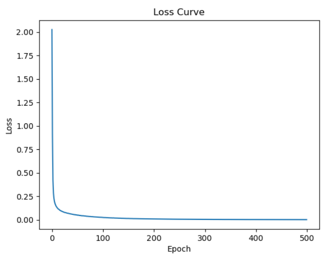
When MLP have shallow layers, models with **ReLU**, **tanh** and **sigmoid** function in hidden layers all have good performance. For example, when we choose the layer size as `[64, 64, 10]`, with learning rate of 0.01 and batch size of 64, Figure 1(a), Figure 1(b) and Figure 1(c) are the results using **ReLU**, **tanh** and **sigmoid**, respectively, with accuracy above 0.95.

When layers go deeper, however, things become different. Models with **ReLU** and **tanh** still have good performance, while **sigmoid** becomes difficult to train and it is hard to find the appropriate hyper-parameters. Figure 1(d) and Figure 1(e) are the results using **ReLU** and **tanh**, respectively, with layer size of `[64, 64, 64, 64, 10]`, learning rate of 0.01 and batch size of 64. Figure 1(f) use **sigmoid** with the same layer size, learning rate of $1e-4$ and batch size of 64.

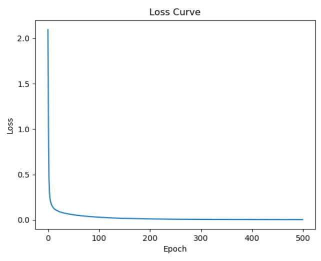
The reason for such phenomenon may comes from the derivatives of the activation functions. The derivative of **sigmoid** function lies in $(0, 0.25)$, meaning that as the network deepens, gradients multiply layer by layer, becoming increasingly smaller. This leads to the vannishing gradient problem. The **tanh**, however, have wider range of derivative. The output of **ReLU** value in $[0, +\infty)$, and its derivative is 1 for positive inputs, which avoids the vanishing gradient problem. Gradients can flow effectively even in deep networks.



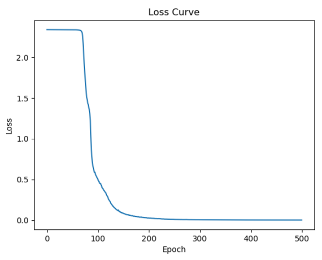
(a) Accuracy=0.9399



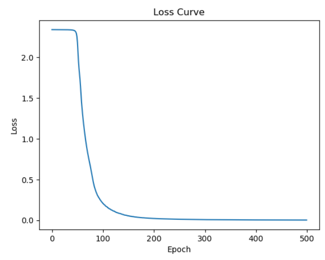
(b) Accuracy=0.9616



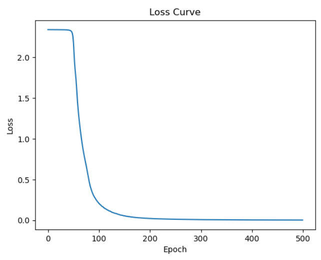
(c) Accuracy=0.9599



(d) Accuracy=0.9466



(e) Accuracy=0.9633



(f) Accuracy=0.9393

Figure 2: Results 2

Next, we try the different layer size with the same activation function. Figure 2(a) and Figure 2(a) are the results of models using **relu**, with layer size of `[64, 8, 10]` and `[64, 128, 10]`, respectively. Figure 2(c) and Figure 2(c) are the results of models using **relu**, with layer size of `[64, 64, 10]` and `[64, 64, 64, 64, 10]`, respectively. Figure 2(e) and Figure 2(f) are the results of models using **tanh**, with layer size of `[64, 64, 10]` and `[64, 64, 64, 64, 10]`, respectively. It reveals that the model with too few or too many parameters may reduce the performance of the model or increase the difficulty of training.

4 Conclusions

1. Different complexity datasets require corresponding size parameters for training.
2. When the network needs to go deeper, as for the activation function in the hidden layers, **ReLU** and **tanh** can perform better than **sigmoid**.