Deep Learning Assignment 1

Oscar Ligthart
10557520
University of Amsterdam
oscarligthart@gmail.com

Collaborated with Jasper Bakker and Aron Hammond.

1 MLP backprop and NumPy implementation

1.1 Analytical derivation of gradients

1.1a

$$\frac{\partial L}{\partial x^(N)} = \begin{bmatrix} -t_1 \frac{1}{x_i} \\ \vdots \\ -t_n \frac{1}{x_n} \end{bmatrix} \tag{1}$$

Use the quotient rule to calculate the derivatives for the softmax function.

$$\frac{\partial x^{(N)}}{\partial \tilde{x}_i^{(N)}} = \frac{\partial}{\partial \tilde{x}_i^{(N)}} \frac{\exp(\tilde{x}^{(N)})}{\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)})} \tag{2}$$

This derivative has two forms, which depend on whether $i = j$ or $i \neq j$. Starting with the case where $i = j$:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}_i^{(N)}} = \frac{\exp(\tilde{x}_j^{(N)}) \sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) - \exp(\tilde{x}_j^{(N)}) \exp(\tilde{x}_j^{(N)})}{\left(\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) \right)^2} \tag{3}$$

$$= \frac{\exp(\tilde{x}_j^{(N)})}{\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}} - \frac{\exp(\tilde{x}_j^{(N)}) \exp(\tilde{x}_j^{(N)})}{\left(\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) \right)^2} \tag{4}$$

$$\tag{5}$$

If we denote $\frac{\exp(\tilde{x}^{(N)}}{\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)})}$ as S, the previous equation will be equal to:

$$\frac{\exp(\tilde{x}_j^{(N)})}{\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)})} - \frac{\exp(\tilde{x}_j^{(N)}) \exp(\tilde{x}_j^{(N)})}{\left(\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) \right)^2} = S_j(1 - S_i) \tag{6}$$

32nd Conference on Neural Information Processing Systems (NIPS 2018), Montréal, Canada.

This will be on the diagonal of the Jacobian, on all other indices we will find $i \neq j$:

$$\frac{\partial x^{(N)}}{\partial \tilde{x}_i^{(N)}} = \frac{0 \cdot \sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) - \exp(\tilde{x}_j^{(N)}) \exp(\tilde{x}_j^{(N)})}{\left(\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) \right)^2} \tag{7}$$

$$= -\frac{\exp(\tilde{x}_j^{(N)}) \exp(\tilde{x}_j^{(N)})}{\left(\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) \right)^2} \tag{8}$$

Once again, we denote $\frac{\exp(\tilde{x}^{(N)}}{\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)})}$ as S:

$$-\frac{\exp(\tilde{x}_i^{(N)}) \exp(\tilde{x}_j^{(N)})}{\left(\sum_{i=1}^{d_N} \exp(\tilde{x}_i^{(N)}) \right)^2} = -S_i S_j \tag{9}$$

The Jacobian will thus have the following form:

$$\begin{bmatrix} S_1(1-S_1) & -S_1 S_2 & \cdots & -S_1 S_N \\ -S_2 S_1 & S_2(1-S_2) & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ -S_N S_1 & \cdots & \cdots & S_N(1-S_N) \end{bmatrix} \tag{10}$$

The derivative of the ReLU module is:

$$f(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases} \tag{11}$$

This can be notated using θ, which is the heavy side function. It returns 1 for positive arguments and 1 for negative arguments. The result looks like this: $\theta(\tilde{x} > 0)$

Which looks like this in matrix form:

$$\frac{\partial x^{(l<n)}}{\partial \tilde{x}} = \begin{bmatrix} \theta(\tilde{x_1} > 0) & 0 & \cdots & 0 \\ 0 & \theta(\tilde{x_2} > 0) & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \theta(\tilde{x_n} > 0) \end{bmatrix} \tag{12}$$

The derivatives of the linear equations can all be derived using matrix derivation rules:

$$\frac{\partial x^{(l)}}{\partial \tilde{x}^{(l-1)}} = W^{T^{(l-1)}} \tag{13}$$

$$\frac{\partial x^{(l)}}{\partial W^{(l)}} = x^{T^{(l)}} \tag{14}$$

$$\frac{\partial x^{(l)}}{\partial b^{(l)}} = \mathbb{1} \tag{15}$$

1.1.1 1.1b

The first part of the backpropagation consists of the matrix multiplication of the derivatives of the loss function and softmax:

$$\frac{\partial L}{\partial x^{(N)}}\frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}} = \begin{bmatrix} -t_1\frac{1}{x_1} \\ \vdots \\ -t_n\frac{1}{x_n} \end{bmatrix} \begin{bmatrix} S_1(1-S_1) & -S_1S_2 & \cdots & -S_1S_N \\ -S_2S_1 & S_2(1-S_2) & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ -S_NS_1 & \cdots & \cdots & S_N(1-S_N) \end{bmatrix} \tag{16}$$

To get the input of a certain layer you have to backtrack through $\frac{\partial L}{\partial \tilde{x}^{(N)}}$ to get the input of the softmax. Afterwards, for every layer take the product of the derivative w.r.t. the linear layer following the derivative w.r.t. the ReLU. This will take on the following form for each layer:

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(N)}} \prod_{i=1}^{l<N} W^{T^{(N-i+1)}} \theta(\tilde{x}^{(N-i)} > 0) \tag{17}$$

In matrix form this looks like this:

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(N)}} \prod_{i=l}^{l<N} W^{T^{(N-i+1)}} \begin{bmatrix} \theta(\tilde{x_1}^{(N-i)} > 0) & 0 & \cdots & 0 \\ 0 & \theta(\tilde{x_2}^{(N-i)} > 0) & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & \cdots & \theta(\tilde{x_n}^{(N-i)} > 0) \end{bmatrix} \tag{18}$$

Now for the linear derivative w.r.t. x. We use the derivative defined in equation 17, which leads to the derivative of the input to a certain layer w.r.t. the loss function.

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}}\frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}} \tag{19}$$

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} W^{T^{(l+1)}} \tag{20}$$

Where $\frac{\partial L}{\partial \tilde{x}^{(l+1)}}$ is the derivation seen in equation 17, up until a certain value of l.

Derivative w.r.t. the weights also uses the derivation seen in equation 17:

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}}\frac{\partial \tilde{x}^{(l)}}{\partial W^{(1)}} \tag{21}$$

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} x^{T^{(l)}} \tag{22}$$

Lastly, the derivative w.r.t. the bias, which also uses the derivation seen in equation 17:

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}}\frac{\partial \tilde{x}^{(l)}}{\partial b^{(1)}} \tag{23}$$

$$\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \mathbb{1} \tag{24}$$

1.1c

The operations within the derivatives and the chained derivatives will not change. However, all derivatives will gain another dimension, meaning vectors will turn into matrices and 2D-matrices will turn into 3D-matrices. The addition of this dimension is caused by the batches. The operations however will still occur on the inner dimensions (and thus parallel so to speak). You can multiply a vector (1xD) by another vector (1xD) for every input in a batch size of N, but you can also multiply two matrices (NxD), to get the same result for the whole batch.

1.2 NumPy implementation

I found out at the last moment that we were supposed to plot the accuracy curve on the test set, but instead I plotted the accuracy curve on the validation and training set for every evaluation. Once my model had finished learning I ran the test set through it once, to obtain the final accuracy of my model on the test set.

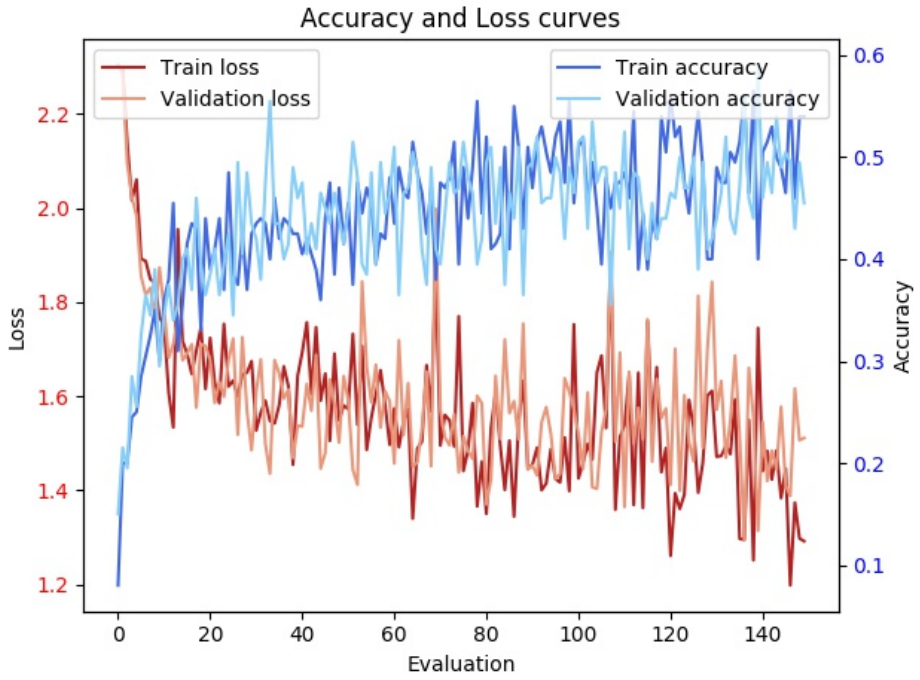

Figure 1: Loss and accuracy values for both the training and validation set on the Numpy implementation of an MLP. Blue lines show the accuracy values for both training and validation, while red lines show loss values for both training and validation. The losses and accuracies were evaluated after every 10 batches consisting of 200 inputs.

When looking at figure 1, it becomes evident that learning has occurred within my implementation of an MLP in numpy. Both training and validation loss seems to drop to a value around 1.4. For the accuracy, both training and validation score around 0.5. Due to the fact that these values appear to be very close to each other for both the training and validation set, I assume that a form of overfitting has not occurred. This is most likely due to the small amount of steps (batches evaluated) within the algorithm. The accuracy on the test set can be seen in table 1. See additional submissions to find the code implementation.

Table 1: Numpy MLP accuracy on the test set.

	Numpy MLP
Accuracy	0.4628

2 PyTorch MLP

With default parameters and the torch random seed set to 42, my PyTorch MLP had a 0.39 accuracy on the test set. The aim was to get 0.53 by changing the hyperparameters.

Table 2: Progress report of hyperparameter and structure tuning of the PyTorch MLP. The grey boxes show changes made, while the green boxes show changes kept (which tended to improve the accuracy). The table does not show all my changes, but the ones that had the most impact in my opinion.

# hidden neurons	Activation func.	Batch norm	Learning rate	Momentum	Batch size	Steps	Test accuracy
100	ReLU	x	2e-3	0	200	1500	0.3916
100	LeakyReLU	x	2e-3	0	200	1500	04044
100, 50	LeakyReLU	x	2e-3	0	200	1500	0.378
100, 50, 25	LeakyReLU	x	2e-3	0	200	1500	0.3906
100, 100	LeakyReLU	x	2e-3	0	200	1500	0.4132
100, 100, 100	LeakyReLU	x	2e-3	0	200	1500	0.406
100, 100	LeakyReLU	✓	2e-3	0	200	1500	0.3287
100, 100	LeakyReLU	✓	2e-3	0.9	200	1500	0.426
100, 100	LeakyReLU	✓	5e-3	0.9	200	1500	0.471
100, 100	LeakyReLU	✓	1e-2	0.9	200	1500	0.4887
100, 100	LeakyReLU	x	1e-2	0.9	200	1500	0.2205
100, 100	LeakyReLU	✓	5e-2	0.9	200	1500	0.4889
100, 100	LeakyReLU	✓	Adaptive	0.9	200	1500	0.4794
100, 100	LeakyReLU	✓	Adaptive	0.9	200	5000	0.5264
100, 100	LeakyReLU	✓	Adaptive	0.9	200	10000	0.5298
100, 100	LeakyReLU	✓	Adaptive	0.9	100	20000	0.5327

2.1 Structure alterations

I first started to use different network structures, by altering the activation functions, adding new layers and adding batch normalization. In order to try to prevent having dead neurons within the network, I altered the ReLU function after each layer into a LeakyReLU function. As can be seen in table 2, this resulted in a slight increase in performance. Afterwards, I started to alter the amount of hidden layers and neurons. After a couple of tests, I concluded that having more than two hidden layers resulted in worse performance. Furthermore, an amount of 100 hidden nodes per layer seemed to perform quite well while having low computational cost. Finally, I added batch normalization in my neural net. To my surprise, this drastically dropped the performance. I hypothesized that this was caused by the relatively small learning rate. As the learning rate increases, so does the impact of the gradient to the weights, thus resulting into bigger changes of the network. These bigger changes will cause the batch mean and variance to further deviate from each other, in which case batch normalization would be more helpful. Therefore, I decided to keep batch normalization while increasing the learning rate. With a big increase of the learning rate (from 2e-3 to 1e-2) I removed it again to check whether it would still diminish performance. It seemed that with an increased learning rate batch normalization became vital to performance.

2.2 Hyperparameter tuning

After having set the structure of my network I decided to alter the hyperparameters given to us. Firstly, I started by adding momentum to the SGD optimizer, which resulted in a big leap in performance. Secondly, I increased the learning rate from a value of 2e-3 all the way up to a value of 5e-2. As can be seen in table 2, improvement caused by increasing the learning rate from 1e-2 to 5e-2 was minimal. Therefore, I decided to implement adaptive learning rates. At first this led to a decrease in performance, but I figured that adding more steps would cause this adaptive learning rate to be better able to find a global optimum. Hence, the amount of steps was increased, leading to a big increase in performance.

2.3 Best configuration

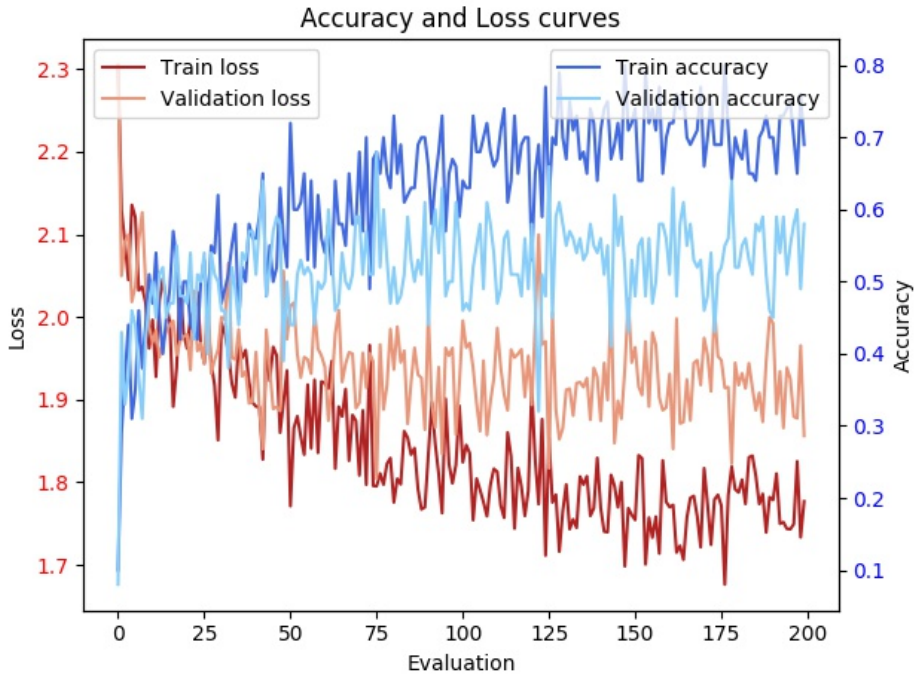

Figure 2: Loss and accuracy values for both the training and validation set on the best configuration of the PyTorch implementation of an MLP. Blue lines show the accuracy values for both training and validation, while red lines show loss values for both training and validation. The losses and accuracies were evaluated after every 10 batches consisting of 200 inputs.

As can be seen in figure 3, the model was successfully able to learn. At the last couple of epochs, loss fluctuated around 1.8 for the training data and around 1.9 for the validation data. From this we can conclude that some form of overfitting has occurred. In the accuracy values this is confirmed, since the training accuracy is significantly higher compared to the validation accuracy. This is most likely caused by the fact that the amount of steps drastically increased within this algorithm (compared to the Numpy implementation). The final test accuracy can be seen in table 3.

Table 3: PyTorch MLP accuracy on the test set.

	PyTorch MLP
Accuracy	0.5327

3 Custom Module: Batch Normalization

3.1 Automatic Differentiation

See code implementations.

3.2 Manual implementation of backward pass

3.2a

Firstly the derivation w.r.t. gamma:

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma_j} \tag{25}$$

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y} \hat{x}_i^s \tag{26}$$

Secondly the derivation w.r.t. beta:

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \frac{\partial L}{\partial y} \frac{\partial y}{\partial \beta_j} \tag{27}$$

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y} 1 \tag{28}$$

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_s \sum_i \frac{\partial L}{\partial y} \tag{29}$$

Finally, the derivation w.r.t. x:

$$\left(\frac{\partial L}{\partial x}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial y_i^s} \frac{\partial y_i^s}{\partial x_j^r} \tag{30}$$

$$\left(\frac{\partial L}{\partial x}\right)_j^r = \sum_s \sum_i \frac{\partial L}{\partial x_j^r} \tag{31}$$

Where:

$$\frac{\partial L}{\partial x_j^r} = \sum_s \sum_i \frac{\partial L}{\partial \hat{x}_j^r} \frac{\partial \hat{x}_j^r}{\partial x_j^r} + \frac{\partial L}{\partial \mu_j} \frac{\partial \mu_j}{\partial x_j^r} + \frac{\partial L}{\partial \sigma_j^2} \frac{\partial \sigma_j^2}{\partial x_j} \tag{32}$$

The derivative of each of these components is as follows for every channel:

$$\frac{\partial \hat{x}_j^r}{\partial x_j^r} = \frac{1}{\sqrt{\sigma_j^2 + \epsilon}} \tag{33}$$

$$\frac{\partial L}{\partial \mu} = \frac{\partial L}{\partial \hat{x}_j^r} \frac{-1}{\sqrt{\sigma^2 + \epsilon}} \tag{34}$$

$$\frac{\partial \mu}{\partial x_j^r} = \frac{1}{B} \tag{35}$$

$$\frac{\partial L}{\partial \sigma^2} = -0.5 \sum_s \frac{\partial L}{\partial \hat{x}_j^r} (x_i^s - \mu)(\sigma^2 + \epsilon)^{-1\frac{1}{2}} \tag{36}$$

$$\frac{\partial \sigma_j^2}{\partial x_j} = \frac{2(x_j - \mu)}{B} \tag{37}$$

4 PyTorch CNN

The CNN was unable to run on my own computer so LISA was used to test the implementation. This script returned a .pickle file holding all the loss and accuracy values for every evaluation. Furthermore, accuracy on the test set was saved.

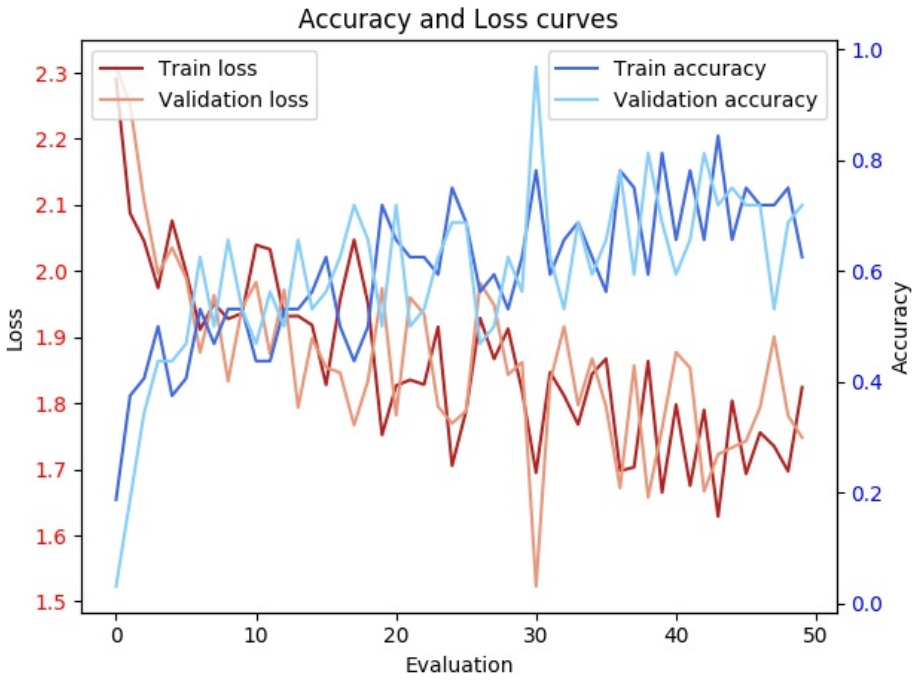

Figure 3: Loss and accuracy values for both the training and validation set on the best configuration of the PyTorch implementation of a CNN. Blue lines show the accuracy values for both training and validation, while red lines show loss values for both training and validation. The losses and accuracies were evaluated after every 100 batches consisting of 32 inputs.

Compared to the previous two classifiers, the CNN seems to perform significantly better compared to the Numpy and PyTorch MLP. Furthermore, training and validation do not seem to differ much from each other, suggesting that overfitting has not occurred. Table 4 shows that this network also performs relatively well compared to the CNN.

Table 4: Accuracy of the CNN on the test set.

	PyTorch CNN
Accuracy	0.714

5 Conclusion

In summary, of all models the CNN performed the best with a test accuracy of 0.714. However, this was also the deepest network of the three. It required a lot more computational power compared to the first two, but the improvement is very significant.

Table 5: Overall comparison of performance.

	Numpy MLP	PyTorch MLP	PyTorch CNN
Accuracy	0.4628	0.5327	0.714