Deep learning assignment 1

Changxin Miao
Changxin.Miao@student.uva.nl

1 MLP backprop and NumPy implementation

1.1 Question 1.1 a)

1.1.1 1

The loss function is defined as $L = -\log(x^{(N)}_{argmax(t)})$, the gradient of cross-entropy could be defined as:

$$\frac{\partial L}{\partial x_i^{(N)}} = \begin{cases} -\frac{1}{x_i^{(N)}}, \text{ if } i = argmax(t) \\ 0, \text{ if } i \neq argmax(t) \end{cases}$$

1.1.2 2

The internal gradient of softmax module is defined as $\frac{\partial x^{(N)}}{\partial \tilde{x}_j^{(N)}}$ where $x^{(N)} = \frac{e^{\tilde{x}^{(N)}}}{\sum_s e^{\tilde{x}_s^{(N)}}}$

$$\frac{\partial x^{(N)}}{\partial \tilde{x}_j^{(N)}} = \frac{e^{\tilde{x}_s^{(N)}}}{(\sum_s e^{\tilde{x}_s^{(N)}})^2}$$

This could lead to two scenarios

$$\frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = \begin{cases} -\frac{e^{\tilde{x}_i^{(N)}} e^{\tilde{x}_j^{(N)}}}{(\sum_j e^{\tilde{x}_j^{(N)}})^2} + \frac{e^{\tilde{x}_i^{(N)}}}{(\sum_i e^{\tilde{x}_i^{(N)}})^2}, \text{ if } i = j \\ -\frac{e^{\tilde{x}_i^{(N)}} e^{\tilde{x}_j^{(N)}}}{(\sum_j e^{\tilde{x}_j^{(N)}})^2}, \text{ if } i \neq j \end{cases} = \begin{cases} -x_i^2 + x_i, \text{ if } i = j \\ -x_i x_j, \text{ if } i \neq j \end{cases}$$

Therefore, $\frac{\partial x^{(N)}}{\partial \tilde{x}_j^{(N)}}$ should be a squared matrix with diagonal elements being $-x_i^2 + x_i$ and other elements being $-x_i x_j$.

$$\begin{bmatrix} -x_1^2 + x_1 & -x_1 x_2 & -x_1 x_3 & \dots & -x_1 x_n \\ -x_2 x_1 & -x_2^2 + x_2 & -x_2 x_3 & \dots & -x_2 x_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -x_n x_1 & -x_n x_2 & -x_n x_3 & \dots & x_n^2 + x_n \end{bmatrix}$$

1.1.3 3

The internal gradient of Relu module is defined as $\frac{\partial x_i^{(l<N)}}{\partial \tilde{x}_j^{(l<N)}}$, it can be calculated as below:

$$\frac{\partial x^{(l<N)}}{\partial \tilde{x}^{(l<N)}} = \begin{cases} 0, \text{ if } i \neq j \\ 0, \text{ if } i = j \text{ and } (x_i^{(l<N)} < 0 \text{ or } x_i^{(l<N)} = 0)) \\ 1, \text{ if } i = j \text{ and } x_i^{(l<N)} > 0 \end{cases}$$

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

1.1.4 4

The gradient of x within linear model is defined as $\frac{\partial x^{(l)}}{\partial x_j^{(l-1)}}$. The linear function is $\tilde{x}^l = W^{(l)}x^{(l-1)} + b^{(l)}$

$$\frac{\partial \tilde{x}_{(l)}}{\partial x_{(l-1)}} = W^{(l)}$$

1.1.5 5

The gradient of w within linear model is defined as $\frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$

$$\frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} = \begin{cases} 0, \text{ if } i \neq j \\ x_k^{(l)}, \text{ if } i = j \end{cases}$$

1.1.6 6

The gradient of b within linear model is defined as $\frac{\partial \tilde{x}^{(l)}}{\partial B^{(l)}}$

$$\frac{\partial \tilde{x}^{(l)}}{\partial B^{(l)}} = I_{dl \times dl}$$

1.2 Question 1.1 b)

1.2.1 1

Backward gradient in softmax layer is $\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \frac{\partial x^{(N)}}{\partial \tilde{x}^{(N)}}$

$$\frac{\partial L}{\partial \tilde{x}^{(N)}} = \frac{\partial L}{\partial x^{(N)}} \odot \begin{bmatrix} -x_1^2 + x_1 & -x_1x_2 & -x_1x_3 & \dots & -x_1x_n \\ -x_2x_1 & -x_2^2 + x_2 & -x_2x_3 & \dots & -x_2x_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -x_nx_1 & -x_nx_2 & -x_nx_3 & \dots & x_n^2 + x_n \end{bmatrix}$$

Where $\odot$ means element-wise multiplication in matrix.

1.2.2 2

Backward gradient for Relu layer is $\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}$

$$\frac{\partial L}{\partial \tilde{x}^{(l<N)}} = \frac{\partial L}{\partial x^{(l)}} \odot \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}}$$

Each element in the gradient matrix is the element-wise multiplication of dout matrix and Relu internal gradient matrix.

1.2.3 3

Backward gradient in Linear layer for x is $\frac{\partial L}{\partial x^{(l<N)}} = \frac{\partial L}{\partial \tilde{x}^{(l+1)}} \frac{\partial \tilde{x}^{(l+1)}}{\partial x^{(l)}}$

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

1.2.4 4

Backward gradient in Linear layer for W is $\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial W^{(l)}}$

$$\begin{aligned} \frac{\partial L}{\partial W^{(l)}} &= \sum_i \frac{\partial L}{\partial \tilde{x_i}^{(l)}} \frac{\partial \tilde{x}_i^{(l)}}{\partial W_{jk}^{(l)}} \\ &= \frac{\partial L}{\partial \tilde{x_i}^{(l)}} x_k^{(l-1)} \end{aligned}$$

1.2.5 5

Backward gradient in Linear layer for b is $\frac{\partial L}{\partial b^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}}$

$$\begin{aligned} \frac{\partial L}{\partial b^{(l)}} &= \frac{\partial L}{\partial \tilde{x}^{(l)}} \frac{\partial \tilde{x}^{(l)}}{\partial b^{(l)}} \\ &= \frac{\partial L}{\partial \tilde{x}^{(l)}} I_{dl \times dl} \end{aligned}$$

1.3 Question 1.1 c)

If a batch size of $B \neq 1$ is used, then the backward propagation gradients in linear models for weight and bias will change. More specifically, the calculate of w and b should take the average of batch size.

1.4 Numpy implementation

For the numpy implementation, I manually implemented the forward and backward propagation for each module. The results could be observed in the figure below. The loss along the training step is around 4 and the accuracy rate achieves 0.47 in the end.

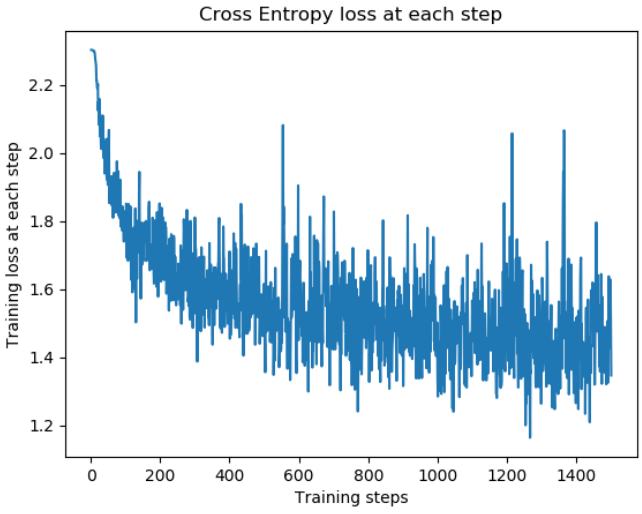

Figure 1: Cross entropy loss numpy implementation

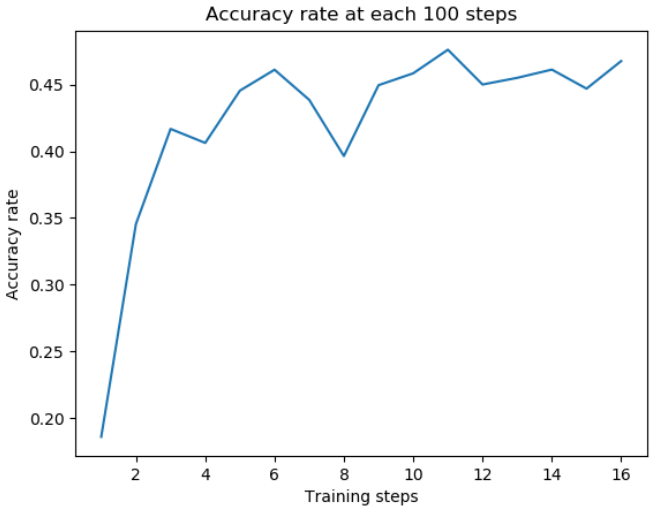

Figure 2: Accuracy rate for numpy implementation

As we could observe from the graph, the cross entropy loss decreases and accuracy rate increases tremendously for the beginning 400 steps. Then the prediction increases in a fluctuating way.

2 PyTorch

The Pytorch implementation is very similar to numpy. The only difference is that as the cross entropy module in pytorch already includes the softmax module, we need to remove the softmax layer. With the default parameters, we could obtain graphs as below.

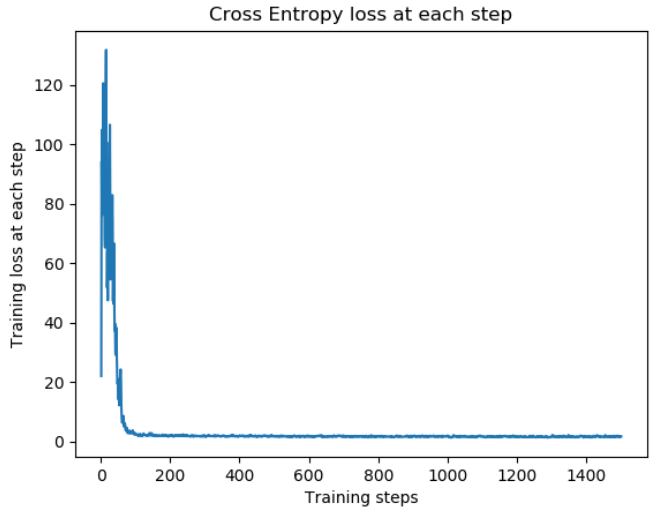

Figure 3: Cross entropy loss Pytorch implementation

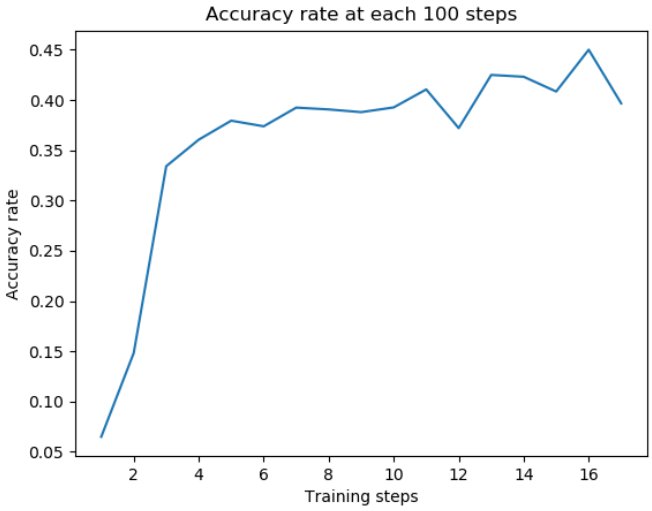

Figure 4: Accuracy rate Pytorch implementation

As we could observe from the above graphs, the scale of loss is quiet high at the beginning, after 150 steps it decreases drastically. With regarding to the accuracy rate, we could observe that the it shows quiet similar pattern to the numpy implementation. In the end, it achieves a accuracy rate of 0.45, which is similar to the result of numpy. In order to achieve a higher level of accuracy rate, I experimented with different parameters. By only changing the number of layers (3 layers) and number of nodes (300) per layer, I achieved an accuracy rate at around 0.49.

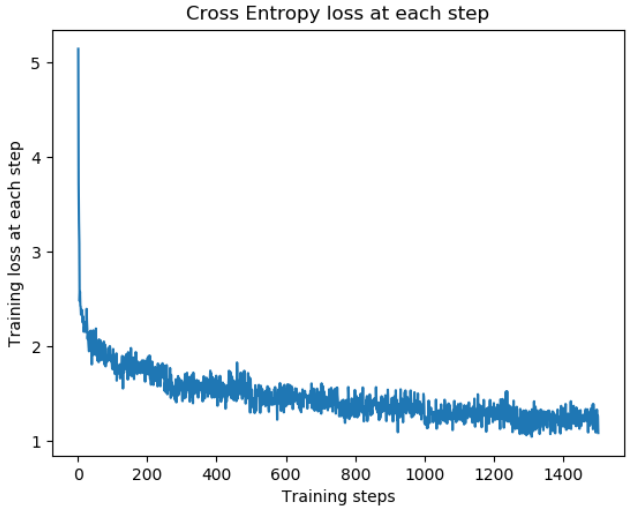

Figure 5: Cross entropy loss (3 layers of 300 nodes)

As it is shown from the graph, the scale of the loss already decreases a lot compared to the default setting. In the end, with 5 layers of 500 nodes, 0.52 accuracy rate is achieved. It further increases with more iterated steps (5000 steps). In the end, I achieved an accuracy rate of 0.54. The training loss and accuracy rate could be visualized as below.

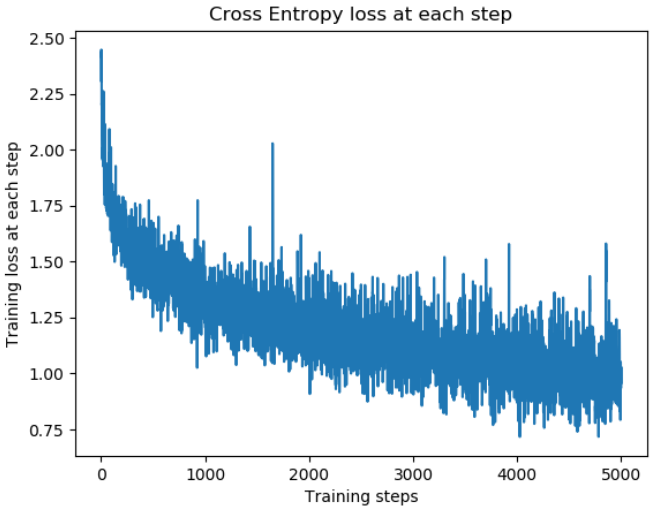

Figure 6: Cross entropy loss (5 layers of 500 nodes)

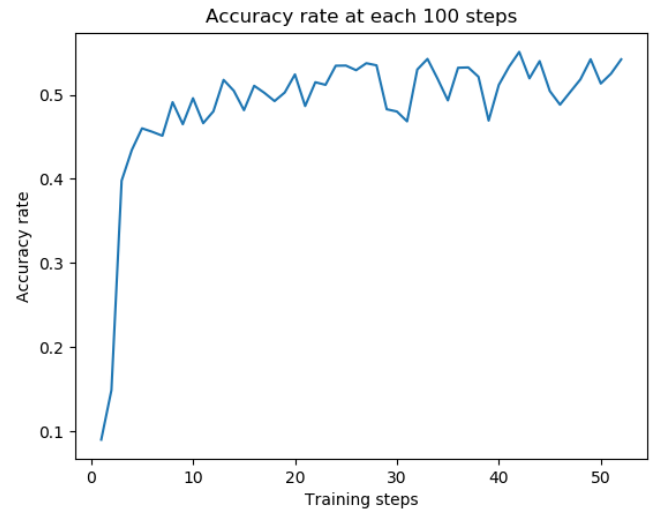

Figure 7: Cross entropy loss (5 layers of 500 nodes)

3 Automatic differentiation

3.1 Question 3.2 a)

We need to calculate the gradient of γ, β and x.

3.1.1 1

$$\frac{\partial L}{\partial \gamma} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \gamma}$$
$$\frac{\partial y_i^{(s)}}{\partial \gamma_j} = \begin{cases} 0, \text{ if } i \neq j \\ \hat{x}_{ij}^{(s)}, \text{ if } i = j \end{cases}$$
$$\frac{\partial L}{\partial \gamma} = \sum_i^B \frac{\partial L}{\partial y_i^{(j)}}\frac{\partial y_i^{(j)}}{\partial \gamma_j}$$
$$= \sum_i^B \frac{\partial L}{\partial y_i^{(j)}}\hat{x}_i^{(j)}$$

3.1.2 2

$$\frac{\partial L}{\partial \beta} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial \beta}$$
$$\frac{\partial y_i^{(j)}}{\partial \beta_j} \begin{cases} 0, \text{ if } i \neq j \\ 1, \text{ if } i = j \end{cases}$$
$$\frac{\partial L}{\partial \beta} = \sum_i^B \frac{\partial L}{\partial y_i^{(j)}}\frac{\partial y_i^{(j)}}{\partial \beta_j}$$
$$= \sum_i^B \frac{\partial L}{\partial y_i^{(s)}}$$

3.1.3 3

The gradient of x could be calculated as $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x}$. After expand the equation into elements, I get the function below:

$$y_i^{(s)} = \gamma_i \hat{x}_i^{(s)} + \beta_i$$
$$= \gamma_i(\sigma_i^2 + \epsilon)^{-\frac{1}{2}}(x_i^{(s)} - \mu_i) + \beta_i$$

With product rule, we could obtain the function below:

$$\frac{\partial y_i^{(s)}}{\partial x_j^{(k)}} = \gamma_i\big[(\sigma_i^2 + \epsilon)^{-\frac{1}{2}}\{\frac{\partial (x_i^{(s)} - \mu_i)}{\partial x_j^{(k)}}\} + (x_i^{(s)} - \mu_i)\{\frac{\partial (\sigma_i^2 + \epsilon)^{-\frac{1}{2}}}{\partial x_j^{(k)}}\}\big] \tag{1}$$
$$\frac{\partial (x_i^{(s)} - \mu_i)}{\partial x_j^{(k)}} = \begin{cases} 1 - \frac{1}{B}, \text{ if } i = j \\ 1, \text{ if } i \neq j \end{cases} \tag{2}$$
$$\frac{\partial (\sigma_i^2 + \epsilon)}{\partial x_j^{(k)}} = \frac{1}{B}\sum_{s=1}^B (2(x_i^{(s)} - u_i))\frac{\partial (x_i^{(s)} - \mu_i)}{\partial x_j^{(k)}} \tag{3}$$
$$= \frac{2}{B}\sum_{s=1}^B (x_i^{(s)} - u_i)\frac{\partial (x_i^{(s)} - \mu_i)}{\partial x_j^{(k)}} \tag{4}$$

Put all terms together, we could get

$$\begin{aligned}
\frac{\partial y_i^{(s)}}{\partial x_j^{(k)}} &= \gamma_i \big[(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\{\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\} - (\sigma_i^2+\epsilon)^{(-\frac{3}{2})}(x_i^{(s)}-\mu_i)\{\frac{2}{B}\sum_{s=1}^{B}(x_i^{(s)}-u_i)\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\}\big] \\
&= \gamma_i(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\big[(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\{\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\} - \frac{(x_i^{(s)}-\mu_i)}{(\sigma_i^2+\epsilon)^{-\frac{1}{2}}}\{\frac{2}{B}\sum_{s=1}^{B}\frac{(x_i^{(s)}-\mu_i)}{(\sigma_i^2+\epsilon)^{-\frac{1}{2}}}\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\}\big] \\
&= \gamma_i(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\big[(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\{\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\} - (\hat{x}_i^{(s)})^2\{\frac{2}{B}\sum_{s=1}^{B}\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\}\big] \\
\frac{\partial L}{\partial y_i^{(s)}}\frac{\partial y_i^{(s)}}{\partial x_i^{(k)}} &= \sum_i^B \gamma_i(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\big[(\sigma_i^2+\epsilon)^{-\frac{1}{2}}\{\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\} - (\hat{x}_i^{(s)})^2\{\frac{2}{B}\sum_{s=1}^{B}\frac{\partial(x_i^{(s)}-\mu_i)}{\partial x_j^{(k)}}\}\big]
\end{aligned}$$

4 Pytorch CNN

The construction of CNN could be easily implemented via Pytorch. After each convolution layer, the batch normalization and ReLu layer is added after that. By changing parameters within the function and adding it to the CNN class, the CNN is constructed. The backward and forward implementation is similar to Pytorch. Due to limited memory and CPU power of the computer, for each evaluation, the first 100 pictures from the test set are taken. After 500 training steps, the model is able to achieve a accuracy rate of at around 75 percent. The training loss and accuracy rate could be easily visualized in graphs below:

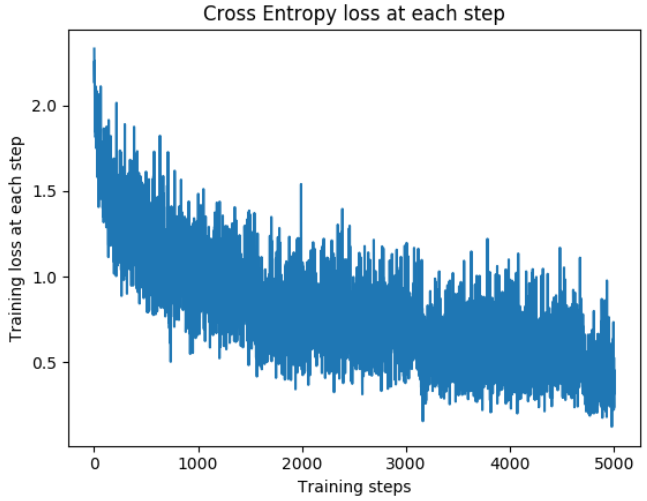

Figure 8: Cross entropy loss cnn implementation

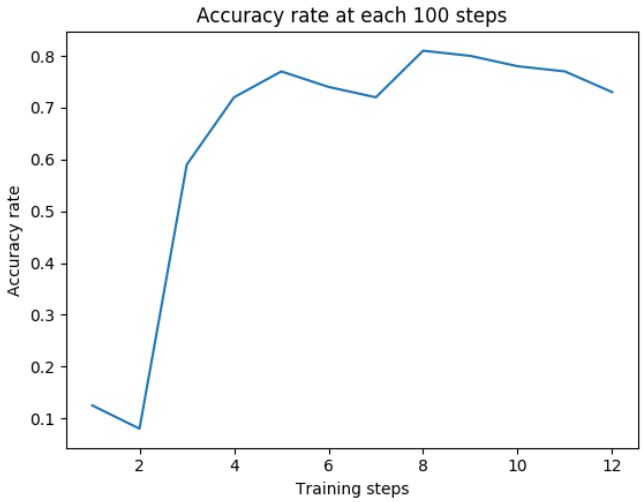

Figure 9: Accuracy rate cnn implementation

As we could observe from the graph, the accuracy rate decreases a little bit after 4000 steps, it might be caused by chance because the cross entropy loss keeps on decreasing. Compared to MLP, CNN is able to achieve higher accuracy rate but it requires more resources to train the model.