2019 Fall 16720B Computer Vision HW5 Report

Eva Mo

November 23, 2019

1 Q1.1

$$softmax(x+c) = \frac{e^{x_1+c}}{\sum_j e^{x_j+c}} = \frac{e^{x_i}e^c}{e^c\sum_j e^{x_j}} = \frac{e^{x_i}}{\sum_j e^{x_j}} = softmax(x)$$

Without the translation c, the value of e^{x_i} could go really high since it is an exponential function. However, if we choose c = -max x_i, the biggest value in numerator becomes e^0, which is 1. All the other numerators would fall in the range between 0 and 1. This way, we keep our calculation in a reasonable range.

2 Q1.2

- the range is $0 < softmax(x) < 1$, and they would sum up to 1.
- probability distrubution.
- – Logits range from negative infinity to positive inginity. Taking exponential of logits turn them into zero or positive.
  – Calculate the sum of all the transformed logits, so we can use it later to normalize each of the transformed logits.
  – Dividing the transformed logit by the sum value just calculated gives us the probability between 0 and 1.

3 Q1.3

Let us consider a feed-forward neural network as depicted in Fig.1. There is no non-linear activation function involved. Derive the output y as the following:

$$
\begin{aligned}
y &= h_2 W_3 + b_3 \\
&= (h_1 W_2 + b_2) W_3 + b_3 \\
&= h_1 W_2 W_3 + b_2 W_3 + b_3 \\
&= (x W_1 + b_1) W_2 W_3 + b_2 W_3 + b_3 \\
&= x W_1 W_2 W_3 + b_1 W_2 W_3 + b_2 W_3 + b_3 \\
&= x W' + b'
\end{aligned}
$$

The output is still a linear combination of the input with bias. Even if we add linear activation function to each layer, it's still the same. Linear combinations of linear functions would always result in linear outputs.

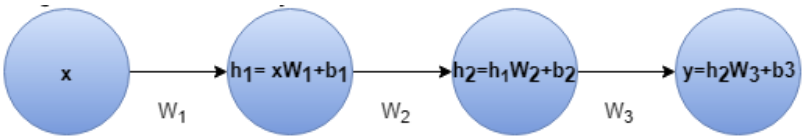


Figure 1: A feed-forward neural network.

4 Q1.4

$$\begin{aligned}
\frac{\partial \sigma(x)}{\partial x} &= \frac{\partial (1+e^{-x})^{-1}}{\partial x} \\
&= (1+e^{-x})^{-2} e^{-x} \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} \frac{e^{-x}}{1+e^{-x}} \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}$$

5 Q1.5

1. $$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} = \delta x^T$$

   δ has the size kx1, while x^T has the size dx1, so we got a gradient matrix with size kxd in the end.

2. $$\frac{\partial J}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial J}{\partial y} = W^T\delta$$

   W^T has the size dxk, while δ has the size kx1, so we got a gradient vector with size dx1 in the end.

3. $$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} = \delta \times 1 = \delta$$

   δ is itself a kx1 vector, which is the same size as what we expect.

6 Q1.6

1. Looking at Fig.4, we can see the sigmoid activation function as well as its first derivative. As can be observed, when the sigmoid function value is either too high or too low, the derivate becomes really small. Also, the maximum of the derivative is around 0.2, with only four layers, the gradient would be as small as 0.0016.

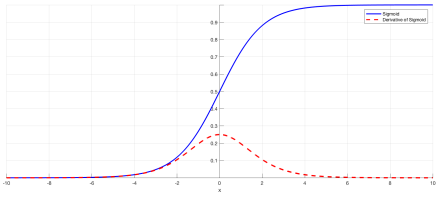


Figure 2: The sigmoid activation function and its derivative.

2. The output range of tanh is $-1 < tanh(x) < 1$, while for sigmoid it is $0 < sigmoid(x) < 1$. Strongly negative inputs to the tanh will map to negative outputs instead of zero for the sigmoid function. Only zero-valued inputs are mapped to near-zero outputs. These properties make the training process less likely to get stuck.

3. Looking at Fig.3, we can see the tanh activation function as well as the its derivative. For input between [-1, 1], we have derivative between [0.41, 1] for tanh, which is much bigger than the derivative of sigmoid. Thus, tanh is preferred for activation function since it can prevent gradients from vanishing better.

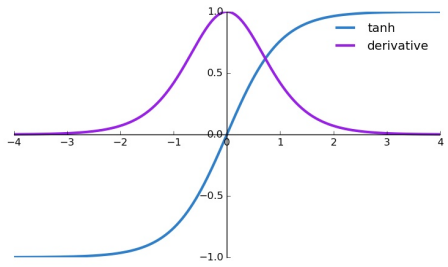


Figure 3: The tanh activation function and its derivative.

4.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}} = \frac{1}{1 + e^{-2x}} - \frac{e^{-2x}}{1 + e^{-2x}} = \sigma(2x) - (1 - \sigma(2x)) = 2\sigma(2x) - 1$$

7 Q2.1.1

Say we have a neural network as shown in Fig.**??**. During forward propagation, each unit in the hidden layer gets the signal: $a_i = \sum_i^N W_{ij} x_i$. No matter what the input is, if all weights are the same, all units in hidden layer will be the same too. Now with weights and bias equal zero, it is even worse. All hidden units get zero signal. In this way, there is symmetry among the weights. All the the neurons of all the layers performs the same calculation, giving the same output and there by making the whole deep net useless.

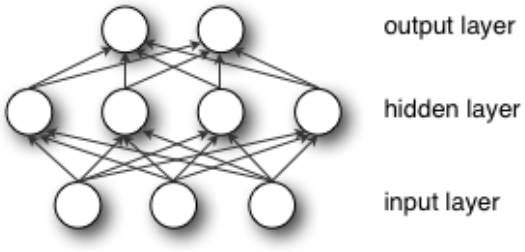


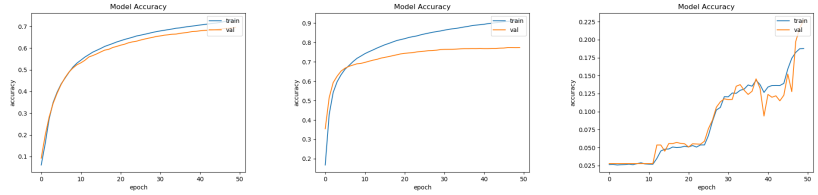
Figure 4: An example of neural network.

8 Q2.1.3

Initializing weights with random numbers helps in breaking symmetry and every neuron is no longer performing the same computation.

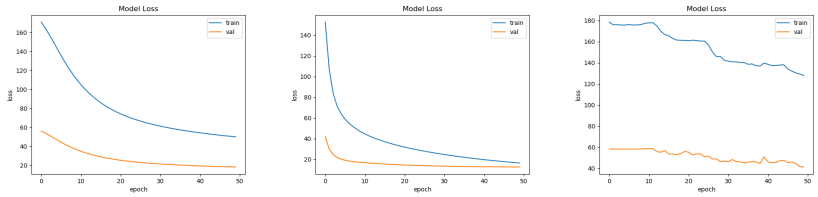
In the same way that the gradients might vanish, if there is a large number of hidden neurons, the weighted sums of them could be really big, causing exploding gradient problem. However, if we scale the initialization by the inverse of layer size, we are able to normalize the activation values for each layer and thus maintain the back-propagated gradients variance within some range. We can then mitigate the chances of exploding or vanishing gradients.

9 Q3.1.2



(a) learning rate = 5e-4. (b) learning rate = 5e-3. (c) learning rate = 5e-2.

Figure 5: Model accuracy with respect to epochs with different learning rates.



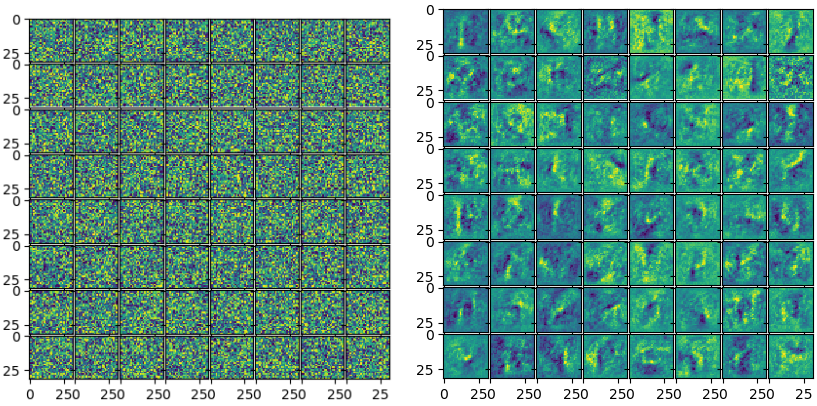
(a) learning rate = 5e-4. (b) learning rate = 5e-3. (c) learning rate = 5e-2.

Figure 6: Model loss with respect to epochs with different learning rates.

Basically, when you have smaller learning rates, the training process goes really slow, since the weights are only updated by a little amount every time. Eventually, it will still converge, but you may not want to spend that much time training. However, if you have a too big learning rate, it might just diverge and explode, or simply keep missing the local minimum because of the big step size. Thus, we want to choose a suitable learning rate to keep our model updating while at the same time not missing the local minimum.

Best network accuracy on test set: 78.17%.

10 Q3.1.3



(a) Original Wlayer1. (b) Learned Wlayer1.

Figure 7: Initial and learned first layer weights.

In Fig.7a, we can see the weights are randomly distributed, and there are no patterns in it. However, looking at Fig.7b, we can see the learned weights represent some sort of patterns for each hidden neuron. For example, the upper-left neuron becomes a filter for a vertical line in the middle (sort of). Actually, those filters all look like some components of a character or a number, which is the reason this classifier works.

11 Q3.1.4

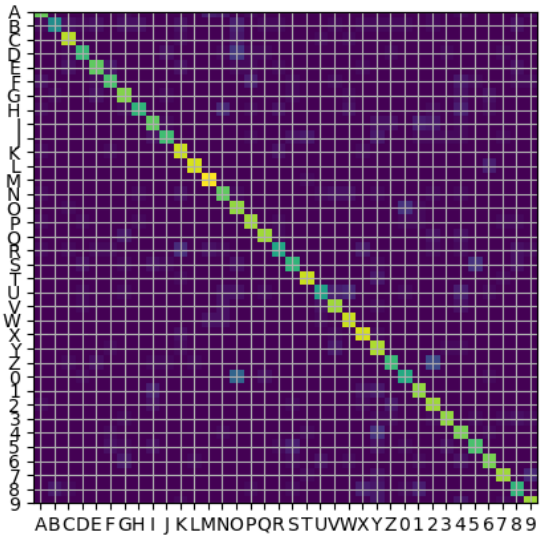


Figure 8: Confusion matrix.

We can see there are some pairs commonly confused, including misclassifying character 'O' as figure '0', character 'Z' as figure '2', figure '5' as character 'S'. They both share a common characteristic, which is they look alike, even we human can sometimes make a mistake.

12 Q4.1

Two assumptions:

1. Each character is connected itself, and any two characters are not connected.

2. Characters have similar sizes.

Fig.9 shows two examples that might fail the detection. In Fig.**??**, the letter 'E' and 'v' are connected, and would be detected as one character. In Fig.**??**, the letter 'v' is too small, and will be filtered out during the area threshold checking.

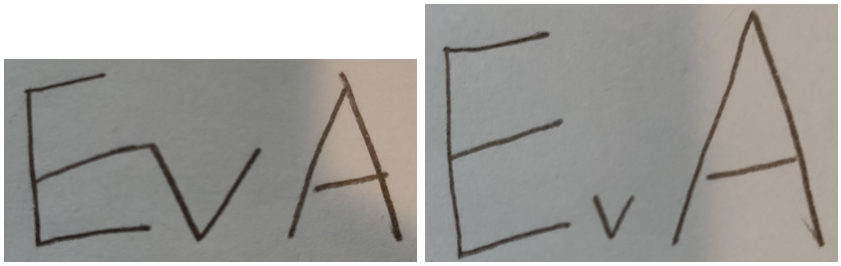
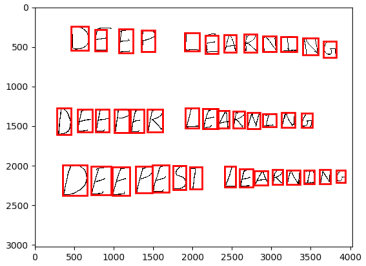
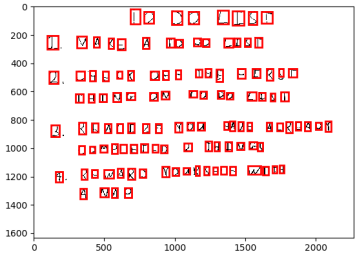


Figure 9: Example images that are expected to fail the character detection.

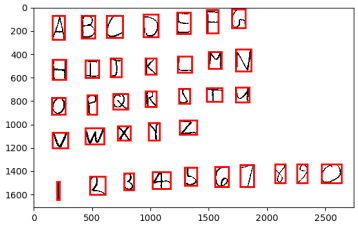
13 Q4.3



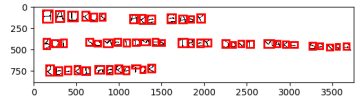
(a) 01_list.jpg.



(b) 02_letters.jpg.



(c) 03_haiku.jpg.

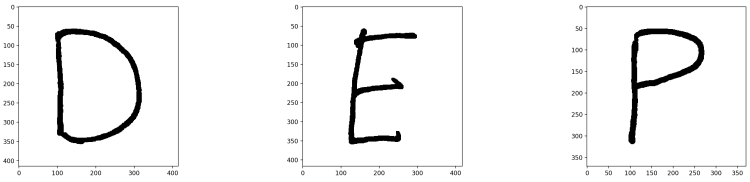


(d) 04_deep.jpg.

Figure 10: Results of findLetters(..) function.

14 Q4.4

Fig.11 shows some deteceted images examples that are postprocessed with crop, padding, erosion, gamma correction, and resizing.



(a) Letter D. (b) Letter E. (c) Letter P.

Figure 11: Examples of cropped images.

Results for the 4 images:

- 01_lists.jpg
    1. Extracted text: "TODOLIST", "IMIXEATDDDLIST","ZCHGCKDFFTHIFIRFT", "THTNGQNTODOLIST", "3RIALIZEY0UHBVERLR6ADT", "C0MPLBTED2THINGS", "4RFWARDYOURSELFWITH", "ANAP"
    2. Accuracy: 80%
- 02_letters.jpg
    1. Extracted text: "IBCDEFG", "HIJKLMN", "OPQKSTU", "VWXYZ", "TX34567XY0"
    2. Accuracy: 80.56%
- 03_haiku.jpg
    1. Extracted text: "HAIKUSAREEASY", "BUTSQMETIMESTREY-DDWTMAKQSZNGG", "REFRIGERAMOR"
    2. Accuracy: 83.33%
- 04_deep.jpg
    1. Extracted text: "JVTPLVJKUIN6", "C55YTTLEAKYING", "VT5TTSTL5ARNING"
    2. Accuracy: 46.34%

15 Q5.2

Fig.12 shows the training loss curve for 100 epochs with momentum update instead of vanilla update. With momentum update, the model is able to reduce its loss significantly right at the beginning of the training process, which is obvious by looking at the steep slope at the start of the curve.

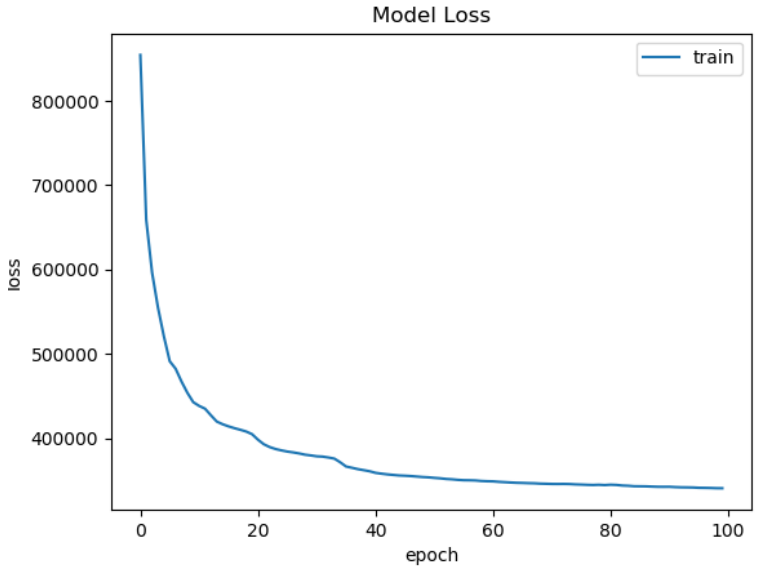


Figure 12: Training loss curve with momentum update.

16 Q5.3.1

Five selected classes: 2, S, L, 7, W, and results are shown respectively from left to right in Fig.13. The reconstructed images are pretty noisy in the background, although, you can still tell which character or figure it is. The image is also blurred.

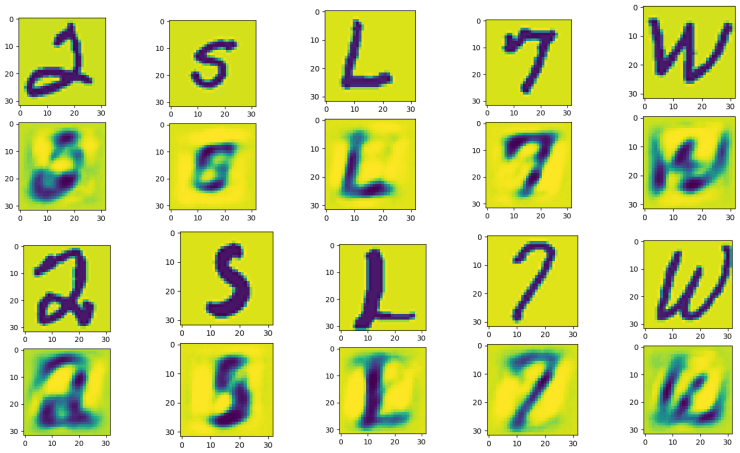


Figure 13: Original and reconstructed validation images for 5 classes.

17 Q5.3.2

Average PSNR across all validation images = 15.70

18 Q6.1

Size of projection matrix: 32 X 1024
Rank: 32
Some examples of origianl and reconstructed test images shown in Fig.14.

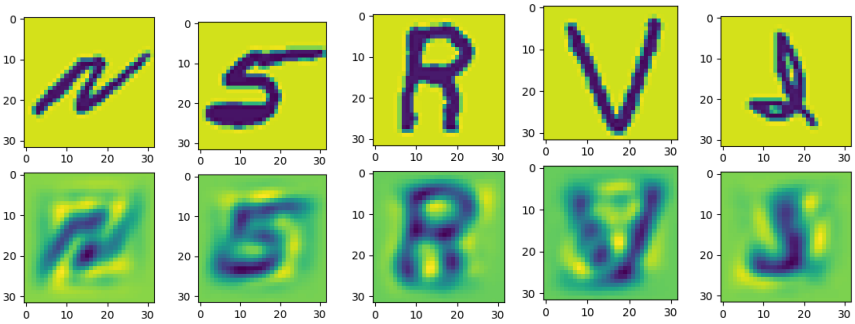


Figure 14: Examples of original and reconstructed images using PCA.

19 Q6.2

Five selected classes: 2, S, L, 7, W, and results are shown respectively from left to right in Fig.15. The reconstructed images are even much noisier than the results we got from autoencoder. Lots of noise is in the background, and sometimes you can not tell which letter it is supposed to be.

Average PSNR on validation set: 16.35.

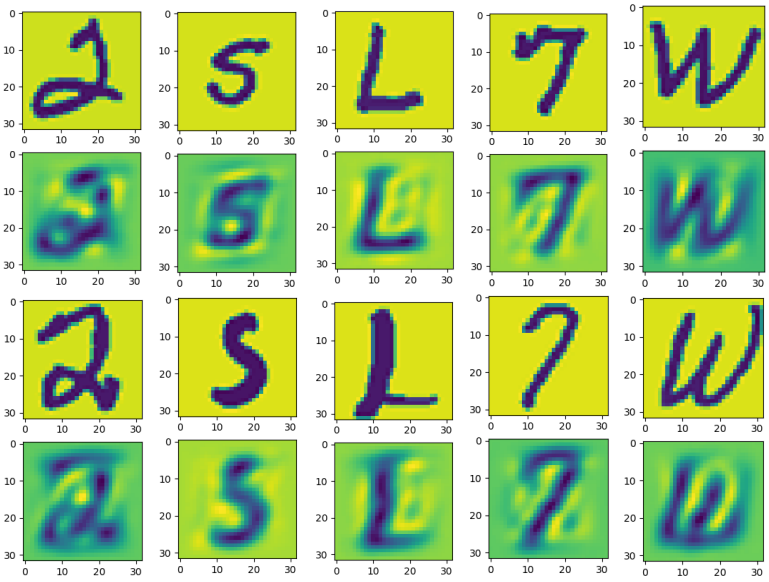
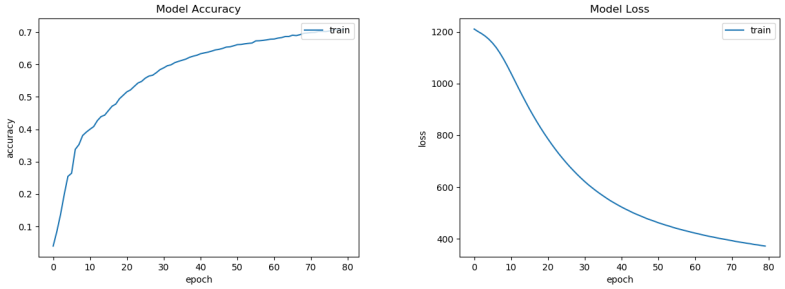


Figure 15: Original and reconstructed validation images for 5 classes using PCA.

20 Q7.1.1

After training for 80 epochs, training accuracy reached 71%.

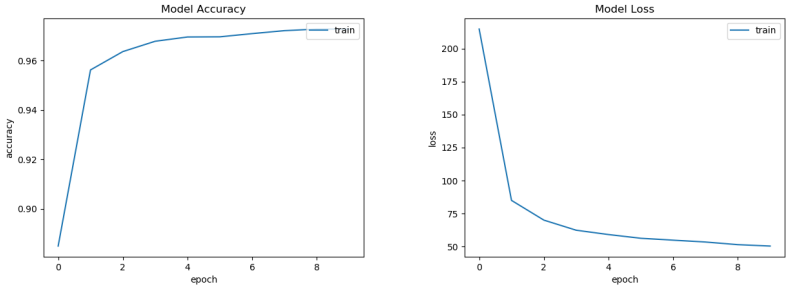


(a) Training accuracy. (b) Training loss.

Figure 16: Training accuracy & loss using PyTorch on NIST36.

21 Q7.1.2

After training for 10 epochs, training accuracy reached 97%.



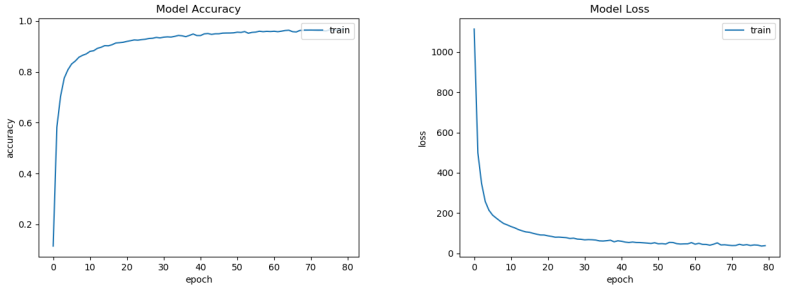
(a) Training accuracy.

(b) Training loss.

Figure 17: Training accuracy & loss using PyTorch on MNIST.

22 Q7.1.3

After training for 80 epochs, training accuracy reached 97%.

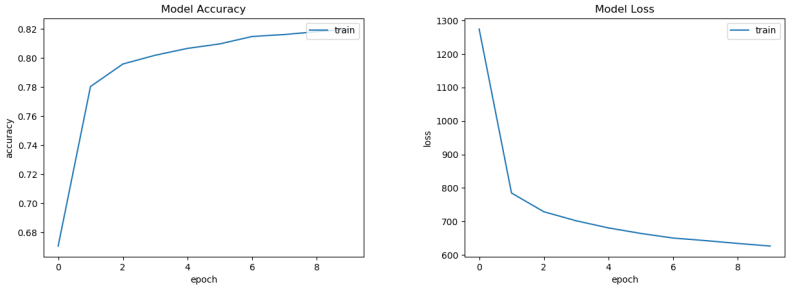


(a) Training accuracy. (b) Training loss.

Figure 18: Training accuracy & loss using PyTorch on NIST36 with CNN.

23 Q7.1.4

After training for 10 epochs, training accuracy reached 82%.



(a) Training accuracy. (b) Training loss.

Figure 19: Training accuracy & loss using PyTorch on balanced EMNIST with CNN.

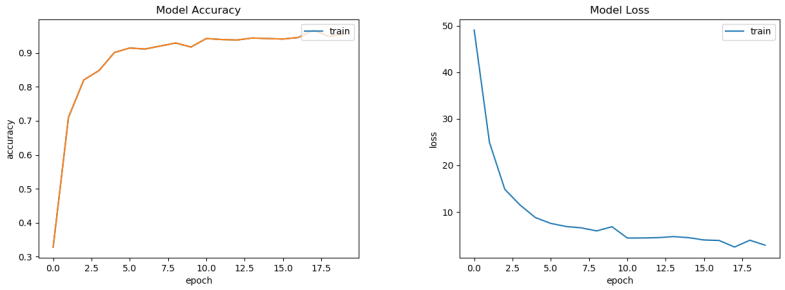
Evaluation on findLetters bounded boxes:

- 01_lists.jpg
  1. Extracted text: "T0D0LIST", "IMAKEAT0D0LIST", "2CHECK0FFTHfFIRfT", "THDNG0Nr0D0LtST", "3RFhLIZEY0UHAVEhLREADI", "C0MPLET8D2THINGS", "tRFWARDY0URSELFWIIH", "ANAP"
  2. Accuracy: 78.26%
- 02_letters.jpg
  1. Extracted text: "ABCDEFG", "HIJKLMN", "OPQKSTV", "VWXYZ", "I234S67890"
  2. Accuracy: 88.89%
- 03_haiku.jpg
  1. Extracted text: "HAIKUSAREEASV", "BUTSOMETIMESTHEV-DONTMAKES8NSF", "REFRIGERAMOR"
  2. Accuracy: 88.89%
- 04_deep.jpg
  1. Extracted text: "OLEPLEARNING", "DEEPERLEARNVING", "DEEP-ISILEARNING"
  2. Accuracy: 87.80%

The result for 04_deep.jpg improved a lot compared to the network trained in Q3.1.1.

24 Q7.2.1

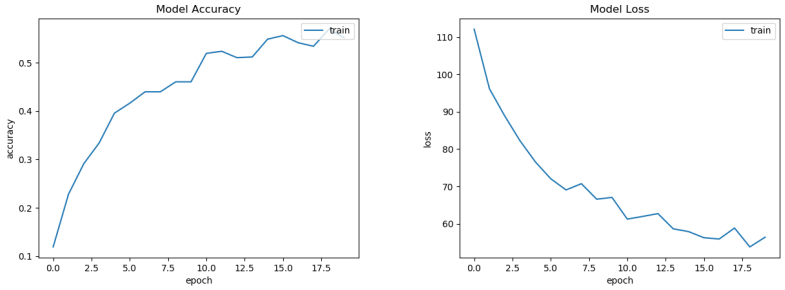
Squeezenet: After fine-tuning classifier for 10 epochs, and training the whole network with small learning rate for another 10 epochs, training accuracy reached 96%, and training loss reduced to 2.87.



(a) Training accuracy. (b) Training loss.

Figure 20: Training accuracy & loss using PyTorch using pretrained Squeezenet.

Custom: After training from scratch for 20 epochs, raining accuracy reached 55%, and training loss reduced to 56.38.



(a) Training accuracy. (b) Training loss.

Figure 21: Training accuracy & loss using PyTorch using custom network.

It's pretty obvious that pretrained network works much better than custom defined network. During the 4th epoch, the pretrained network is able to reach training accuracy over 90%, while after 20 epochs of training, the custom network can't reach over 60% accuracy. Thus, the comparison explains the fact that most images share common properties and so share common weights.