16720-B HW3

Jacob Miller

October 2020

Study Group I formed a study group with Ben Kolligs, Dan McGann, Tom Xu at 2 pm on Sunday.

1.1 Softmax is defined as

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \text{ for index i in vector x}$$

Softmax is invariant to translation, which is shown by

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

Setting $c = -max(x_i)$ would be advantageous because it would drastically reduce the range of values being computed. With $c = -max(x_i)$, the maximum possible value of e^{x_i} for any i would be 1.

1.2 Since, $s_i = e^{x_i}$, $s_i \in (0,1)$. The sum over all of the elements is equal to 1 (i.e. $\sum_i s_i = 1$). It can be said that softmax takes an arbitrary real valued vector x and turns it into a vector of probabilities. The first step of softmax turns x_i into a positive real number s_i. The second step calculates the sum over the entire probability distribution. The third step computes each individual probability.

1.3 Linear regression is a machine learning method that can fit linear equations based on training data. Without a non-linear activation function, the output of layer i and input to node k in layer i+1 in a neural network can be defined as $y_{L_{ik}} = b_k + \sum_j x_j w_{ij}$, which is a simple linear equation. If this output is then passed through another layer, the output of layer i+1 can be defined as $O = b + \sum_k y_k w_{i+1,k} = b + \sum_k (b_k + w_{i+1,k} \sum_j x_j w_{ij})$, which is still a linear equation.

Linearity is defined by the property that $f(\alpha_1 * x_1 + \alpha_2 * x_2) = \alpha_1 * f(x_1) + \alpha_2 * f(x_2)$. Since there is not a non-linear activation function, the property of linearity holds. Therefore, a multi-layer neural network without non-linear activation functions is equivalent to fitting a linear equation to data, AKA linear regression.

1.4 The derivative of the sigmoid function is derived as follows:

$$\begin{aligned}\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{(1+e^{-x})*0 - 1*(-e^{-x})}{(1+e^{-x})^2} \\ &= \frac{e^{-x}}{(1+e^{-x})(1+e^{-x})} = \frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}} \\ &= \sigma(x)\frac{e^{-x}+1-1}{1+e^{-x}} = \sigma(x)(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}) \\ &= \sigma(x)(1-\sigma(x))\end{aligned}$$

1.5 Given $y = x^T W + b$ and $\frac{\partial J}{\partial y} = \delta$, $\frac{\partial J}{\partial W}$ can be calculated as follows:

$$\begin{aligned}\frac{\partial J}{\partial W} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial W} = \frac{\partial y}{\partial W}\frac{\partial J}{\partial y} \\ \frac{\partial y}{\partial W} &= \frac{\partial}{\partial W}(x^T W + b) = \frac{\partial}{\partial W}(x^T W)\end{aligned}$$

Let $z_k = \sum_{l=1}^{m} x_l W_{kl}$, then $\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{m} x_l \frac{\partial W_{kl}}{\partial W_{ij}}$. $\frac{\partial z_k}{\partial W_{ij}}$ equals 1 if i=k and 0 otherwise. Therefore, $\frac{\partial y}{\partial W} = x^T$. Substituting into the original equation gives $\frac{\partial J}{\partial W} = x\delta^T$.

$\frac{\partial J}{\partial x}$ can be calculated as follows:

$$\begin{aligned}\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial x} = \frac{\partial y}{\partial x}\frac{\partial J}{\partial y} \\ \frac{\partial y}{\partial x} &= \frac{\partial}{\partial x}(x^T W + b) = W^T \\ \frac{\partial J}{\partial x} &= W\delta\end{aligned}$$

$\frac{\partial J}{\partial b}$ can be calculated as follows:

$$\begin{aligned}\frac{\partial J}{\partial b} &= \frac{\partial J}{\partial y}\frac{\partial y}{\partial b} \\ &= \frac{\partial J}{\partial y}\frac{\partial}{\partial b}(x^T W + b) = \delta\end{aligned}$$

1.6 Given the definition of cross-correlation, we can write that

$$y_{c,i,j} = \sum_k \sum_l W_{d,c,k,l} x_{c,i+k-1,j+l-1} + b_c$$

This equation can then be used to calculate $\frac{\partial J}{\partial W}$, $\frac{\partial J}{\partial b}$, and $\frac{\partial J}{\partial x}$.

$$\begin{aligned}\frac{\partial J}{\partial b_c} &= \sum_i \sum_j \frac{\partial J}{\partial y_{c,i,j}} \\ \frac{\partial J}{\partial W_{d,c,k,l}} &= \frac{\partial J}{\partial y_{c,i,j}}\frac{\partial y_{c,i,j}}{\partial W_{d,c,k,l}} = \frac{\partial J}{\partial y_{c,i,j}} x_{c,i+k-1,j+l-1} \\ \frac{\partial J}{\partial x_{c,i,j}} &= \frac{\partial J}{\partial y_{c,i,j}}\frac{\partial y}{\partial x_{c,i,j}} = \frac{\partial J}{\partial y_{c,i,j}} \sum_d \sum_i \sum_j W_{d,c,m-i+1,n-j+1}\end{aligned}$$

1.7

1. The gradient of the sigmoid function can lead to the "vanishing gradient" problem in deep networks because the gradient is much smaller than the function itself as seen in 1. As this propagates through a deep network, it will continue to shrink the gradients until the gradients are essentially 0.

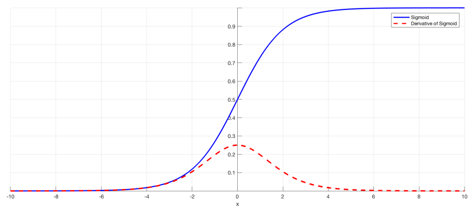

Figure 1: A graph of the sigmoid function and its gradient (https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e)

2. The sigmoid function can range between 0 and 1, while tanh can range between -1 and 1. Tanh might be preferred because it has more expressive power and the derivative is larger than that of sigmoid.

3. The derivative of the tanh function can vary between 0 and 1, which is much larger than that of the sigmoid function. Because of this, there is less of a vanishing gradient problem.

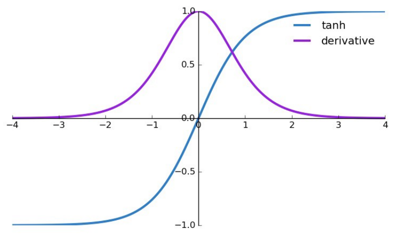

Figure 2: A graph of the tanh function and its gradient (https://medium.com/@omkar.nallagoni/activation-functions-with-derivative-and-python-code-sigmoid-vs-tanh-vs-relu-44d23915c1f4)

4. Tanh can be written as a scaled and shifted version of a sigmoid as follows:

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

2.1.1 If the weights and biases are initialized as all zeros, the network will not be able to properly "learn". The weights will change together because they were all initialized with the same value and have the same gradient, which severely limits the ability of backpropagation to find the minimal cost.

2.1.3 Initializing weights with random numbers helps prevent the problem of weights in the same layer being initialized to the same value, which hampers backpropagration from learning each weight independently. Scaling the initialization by the layer size helps prevent the problem of having weights that are too small or too large. If the weights are too small or too large, gradient descent will struggle to converge.

3.1.2 The best validation accuracy was 75.61% with a learning rate of 0.01. Figure 3 shows graphs of the training loss and accuracy over the training epochs for each of the three learning rates. The graph with the best learning rate shows a quick but steady drop in the loss over the epochs and a sharp, then gradual rise in accuracy. The graph with the lower learning rate is similar to that of the best learning rate, but the drop of the loss and the rise of the accuracy is more gradual. The graph with the higher learning rate did not see a decrease in loss or an increase in accuracy. Instead of finding a local minima, gradient descent took too large of steps and oscillated around the minima of the loss function.

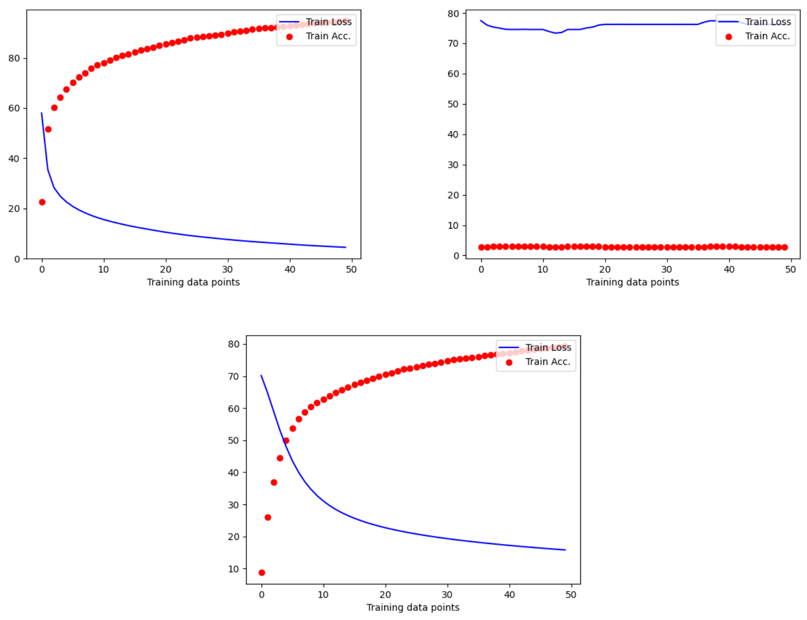

Figure 3: The training loss and accuracy of each epoch for a learning rate of 0.01 (upper left), 0.1 (upper right), and 0.001 (bottom).

3.1.3 Figure 4 and 5 shows the randomly initialized and learned weights for the first layer. Some of the weights look like vertical, horizontal, or diagonal edge filters because it appears dark on one side and light on the other. Other look for like the DoG filter with a dark middle surrounded by a lighter circle or vice versa.

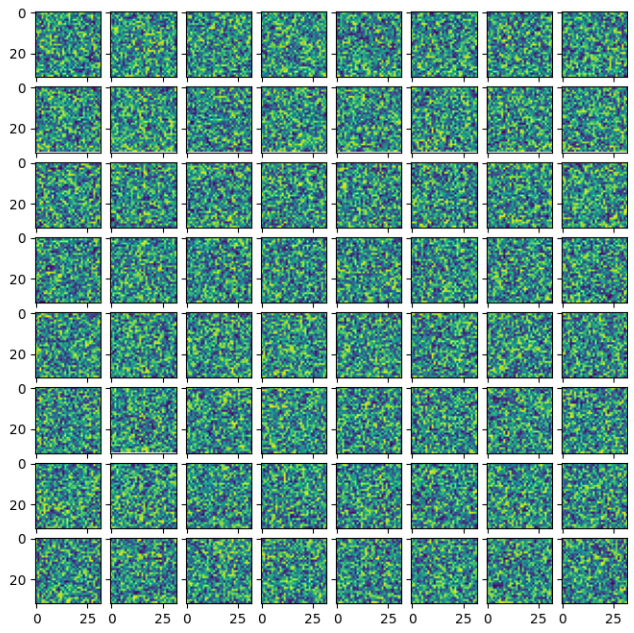

Figure 4: A visualization of the first layer weights before training.

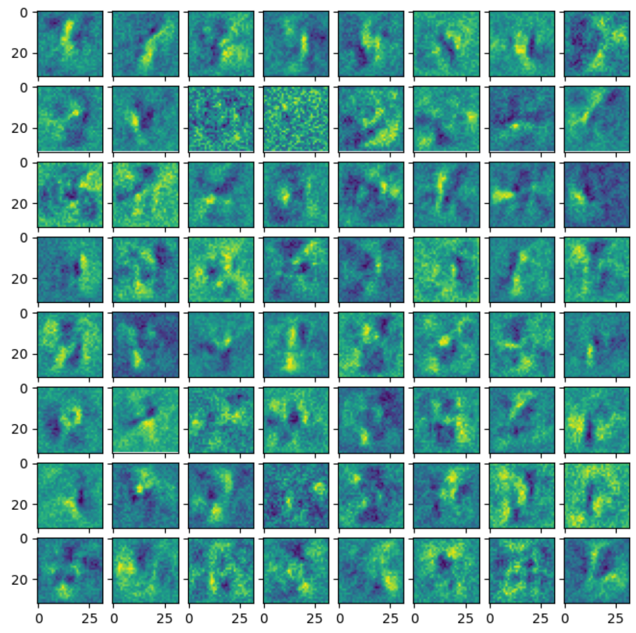

Figure 5: A visualization of the first layer weights after training.

3.1.4 Figure 6 shows visualizations of the second layer weights and the associated image from the validation set. These weights look more refined than the first layer weights and some appear to resemble the shapes of various letters and numbers.

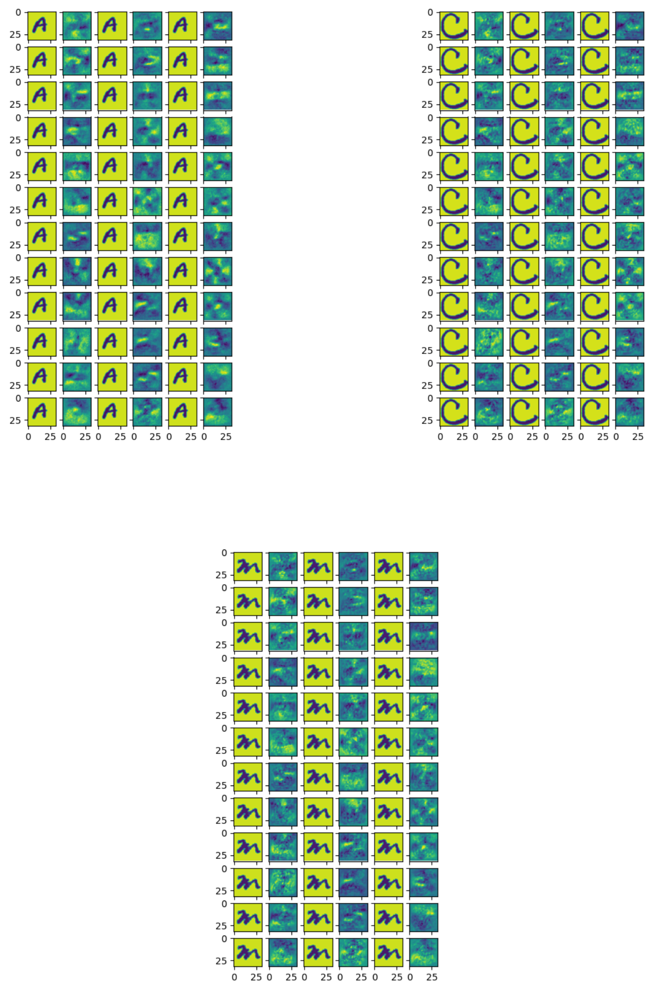

Figure 6: Caption

3.1.5 Figure 7 shows the confusion matrix for the best model. The characters that get most confused are O and 0, S and 5, and Z and 2. These characters have similar shapes and would be difficult to distinguish with different handwriting.

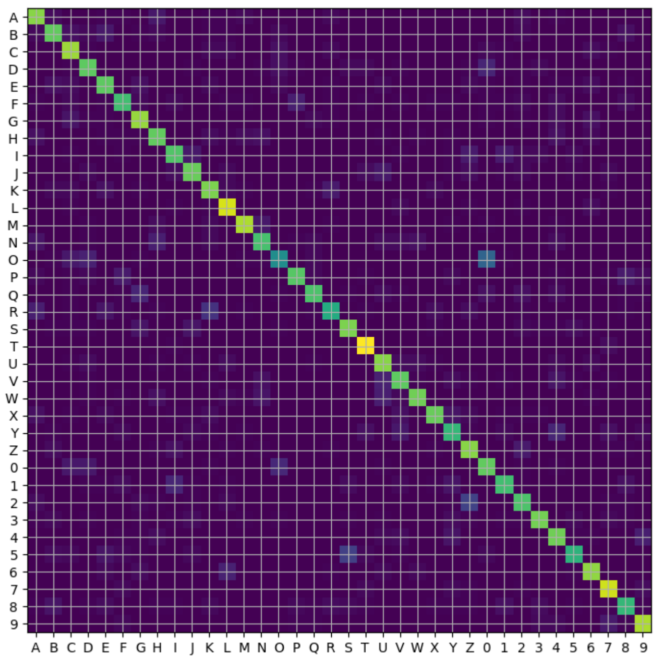

Figure 7: The confusion matrix for the best model on the NIST36 dataset.

4.1 Two big assumptions made by the sample method are

1. The method assumes that each part of the character is connected. It will fail if a letter is made up of two separated parts (see Figure 8).

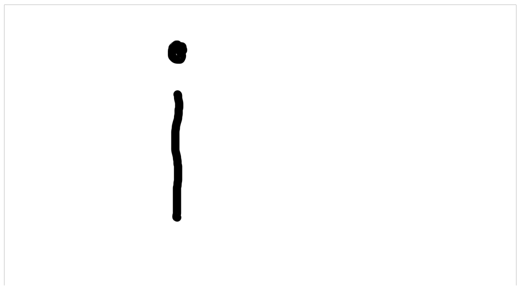

Figure 8: An example handwritten letter that the sample method would fail to recognize.

2. The method assumes that characters are separated with sufficient space. It will fail when letters are connected (see Figure 9).

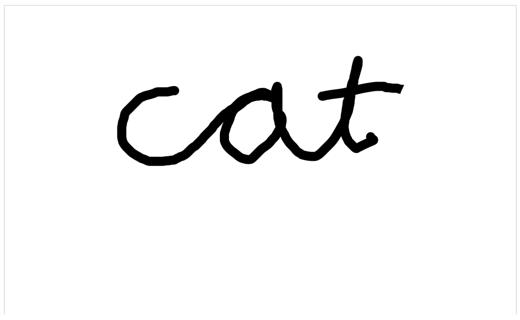

Figure 9: An example handwritten word that the sample method would fail to recognize as separate letters.

4.3 Figure 10 shows the test images with bounding boxes drawn around the letters. The findLetters algorithm finds almost every letter (only misses one letter in the image shown in the upper-right) and has only a few false positives.

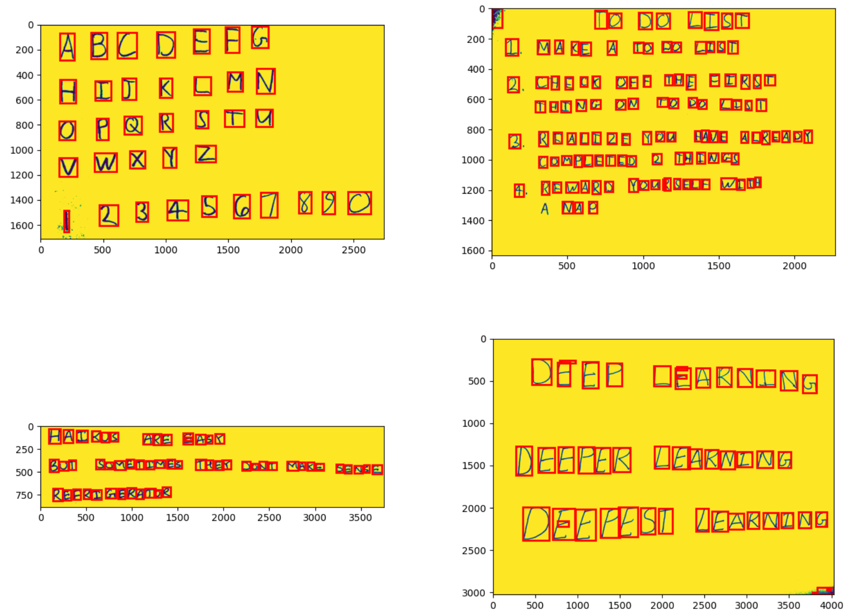

Figure 10: Bounding boxes drawn on the extracted letters in the test images.

4.4 The text extracted from each image was as follows:

1. ABCDFFG HIIKLMN QPQRSTW VWXYZ Z3GS67IYJ
2. F T0 DJ LIST I NDK6 A TD DQ LIST 2 CHLCX QFF THI FIRST THING QN TQ DQ LIST 3 CROIMAPLLIFTZLED YZ0U THHIANUEGS2LR6ADT 9 8FWARD YOURSELF WITH NAP
3. HAIKGS ARG GMASY BWT SQMETIMES TREY DDWT MAK2 SGNGE RBFRIGERATOR
4. CCFP CCARMIXG DFFFIK LCAKNING CPFCCISC LFAKNINGC

The text extraction algorithm works fairly well on the first image, finding all of the letters and classifying the majority of them correctly. You can make out words or partial words from the other images, but not complete sentences. On image 2, the algorithm only found 7 lines and in image 4 the algorithm found 4 lines; this will cause a scrambling of letters. There are also numerous misclassified letters.

7.1.1 See Figure 11.

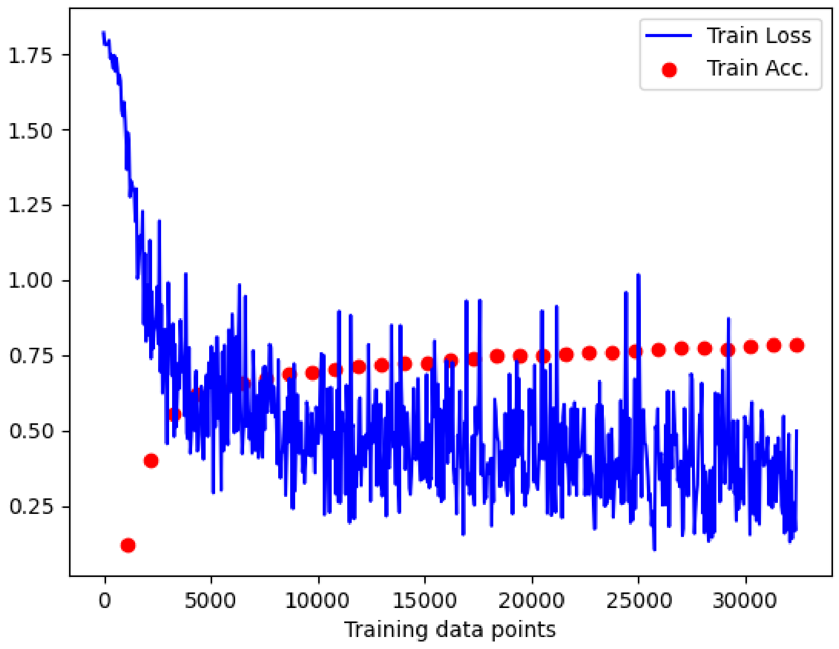

Figure 11: The training loss and accuracy for a FCN on NIST36.

7.1.2 See Figure 12.

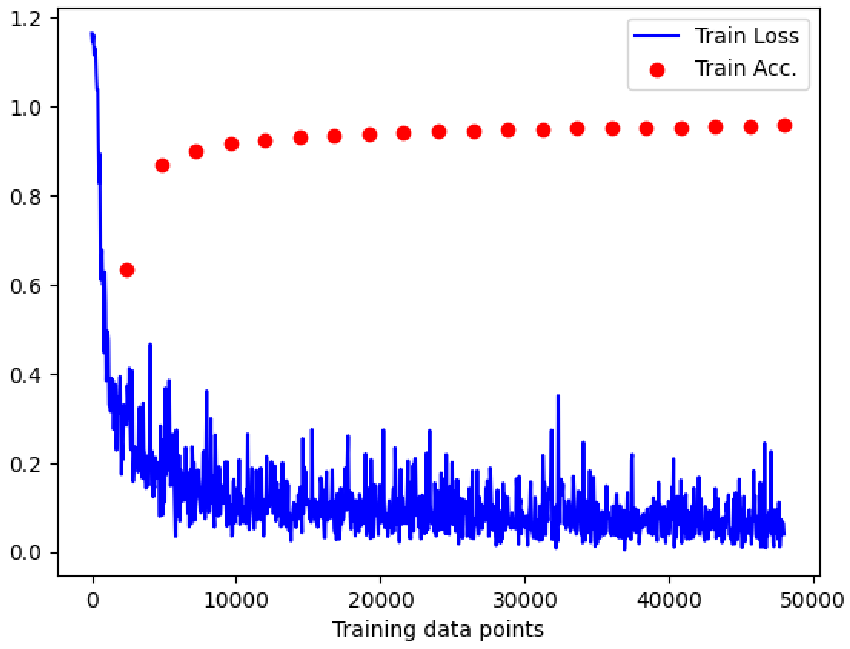

Figure 12: The training loss and accuracy for a CNN on MNIST.

7.1.3 See Figure 13.

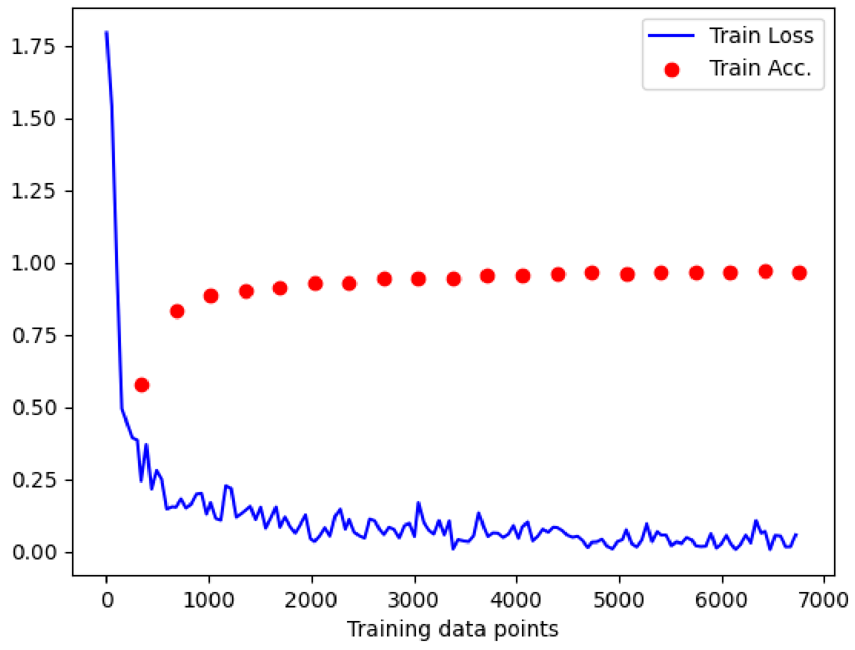

Figure 13: The training loss and accuracy for a CNN on NIST36.

7.1.4 See Figure 14

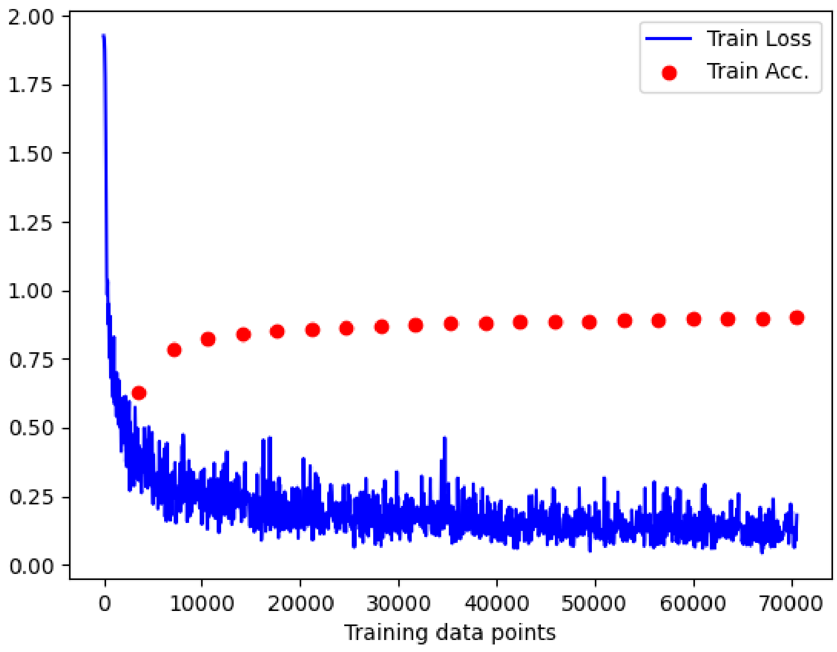

Figure 14: The training loss and accuracy for a CNN on EMNIST.

The text extracted from each image was as follows:

1. BQMMKBqQnnMbMWQHQKYMgMMMNRUBYgBMBQB
2. JnQDQQMBnnMMBBMn0BMQNBNM0MBWM0YKnHBBRBBMNMnMB0MGQMMUBMn BMMQBWQMMMNBMnMRMWnQMnMHMMMMBBBMKMBQMMG-BBMMMWNMMMMBKRBMNMWMQ
3. GQNQ0BQRBGZQGKGOUBBQYMUQQfnHZhdQYnKHQgGgNBgQRBBnGBRQnMg
4. MGQQQQBRNnBBMGBBQQKQBRMMMBQgMgMMgnMBBMQnMBa

7.2.1 The model that used the pretrained SqueezeNet features with fine tuning achieved a training accuracy of 51.5%. The custom CNN model saw similar performance and achieved 50% accuracy.

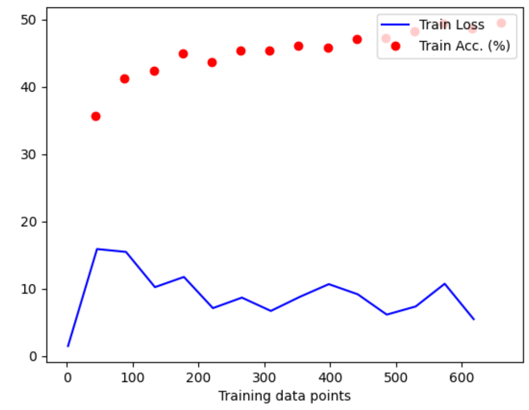

Figure 15: The training loss and accuracy on Oxford Flowers 17 using the pretrained SqueezeNet features.

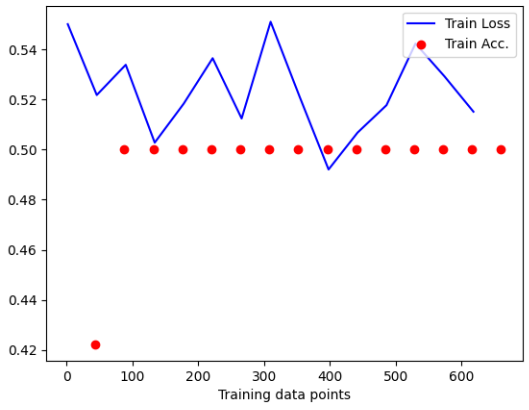

Figure 16: The training loss and accuracy using a custom CNN on Oxford Flowers 17.