1.1. Softmax is invariant to translation; that is, softmax(x) = softmax(x+c). We can derive this using the definition of softmax:

$$softmax(x_i) = \exp(x_i) / \sum (\exp(x_j))$$

$$softmax(x_i+c) = \exp(x_i+c) / \sum (\exp(x_j+c)) = \exp(c)*\exp(x_i) / \sum (\exp(c)\exp(x_j))$$

Factoring exp(c) in the summation, and then pulling it out gives us:

$$softmax(x_i+c) = \exp(c)*\exp(x_i) / (\exp(c) \sum (\exp(x_j))) = \exp(c)/\exp(c)*\exp(x_i) / \sum (\exp(x_j))$$

We see that exp(c)/exp(c) = 1. This gives us:

$$softmax(x_i+c) = \exp(x_i) / \sum (\exp(x_j)) = softmax(x_i)$$

This shows that softmax is invariant to translation.

Often times, we set $c = -max(x_i)$. If x is a vector of numbers, then $x - max(x)$ will be a vector containing only non-positive numbers (negatives or zeros). This in turn will result in $\exp(x - max(x))$ being a vector containing numbers ranging from 0 to 1. This is desirable as it guarantees the numerator of softmax will be at most 1, which is good for numerical stability (you don't have to worry about enormous numerators; exponential are known for producing huge numbers, which can give computers some trouble)

1.2.

- $softmax(x_i)$ will produce a value between 0 and 1. If softmax is computed for each element in x, then the sum of all the softmaxes will total to 1.

- probability

- $s_i = \exp(x_i)$ computes the numerators of the softmax function, computing the exponential of each element in x. $S = \sum(s_i)$ computes the denominator of the softmax function, getting the total sum of the exponential of each element in x. The last step computes the softmax of each element by combining the numerator and denominator: $softmax(x_i) = s_i/S$

1.3. A multi-layer neural network without a non-linear activation function means all the activation functions are linear. Before looking at neural networks, let's look at linear systems in general. Let's define the following functions:

$$f(x) = ax + b, g(x) = cx + d$$

First, observe that the composition of two linear systems is also linear:

$$g(f(x)) = c(ax+b) + d = (ac)x + (bc+d)$$

Second, observe that the linear combination of two linear systems is also linear:

$$Af(x) + Bg(x) = A(ax+b) + B(cx+d) = (Aa+Bc)x + (Ab+Bd)$$

Let us now return to neural networks. To compute the output of a neuron, we get the weighted sum of its input, then apply a linear activation function. Let's examine each step:

- The input to the neuron is either the input to the network, which is a vector of numbers, or the output of some other neuron. We can consider the vector of numbers to be linear. For now, let's assume the output of previous neurons are linear.
- Computing the weighted sum of the input to the neurons is equivalent to computing the linear combination of linear systems. That is, if each input is considered a linear function, then the weighted sum is just the sum of many linear functions, each scaled by some constant. Therefore, the weighted sum is a linear system.
- Applying the activation function is same as taking the composite of two functions. The inner function is the weighted sum, which is linear. The outer function is the activation function, which is also linear. Therefore, the output of the neuron is the composition of two linear systems, so the output is also linear. This proves the assumption we made earlier in this proof.

In a multi-layer neural network, we repeat this process for every neuron. The output of every neuron, then, is linear, which means the output of the entire network is linear.

Now that we have proven that the entire neural network is equivalent to a linear function, we must prove that we are doing linear regression. When training the network, we are turning the weights and biases of all the neurons in the network. This is equivalent to tuning the parameters of a linear function to best fit the provided data. In other words, by turning the weights and biases during training, we are fitting the linear system the neural network produces to match the data. This process is equivalent to linear regression, where we compute the slope and offset for a linear system to best match provided data.

1.4. We can rewrite the sigmoid function:

$$\sigma(x)=1/(1+\exp(-x))=(1+\exp(-x))^{-1}$$

Taking the derivative:

$$d\,\sigma(x)/dx=\exp(-x)*(1/(1+\exp(-x)))^2$$

Rearranging the definition of sigmoid, we get the following equation:

$$\exp(-x)=1/\sigma(x)-1$$

Substituting the equation above, we get the following for the derivative:

$$d\,\sigma(x)/dx=(1/\sigma(x)-1)(\sigma(x))^2=\sigma(x)(1-\sigma(x))$$

1.5. Using $y_i = \sum_{j=1}^{d} \left(x_j W_{ij} + b_i \right)$, we get the following partial derivatives:

$dy_i / dW_{ij} = x_j, dy_i / dx_j = W_{ij}, dy_i / db_i = 1$

To compute the partial derivative of the cost, we do the following:

$dJ / dW_{ij} = dJ / dy_i * dy_i / dW_{ij} = \delta_i x_j$

We see that W_{ij} indexes over i and j. This gives a (k x d) matrix for the derivative. Vectorizing the equation above, we get:

$dJ / dW = \delta x^T$

Let's repeat the process for input:

$dJ / dW_{ij} = dJ / dy_i * dy_i / dW_{ij} = \delta_i x_j$

We see that x_j indexes over j. This gives a (d x 1) matrix for the derivative. Vectorizing the equation above, we get:

$dJ / dx = W^T \delta$

Lastly, let's get the derivative of the bias:

$dJ / db_i = dJ / dy_i * dy_i / db_i = \delta_i$

Vectorizing, we get:

$dJ / db = \delta$

1.6.

1. $\sigma(x)$ ranges from 0 to 1. However, its derivative ranges from 0 to 0.25. This means each time the derivative of the sigmoid function is used in multiplication, it reduces the value that it is multiplied with by a factor of 4 *at least*. In deep neural networks, the derivative of the sigmoid is used in multiplication (during gradient calculation) multiple times. This means deep neural networks will have extremely small gradients due to the scaling down effect mentioned earlier. This result is called the "vanishing gradient" problem.

2. As mentioned earlier, $\sigma(x)$ ranges from 0 to 1. $\tanh(x)$ ranges from -1 to 1. Since tanh has a larger range, it is more descriptive or expressive. In other words, it has more discerning power, which is beneficial for neural networks. tanh also has a greater derivative than sigmoid.

3. As mentioned earlier, the derivative of the sigmoid is at most 0.25. However, the derivative of tanh ranges from 0 to 1. Since the maximum derivative of tanh is larger than the sigmoid (and the derivative of tanh is larger than sigmoid in general), the gradient during gradient calculation shrinks more slowly. This reduces the vanishing gradient problem.

4. In 1.4, we saw:

$\exp(-x)=1/\sigma(x)-1$

We can double x to get:

$\exp(-2x)=1/\sigma(2x)-1$

Using this relationship, we can rewrite tanh:

$\tanh(x)=(1-(1/\sigma(2x)-1))/(1+(1/\sigma(2x)-1))=(2-1/\sigma(2x))/(1/\sigma(2x))=2*\sigma(2x)-1$

2.1.1. Imagine we set all the weights and biases in a network to zero. In that case, the first hidden layer would receive the input to the network and multiply it by 0 to get the weighted sum. This means the first hidden layer, and therefore the entire neural network, doesn't do anything with the input. Therefore, learning cannot occur. The problem also occurs at the output of the network. The output layer will multiply its input by 0 to get the weighted sum, producing 0. Then, it would use its bias to introduce an offset, which is also 0. Therefore, the output of the network is 0. Since the input is unused, and the output is useless, a zero-initialized network cannot learn, so it is useless.

2.1.3. We initialize with random numbers as we do not know the desired weights and biases we should use for our network. By initializing with random numbers, we land on a random point in the cost curve (cost varies as a function of weights and biases). We can then use gradient descent from there to navigate our way towards better weights and biases.

During initialization, we scale for layer size to maintain activation variances and back-propagated gradients variance as we move up and down the network. Figure 6 shows how normalized initialization results in activation values having similar distribution between layers, while figure 7 shows a similar result for back-propagated gradients.

3.2. Learning rate = 0.01 (best), 0.1, 0.001 in that order:
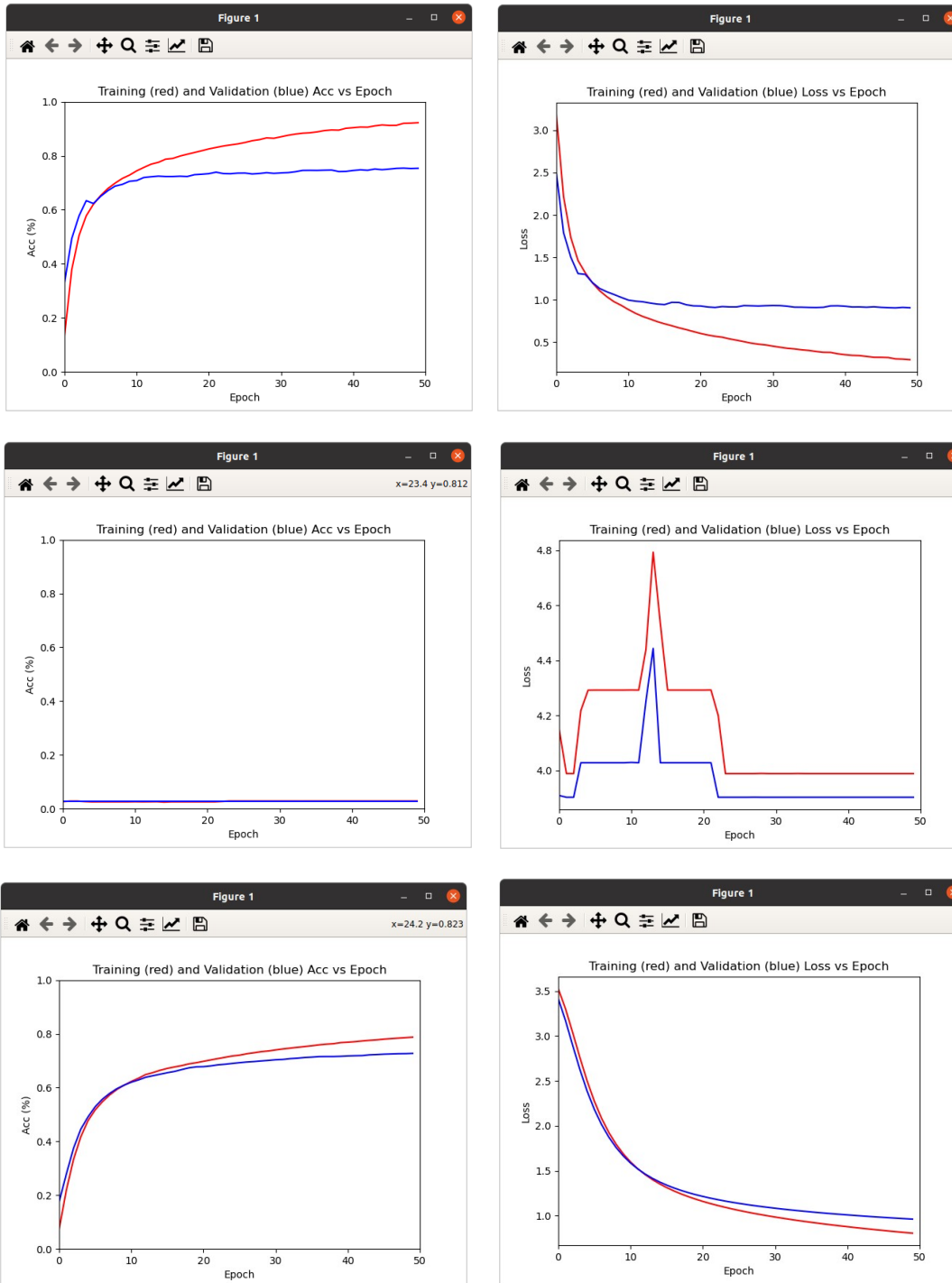
Learning rates have a large impact on how the network trains. At the best learning rate, training accuracy keeps increasing while losses keep decreasing, while the validation performance plateaus for both metrics. At 10 times the learning rate, no learning occurs; the training is too extreme, and the network's parameters jump between multiple bad weights & biases. At 1/10 times the learning rate, learning occurs but slowly; validation and training results are fairly similar. Note that at the end of training at a reduced learning rate, the network does not perform as well as at best learning rate, which means the network should be trained more aggressively.
The best accuracy of the network on the test dataset is 76%.

3.3. First layer weights at initialization (left) and after training (right):
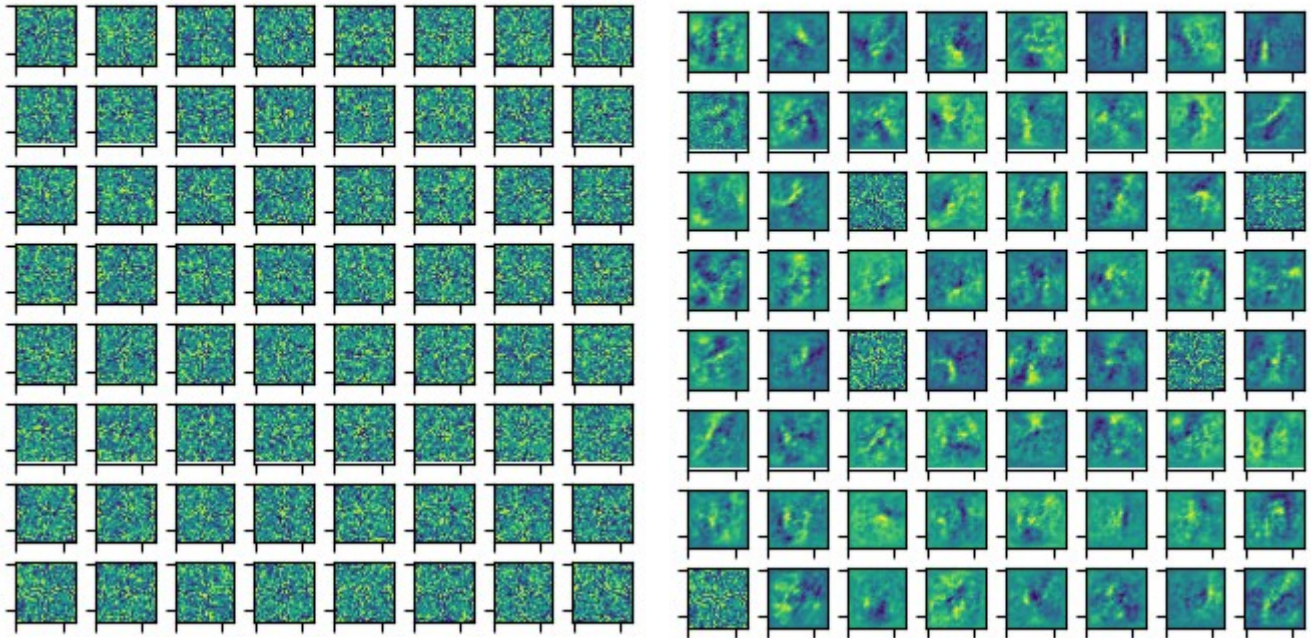
The visualization shows that the weights before training begins is basically just noise, which is expected given we used a uniform random distribution. The key distinct characteristic is the weights look like static.

After training, most of the weights have formed into patterns. They're not obvious to us; I can see a couple of hard edges or curves here and there, but these are the patterns the computer or machine looks for in the images that it is given. It's interesting to note that there are still some nodes who's weights still look like static; either these nodes aren't learning much, or there is merit to looking for patterns that resembles static.
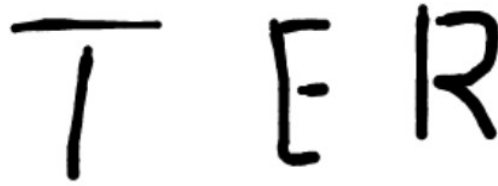
3.4. Here is my confusion matrix:
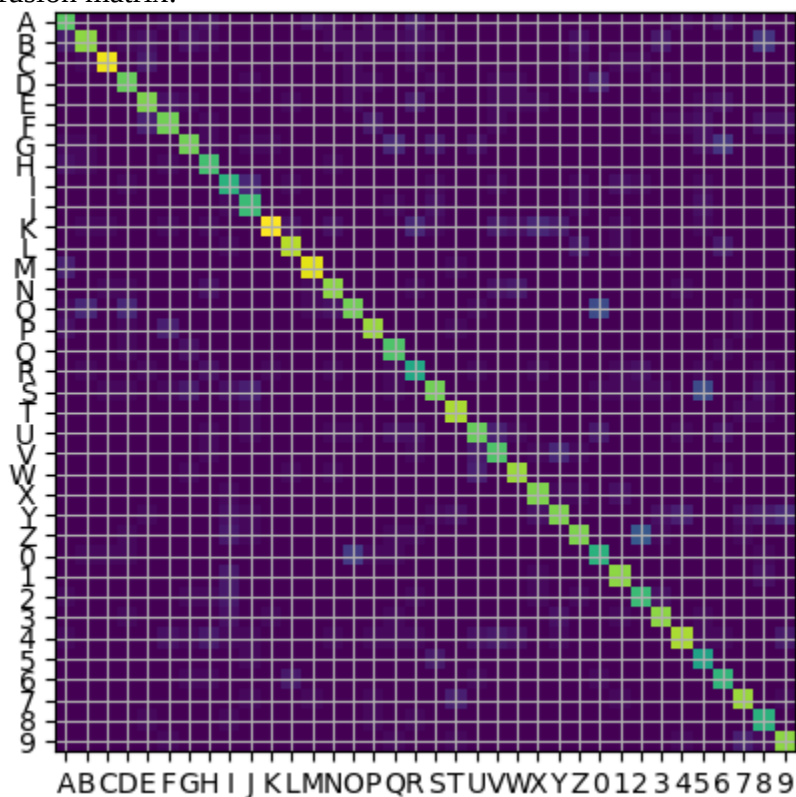
The elements along the diagonal of the confusion matrix are where the neural network correctly classified the input; the elements off of the diagonal are where the network was wrong. Let's look at some of the classes where mistakes occur:

- The network frequently mistakes 0 (zero) and O (the letter). This is very understandable, as even people usually use context to tell the difference between the two, rather than any features in the image.
- 5 and S are also often confused. This is understandable as well, as the two shapes are in general very similar; the only difference is that five has a couple of hard edges in place of curves.
- 2 and Z are also often confused. This is similar to confusing 5 and S.

4.1. One assumption is that a single character is fully connected with itself. Some letters may have parts that are disconnected from the rest of the letter, resulting in incorrect grouping:

Note that the top part of T, the middle part of E, and the right part of R are disconnected from the rest of the letter.

Another assumption is characters are of similar intensity. Since we're using thresholding, if some characters are fainter than others, they may be below the threshold, causing them to not be detected:

There are 6 letters in the image above. However, since the lower 3 letters are so much fainter, they may not be detected due to being below the threshold.

4.2. Images in grayscale:

A B C D E F G
H I J K L M N
O P Q R S T U
V W X Y Z

1 2 3 4 5 6 7 8 9 0

HAIKUS ARE EASY

BUT SOMETIMES THEY DONT MAKE SENSE

REFRIGERATOR

DEEP LEARNING

DEEPER LEARNING

DEEPEST LEARNING

TO DO LIST

1. MAKE A TO DO LIST

2. CHECK OFF THE FIRST
   THING ON TO DO LIST

3. REALIZE YOU HAVE ALREADY
   COMPLETED 2 THINGS

4. REWARD YOURSELF WITH
   A NAP

4.4. The result for the four images are:

| |
|---|
| ABLDEFG
HIJKLMN
QPQR5TH
VWXTZ
1Z3GS6789D |
| HAIKUS ARE RAST
BUT SQMETIMES THEY 2DWT MAKE SEMGE
RBFRIGERATQR |
| DEEP LEARMIN6
DE8PER LEARHING
DEBPE5T LEARNIMG |
| TQ D0 LIST
I MAXE A TQ DQ LI5T
Z ZHEEE DFF TH8 FIRST
THING QN TQ DQ LIST
3 RFALIZE YQU NVE 2LREADT
SQMPLET6D Z THINGI
5 REWARD YQURSEL8 WITA
A NAP |

5.2. The loss over epochs is:

Training Loss vs Epoch
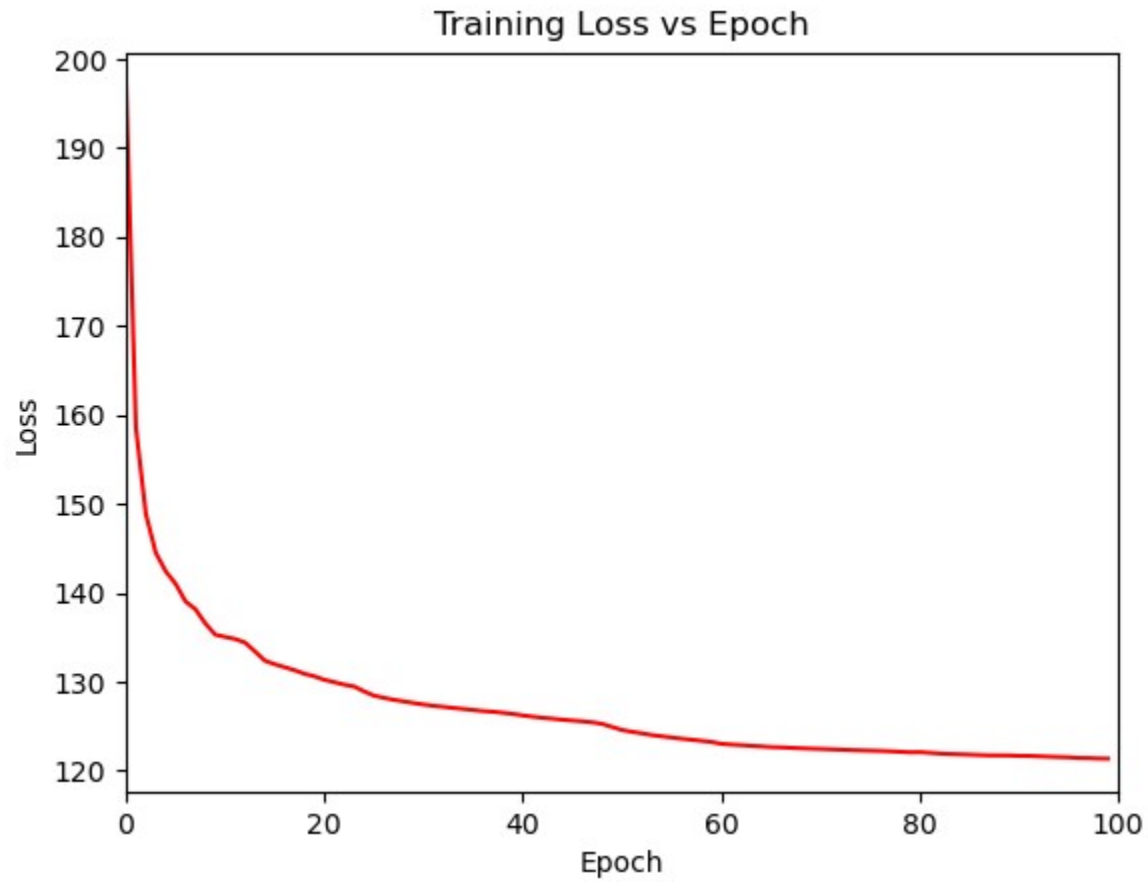
As can be seen, the loss decreases extremely rapidly in the first 10 epochs, then the rate of change decreases. There isn't much change between epochs 80 and 100.

5.3.1. Let's look at some inputs and outputs of the autoencoder. Note that while the general shape is preserved, the end result is generally blurry, and certain letters look like hybrids of multiple letters.
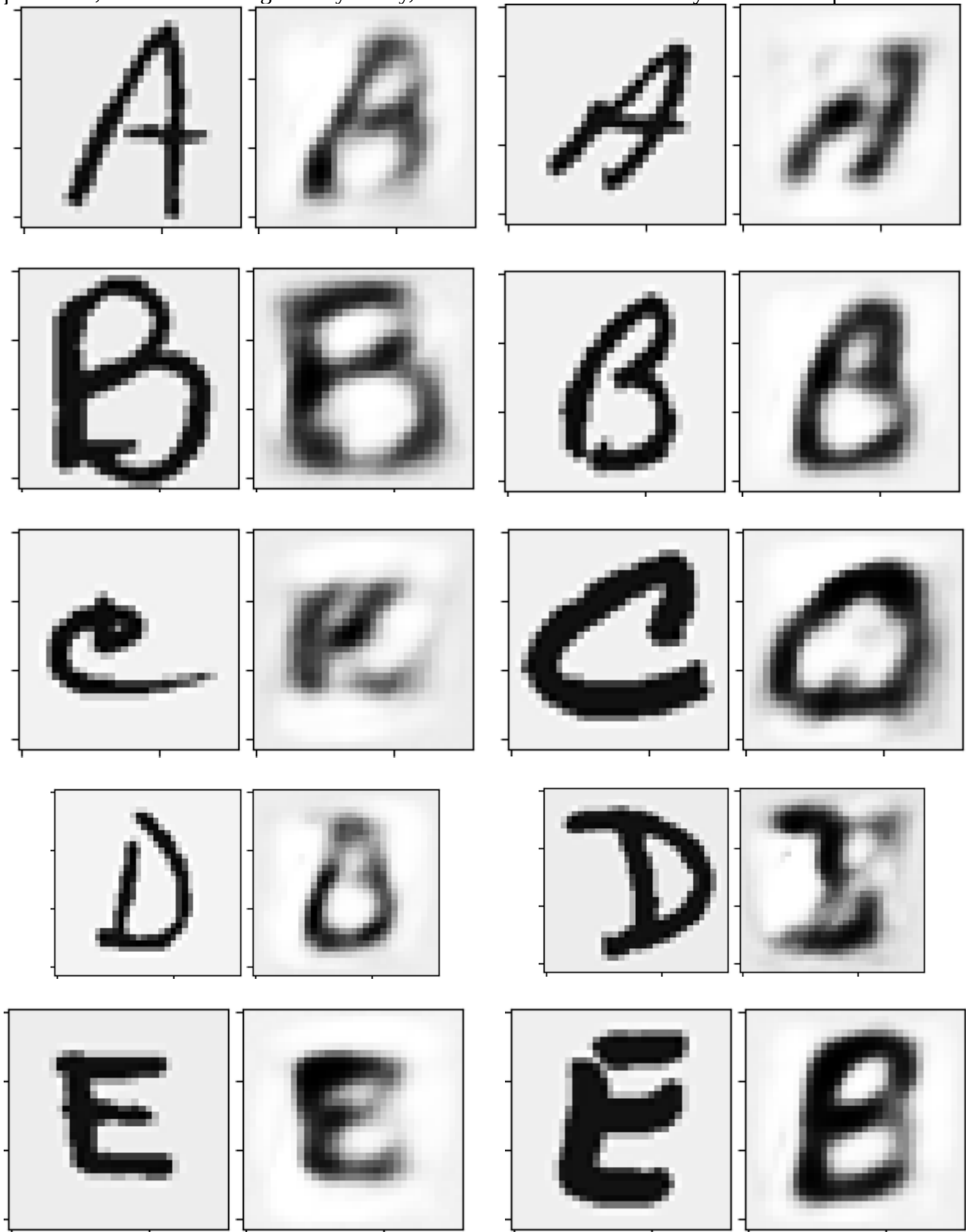
5.3.2. Average PSNR over validation dataset = 15.36935692905095