Inverting Neural Networks to Find the Best Input

Jerry Lai

August 14, 2021

Abstract

Neural Networks are black boxes. This paper attempts a task that may help shed light into how a classification model works. Its goal is to, for each class, find a "best" input vector that produces a perfect output to match that class.

The hope is that, if we know which input vector is the best, we can learn about how the neural network takes the different features into account when making predictions.

This paper compares two methods of doing so - a naive empirical method that serves as a baseline, and an analytical method that attempts to take the inverse of all components of the neural network and work backward from the best output to the best input.

The results obtained through experimentation show that, even after post-processing methods such as filtering, the analytical method actually performed worse than the baseline in most cases.

Link to code

`https://github.com/rifftu/inverting_nn_presentation/blob/main/presentation.ipynb`

Contents

1 Introduction

1.1 Background and Motivation

Neural Networks are very powerful, but a significant hurdle in more widespread adoption is that they are often black boxes. Their complexity allows them to form accurate models on complex data, but it also makes it difficult for humans to understand how they reach their conclusions.

This project attempts a novel way of examining trained classification neural networks by inverting the model on perfect outputs to calculate and visualize perfect inputs.

1.2 Problem Statement

Given a previously trained classification neural network, can we analytically reverse the weights of the neural network, using inverse matrices and inverse activations, to find the ideal input that produces the perfect output for each class?

We will try this for both a **Single-layer perceptron** and a **Multi-layer perceptron**, both trained on the **MNIST handwritten digits** dataset. Instead of using the entire dataset, this project only uses half of it: the entries labeled 0-4. This is done to make the results easier to visualize.

The results will be compared against a baseline result obtained through an empirical, pixel-by-pixel method of finding an ideal input.

Throughout this paper, the method that involves inverting the weights and models shall be referred to as the **Analytical method**, whereas the pixel-by-pixel method (used to create the baseline) shall be referred to as the **Empirical Method**.

1.3 Names and Definitions

1.3.1 Single-layer perceptron

The single-layer perceptron is trained on 21416 rows of MNIST handwritten digits.

It takes 784 features (for a 28x28 image) and has 5 outputs, one for each possible one-hot encoded classification in 0-4 inclusive. The outputs go through a softmax activation.

Mathematical expression of the SLP model:

Let $x \in [-1, 1]^{784}$: input vector, float. Normalized and scaled to fall between -1 and 1 for each pixel. In the original data, it was an integer between 0 and 255.

Let $A \in \mathbb{R}^{784 \times 5}$: trained weights

Let $b \in \mathbb{R}^{5}$: biases

Let $y \in \mathbb{R}^5$ output (prediction), with one-hot encoding

The model is defined as

$$z = x \cdot A + b$$
$$y = softmax(z)$$

The variable $z \in \mathbb{R}^5$ is used to express the intermediate value that the model produces before the softmax activation.

In the example used in this paper, the SLP is trained for 200 iterations using the `adam` optimizer. It has an accuracy score of about 0.94. (Code: Appendix 6.1.1)

1.3.2 Multi-layer perceptron

Similarly to the SLP, the Multi-layer perceptron is trained on 21416 rows of MNIST handwritten digits.

It takes 784 features (for a 28x28 image) and has 5 outputs, one for each possible one-hot encoded classification in 0-4 inclusive. The outputs go through a softmax activation.

It has 2 hidden layers, with 30 nodes in each hidden layer. Each hidden layer uses the Leaky Relu activation with $\alpha = 0.01$.

Mathematical expression of the MLP model:

Let $x \in [-1, 1]^{784}$: input vector, float. Normalized and scaled to fall between -1 and 1 for each pixel. In the original data, it was an integer between 0 and 255.

Let $h_0,\ h_1 \in \mathbb{R}^{30}$: The outputs of the hidden layers

Let $z \in \mathbb{R}^5$: The output of the model before softmax activation

Let $y \in \mathbb{R}^5$ output (prediction, after softmax), with one-hot encoding

Let $A_0 \in \mathbb{R}^{784\times30}, A_1 \in \mathbb{R}^{30\times30}, A_2 \in \mathbb{R}^{30\times5}$: trained weights

Let $b_0 \in \mathbb{R}^{30}, b_1 \in \mathbb{R}^{30}, b_2 \in \mathbb{R}^5$: trained biases

The model is defined as:

$$h_0 = leakyrelu(x \cdot A_0 + b_0)$$
$$h_1 = leakyrelu(h_0 \cdot A_1 + b_1)$$
$$z = h_1 \cdot A_2 + b_2$$
$$y = softmax(z)$$

Leaky Relu ($\alpha = 0.01$) is defined as:

$$leakyrelu(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases}$$

In the example used in this paper, the MLP is trained until converge (categorical training accuracy of 1.0000) using `adam` optimizer, which took about 90 epochs. (Code: Appendix 6.1.2)

1.3.3 Target variables

In addition to the values of the models, the following 3 variables are used throughout this paper:

- $y'_n \in \mathbb{R}^5$. For $n \in [0, 1, 2, 3, 4]$, y'_n refers to the perfect softmax output to predict the n label. Specifically:
$$y'_n[i] = \begin{cases} 1 & \text{if } i = n \\ 0 & \text{if } i \neq n \end{cases}$$
For example, if we are trying to achieve an output with a perfect classification to the Digit 1, then $y'_1 = [0, 1, 0, 0, 0]$.
- $z'_n \in \mathbb{R}^5$, is a manually specified value for z that attempts to satisfy $y'_n = softmax(z)$. Due to the nature of softmax, the true value of z'_n is infinite, so in this paper we use approximate values, such that
$$y'_n \approx softmax(z'_n)$$
- $x'_n \in \mathbb{R}^{784}$ is the hypothetical "ideal" input to produce the label n, so that $z'_n \approx x'_n \cdot A + b$. x'_n is subject to a constraint in terms of range. In the training data for the model, each pixel falls in the range of [-1,1], so x'_n must also fall into this range - otherwise, it wouldn't be able to be converted back into a visible image.

  The ideal x'_n value that we are trying to predict satisfies the following condition:
$$\forall x \in [-1, 1]^{784}, \;\; \|(z'_n - z(x)\| \geq \|(z'_n, z(x'_n))\|$$
  Where $z(x)$ refers to the z value obtained by passing x through the model.

  The norm in the expression above may vary - in the **analytical method**, we optimize for the ℓ_2 norm, whereas in the **empirical method** we optimize for the ℓ_1 norm for ease of calculation. The results suggest that, due to the inaccuracy of both methods, the type of norm that is used does not make a significant impact.

2 Method

2.1 Algorithm (Empirical Method)

Before exploring the analytical method, we used a simple algorithm to establish a baseline for reasonable approximations of x'. This method is not meant to be accurate - rather, it is intended to be intuitive and show what could be achieved with a naive method.

Later, the results from the analytical method will be compared against the results from this algorithm, to see if a mathematical model can yield better results.

For the empirical method, the same algorithm is used for both the SLP and MLP.

Algorithm description: (Code in appendix 6.1.3)

Algorithm 1 Empirical Method

```
Input: model, $n \in \{0, 1, 2, 3, 4\}$, $z'_n$
Output: $x'_n \in [-1, 1]^{784}$
 1: List of “importance” $L \leftarrow [0]^{784}$                  ▷ Used to rank each pixel
 2: for index $i$ from $[0, 784)$ do                                ▷ Iterate through each pixel
 3:     $t_i \in \{0, 1\}^{784} :=$                                  ▷ Create a one-hot vector for that pixel
            $\begin{cases} t_i[j] \leftarrow 1 \text{ if } i = j \\ t_i[j] \leftarrow 0 \text{ if } i \neq j \end{cases}$
 4:     $L[i] \leftarrow -\|z'_n - z(t_i)\|_1$                      ▷ Importance is negative loss
 5: end for
 6: $x'_n \leftarrow [0]^{784}$
 7: $color \leftarrow 255$
 8: for index i of top 254 entries of $L$ do
 9:     $x'_n[i] \leftarrow color$
10:     $color \leftarrow color - 1$
11: end for
12: Return $x'_n$
```

First, we create an array that tracks the “importance” of each pixel. In lines 2-5, we iterate through all of the pixels, creating a one-hot input vector for each pixel and passing it through the model to get the z value for each input vector. The ones with the best z get the highest importance.

Then, in lines 7-11, we sort the pixels based on their importance. The most important pixel gets a color value 255 (white). The second most important pixel gets 254, and so on.

2.2 Analytical Method

2.2.1 Defining z'_n

Recall from section 1.3.3 that we want to have y'_n be a perfect one-hot vector for index n. However, both the MLP and SLP have a softmax activation for its output, specifically, $y'_n = softmax(z'_n)$.

In order to get a perfect result from softmax, z'_n would need to contain $\pm\infty$. This will make it impossible to use it in calculations, so we can only approximate z'_n. We opted to use

$$z'_n[i] = \begin{cases} 10 & \text{if } i = n \\ -10 & \text{if } i \neq n \end{cases}$$

This allows us to get reasonably close to perfect classification, as

$$softmax([10, -10, -10, -10, -10])$$

$$= [0.99999999, 0.000000002, 0.000000002, 0.000000002, 0.000000002]$$

2.2.2 Inverting the SLP

The model of the SLP is $z = x \cdot A + b$

For the purpose of simplicity, we have decided to omit the bias layer when trying to invert the SLP. That is because the bias has very little effect on the output of the neural network: we found that the bias values were all within ± 0.09 of 0 (seen from the code snippet in Appendix 6.1.4), making its impact on z mostly insignificant.

When we are inverting the SLP, we decided to use

$$x'_n = z'_n \cdot A^+$$

A^+ is the Moore-Penrose Inverse of A.

The Moore-Penrose Inverse (Penrose 1955) is used here because it provides a value for x'_n that minimizes the squared error between z'_n and $z(x)$, and is easily calculable by a computer.

This process is repeated for each n in $\{0, 1, 2, 3, 4\}$. The code for this is included in appendix 6.1.5.

2.2.3 Inverting the MLP

For the MLP, a similar strategy to SLP inversion is repeatedly used, inverting each layer independently starting from the one closest to the output. To invert the Leaky Relu activation, a function (`anti_leakyrelu`) was written to undo its effects.

To reiterate the definition of the MLP from section 1.3.2:

$$h_0 = leakyrelu(x \cdot A_0 + b_0)$$

$$h_1 = leakyrelu(h_0 \cdot A_1 + b_1)$$
$$z = h_1 \cdot A_2 + b_2$$

The following was done to invert the MLP, starting with each z' defined in 2.2.1.

$$h'_1 \leftarrow (z' - b_2) \cdot A_2^+$$
$$h'_1 \leftarrow \texttt{anti\_leakyrelu}(h'_1)$$
$$h'_0 \leftarrow (h'_1 - b_1) \cdot A_1^+$$
$$h'_0 \leftarrow \texttt{anti\_leakyrelu}(h'_0)$$
$$x' \leftarrow (h'_0 - b_0) \cdot A_0^+$$

This process is repeated for each n in $\{0, 1, 2, 3, 4\}$. The code used to do this is included in Appendix 6.1.6

2.2.4 Filtering the inverse matrices

The results from both the inverse SLP and inverse MLP resulted in highly noisy outputs. In order to attempt to decrease the noise, we applied 2 different filters to the inverse matrices, the **top_and_bottom** filter and the **middle** filter. These filters were applied to both the inverse matrices in the SLP model and the MLP model. However, only the ones done for the SLP model are presented in the paper because applying these filters on the inverse MLP model made no visually discernible difference.

Algorithm 2 top_and_bottom filter

```
Input: $A^+$: inverse matrix to filter, $p$: percentile to filter
Output: Filtered $A^+$
1: for element $e$ in $A^+$ do
2:     if $e$ is NOT in top $p$ or bottom $p$ of all elements then
3:         Set $e$ to zero
4:     end if
5: end for
6: Return $A^+$
```

Algorithm 3 middle_filter

```
Input: $A^+$: inverse matrix to filter, $p$: percentile to filter
Output: Filtered $A^+$
1: Let $c_{min}$, $c_{max}$ be the values that are exactly the bottom and top $p$ percentile of $A^+$
2: for element $e$ in $A^+$ do
3:     $e \leftarrow \min(e, c_{max})$
4:     $e \leftarrow \max(e, c_{min})$
5: end for
6: Return $A^+$
```

The purpose for the first filter is to reduce the amount of noise in the inverse matrices by deleting those with small magnitudes. The purpose of the second filter is to reduce the range of values in the inverse matrices by reducing the magnitude of the largest values.

When the filters were applied, the percentile ranges of [2%, 98%] were used for the top_and_bottom filter, and [20%, 80%] for the middle_filter. These percentiles were chosen because the resulting A^+ for both were able to cut away as much as possible while still returning x' values that, when passed through the model, provided y values that equaled y' up to 2 decimal points.

3 Results

For each method described in the previous section, the x'_n is scaled to between 0-255 and displayed below. Each row contains the x'_n for each n from 0-4, from left to right.

3.1 SLP

3.1.1 Empirical Method - results

Method to obtain this is described in section 2.1

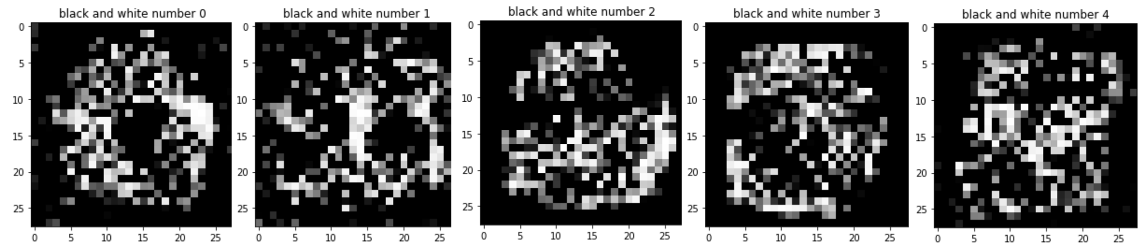


Fig 3.1: Empirical Method, SLP

In this result, the outlines for the digits are clearly visible. In addition to the numbers themselves, also visible are white patches around the outer rims of digits 1 and 4. These are lightly highlights of pixels that are the least likely to be white in other digits.

3.1.2 Analytical Method - results

Method to obtain this is described in section 2.2.2

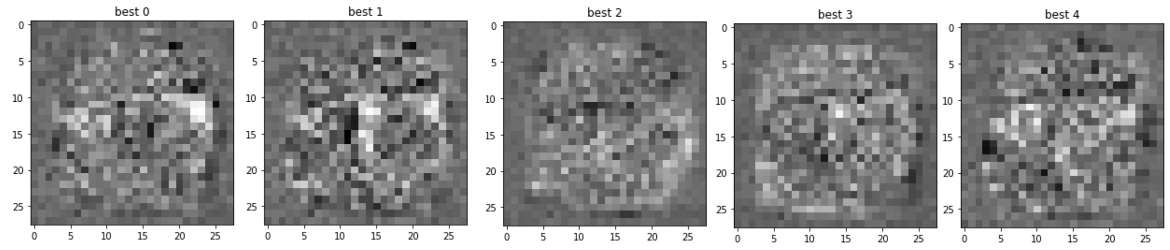


Fig 3.2: Analytical Method, SLP

This result bears similarities to the previous one, but is much more noisy. It is difficult but still possible to make out some of the shapes of the digits.

3.1.3 Analytical Method - results, with filtering

Results after conducting top_and_bottom filtering (Algorithm from section 2.2.4)

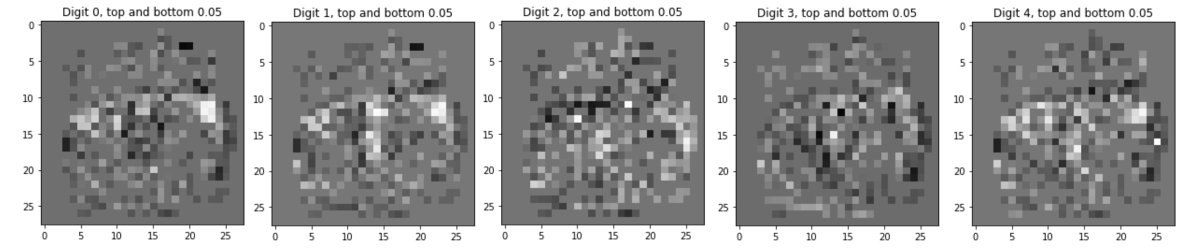


Fig 3.3: Analytical Method with top_bottom_filter, SLP

This row highlights the pixels that are important in classification, and have high positive (or negative) correlation to the classification. However, although these images, when fed to the model, produce very good classification, it is impossible for the human eye to discern any digits.

Results after conducting middle_filter (Algorithm from from section 2.2.4)

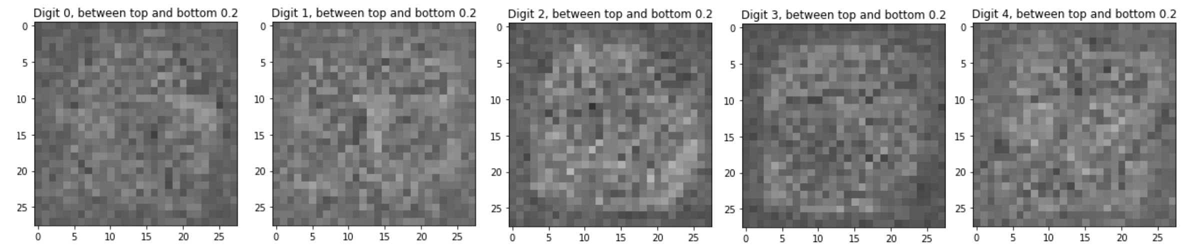


Fig 3.4: Analytical Method with middle_filter, SLP

This is similar to Figure 3.2, but without the brightest and darkest spots, it is easier to see the outlines of the digits. However, it is still not as visually informative as the results from Figure 3.1.

3.2 MLP

3.2.1 Empirical Method - results

Method to obtain this is described in section 2.1

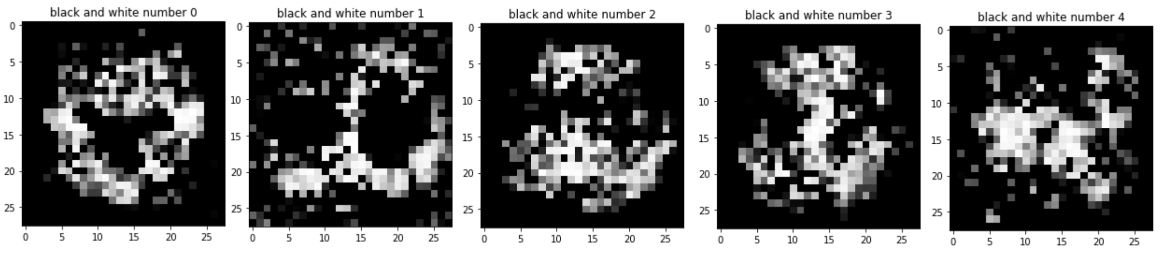


Fig 3.5: Empirical Method, MLP

These offer approximately the same amount of detail as the results from the SLP. The outlines of the digits are roughly visible, and the dark areas of each class also appear as lighter areas of other classes. A viewer can use this to learn about how the MLP makes its decisions.

3.2.2 Analytical Method - results

Method to obtain this is described in section 2.2.3

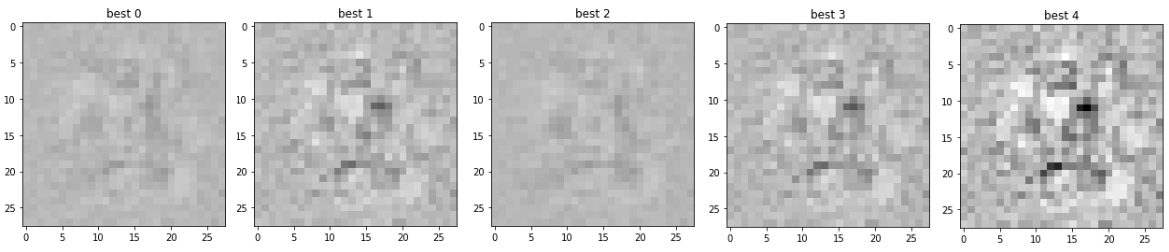


Fig 3.5: Empirical Method, MLP

Unlike in the SLP example, the results shown here bear no resemblance to the results obtained from the empirical method. They also bear no resemblance to the real digits. However, they still somehow achieve perfect classification outputs in the neural network.

4 Conclusion

In summary, the analytical method does not produce as visually informative results as the empirical method.

In both the SLP and the MLP models, the empirical method provided results that resembled real data more closely than the analytical method. Additionally, looking at the results from the empirical method would reveal more about which features (pixels) contribute to each classification.

Another drawback of the analytical method was that the results were dominated by noise in a way that the results from the analytical method was not. However, picking out the top pixels in the analytical method yielded pixels in entirely different regions than doing so in the empirical method, which inherently picks out the top 254 pixels for each number.

Perhaps doing filtering for the top pixels of the result, rather than the inverse matrices, will make it more similar to the processes of the empirical method and possibly achieve better results. However, that is out of the scope of this paper.

The performance of the analytical method differs between the SLP and the MLP. In the SLP, the analytical method returned similar results as the empirical method, albeit with more noise.

On the other hand, the results from the analytical method for the MLP returned much less informative results than the empirical method.This is surprising because, while the empirical method makes sense, mathematically, for the SLP, it is not a mathematically sound method for approximating x' for an MLP, as the pixels in an MLP are not independent of each other. Therefore, the empirical method wasn't expected to do as well for an MLP. The results showed otherwise.

The results from the analytical method for MLP is another example of the **black-box problem** of deep neural networks: Although shapes do exist in the results, it is meaningless to a human viewer. On the other hand, the trained neural network can make excellent predictions based on them.

Another possible explanation for the lack of resemblance in the results of the analytical method is the space available for the x' to occupy:

- In the training data, when writing handwritten digits, although there are 784 pixels, only a few of them can be selected, and most of the time they have to be adjacent to each other.
- In the training data, the entries are [-1,1] for each pixel. However, this constraint wasn't applied when inverting the models.

Since the empirical method only uses a small fraction of the pixels, and the pixel values are manually assigned into the visible range, it doesn't deviate from the training data by much; however, without these constraints, it makes sense that the analytical method may return results that are very different than any answer that exists within these constraints.

It may be worth pursuing different analytical methods that are subject to the constraints listed above. However, these would require more complicated methods, such as iterative optimization.

An unexpected outcome is how well the empirical method worked to approximate the best input, even on a deep neural network. It may also be worthwhile to pursue improved versions of this method, since the one used here was intentionally basic.

5 References

- MNIST Handwritten digits dataset, http://yann.lecun.com/exdb/mnist/
- Penrose, R. (1955). A generalized inverse for matrices. Mathematical Proceedings of the Cambridge Philosophical Society, 51(3), 406-413. doi:10.1017/S0305004100030401
- Software packages used:

  Numpy

  Scikit-learn

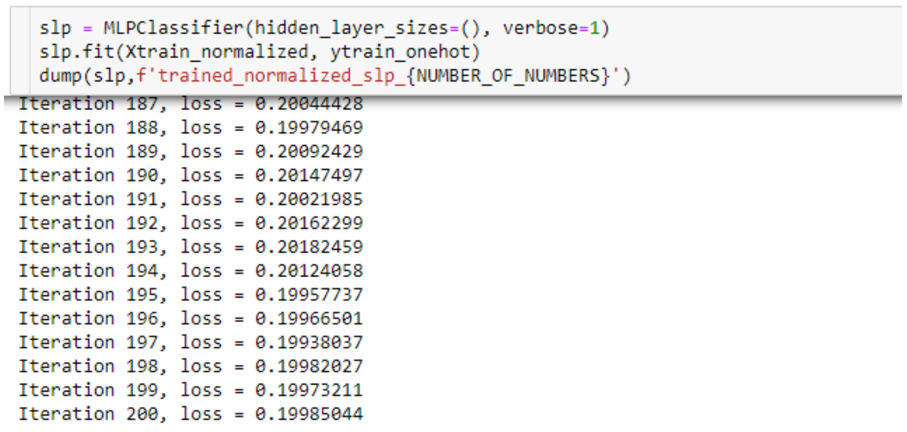
  Tensorflow/Keras

All code used for this project can be found at
`https://github.com/rifftu/inverting_nn_presentation/blob/main/presentation.ipynb`

6 Appendix

6.1 Code

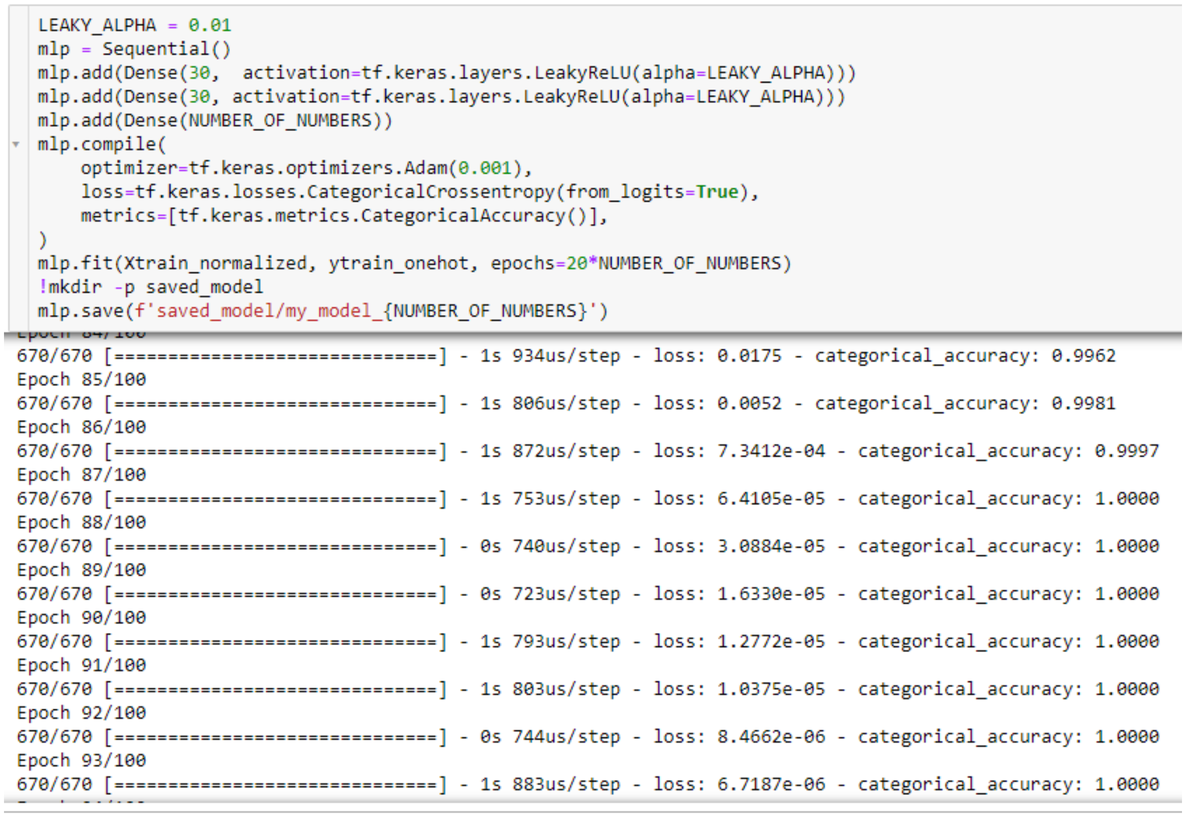
6.1.1 Training SLP



```
slp = MLPClassifier(hidden_layer_sizes=(), verbose=1)
slp.fit(Xtrain_normalized, ytrain_onehot)
dump(slp,f'trained_normalized_slp_{NUMBER_OF_NUMBERS}')
```

```
Iteration 187, loss = 0.20044428
Iteration 188, loss = 0.19979469
Iteration 189, loss = 0.20092429
Iteration 190, loss = 0.20147497
Iteration 191, loss = 0.20021985
Iteration 192, loss = 0.20162299
Iteration 193, loss = 0.20182459
Iteration 194, loss = 0.20124058
Iteration 195, loss = 0.19957737
Iteration 196, loss = 0.19966501
Iteration 197, loss = 0.19938037
Iteration 198, loss = 0.19982027
Iteration 199, loss = 0.19973211
Iteration 200, loss = 0.19985044
```

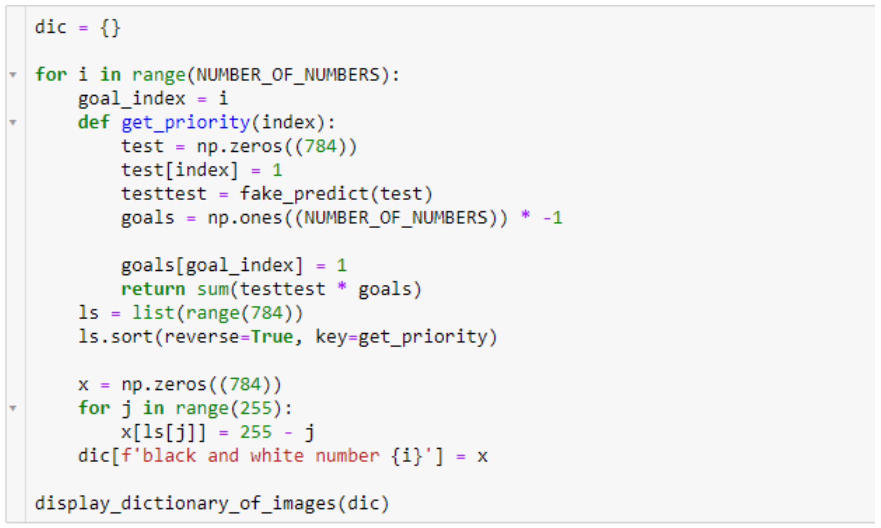
6.1.2 Training MLP



```
LEAKY_ALPHA = 0.01
mlp = Sequential()
mlp.add(Dense(30,  activation=tf.keras.layers.LeakyReLU(alpha=LEAKY_ALPHA)))
mlp.add(Dense(30, activation=tf.keras.layers.LeakyReLU(alpha=LEAKY_ALPHA)))
mlp.add(Dense(NUMBER_OF_NUMBERS))
mlp.compile(
    optimizer=tf.keras.optimizers.Adam(0.001),
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=[tf.keras.metrics.CategoricalAccuracy()],
)
mlp.fit(Xtrain_normalized, ytrain_onehot, epochs=20*NUMBER_OF_NUMBERS)
!mkdir -p saved_model
mlp.save(f'saved_model/my_model_{NUMBER_OF_NUMBERS}')
```

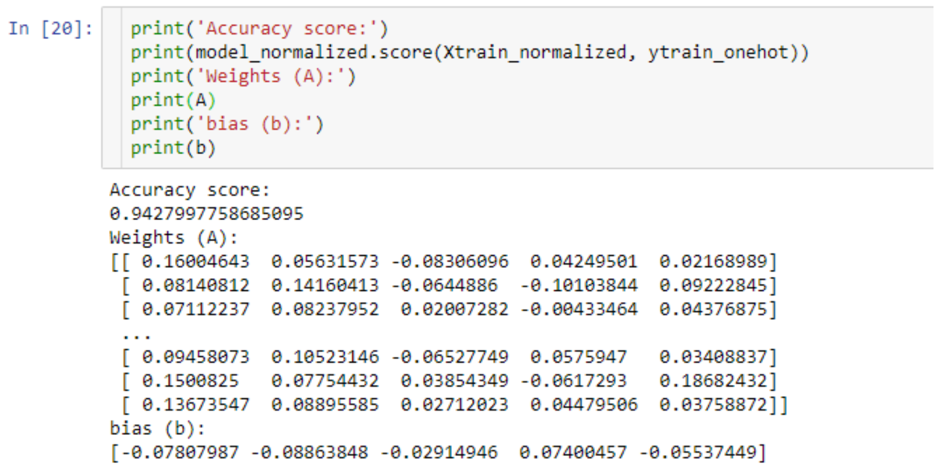
```
670/670 [==============================] - 1s 934us/step - loss: 0.0175 - categorical_accuracy: 0.9962
Epoch 85/100
670/670 [==============================] - 1s 806us/step - loss: 0.0052 - categorical_accuracy: 0.9981
Epoch 86/100
670/670 [==============================] - 1s 872us/step - loss: 7.3412e-04 - categorical_accuracy: 0.9997
Epoch 87/100
670/670 [==============================] - 1s 753us/step - loss: 6.4105e-05 - categorical_accuracy: 1.0000
Epoch 88/100
670/670 [==============================] - 0s 740us/step - loss: 3.0884e-05 - categorical_accuracy: 1.0000
Epoch 89/100
670/670 [==============================] - 0s 723us/step - loss: 1.6330e-05 - categorical_accuracy: 1.0000
Epoch 90/100
670/670 [==============================] - 1s 793us/step - loss: 1.2772e-05 - categorical_accuracy: 1.0000
Epoch 91/100
670/670 [==============================] - 1s 803us/step - loss: 1.0375e-05 - categorical_accuracy: 1.0000
Epoch 92/100
670/670 [==============================] - 0s 744us/step - loss: 8.4662e-06 - categorical_accuracy: 1.0000
Epoch 93/100
670/670 [==============================] - 1s 883us/step - loss: 6.7187e-06 - categorical_accuracy: 1.0000
```

6.1.3 Empirical Algorithm



```
dic = {}

for i in range(NUMBER_OF_NUMBERS):
    goal_index = i
    def get_priority(index):
        test = np.zeros((784))
        test[index] = 1
        testtest = fake_predict(test)
        goals = np.ones((NUMBER_OF_NUMBERS)) * -1

        goals[goal_index] = 1
        return sum(testtest * goals)
    ls = list(range(784))
    ls.sort(reverse=True, key=get_priority)

    x = np.zeros((784))
    for j in range(255):
        x[ls[j]] = 255 - j
    dic[f'black and white number {i}'] = x

display_dictionary_of_images(dic)
```

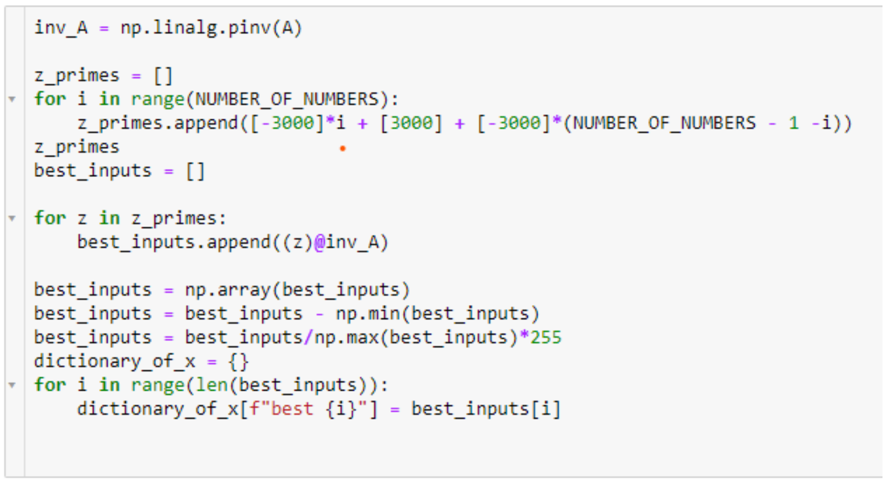
6.1.4 SLP Information



```
In [20]:  print('Accuracy score:')
          print(model_normalized.score(Xtrain_normalized, ytrain_onehot))
          print('Weights (A):')
          print(A)
          print('bias (b):')
          print(b)
```

```
Accuracy score:
0.9427997758685095
Weights (A):
[[ 0.16004643  0.05631573 -0.08306096  0.04249501  0.02168989]
 [ 0.08140812  0.14160413 -0.0644886  -0.10103844  0.09222845]
 [ 0.07112237  0.08237952  0.02007282 -0.00433464  0.04376875]
 ...
 [ 0.09458073  0.10523146 -0.06527749  0.0575947   0.03408837]
 [ 0.1500825   0.07754432  0.03854349 -0.0617293   0.18682432]
 [ 0.13673547  0.08895585  0.02712023  0.04479506  0.03758872]]
bias (b):
[-0.07807987 -0.08863848 -0.02914946  0.07400457 -0.05537449]
```

6.1.5 SLP Inversion



```
inv_A = np.linalg.pinv(A)

z_primes = []
for i in range(NUMBER_OF_NUMBERS):
    z_primes.append([-3000]*i + [3000] + [-3000]*(NUMBER_OF_NUMBERS - 1 -i))
z_primes
best_inputs = []

for z in z_primes:
    best_inputs.append((z)@inv_A)

best_inputs = np.array(best_inputs)
best_inputs = best_inputs - np.min(best_inputs)
best_inputs = best_inputs/np.max(best_inputs)*255
dictionary_of_x = {}
for i in range(len(best_inputs)):
    dictionary_of_x[f"best {i}"] = best_inputs[i]
```

6.1.6 invert MLP

```
weights = [layer.get_weights() for layer in mlp.layers]

As, bs = [],[]

for weight in weights:
  As.append(weight[0])
  bs.append(weight[1])

def anti_leakyrelu(X, alpha):
    def myfunc(x):
        if x >= 0:
            return x
        else:
            return x / alpha
    myfunc_vec = np.vectorize(myfunc)
    return myfunc_vec(X)

def reverse_z(z_prime, As, bs):
    z = z_prime
    for layer in [2,1,0]:
        z_old = z[:]
        if layer != len(As) - 1:
            z = anti_leakyrelu(z, LEAKY_ALPHA)
        z = z - bs[layer]
        inv_A = np.linalg.pinv(As[layer])
        z = z@inv_A
```

```python

    return z


best_inputs_unscaled = []

for z in z_primes:
    best_inputs_unscaled.append(reverse_z(z, As,bs))

best_inputs_unscaled = np.array(best_inputs_unscaled)
```

6.1.7 Filtering

```python
def get_filtered_inputs(z_prime, inv_A, b, digit):


    percentiles = np.array([0,0.005, 0.01, 0.02, 0.05, 0.1, 0.2,
   0.3, 0.5, 0.7, 0.8, 0.9, 0.95, 0.98,  0.99, 0.995,1])
    percentile_indices = np.maximum((percentiles * np.size(inv_A)).
   astype(int) - 1, 0)

    cutoffs = np.sort(np.array(inv_A).ravel())[percentile_indices]

    global_min = np.min(z_prime@inv_A)
    global_range = np.max(z_prime@inv_A) - global_min

    def invert(cut_inv_A):
        return ((z_prime - b)@cut_inv_A)

    def scale(x):
        x = x - global_min
        x = x*255/global_range
        return x

    high_pass = {}
    high_pass_unscaled_list = []
    low_pass = {}
    low_pass_unscaled_list = []

    both_ends = {}
    middle = {}


    for i in range(len(percentiles)):
        above_iA = filter_values(inv_A, cutoffs[i], 1)
        above_x = invert(above_iA)
        above_x_scaled = scale(above_x)

        high_pass[f"Digit {digit}, only above {cutoffs[i].round(3)} which is {percentiles[i]} percentile"] = above_x_scaled
        high_pass_unscaled_list.append(above_x)

        below_iA = filter_values(inv_A, cutoffs[i], 0)
        below_x = invert(below_iA)
        below_x_scaled = scale(below_x)
        low_pass[f"Digit {digit}, only below {cutoffs[i].round(3)} which is {percentiles[i]} percentile"] = below_x_scaled
        low_pass_unscaled_list.append(below_x)

    for i in range(math.ceil(len(percentiles)/2)):
        top_and_bottom = low_pass_unscaled_list[i] + high_pass_unscaled_list[-i-1]
        both_ends[f"Digit {digit}, top and bottom {percentiles[i]}"] = scale(top_and_bottom)

        middle_iA = limit_values(inv_A, cutoffs[i], cutoffs[-i-1])

        middle_x = invert(middle_iA)
        middle_x_scaled = scale(middle_x)
        middle[f"Digit {digit}, between top and bottom {percentiles[i]}"] = middle_x_scaled

    return high_pass, low_pass, both_ends, middle


# Set all values of the array that is either higher or lower than the cutoff to 0
def filter_values(array, cutoff, high_pass):
    array = np.copy(array)
    if high_pass:
        delete_indices = (array <= cutoff)
    else:
        delete_indices = (array > cutoff)
    array[delete_indices] = 0
    return array
def limit_values(array, cutoff_low, cutoff_high):
    array = np.maximum(array, cutoff_low)
    array = np.minimum(array, cutoff_high)
    return array

high_passes = []
```

```python
low_passes = []
both_ends = []
middles = []

for i in range(NUMBER_OF_NUMBERS):
    high_pass_i, low_pass_i, both_ends_i, middle_i = get_filtered_inputs(z_primes[i], inv_A, b, i)
    high_passes.append(high_pass_i)
    low_passes.append(low_pass_i)
    both_ends.append(both_ends_i)
    middles.append(middle_i)

def get_dict_of_filtered_inputs(percentile_ends, percentile_middle):
    dic = {}
    for digit in range(NUMBER_OF_NUMBERS):
        if percentile_ends != None:
            string = f'Digit {digit}, top and bottom {percentile_ends}'
            dic[string] = both_ends[digit][f'Digit {digit}, top and bottom {percentile_ends}']
        if percentile_middle != None:
            string = f"Digit {digit}, between top and bottom {percentile_middle}"
            dic[string] = middles[digit][f"Digit {digit}, between top and bottom {percentile_middle}"]
    return dic
display_dictionary_of_images(get_dict_of_filtered_inputs(0.05, 0.2))
```