Neural Networks for Images (67103): Exercise 2 - Autoencoders

By: Amit Halbreich, ID: 208917393

Theoretical Questions – Answers

1. (I) Let A and B be linear transformation matrices, and let x be a vector. Then we have:

$$(A \circ B)(x) = A(B(x))$$

Let $y = B(x)$, so that:

$$(A \circ B)(x) = A(y)$$

Since A is a linear transformation, to prove additivity trait we have:

$$A(y) = A(B(x)) = A(Bx1 + Bx2) = A(Bx1) + A(Bx2)$$

$$for\ any\ x1\ and\ x2\ in\ R^n.$$

Therefore, we have:

$$(A \circ B)(x1 + x2) = A\big(B(x1 + x2)\big) = A(Bx1 + Bx2) = A(Bx1) + A(Bx2) =$$

$$= (A \circ B)(x1) + (A \circ B)(x2)$$

Similarly, to prove homogeneity trait, we have:

$$(A \circ B)(kx) = A(B(kx)) = A(kBx) = kA(Bx) = k(A \circ B)(x)$$

Therefore, the composition of linear functions is a linear function because it respects both additivity and homogeneity traits.

Define the functions as follows:

$$f(x) = ax + b$$

$$g(x) = cx + d$$

For any $a, b, c, d \in \mathbb{R}$ we have:

$$(f \circ g)(x) = f\big(g(x)\big) = f(cx + d) = acx + ad + b$$

Denote $ac = \lambda$ and $ad + b = \eta$ we have:

$$(f \circ g)(x) = \lambda x + \eta$$

And there we have a linear function with respect to homogeneity and linearity as in matrix form.

1. (II) To show that the composition of affine transformations remains an affine function, let $f(x)$ and $g(x)$ be two affine functions, such that $f(x) = Ax + b$ and $g(x) = Cx + d$, where A, C are matrices and b, d are vectors. Then the composition of f and g is given by:

$$(f \circ g)(x) = f\big(g(x)\big) = A(Cx + d) + b = ACx + Ad + b$$

Since A and C are matrices, their product AC is also a matrix, and hence the composition of f and g can be written as:

$$(f \circ g)(x) = ACx + (Ad + b)$$

Since A and C are matrices and $Ad + b$ is a vector, the composition of f and g is also an affine function.

2. (I) The stopping condition for the Gradient Descent algorithm:

$$\theta^{n+1} = \theta^n + \alpha \nabla f_{\theta_n}(x)$$

Generally, we will strive to get to the value of 0 for the gradient because that indicates the algorithm is converged to a minimum.

But since we cannot guarantee a convergence to 0 we can define several different stopping conditions:

Gradient Convergence threshold: One common stopping condition is to monitor the magnitude of the gradient of the objective function with respect to the parameter θ at the nth iteration $\nabla f_{\theta_n}(x)$ and check if it falls below a certain threshold. This can be expressed as:

$$\left\|\nabla f_{\theta_n}(x)\right\| < \epsilon$$

Where $\left\|\nabla f_{\theta_n}(x)\right\|$ denotes the magnitude of the gradient at the current iteration, ϵ is a predefined threshold, and the algorithm stops when this condition is satisfied. This indicates that the algorithm has converged to a local minimum of the objective function, as the gradient is close to zero, and further iterations are not likely to result in significant changes to the parameter values.

Minor Change in θ values in successive iterations:

Another stopping condition is to monitor the change in the parameter values (θ) at each iteration and check if it falls below a certain threshold. This can be expressed as:

$$\|\theta^{n+1} - \theta^n\| < \delta$$

where θ^{n+1} and θ^n represent the parameter values at the (n+1)-th and n-th iterations respectively, δ is a predefined threshold, and the algorithm stops when this condition is satisfied. This indicates that the algorithm has reached a point where the parameter values are not changing significantly, and further iterations are not necessary.

The choice of the threshold $(\varepsilon \; or \; \delta)$ depends on the specific problem and the desired accuracy of the optimization. A smaller threshold will result in a more accurate solution but may require more iterations, while a larger threshold may result in faster convergence but a less accurate solution.

In addition to the threshold-based stopping conditions, other stopping conditions can also be used, such as a maximum number of iterations or a predefined runtime limit.

Maximum Iterations value: We can predefine a fixed number of iterations that the algorithm will run, independently from the target function.

Runtime limit: We can limit the Gradient Descent algorithm to stop after a certain amount of runtime regardless of the target function.

Target Function fixed target value: We can set a fixed desired value for the target function such that if the target function reaches this value the algorithm will stop.

It is important to choose an appropriate stopping condition and consider all the possibilities to balance the trade-off between convergence accuracy and computational efficiency in the Gradient Descent optimization process.

2. (II) To derive the conditions for classifying a stationary point as a local maximum or minimum using the second-order multivariate Taylor theorem, we need to examine the Hessian matrix, which is the matrix of second partial derivatives of the objective function with respect to the parameter θ. The Hessian matrix is given by:

$$H_{ij} = \frac{\partial^2 f(x)}{\partial \theta_i \partial \theta_j}$$

At a stationary point, the gradient is zero, i.e., $\nabla f_{\theta_n}(x) = 0$. We can then use the second-order multivariate Taylor theorem to approximate the objective function near the stationary point as:

$$f(\theta^{n+1}) = f(\theta^n) + (\theta^{n+1} - \theta^n)^T * \nabla f_{\theta^n}(x) + \frac{1}{2} * (\theta^{n+1} - \theta^n)^T * H(f(\theta_n)) * (\theta^{n+1} - \theta^n) + O(||\theta^{n+1} - \theta^n||^3)$$

where θ_{n+1} is the next iterate of the parameters, and the $O(||\theta^{n+1} - \theta^n||^3$ term represents higher-order terms that can be ignored as the step size approaches zero.

For a stationary point to be a **local minimum**, the second-order term in this expansion (i.e., the term involving the Hessian matrix) must be **positive-definite**, i.e., all of its eigenvalues must be **positive**. For a stationary point to be a **local maximum**, the second-order term must be **negative-definite**, i.e., all of its eigenvalues must be **negative**. If the **Hessian matrix has both positive and negative eigenvalues**, the stationary point is **a saddle point**.

3. To account for the circularity of the predicted angle in the loss function, we can use the circular mean absolute error (CMAE) loss function. The CMAE loss function considers the circular nature of angles by calculating the absolute difference between the predicted and actual angles along the shortest path on a circle. The formula for the CMAE loss function is:

$$CMAE = |sin(\frac{y_{pred} - y_{true}}{360 \cdot 2\pi})|$$

where y_pred is the predicted angle and y_true is the true angle. This formula calculates the absolute difference between the predicted and true angles along the shortest path on a circle, by first converting the angles to radians, then dividing by 360 to obtain a fraction of a full circle, and finally multiplying by 2π to convert to radians. The sin function is then applied to this fraction to ensure that the difference is always positive.

In programmable mostly differentiable pseudo-code, the CMAE loss function can be written as:

```python
def circular_loss(prediction, target):
    """
    Calculates circular loss between two angles in degrees.
    """
    # Convert scalar float values to torch.Tensor objects
    prediction = torch.tensor(prediction)
    target = torch.tensor(target)

    diff = torch.abs(prediction - target)
    diff = torch.where(diff > 180, 360 - diff, diff)
    loss = torch.mean(torch.square(diff))
    return loss
```

The function first calculates the absolute difference between the prediction and target angles using torch.abs(). Then, it uses torch.where() to check if the absolute difference is greater than 180 degrees. If it is, it subtracts the absolute difference from 360 degrees, otherwise, it keeps the absolute difference unchanged. This step accounts for the circular nature of angles, where angles greater than 180 degrees wrap around to the other side of the circle.

Next, the function calculates the mean of the squared differences using torch.mean(torch.square(diff)), which represents the circular loss between the prediction and target angles. Finally, the calculated circular loss is returned as the output of the function.

a. Using the chain rule, first we can first take the partial derivative of f with respect to its first argument $(x + y)$, which is 1. Then, we can take the partial derivative of the first argument with respect to x, which is 1. Second, we can take the partial derivative of the second argument 2x with respect to x, which is 2. Finally, we can take the partial derivative of the third argument z with respect to x, which is 0. Putting it all together, we get:

Denote by u, v, w respectively the following functions $u(x, y) = x + y, v(x) = 2x, w(z) = z$

Then we have:

$$f(x + y, 2x, z) = f(u(x, y), v(x), w(z))$$

For the functions u, v, w we have, by applying the chain rule on each one of the functions with respect to x:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u} \cdot \frac{\partial u}{\partial x} + \frac{\partial f}{\partial v} \cdot \frac{\partial v}{\partial x} + \frac{\partial f}{\partial w} \cdot \frac{\partial w}{\partial x}$$

Calculating each partial derivatives yields:

$$\frac{\partial u}{\partial x} = 1, \qquad \frac{\partial v}{\partial x} = 2, \qquad \frac{\partial w}{\partial x} = 0$$

Putting it all together, we get:

$$\frac{\partial f(x + y, 2x, z)}{\partial x} = \frac{\partial f}{\partial u} + 2\frac{\partial f}{\partial v}$$

B. First, denote the derivative with respect to x of first order as follows:

$$\frac{\partial}{\partial x} f_n(x) = f_n'(x)$$

Denote $f_n(x) = u$, for the first order derivative with respect to x of $f_{n-1}(x)$ we have:

$$\frac{\partial}{\partial x} f_{n-1}(u) = f_{n-1}'(u) \cdot \frac{\partial}{\partial x} f_n(x) = f_{n-1}'\big(f_n(x)\big) \cdot f'_n(x)$$

Applying the chain rule over and over iteratively we finally get:

$$\frac{\partial\left(f_1\left(f_2(\dots f_n(x))\right)\right)}{\partial x}$$

Using the chain rule, we can take the derivative of f_1 with respect to its input, and then the derivative of f_2 with respect to its input, and so on up to f_n. To simplify notation, we can define $g_i(x) = f_i(f_{i+1}(\dots f_n(x))), so\ that\ f_1(x) = g_1(x)$. Then, we have:

$$\frac{\partial\left(f_1\left(f_2(\dots f_n(x))\right)\right)}{\partial x} = f_1'\left(f_2\left(f_3(\dots f_n(x))\right)\right) \cdot f_2'\left(f_3(\dots f_n(x))\right) \cdot \dots \cdot f_{n-1}'\big(f_n(x)\big) \cdot f'_n(x)$$

c. First, denote the derivative of first order as follows:

$$\frac{\partial}{\partial x} f_n(x) = f_n'(x), u = f(x)$$

Using notations from the previous section 4b we get:

$$\frac{\partial}{\partial x} f_{n-1}(x,u) = \frac{\partial}{\partial x} f_{n-1}(x, f_n(x)) + \frac{\partial}{\partial u} f_{n-1}(x,u) \cdot \frac{\partial}{\partial x} f_n(x)$$

In other terms we can switch according to $\frac{\partial}{\partial x} f_n(x) = f_n'(x)$ and get:

$$\frac{\partial}{\partial x} f_{n-1}\big(x, f_n(x)\big) = f_{n-1}'(x,u) + f_n'(x)\ \frac{\partial}{\partial u} f_{n-1}(x,u)$$

If we operate iteratively, we will have:

$$f_1{}'(x, f_2\left(x, f_3\left(\dots\left(f_{n-1}\big(x, f_n(x)\big)\right)\right)\right) + \sum_{j=2}^{n} f_j(x, f_{j+1}(\dots f_n(x)))$$

$$\cdot \prod_{k=j+1}^{n-1} \frac{\partial}{\partial u_k} f_k\left(x, f_{k+1}(\dots f_n(x))\right)$$

$$for\colon \quad u_k = f_{k+1}(x, f_{k+2}(\dots f_n(x) \dots))$$

d. Define $u(x) = x + g\big(x + h(x)\big)$ from the chain rule we have:

$$\frac{\partial u}{\partial x} = 1 + g'(x + h(x)) \cdot (1 + h'(x))$$

Overall, we have:

$$\frac{\partial\left(f\left(x+g(x+h(x))\right)\right)}{\partial x}=\frac{\partial f}{\partial u}\cdot\frac{\partial u}{\partial x}=$$

$$=f'\left(x+g(x+h(x))\right)\cdot(1+g'(x+h(x))\cdot(1+h'(x))$$

PRACTICAL QUESTIONS – ANSWERS

1. (I) Selecting the latent space dimension can have a great impact on the Autoencoders network overall performance. We can conclude after analyzing the results that as we increase the latent space dimension, we get better results – a lower MSE Loss value at the end of the model training session. But we need to remember that we wish to store the encoded images in a way that they will be encoded in the most compact version of latent code dimension e.g. with a small number of pixels representing the encoded image.

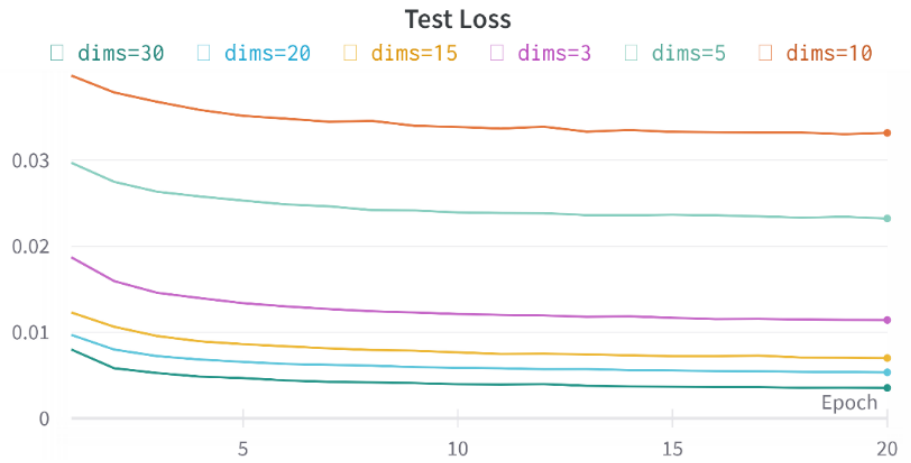
Generally enlarging the Latent Code Space vector can make the encoder classify the images more accurately and decode the images with better results, while reducing the Latent Code Space vector can lead to a more compact encoding of the images, but may result in a loss of data and key features and patterns in the reconstructed image. If we set a large Latent Code vector the model will more likely to identify feature and reconstruct the images successfully, with a small loss value. But if the train dataset will be smaller, using large Latent Code the model will be in overfitting and bigger test loss value. In the case of much compact train data set we will strive to use smaller Latent Code vector to avoid overfitting.

I tried to build different Autoencoder networks which consists of an Encoder and a Decoder, the parameters I used are:

- $MSE\ Loss\ Criterion.$
- $AdamW\ Optimizer.$
- $2\ Convolutional\ layers\ and\ one\ FC\ Layer\ in\ both\ the\ Encoder\ and\ the\ Decoder.$
- $Capacity\ of\ 128\ Channels.$
- $Learning\ Rate\ =\ 0.003.$

The overall reconstruction error presented in the graph below, for each epoch we calculate the MSE Loss function, for each one of the Latent Code values:

$$Latent\ Code\ Dimensions\ \epsilon\ \{3, 5, 10, 15, 20, 30\}$$



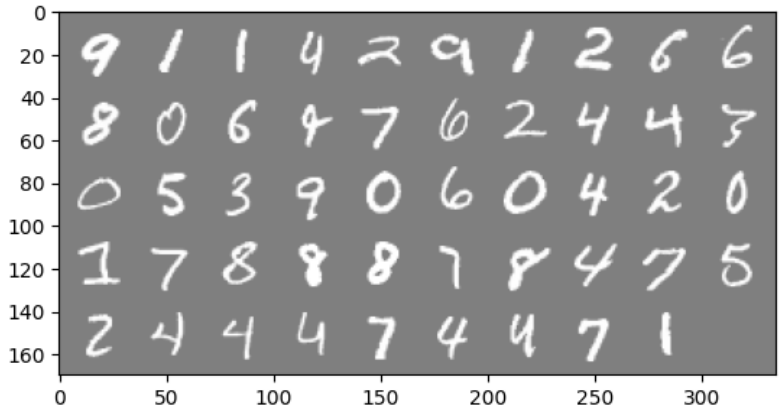
Final Results

Name (7 selected)	dataset	learning_rate	AutoEncoder Parameters	Test Loss	Train Loss	Latent Code	Epoch
dims=30	MNIST	0.003	1818399	0.003427	0.00302	30	20
dims=20	MNIST	0.003	1567509	0.005308	0.004699	20	20
dims=15	MNIST	0.003	1442064	0.007143	0.006411	15	20
dims=10	MNIST	0.003	1316619	0.01172	0.01086	10	20
dims=8	MNIST	0.003	1266441	0.01458	0.01375	8	20
dims=5	MNIST	0.003	1191174	0.02338	0.02234	5	20
dims=3	MNIST	0.003	1140996	0.03338	0.03166	3	20

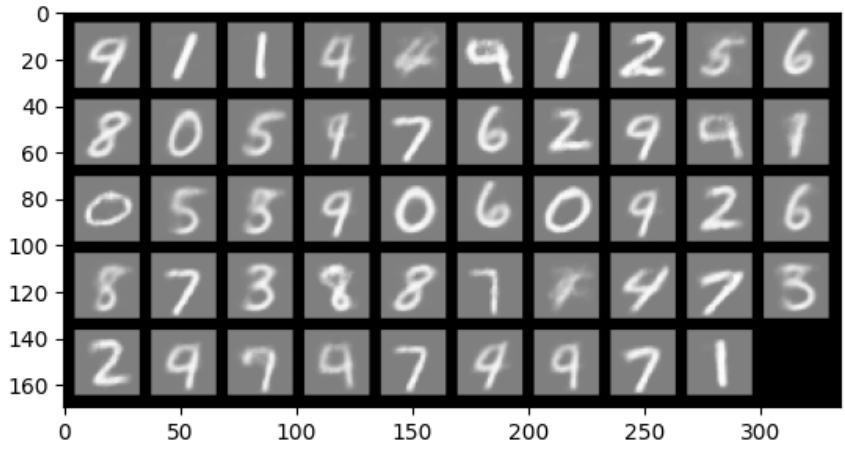
For each one of the Latent Code Dimensions, I will present the results on the Test dataset overall reconstruction and analyze them.

$$Latent\ Code\ Dimension = 3,$$

$$Original\ Images$$

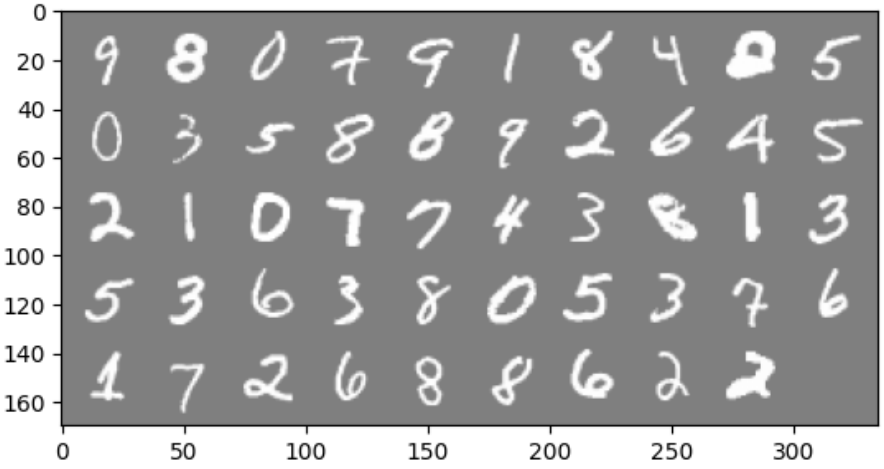


$$Reconstructed\ Images$$

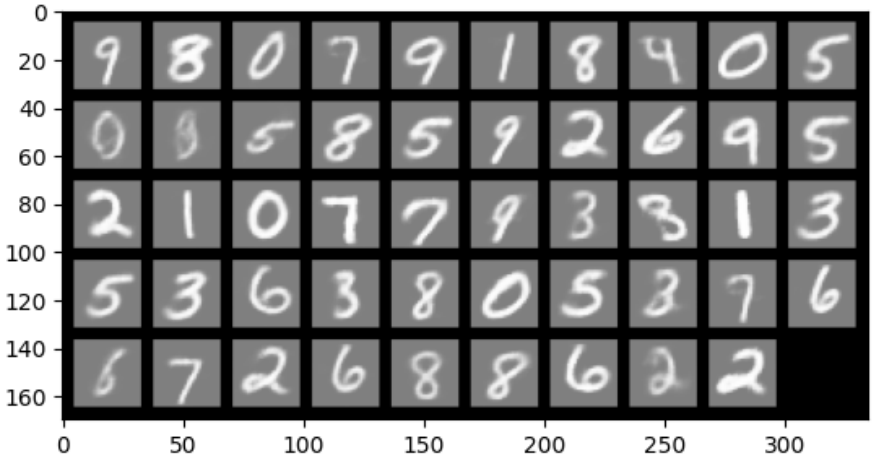


Latent Code Dimension = 5

Original Images

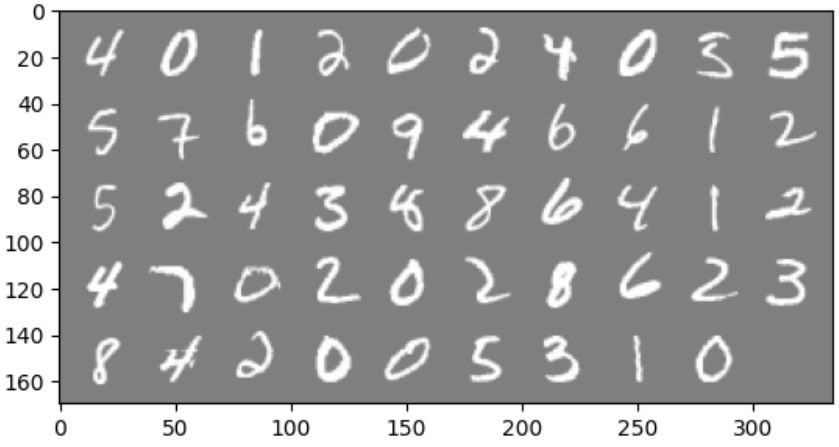


Reconstructed Images

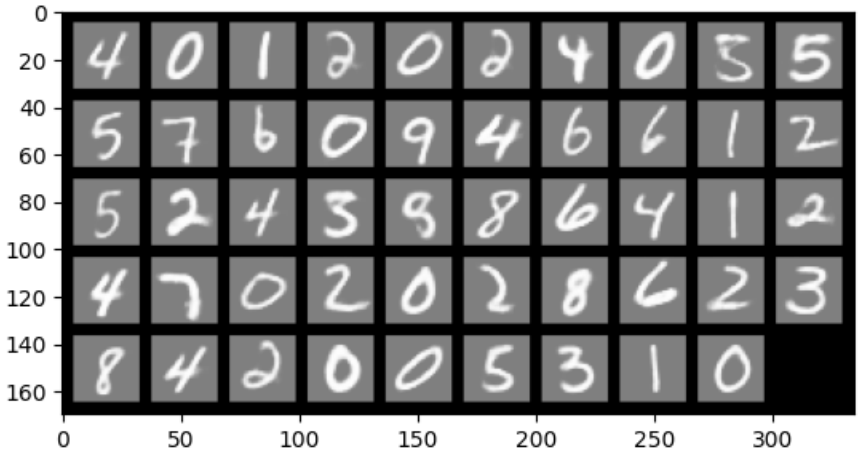


Latent Code Dimension = 10

Original Images

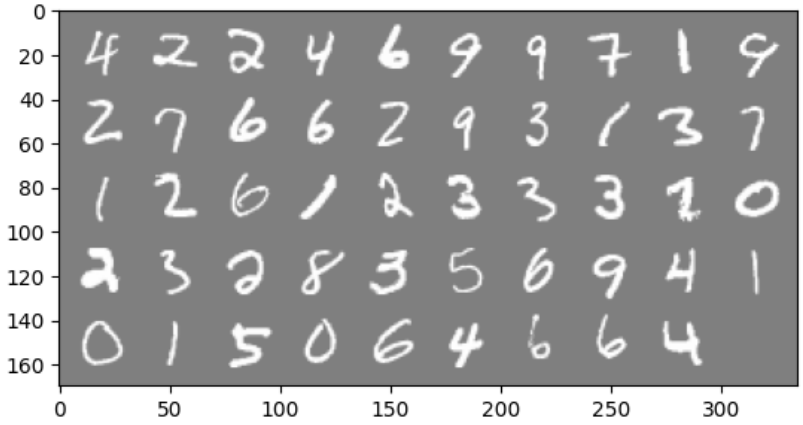


Reconstructed Images

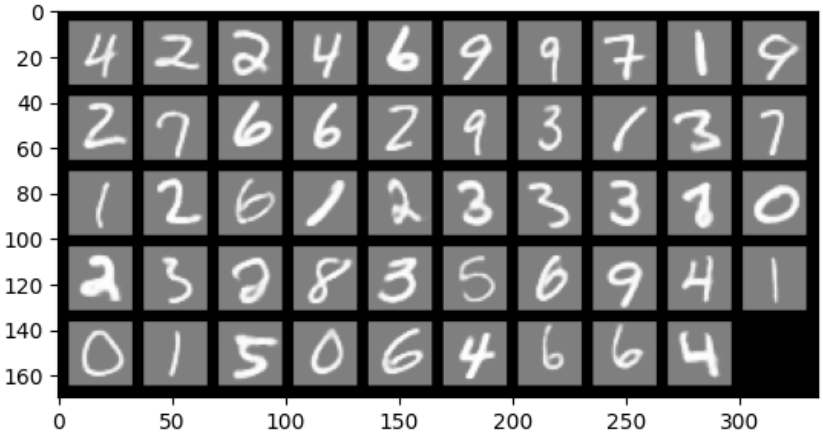


$Latent\ Code\ Dimension = 15$

Original Images

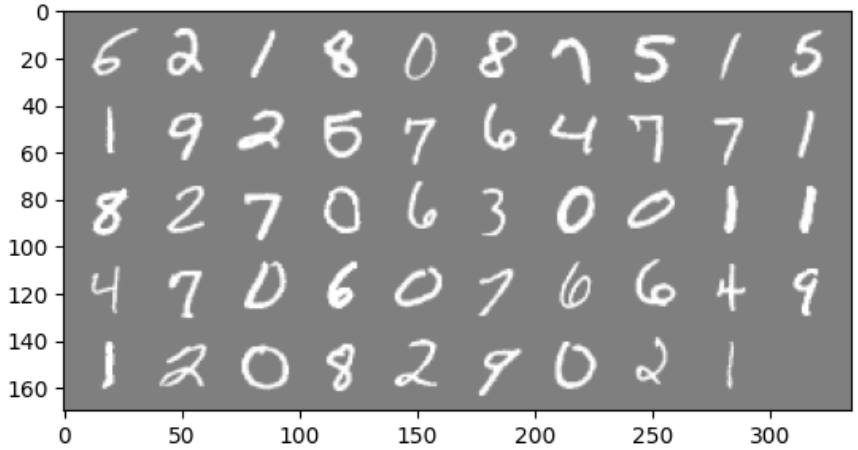


Reconstructed Images

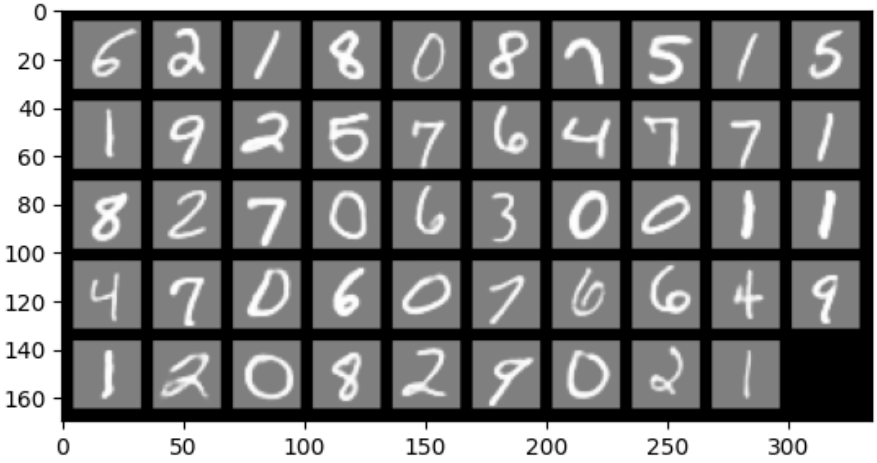


$$Latent\ Code\ Dimension = 30$$

Original Images



Reconstructed Images



1.(I) ANALYSIS

As we can spot that images reconstruction in lower Latent Code Space autoencoders model as 3,5 doesn't reconstruct the images fully and we get a blurry result due to the fact that we got a greater loss training these networks, so when we wish to reconstruct images from the test dataset we don't get good enough results.

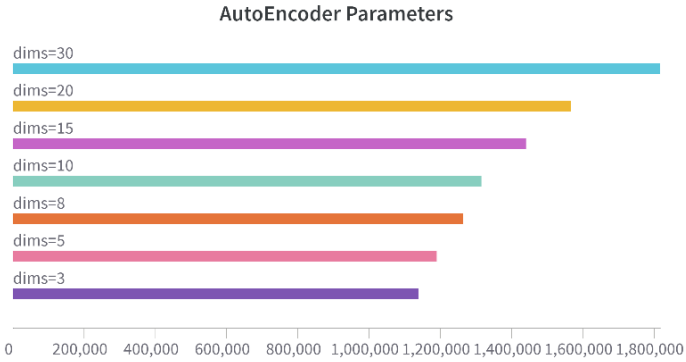
From the value $Latent\ Code\ Space = 10$ and greater values up until 30 the image reconstruction is good and sharp as we would like to have.

We can conclude that the optimal $Latent\ Code\ Space$ for this task is around 10 or 15 if we wish to keep the encoded images mapping to a compact enough Latent Code Space vector which will suffice our cause.

1. (II) In this section we will set the Latent Code to a fixed value – I chose $Latent\ Code\ Dimension = 15$ and the same architecture as in the previous section 1 (i).

But in this section, we will change the capacity of the AE given initial Channels given to the modules of the Encoder and the Decoder.

Overall Learned Parameters:



Reconstruction error VS epoch graphs with:

$$Channels\ Capacity \in \{128, 64, 32, 16, 8, 4\}$$

As we can spot, there is a direct connection between the number of initial Channels we provide to the first convolutional Layer which according to my code influences also the second convolutional layer and the FC Layer number of learned parameters as we can see in the diagram presented below:

2. Now, we will use the same trained Autoencoder architecture as in the previous questions to test the interpolation process of 2 images.

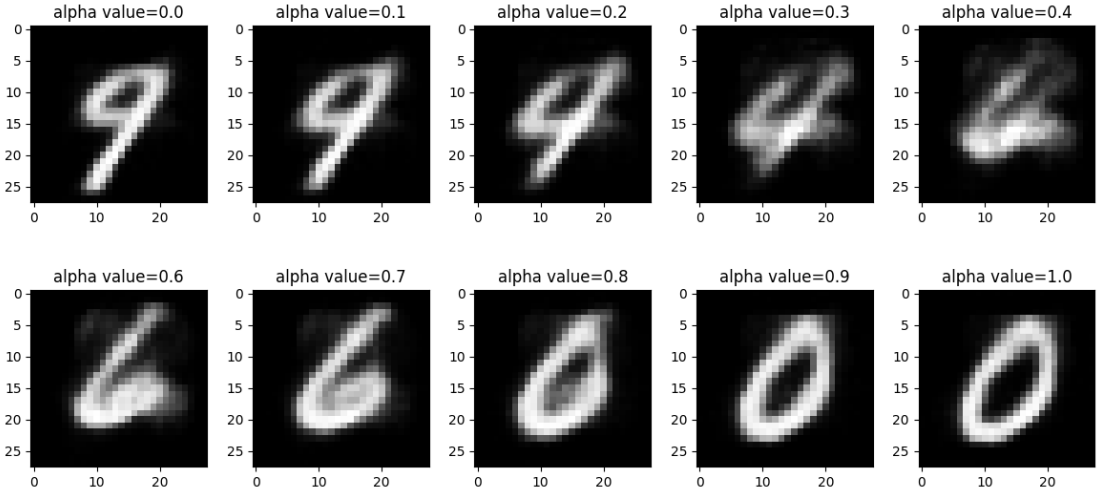
Given 2 images I_1, I_2 of different digits, we will calculate the interpolation defined as:

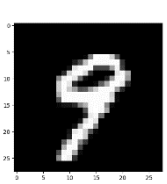
$$D\Big((E(I_1)\cdot\alpha) + \big(E(I_2)\cdot(1-\alpha)\big)\Big), \qquad \alpha \in [0,1]$$

Where D denotes the decoder and E denotes the encoder.

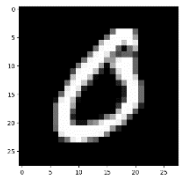
(I) The resulting images we got, using different values of $\alpha \in [0,1]$ values in steps of 0.1 each time and presented next to each other where $\alpha = 0, \alpha = 1$ represents I_1, I_2 accordingly:

$$Latent\ Code\ Space = 3$$



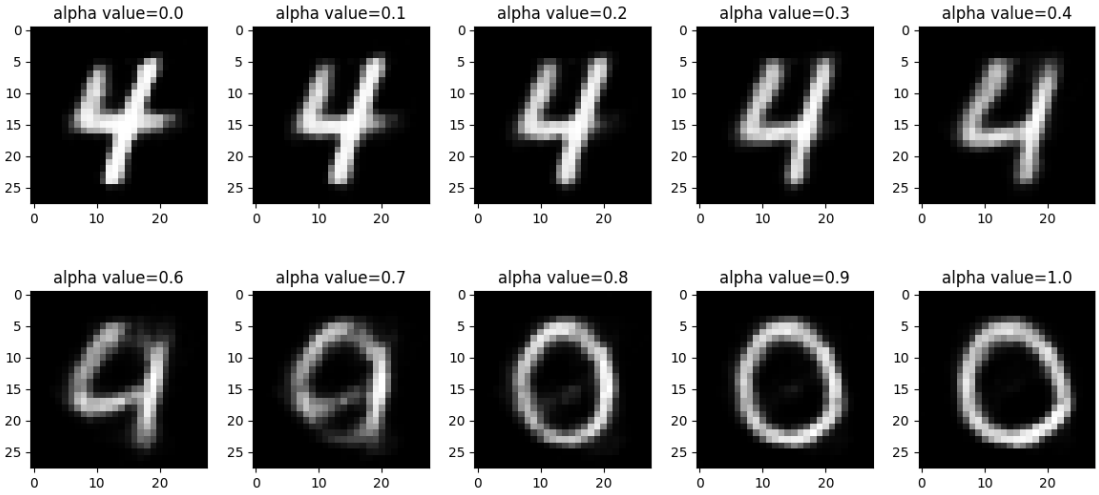


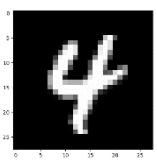
$Image\ 1$

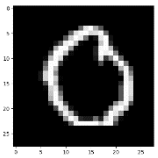


$Image\ 2$

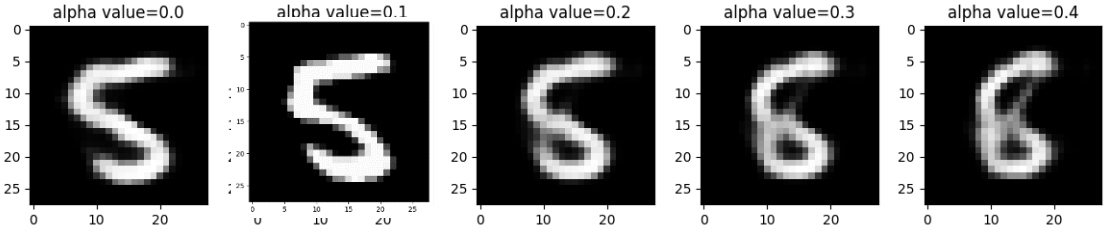
$$Latent\ Code\ Space = 5$$

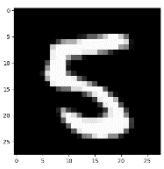




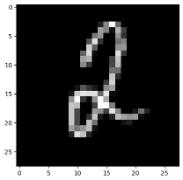


$Latent\ Code\ Space = 10$



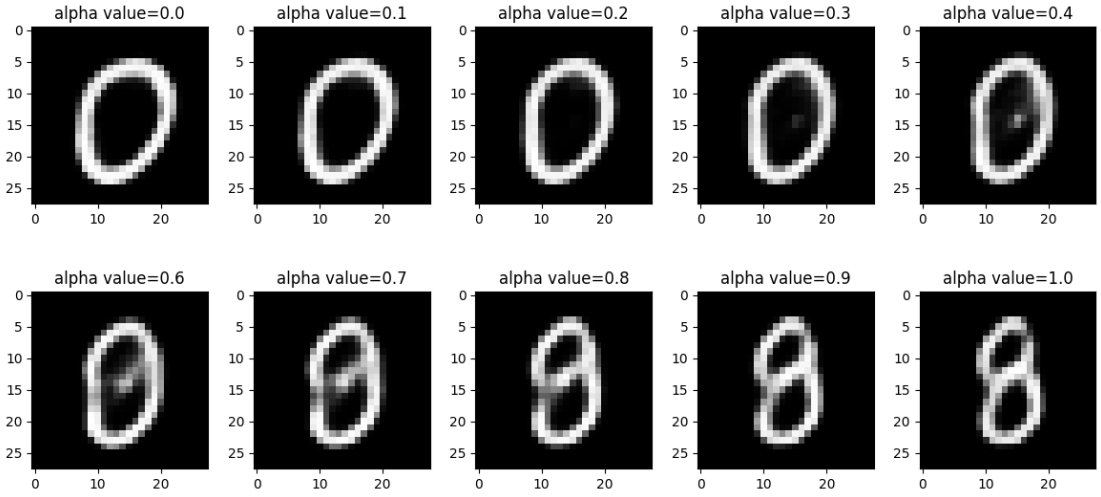


$Image\ 1$



$Image\ 2$

$Latent\ Code\ Space = 15$



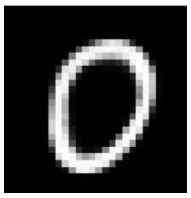


Image 1

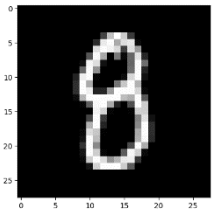
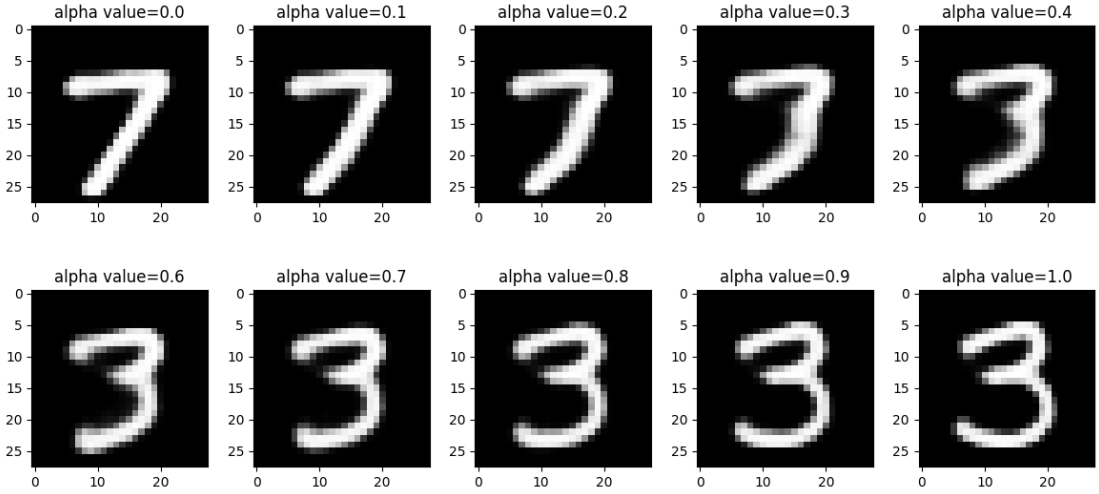


Image 2

$$Latent\ Code\ Space = 20$$



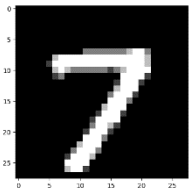


Image 1

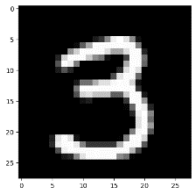
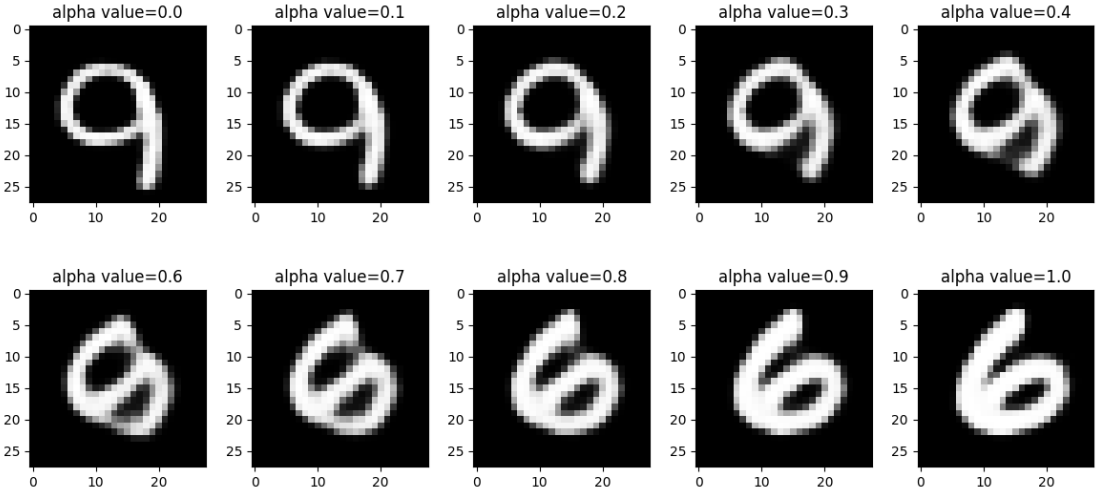
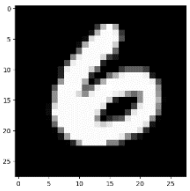


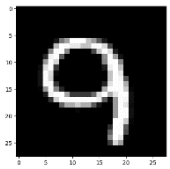
Image 2



$$Latent\ Code\ Space = 30$$



$Image\ 1$



$Image\ 2$

ANALYSIS

As we can spot when α value is getting close to the value of $\alpha \approx 0.5$ and in the range of $0.4 - 0.6$ especially the decoder returns an interpolated and distorted image which is a combination of the 2 images I_1, I_2 which seems very blurry and displays an unclear image. We get these results since $\alpha = 0.5$ is an intermediate value in which we get the same contribution from both images I_1, I_2 in the interpolation process and values that are close to 0.5 are more likely to produce mixed and distorted images of digits. We get distorted and unclear reconstructed digits mainly if the given images and digits are very different in their patterns.

Further analysis discovers that there is a direct connection between the Latent Code Space value and the reconstructed interpolated images. As we increase the Latent Code Space, we get images of digits a lot sharper and clearer, this is due to the fact that a larger Latent Code Space dimension can hold more encoded representations in number and will keep more complex and diverse patterns of the given digits dataset of MNIST.

We can spot that in very small $Latent\ Code\ Space = 5$ in the interpolation process the decoder confused between some of the digits. The confusion was between $0,9$ that were the original digits of I_1, I_2 in correspondence and the digits 6 and 4 which the decoder misinterpreted because of the low Latent Code Space dimension as explained above.

We should be aware of the fact that increasing the Latent Code Space dimension too much can produce an overfitting result, so we should use an intermediate value of dimension.

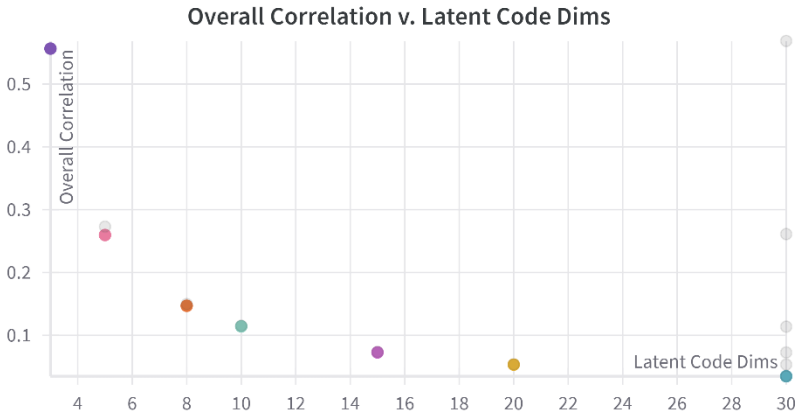
3. (I) In this section we will investigate the connection between the decrease in Latent Code Space and the overall correlation between different pairs of coordinates in the Latent Code vector. For each value of Latent Code Dimension, we will calculate the value of the Pearson correlation coefficients based on a 10,000 images and use all the encoded images Pearson correlation coefficients to calculate the overall correlation represented as single value.

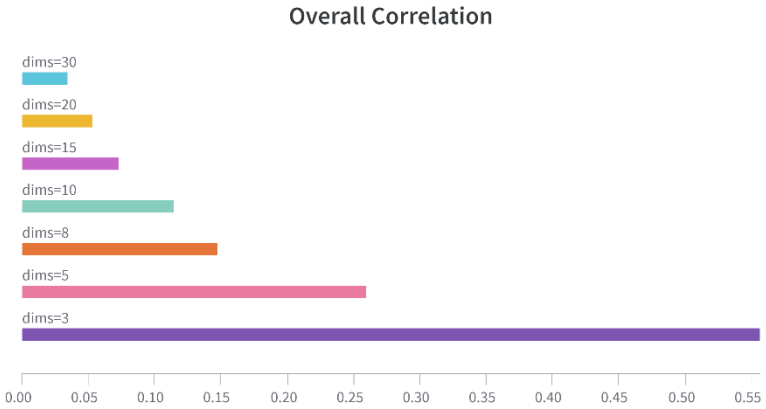
To do so we will calculate the correlation matrix of the coordinates and values stored in the Latent Code Space vector and it will be filled with values in the range of $Corr \in [-1,1]$, so we will perform absolute value because we only want the magnitude of the correlation, and sum over all the coordinates which are not on the main diagonal since we want only pairs of odd coordinates. Then we will normalize the correlation by dividing by the number of counted pairs of coordinates. High value of overall correlation indicates a strong connection between the coordinates and low value indicates a weaker connection between coordinates.

To explore the results of the change of Overall Correlation between coordinates as a result of Latent Code dimension change, we will take varying values of Latent Code Space dimensions:

$$Latent\ Code\ Space \in \{3, 5, 8, 10, 15, 20, 30\}$$

The results are presented in the graphs below:





As expected, we can spot a decrease in the overall correlation between coordinates of the Latent Code vector when we increase the Latent Code vector's dimension.

This is reasonable due to several reasons:

Increased encoding capacity: As the size of the latent code space vector (latent space dimension, d) increases, the encoding capacity of the autoencoder also increases. This means that the autoencoder may have more room to encode redundant or irrelevant information, leading to a decrease in overall correlation of pairs of coordinates in the latent code space vector. With higher encoding capacity, the autoencoder may not be forced to capture the most salient and informative features in the data, resulting in a less meaningful and correlated latent code space.

Increased complexity: With higher latent space dimensions, the complexity of the latent code space also increases. This means that the interactions and dependencies between different coordinates in the latent code space vector may become more complex, making it harder to maintain overall correlation. As the Latent Code Space dimension increases, the

likelihood of capturing redundant or irrelevant information in different coordinates of the latent code space also increases, leading to a decrease in overall correlation.

We can spot that the overall correlation value decreases almost exponentially from the second graph and a reasonable value to maintain high correlation between the coordinates of the Latent Code Space vector is 15 which has an intermediate value of overall correlation.

4. (I) In this section we will use Transfer Learning technique to check the overall accuracy of the autoencoder in the task of classifying digits to its correct labels in the MNIST test dataset. We will first train an autoencoder and use it as a pretrained autoencoder model. We will add a small MLP network sequential to the autoencoder that will help us in the digit classification task.

We will compare between the performance of the Transfer Learning and Full Network optimizations:

- Transfer Learning: Use the pretrained autoencoder, exclude the autoencoder's weights from the training optimization and include just the MLP weights in the training.
- Full Network training: Use the pretrained autoencoder, include both autoencoder's weights and MLP weights in the training optimization.

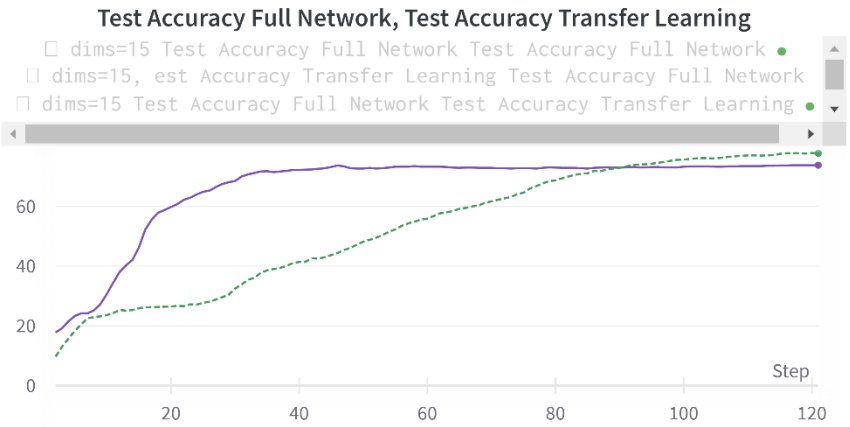
We will run the training of both the Transfer Learning network and the Full Network (Autoencoder + MLP Optimization) on a subset of 100 random images from the train dataset.

Then we will calculate alongside the training session the progress in the Test Accuracy and Loss of the classification of the digits in the test dataset using a few thousands of images.

We will set the following parameters:

- $Latent\ Code\ Space\ =\ 15 -$
  $Provides\ Good\ Overall\ Correlation\ and\ low\ loss\ values.$
- $Optimizer -\ AdamW.$
- $Loss\ Function\ \text{-}\ CrossEntropyLoss - Good\ for\ classification\ tasks.$
- $Learning\ Rate = 0.003$
- $MLP\ Network\ sequantial\ to\ the\ Autoencoder -$ with 3 FC Layers and ReLU activation function, expanding the dimensions from 10 to 64, 64 to 32 and 32 back to Latent Vector of size 10.

The graphs of Test Accuracy VS Epochs from both training sessions:



As expected, from the graph of Test Accuracy VS Epoch we can clearly see that the approach of Transfer Learning network produced better results in performance and has higher Test Accuracy rate.

This is a reasonable result since the Transfer Learning method network uses the pretrained encoder to identify features and patterns from the early training session given images from test dataset. The pretrained encoder has learned a big amount of data, features and complex patterns from the train dataset and it is reflected in the classification task on the test dataset which performs better and has higher Test Accuracy. Given a small train dataset and training the small MLP network only (and not the pretrained encoder) the accuracy of the Transfer Learning method network is higher than full network training.

In the second method, when we train both Encoder and MLP network altogether, while allowing the optimization of the encoder weights, it may seem like a better solution, but in fact because of the small amount of data from the train dataset given to the network to train on, we may get overfitting result and get lower accuracy since the Full Network's encoder has no sufficient training on the train data set and therefore the Test Accuracy for the Full Network training method is lower.