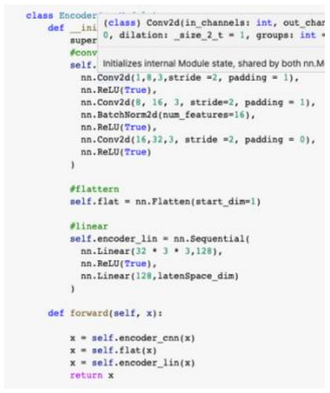
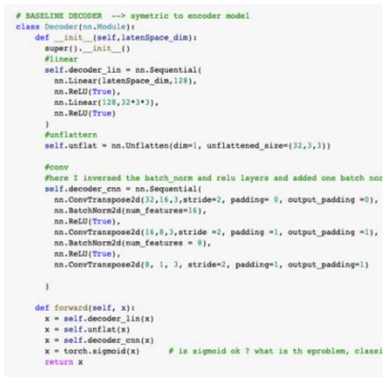
Color code.
Quotes
Expectations:
Results
Unanswered questions
I left some unanswered questions I had. Hope you could help with these.
In some of the figures I added the test and train losses.

Auto_Encoding:

This is the baseline Auto Encoder I started with. The decoder is based on the baseline model from the previous exercise (with the addition of Batch Normalization layers) as well as from some models I found online. The decoder is symmetric to the encoder. Also, I noted that the loss seems to converge quite fast as the number of epochs increases. At around 7 epoch the loss curve stabilizes. I therefore allowed myself to train for 20 epochs. With that, take into consideration that I have a batch size of 128 and in most of the exercise one of 256.



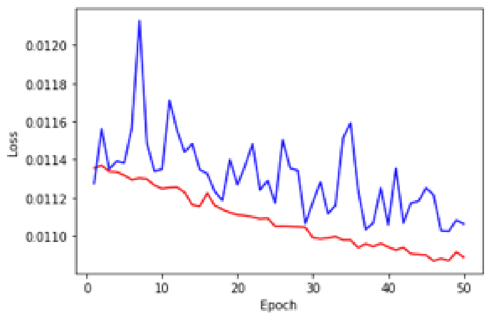
```
class Encoder
    def __ini   (class) Conv2d(in_channels: int, out_char
                0, dilation: _size_2_t = 1, groups: int =
        super
        #conv
        self.   Initializes internal Module state, shared by both nn.M
          nn.Conv2d(1,8,3,stride =2, padding = 1),
          nn.ReLU(True),
          nn.Conv2d(8, 16, 3, stride=2, padding = 1),
          nn.BatchNorm2d(num_features=16),
          nn.ReLU(True),
          nn.Conv2d(16,32,3, stride =2, padding = 0),
          nn.ReLU(True)
        )

        #flattern
        self.flat = nn.Flatten(start_dim=1)

        #linear
        self.encoder_lin = nn.Sequential(
          nn.Linear(32 * 3 * 3,128),
          nn.ReLU(True),
          nn.Linear(128,latenSpace_dim)
        )

    def forward(self, x):

        x = self.encoder_cnn(x)
        x = self.flat(x)
        x = self.encoder_lin(x)
        return x
```



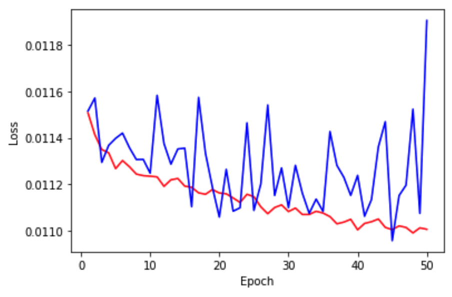
```
# BASELINE DECODER  --> symetric to encoder model
class Decoder(nn.Module):
    def __init__(self,latenSpace_dim):
      super().__init__()
      #linear
      self.decoder_lin = nn.Sequential(
        nn.Linear(latenSpace_dim,128),
        nn.ReLU(True),
        nn.Linear(128,32*3*3),
        nn.ReLU(True)
      )
      #unflattern
      self.unflat = nn.Unflatten(dim=1, unflattened_size=(32,3,3))

      #conv
      #here I inversed the batch_norm and relu layers and added one batch nor
      self.decoder_cnn = nn.Sequential(
        nn.ConvTranspose2d(32,16,3,stride=2, padding= 0, output_padding =0),
        nn.BatchNorm2d(num_features=16),
        nn.ReLU(True),
        nn.ConvTranspose2d(16,8,3,stride =2, padding =1, output_padding =1),
        nn.BatchNorm2d(num_features = 8),
        nn.ReLU(True),
        nn.ConvTranspose2d(8, 1, 3, stride=2, padding=1, output_padding=1)

      )

    def forward(self, x):
      x = self.decoder_lin(x)
      x = self.unflat(x)
      x = self.decoder_cnn(x)
      x = torch.sigmoid(x)      # is sigmoid ok ? what is th eproblem, classi
      return x
```

I decided to go with larger batch sizes and increase the learning rate by an equal factor. (see https://arxiv.org/pdf/1706.02677.pdf). Thus, I decided for the very beginning to check different batch sizes and learning rate combinations. Here are the results.

Batch size: 256: (lr =0.001)

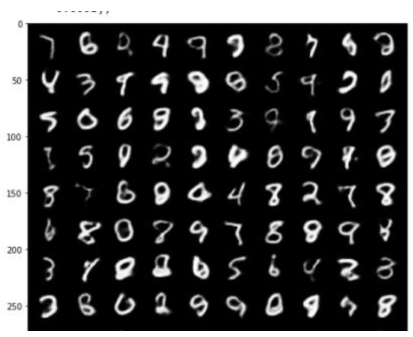


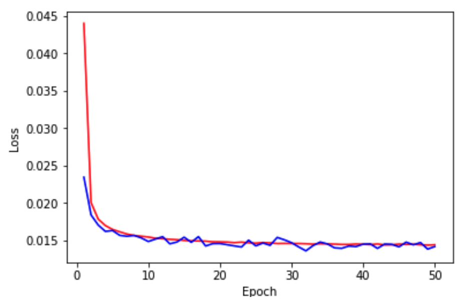
Batch size: 128, lr= 0.001



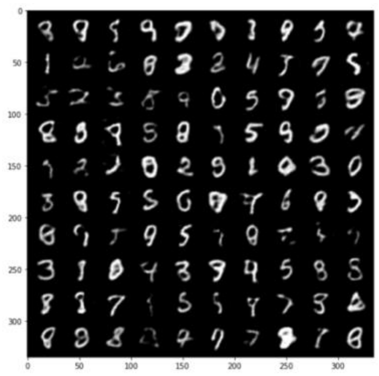
- Although the test loss is low, note that the curve fluctuates. This is a clear sign that the weights converge and diverge, usually as a result of a high learning rate. Nonetheless here the learning rate is not high and the scaling reveals that the fluctuation are quite minimal. The question is whether we always want to compare the test loss against the test loss curve or is there also an objective independent measure on the fluctuations of the test loss? We will experiment with the learning rate to see if we can get a smoother curve. (what are the importance of a smooth curve?)

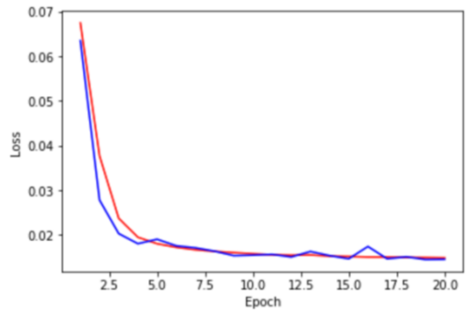
batch size = 128, laten dimension = 10, lr = 0.008 (according to paper)



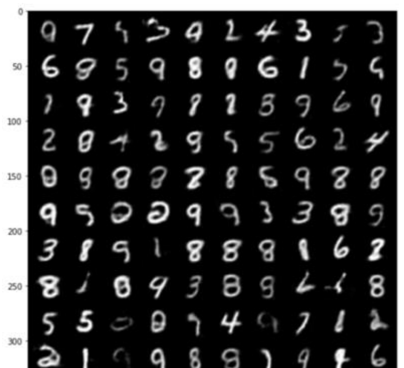


batch size = 256, laten dimension = 12, lr =0.016(according to paper)





batch size =256, laten dimension = 6, lr=0.016(according to paper)

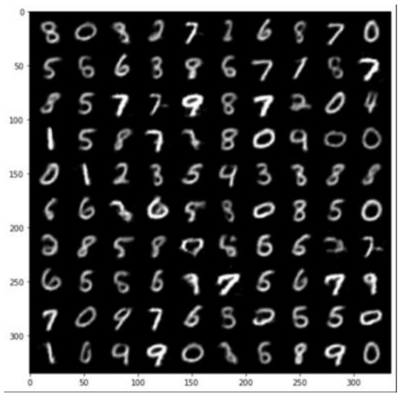


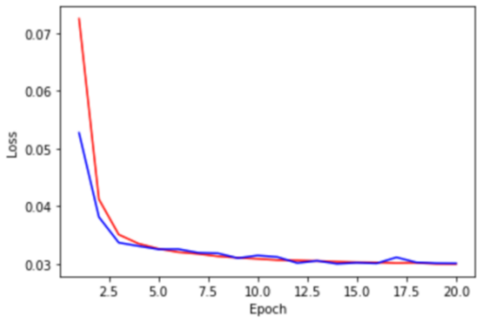
Test and Train loss final value.

```
0.023361295604325354
```

```
0.023504116619005
```

batch size =256/ laten dimension = 4/ lr=0.016(according to paper)

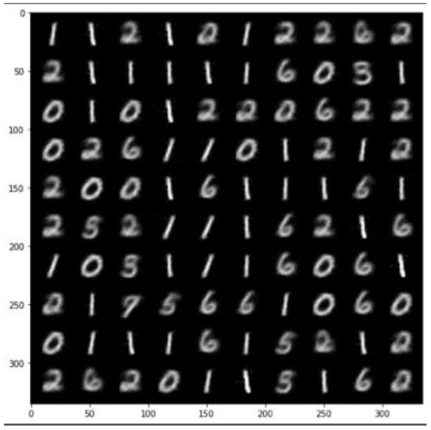


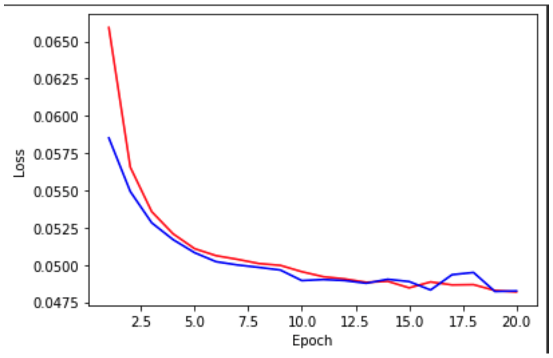


```
0.029955046988548117
0.030066601699218154
```

I decided to use the baseline network (see first page) and play with the dimensions of the laten space. Here are the results.

- **D=1**

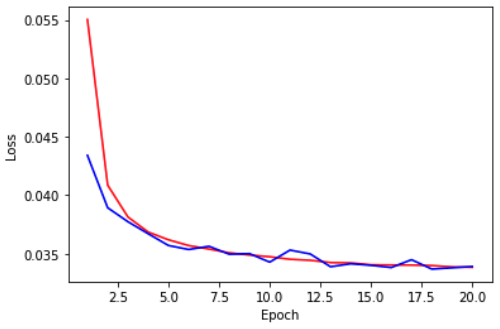


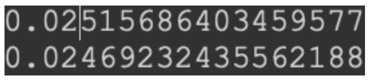


```
0: 9.871666666666666%
1: 11.236666666666666%
2: 9.93%
3: 10.218333333333334%
4: 9.736666666666666%
5: 9.035%
6: 9.863333333333333%
7: 10.441666666666666%
8: 9.751666666666667%
9: 9.915000000000001%
```

Here we can see an interesting pattern. At D=1 as well as for D=3 we can see that although the loss curve doesn't show any signs of overfitting, we can clearly see that the network didn't learn well. Specifically, we can see that the network fixed upon reoccurring configurations that reduce the loss. I checked the distribution of the data to see if there is any correlation with the choice of pattern fixing of the network. I found that there is. For example, we can see that the data consists more of "ones" than any other number which is consistent with the results we got. See above.

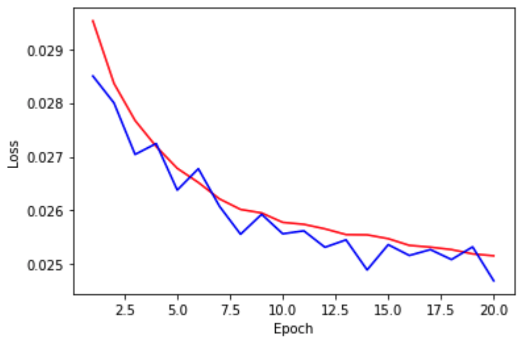
D=3

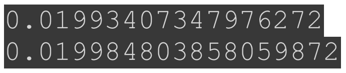




```
0.02515686403459577
0.02469232435562188
```

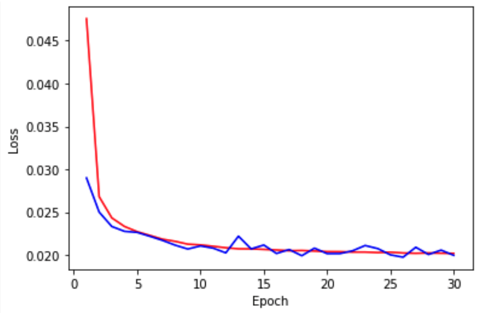
D=5

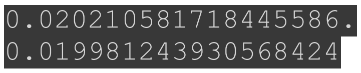




```
0.01993407347976272
0.019984803858059872
```

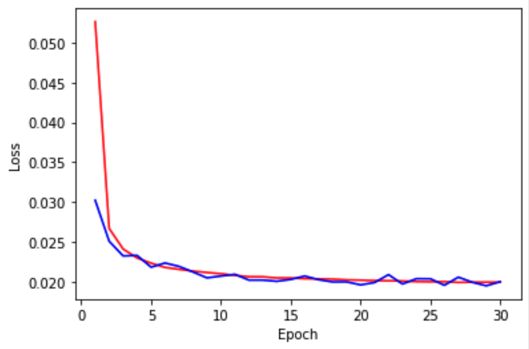
D=7

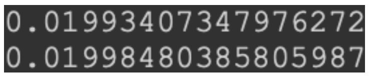




```
0.020210581718445586.
0.019981243930568424
```

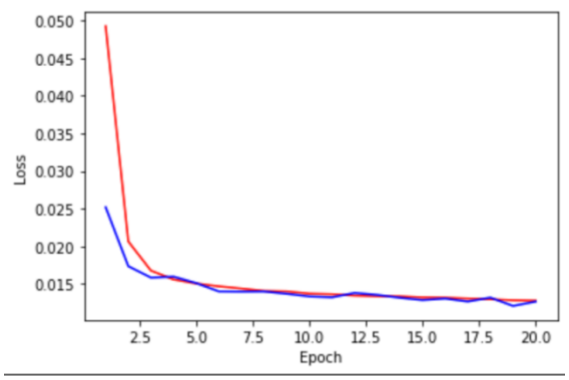
D =10

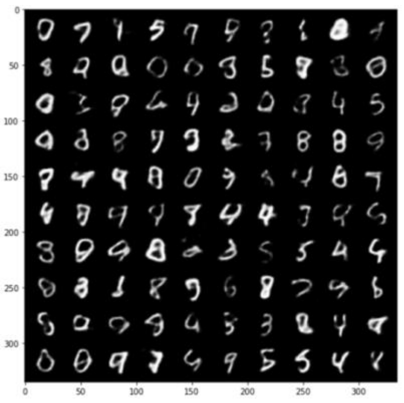




```
0.01993407347976272
0.01998480385805987
```

D =13

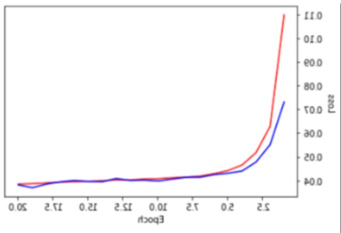




```
0.012801934603546093
0.012667273395235025
```

First I noticed that as we increase the dimension of the laten space over a certain point, we receive good test loses values yet with an output which is quite bad. Second, I noticed that as we decrease the laten dimension below a certain point the encoder is not able to create a meaningful enough laten space to allow the decoder to achieve an acceptable loss.
Here we understand the importance of restricting the latent space as well as the limitation of doing so. It will on one hand force the model to learn underlying features that can be encoded onto a smaller dimension, On the other hand, if taken too far, the restriction will impair the process. Is there a scale or doctrine for the projection of different types of data onto different laten spaces or laten spaces dimensions? Maybe training the network better would help achieving lower dimensions with acceptable true performance.

I surmise that the discrepancy between the real performance and the loss value lies in our choice of loss function (MSEloss). We will check same configuration with L1 loss function instead. Here are the results:

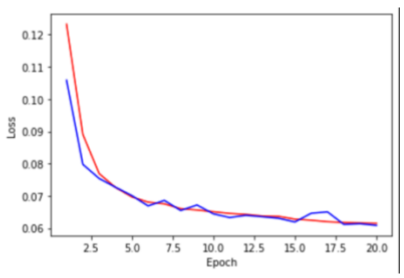


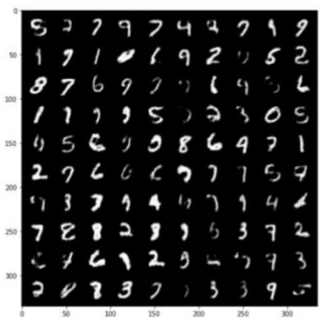
```
0.03849351965288109
```

We got an overall worse real performance than when we used the MSE loss. This shows the advantage of the MSE loss over L1 loss. However, these results do not allow us to answer the above since although the loss value did increase, it comes with a decrease in true performance as well.

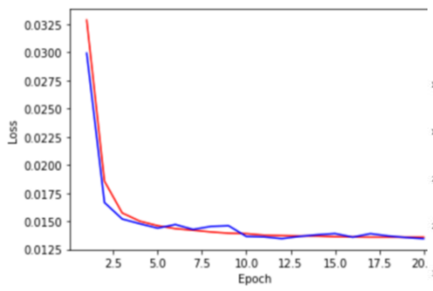
- Lastly, we will check one set of configuration where we keep a constant model and change only the loss functions.
  Model: batch size =256, laten dimension = 5/ lr=0.016

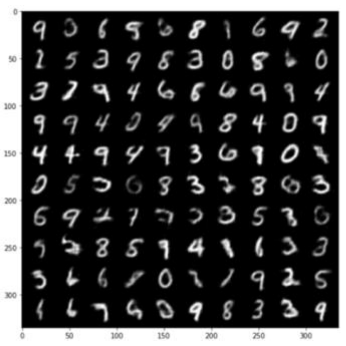
Loss function L1



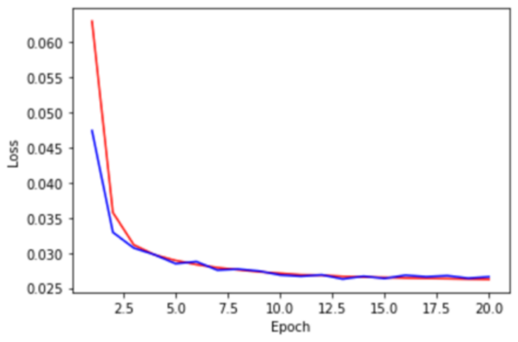


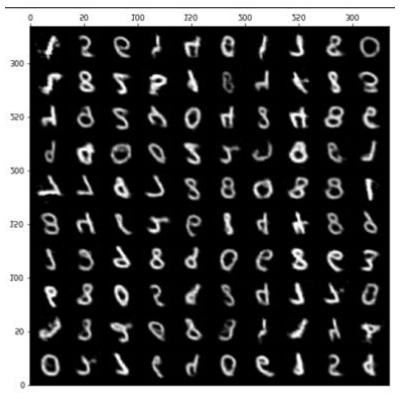
Loss function: HuberLoss





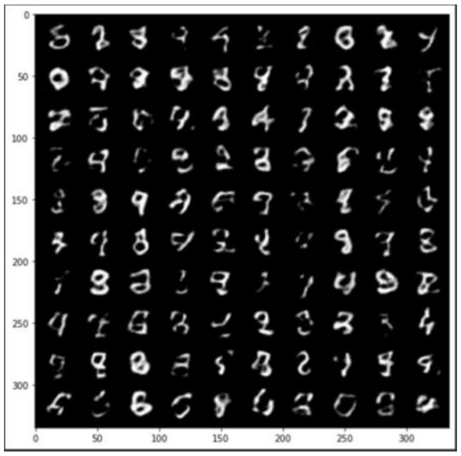
Loss function: MSELoss

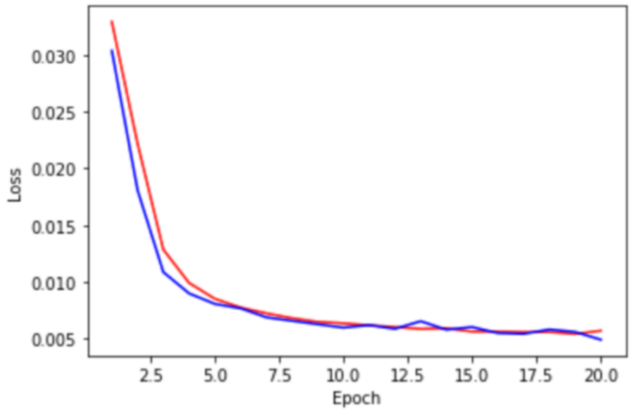




- **As we can see the Huber loss provides the best results.**

Lastly, I checked the same model (using the Huber Loss) yet with a laten space of dimension of 50



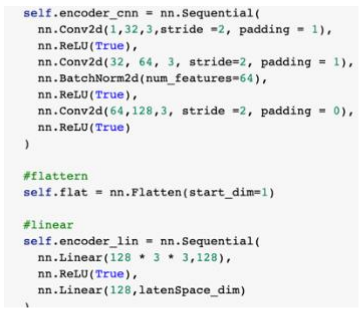


This is the same results as before for D=13 yet more extreme. We can clearly see the discrepancy between the very low-test loss value and the poor performance. Thus, my guess was wrong since here we used a different loss function altogether,

I took two different architectures I thought interesting from the previous exercise.

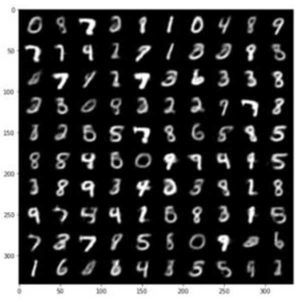
Fat-Man-Theory

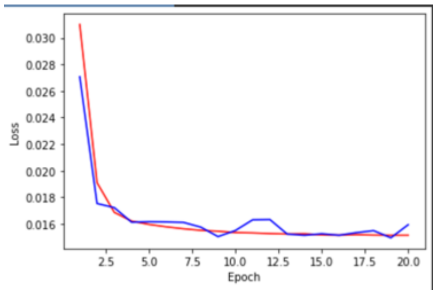
The fat man believes that by increasing the network size, the dimensionality reduction will be performed better. We will use the network I presented in the past exercise.



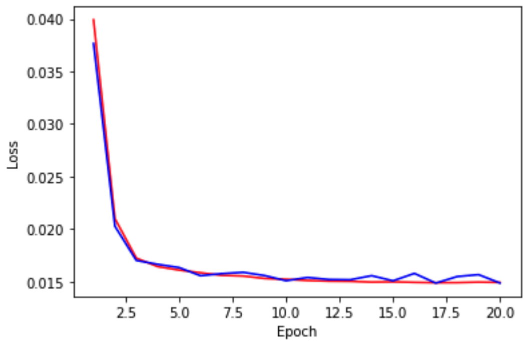
```
self.encoder_cnn = nn.Sequential(
  nn.Conv2d(1,32,3,stride =2, padding = 1),
  nn.ReLU(True),
  nn.Conv2d(32, 64, 3, stride=2, padding = 1),
  nn.BatchNorm2d(num_features=64),
  nn.ReLU(True),
  nn.Conv2d(64,128,3, stride =2, padding = 0),
  nn.ReLU(True)
)

#flattern
self.flat = nn.Flatten(start_dim=1)

#linear
self.encoder_lin = nn.Sequential(
  nn.Linear(128 * 3 * 3,128),
  nn.ReLU(True),
  nn.Linear(128,latenSpace_dim)
)
```

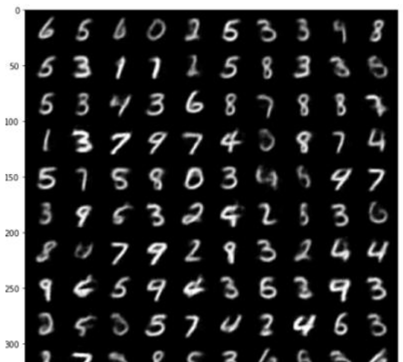
<u>Increased output maps to 64,batch size =256,lr=0.016,dimention =4</u>





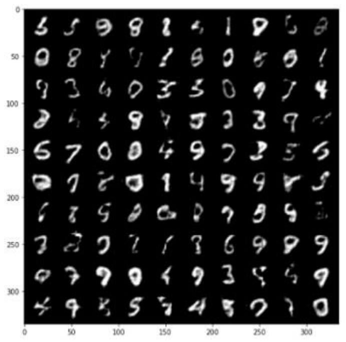
Let’s further increase the number of activation maps. Everything else constant.
Below is the best model so far. The true performance is high, and the loss value is low. This may mean that our encoder wasn’t strong enough or maybe that that our decode wasn’t. (The symmetry in the autoencoder doesn’t seems to make sense to me; why couldn’t it be that for some type of data the decoders task would be harder than the encoder’s. here I think looking at characteristic of the data and task at hand as well as the laten dimension valuable. We can conclude that lightly overtraining the AE is reasonable. Also it seems that as we decrease the laten dimension, the STD of the laten code increases quite drastically

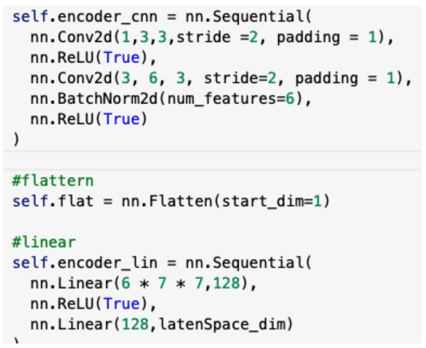




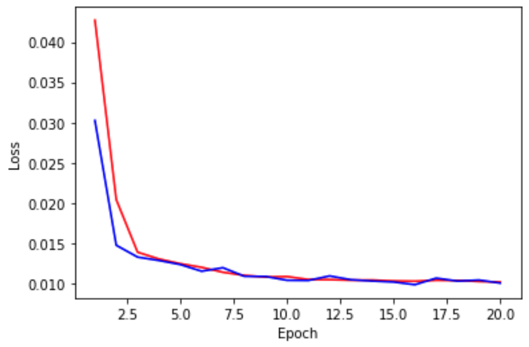
Skinny-Man-Theory

His rational is practical, the task at hand is simple therefore a simple encoder and decoder will suffice if the latten space is not too small. We will check this theory.
We will set up our network as shown below (decoder is symmetric) with a laten dimension of 10, lr=0.016,batch size =256.

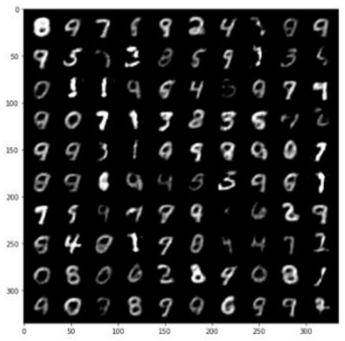


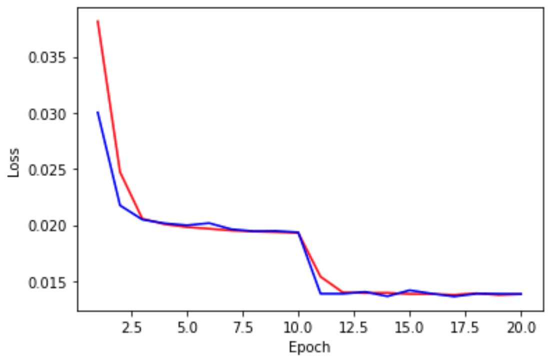


```
self.encoder_cnn = nn.Sequential(
  nn.Conv2d(1,3,3,stride =2, padding = 1),
  nn.ReLU(True),
  nn.Conv2d(3, 6, 3, stride=2, padding = 1),
  nn.BatchNorm2d(num_features=6),
  nn.ReLU(True)
)

#flattern
self.flat = nn.Flatten(start_dim=1)

#linear
self.encoder_lin = nn.Sequential(
  nn.Linear(6 * 7 * 7,128),
  nn.ReLU(True),
  nn.Linear(128,latenSpace_dim)
)
```



I must say that the results are quite deceiving. I will try first to decrease the laten space a little more (dim =6) in order to push the network to learn underlying features. This will further provide us intuition on whether the decoder is strong enough to deal with the data minimization or whether it is underfitted. As we can see there are some improvements although nothing like the fat man NN. What is this fall in the loss curve?

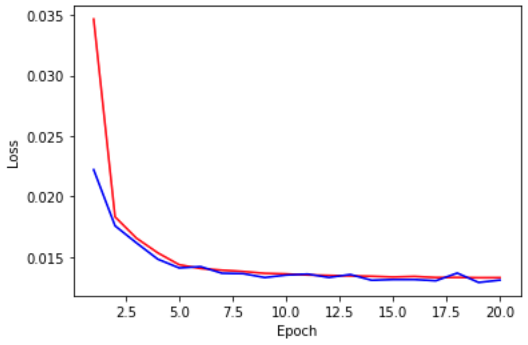


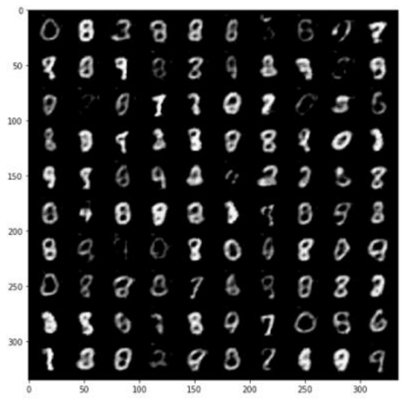


We will check another configuration I thought interesting.
We will uphold for now the skinny's man theory and check it further.
Here I further simplified the network by reducing the activation in the fully connected layer. To 32 instead of 128 And increased the laten dimension to 32. I expect the variance vector from the laten space to decrease drastically. Here are the results. I wonder what type of correlation there is between the mean of the variance vector as a function of laten dimension and test error.

I think it is safe to conclude (although more experiments are needed) that as we increase the dimension of the laten space over a certain point, the AE doesn't learn the deeper underlying features which are a function of how constrained the model is by the reduction of dimensionality. Therefore, the laten space in question isn't representative of the right features. However the variance of the features that are represented reduces as we increase the laten space. We already know that a low variance doesn't imply low test loss or increase in true performance. We also know that increasing laten space decreases laten variance. why? Is it plausible that in a well-trained model, the low variance is a consequent of the clustering of the laten space? or is it the inverse?

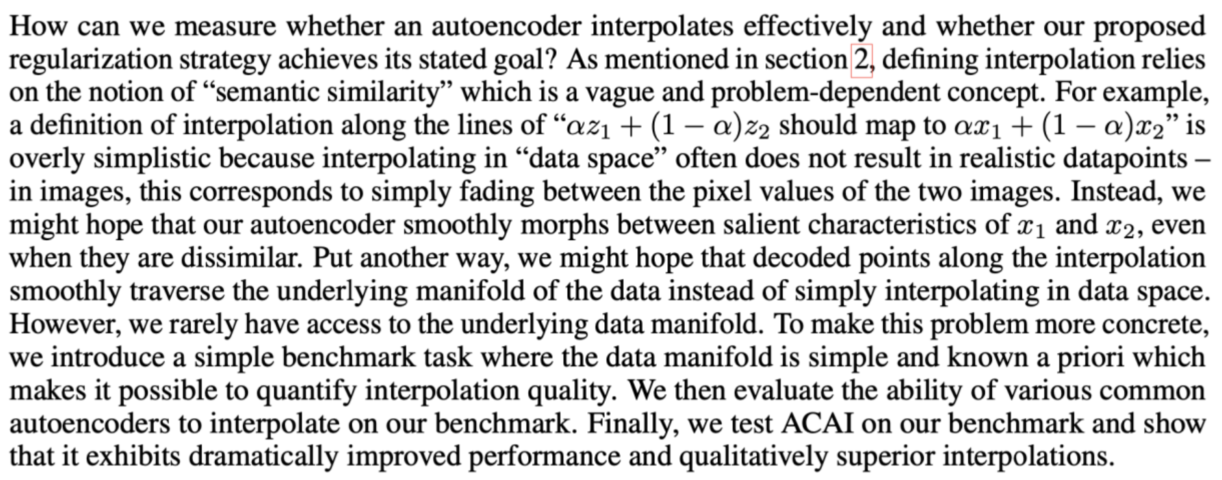
```
0.013304840844679386
0.013103344733826816
```





Interpolation:

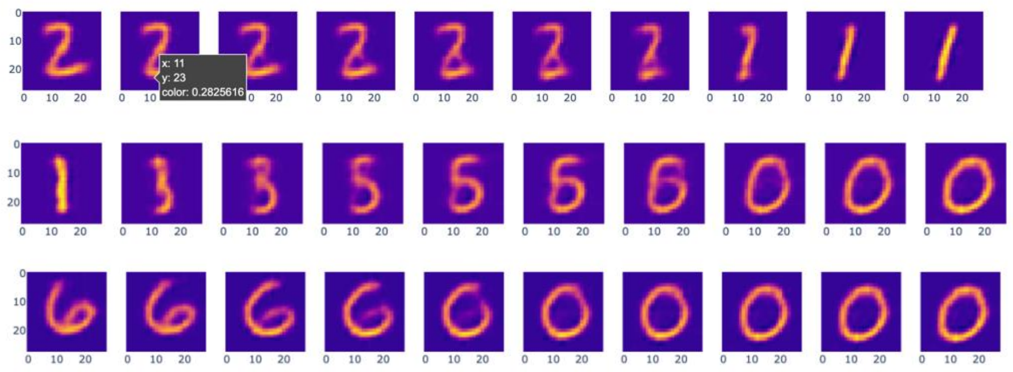
Interpolation has been frequently reported as a qualitative experimental result in studies about autoencoders. (https://openreview.net/pdf?id=S1fQSiCcYm) That is, if the interpolation is good then our latten space is also. I think Ian Goodfellow explains this better:



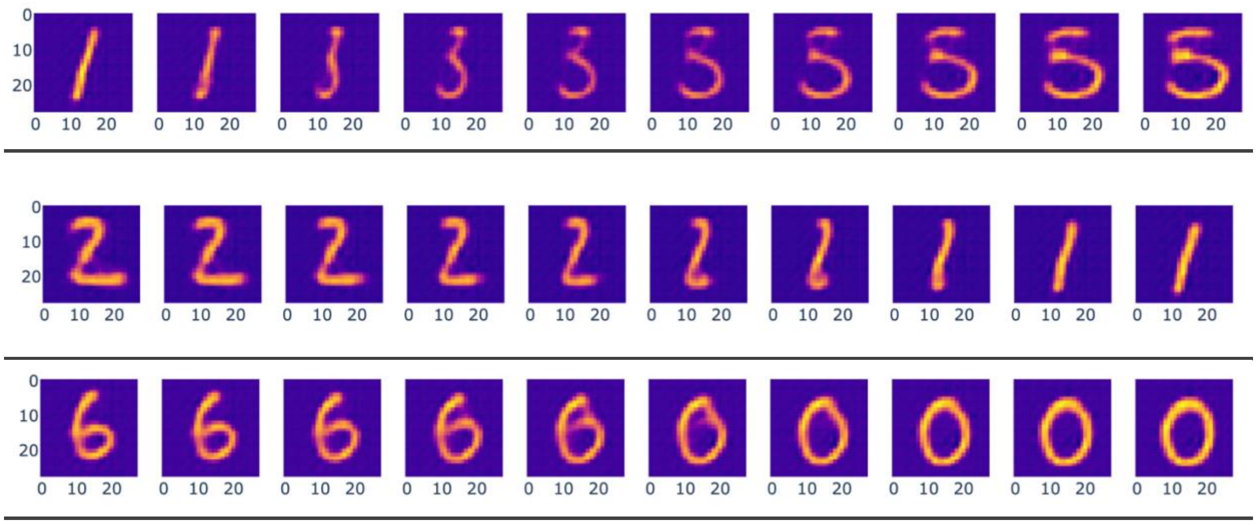
How can we measure whether an autoencoder interpolates effectively and whether our proposed regularization strategy achieves its stated goal? As mentioned in section 2, defining interpolation relies on the notion of "semantic similarity" which is a vague and problem-dependent concept. For example, a definition of interpolation along the lines of "$\alpha z_1 + (1-\alpha) z_2$ should map to $\alpha x_1 + (1-\alpha) x_2$" is overly simplistic because interpolating in "data space" often does not result in realistic datapoints – in images, this corresponds to simply fading between the pixel values of the two images. Instead, we might hope that our autoencoder smoothly morphs between salient characteristics of x_1 and x_2, even when they are dissimilar. Put another way, we might hope that decoded points along the interpolation smoothly traverse the underlying manifold of the data instead of simply interpolating in data space. However, we rarely have access to the underlying data manifold. To make this problem more concrete, we introduce a simple benchmark task where the data manifold is simple and known a priori which makes it possible to quantify interpolation quality. We then evaluate the ability of various common autoencoders to interpolate on our benchmark. Finally, we test ACAI on our benchmark and show that it exhibits dramatically improved performance and qualitatively superior interpolations.

If we have a smooth transition, we can be confident that our laten space is well generated and that the encoder is trained well enough to make sense out of it. We will therefore check the performance of our model by looking at interpolation results. We will also sample two nearby laten vectors and see whether they will result in semantically similar points. This is important since successful interpolation suggests that semantically similar points may be clustered together in the latent space.

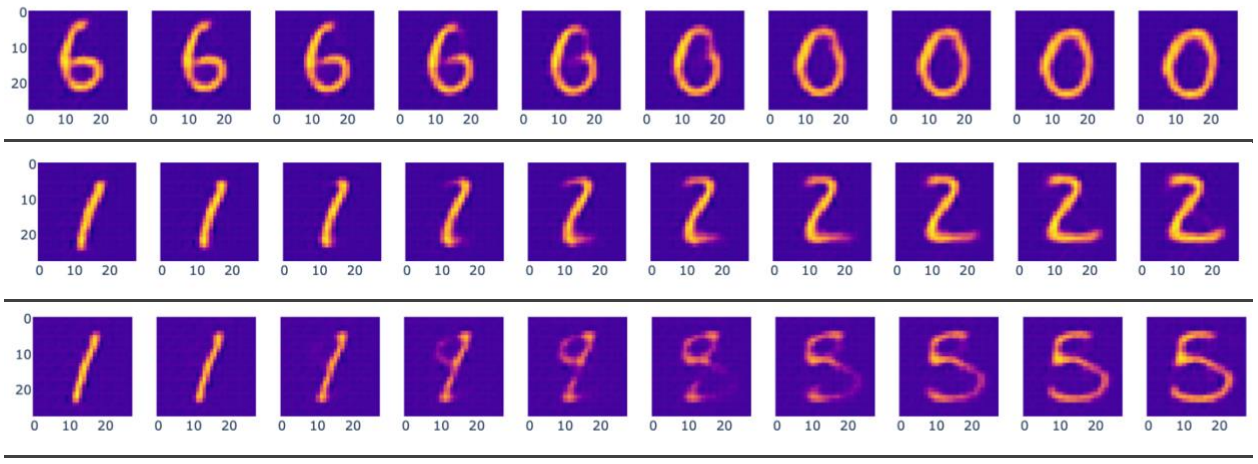
Fat Man – Laten Dim =4



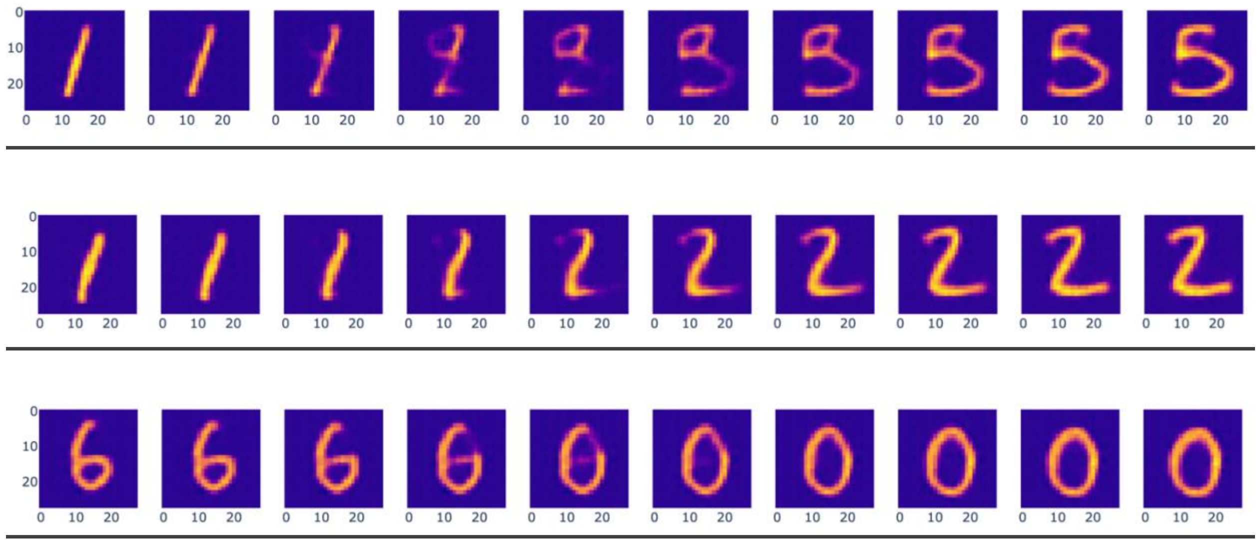
Fat Man – Laten Dim =10



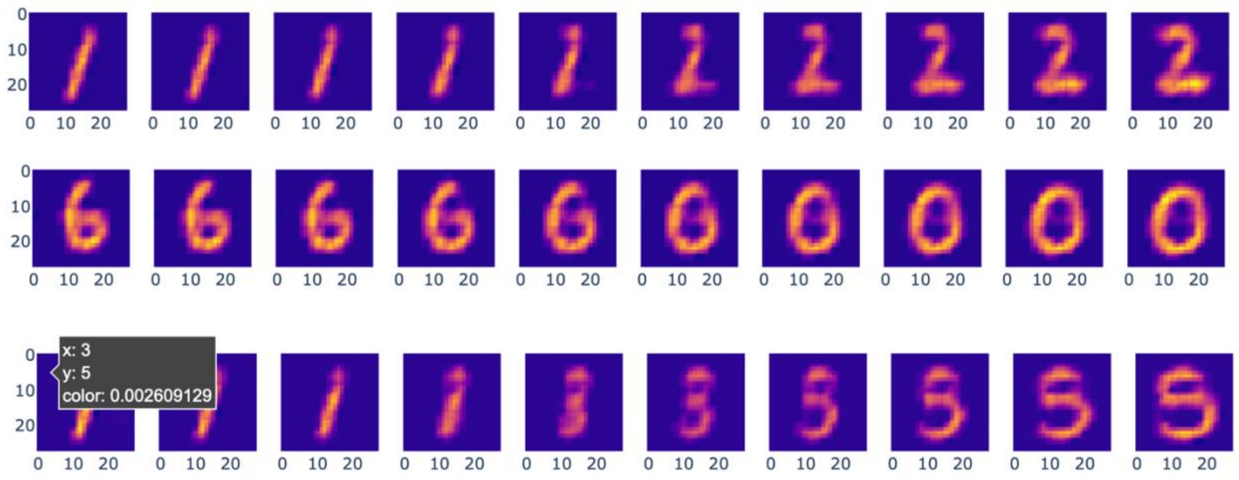
Fat Man – Laten Dim =15



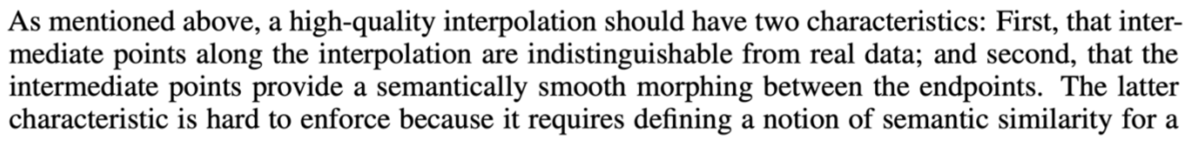
Fat Man – Laten Dim = 20



Model: The skinny man default laten dimension = 5. Note that this is only for the sake of interest. I answered this question using the better performing network (The Fatman)



From the above article



As mentioned above, a high-quality interpolation should have two characteristics: First, that intermediate points along the interpolation are indistinguishable from real data; and second, that the intermediate points provide a semantically smooth morphing between the endpoints. The latter characteristic is hard to enforce because it requires defining a notion of semantic similarity for a

First note that the skinny man model provides a smearing interpolation and an overall worse performance which is consistent with our previous findings. The other models had pretty good resemblance to the real data. If you look at the transition from 1 to 5 at D=10 we can see that throughout the interpolation process the images still resemble real images. For D=20 this is not the case. The reason for that I believe is the lower dimension the laten space which is well clustered and has an overall better structure.
I noticed that there is a “distance” in the clustering. That is, an interpolation between a 1 and a 2 occurs swiftly and the transition is almost indiscernible implying that the cluster of twos and the cluster of ones are themselves clustered together. Cluster to the second degree if you may.
A 5 and a 1 are far apart in that respect. Other than that, a semantically smooth morphing isn’t a reliable measure and I believe that any observation based on that are highly inconclusive.

Decorrelation:

In this section I extracted the correlation matrix of different coordinates in laten spaces of a given dimension. I opted for different methods to come up with a single value.

Method 0 - Subtract the diagonal values in the matrix (all one) and take mean of rows and then take mean again. Note that the correlation matrix is symmetric. Summing up the rows is therefore sufficient. I then took the mean again.

Method1 – Take the eigen values of the covariance matrix. eigen value of zero tells us that there is linear dependency. In this case I essentially looked at the dimension of the null space.

Method 2 – Calculate the determinant of the correlation matrix. The determinant is an indicator of linear dependence yet in this context it is also a normalized indicator of the correlation. A high correlation result in a determinant closer to 0.

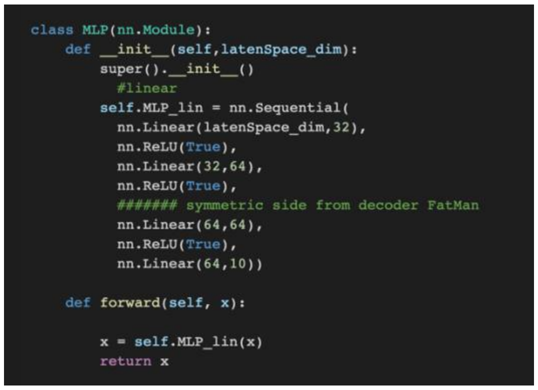
Method 3 - Orthogonally project each column of the matrix onto the same space and calculate mean/correlation on these vectors. What do you think about this?

Findings
I cant think of a way to interpret the results of method 1. This measure is unintelligible as far as I can see. There is no discernable pattern at all. Method 2 on the other hand is quite fruitful. I used the baseline network (the Fat man) and played with the laten space in the same way that we did in the first question. I found that at very low laten dimensions (1,2) we get a low determinant, and a corresponding high correlation. Since these dimensions are too restrictive as we saw before I believe that I cannot infer any useful conclusion. As I further increased the laten space to the sweet spot (4,5) the determinant was still close to zero (High correlation). I then noticed a linear decrease in correlation as I further increased the laten space (10,15,20). Note that this is consistent with our measurements of variances in question 2. What is the relation?
I believe that correlation in the laten space is an instructive measurement (using the determinant) and that a high correlation is desirable if the laten space is not too restricted. Lastly, I tried to do the same with the skinny man model. I got similar results yet for overall higher values for the dimension of the laten space. (This seems logical)

Transfer Learning

I must say that thinking back, I chose the FatMan network because it had the highest real performance. Yet I come to realize that it is overly fat, especially if we were to connect it with a small MLP. I mean, it is almost a MLP itself. I could have reduced the weights in the FC layers in the model. Maybe also reduce the channels to 64 instead of 128. trim or not trim the fat man?

I used as my base model the encoder from the FatMan AE. I attached to it a symmetric MLP. I essentially took the linear layers of the decode and decoder and combined them while reducing the weight. Here is the model.

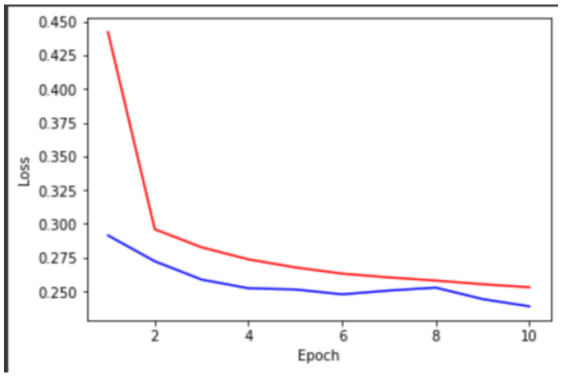


```
class MLP(nn.Module):
    def __init__(self,latenSpace_dim):
        super().__init__()
          #linear
        self.MLP_lin = nn.Sequential(
          nn.Linear(latenSpace_dim,32),
          nn.ReLU(True),
          nn.Linear(32,64),
          nn.ReLU(True),
          ####### symmetric side from decoder FatMan
          nn.Linear(64,64),
          nn.ReLU(True),
          nn.Linear(64,10))

    def forward(self, x):

        x = self.MLP_lin(x)
        return x
```

Note since I decreased the batch size considerably to 8, I had to reduce the learning rate by the same amount in order to get the same performance using the same model.

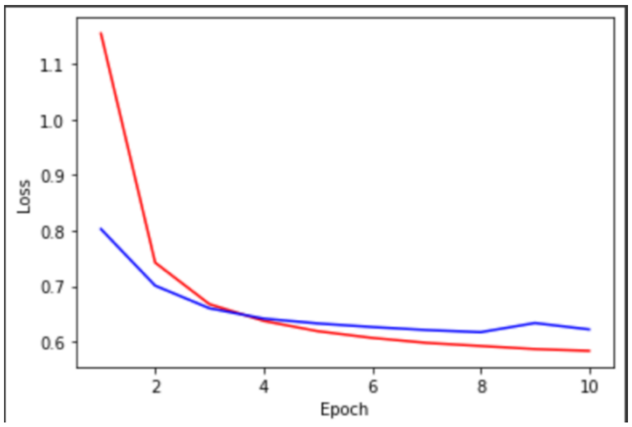
MLP and pretrained Encoder

At first, I trained the encoder with a batch size of 256 just as I did throughout the exercise. However, I surmised that the subsequent training of the network (MLP + encoder) on a considerably smaller batch size would result in undesirable effects on the performance. This however seems to be of no consequence as I didn't see any notable difference. I must also say that I was impressed with the loss value since although we got an overall higher loss than the Fatman model, we did get impressive results for a training set of only 80 pictures.



Training MLP and Encoder together
Here the results were bad although expected. The model didn't learn, and the loss curve didn't even converge. 80 pictures are simply not enough to train such a network. Here I understand the value of transfer learning. I also tried to play with the laten space although without any substantive changes. I also tried the skinny man model which is lighter. This however didn't solve the issue. With 80 images even a very light network seems to be underfitted.

Finally, to make sure the model did learn I increased the batch size considerably to 256. The results show that the network does learn. Even after 10 epoch (2560 training images) the loss curves seem to converge to a value that is higher than our original network.



1. Show that the composition of linear functions is a linear function. Show that the composition of affine transformations remains an affine function.

let g, f linear functions:

$$f\circ g(x+y) = f(g(x+y)) \overset{\text{L of } g}{=} f(g(x)+g(y)) \overset{\text{L of } f}{=} f(g(x)) + f(g(y))$$
$$= f\circ g(x) + f\circ g(y)$$

scalar: let $a \in \mathbb{F}$. $\quad a(f\circ g(x)) = af(g(x)) \overset{\text{Linearity } f}{=} f(ag(x)) = f(g(ax))$

$\square$.

We will prove the same for Affine transformations:

Intuition: Note that we actually do not need to prove since we can just take any Affine transformation, say $g(x) = Ax+b$ for A matrix, and define:

$$B = \left[\begin{array}{c|c} \vdots & A \end{array}\right] \quad \text{and} \quad X' = \begin{bmatrix} b \\ \hline X \end{bmatrix}$$

Is this 100% right??

we get that $g(x) = BX'$ which is a linear transformation

Proof: for $g(x) = Ax + a$, $f(x) = BX + b$, s.t $A, B \in G_{Ln}(\mathbb{F})$

We get $f\circ g(x) = f(g(x)) = B(g(x)) + b = B(Ax+a) + b$

$$= \underbrace{BA}_{M}x + \underbrace{Ba+b}_{\text{scalar}}$$

Note that since g, f are affine $\rightarrow A, B \in G_{Ln}(\mathbb{F})$ (Invertible)

We Also get that $BA \in G_{Ln}(\mathbb{F})$

$\Rightarrow g\circ f$ is also Affine $\square$.

2. The calculus behind the Gradient Descent method:
   a. What is the stopping condition of this iterative scheme,

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

   b. Use the second-order multivariate Taylor theorem,

$$f(x+dx) = f(x) + \nabla f(x) \cdot dx + dx^T \cdot H(x) \cdot dx + O(\|dx\|^3),$$
$$H_{ij}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

a) we can clearly see that when the expression $\nabla f_{\theta^n}(x) = 0$ we get that $\forall i > n \quad \theta^i = \theta^n$

Note our training epochs is finite and thus we will obviously stop at some point. The question is whether we arrive to a extremum point before the end of the training. Here, the inverse consideration arises:

$\longrightarrow$ we may like to stop the Gradient decent Algorithm for that a specific weight even if the training is not over yet.

If for some $\varepsilon > 0$ $|\theta^{n+1} - \theta^n| < \varepsilon$ we may want to stop.

(This is one stopping condition)

2. The calculus behind the Gradient Descent method:
   a. What is the stopping condition of this iterative scheme,

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

   b. Use the second-order multivariate Taylor theorem,

$$f(x+dx) = f(x) + \nabla f(x) \cdot dx + dx^T \cdot H(x) \cdot dx + O(\|dx\|^3),$$
$$H_{ij}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

a) we can clearly see that when the expression $\nabla f_{\theta^n}(x) = 0$ we get that $\forall i > n \quad \theta^i = \theta^n$

Note our training epochs is finite and thus we will obviously stop at some point. The question is wether we arrive to a extremum point before the end of the training. Here, the inverse consideration arises:

$\rightarrow$ we may like to stop the Gradient decent Algorithm for that a specific weight even if the training is not over yet.

IF for some $\varepsilon > 0$ $|\theta^{n+1} - \theta^n| < \varepsilon$ we may want to stop.

(This is one stopping condition)

3. Assume the network is required to predict an angle (0-360 degrees). How will you define a prediction loss that accounts for the circularity of this quantity, i.e., the loss between 2 and 360 is not 358, but 2 (since 0 is 360..). Write your answer in a pytorch-like pseudo-code.

Note that the absolute diff between two angles x, Y is at most 180°

```
loss = $|X-Y|$
If loss ≤ 180 :
    Return loss
return 360 - loss
```

4. Chain Rule. Differentiate the following terms (specify the points of evaluation of each function):
   a.
   $$\frac{\partial}{\partial x} f(x+y, 2x, z)$$
   b.
   $$f_1\Big(f_2\big(\ldots f_n(x)\big)\Big)$$
   c.
   $$f_1\Big(x, f_2\big(x, f_3(\ldots f_{n-1}(x, f_n(x)))\big)\Big)$$
   d.
   $$f\Big(x + g\big(x + h(x)\big)\Big)$$

we will use the chain Rule to solve the above:

$$a) = \frac{\partial F(x+y, 2x, z)}{\partial(x+y)} \cdot \cancel{\frac{\partial(x+y)}{\partial x}}^{1} +$$

$$+ \frac{\partial F(x+y, 2x, z)}{\partial(2x)} \cdot \cancel{\frac{\partial F(2x)}{\partial x}}^{2} +$$

$$+ \frac{\partial F(x+y, 2x, z)}{\partial(z)} \cdot \cancel{\frac{\partial(z)}{\partial(x)}}^{0}$$

$$= \frac{\partial F(x+y, 2x, z)}{\partial(x+y)} + 2 \cdot \frac{\partial F(x+y, 2x, z)}{\partial(2x)}$$

a

b) $\frac{\partial}{\partial x} f_1(f_2(\cdots(f_n(x))\cdots) =$

$$f_1'(\cdots f_n(x)) \cdot f_2'(\cdots f_n(x)) \cdot \cdots \cdot f_n'(x).$$

c) $= \frac{\partial}{\partial x} f_1(x, f_2(x, f_3(\cdots f_{n-1}(x, f_n(x))))) +$

$$+ \frac{\partial f_1(x, f_2(\cdots f_{n-1}(x, f_n(x))\cdots)}{\partial(f_2(x, f_3(\cdots f_{n-1}(x, f_n(x))))} \cdot \frac{\partial(f_2(\cdots f_{n-1}(x, f_n(x)))}{\partial x}.$$

$$= \frac{\partial f_1(\cdots f_{n-1}(x, f_n(x))\cdots)}{\partial x} + \frac{\partial f_1(\cdots f_{n-1}(x, f_n(x))\cdots)}{\partial(f_2 \cdots f_{n-1}(x, f_n(x))\cdots)} \cdot$$

$$\cdot \left[\frac{\partial f_2(\cdots f_{n-1}(x, f_n(x))\cdots)}{\partial x} + \frac{\partial(f_2 \cdots f_{n-1}(x, f_n(x))\cdots)}{\partial(f_3 \cdots f_{n-1}(x, f_n(x))\cdots)} \cdot \left[\frac{\partial(f_3 \cdots f_{n-1}(x, f_n(x))\cdots)}{\partial x} \cdots\cdots \left(\frac{\partial F_n}{\partial x}\right)\right]\right.$$

In the same manner as above.

d) $\frac{\partial}{\partial x} f(x+g(x+h(x)) \overset{\text{Using the chain Ru}}{=}$

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

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

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

$\square$.