[OTP8] Report on lab session #3 - Part II

BADILLO, Jose - MARTINEZ, Gabriela
jose.badillo@student-cs.fr - gabriela.martinez@student-cs.fr

Université Paris-Saclay, CentraleSupélec — December 5, 2019

1 Objective

The objective of this practical session is to build and train a Variational Autoencoder (VAE) with Pytorch. The main goal of such generative model is to provide new random samples close to the real content used to train the network.

For the training phase, we will present the training subset of MNIST data-set which will generate the output vectors of the mean and variance that properly represents the Gaussian distribution of the data.

During the test phase we will meassure how close the test samples are with regards to the network distribuition.

Finally we will generate new samples coming from such trained model.

2 Variational Autoencoder Architecture

A variational autoencoder keeps an encoding distribution regularized and improved during the training phase so the latent space is good enough to produce new samples. The VAE was named after the Variational Inference method used in statistics.

The data was previously accessed from http://deeplearning.net/data/mnist/mnist.pkl.gz. and divided into training, dev, and test sets.

2.1 Encoder

The implemented architecture starts with the *encoding* module implementing a network with one hidden layer and two different output layers representing the parameters of the Gaussian distribution which characterize the data distribution over the latent space.

```
1
2  class Encoder(nn.Module):
3
4      def __init__(self, input_dim, hidden_dim, z_dim):
5          super(Encoder,self).__init__()
6
7          self.linear = nn.Linear(input_dim, hidden_dim)
8          self.mu = nn.Linear(hidden_dim, z_dim)
9          self.var = nn.Linear(hidden_dim, z_dim)
10
11     def forward(self, x):
12         # x is of shape [batch_size, input_dim]
13         hidden = F.relu(self.linear(x))
14         # hidden is of shape [batch_size, hidden_dim]
15         z_mu = self.mu(hidden)
16         # z_mu is of shape [batch_size, latent_dim]
17         z_var = self.var(hidden)
18         # z_var is of shape [batch_size, latent_dim]
19         return z_mu, z_var
```

2.2 Decoder

A decoder is a multi-layer perceptron that will take as input a sample from the latent distribution and after passing trough the network it will deliver a new representation of such sample in the original dimensions.

```
class Decoder(nn.Module):

    def __init__(self, z_dim, hidden_dim, output_dim):
        super(Decoder,self).__init__()

        self.linear = nn.Linear(z_dim, hidden_dim)
        self.out = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # x is of shape [batch_size, latent_dim]
        hidden = F.relu(self.linear(x))
        # hidden is of shape [batch_size, hidden_dim]
        predicted = torch.sigmoid(self.out(hidden))
        # predicted is of shape [batch_size, output_dim]
        return predicted
```

2.3 VAE

The creation of the auto-encoder consist on the conjunction of the encoder and the decoder like any other model of such type. The particularity of this generative model is that in order to obtain the newly created output image, we obtain a random sample from the latent space z and during training we are supposed to back-propagate the error. Nevertheless due to the randomness, we cannot be sure on where such gradients should be back-propagated. In order to solve such limitation we used a reparametrization trick.

This is possible taking into account that the posterior is normally distributed and then we can make an approximation using another normal distribution with another parameters.

```
    class VAE(nn.Module):
    def __init__(self, enc, dec):
        super(VAE, self).__init__()

        self.enc = enc
        self.dec = dec

    def forward(self, x):
        # encode
        z_mu, z_var = self.enc(x)
        # sample from the distribution having latent parameters z_mu, z_var
        # reparameterize
        std = torch.sqrt( torch.exp( z_var ) )
        eps = torch.normal(0, 1., z_mu.shape)
        x_sample = z_mu + eps * std
        # decode
        predicted = self.dec(x_sample)
        return predicted, z_mu, z_var
```

3 Network training

3.1 The training loop

In order to avoid over-fitting and be sure that we could use the auto-encoder as a generative model it is necessary to add a regularization during the training process. The regularization performed for such

purposes is the encoding as a distribution on the latent space. The main steps to train a variational auto-encoder are depicted below [1]:

1. The input is encoded as distribution over the latent space
2. A point from the latent space is sampled from that distribution
3. The sampled point is decoded and the reconstruction error can be computed
4. The reconstruction error is back-propagated through the network

```
def train():

    # set the train mode
    model.train()

    # Use gradient clipping
    torch.nn.utils.clip_grad_value_(model.parameters(), 5.)

    train_loss = 0

    for i, (x, _) in enumerate(train_iterator):

        # reshape the data into [batch_size, 784]
        x = x.view(-1, 28 * 28)
        x = x.to(device)

        # update the gradients to zero
        optimizer.zero_grad()

        # forward pass
        x_sample, z_mu, z_var = model(x)

        # total loss
        loss = loss_function( x_sample, x, z_mu, z_var )

        # backward pass
        loss.backward()
        train_loss += loss.item()

        # update the weights
        optimizer.step()

    return train_loss
```

The loss function to be minimized during training comes from the Variational Inference statistics method, and it is composed of a "reconstruction" and a "regularization term". The latter is expressed as the Kullback-Leiber divergence between the returned distribution and a normal Gaussian.

```
# Reconstruction + KL divergence losses summed over all elements and batch
def loss_function(reconstruction_x, x, z_mean, z_var):

    # reconstruction loss
    BCE = F.binary_cross_entropy(reconstruction_x, x.view(-1, 784), reduction='sum')
    # kl divergence loss
    KLD = -0.5 * torch.sum(1 + z_var - z_mean.pow(2) - z_var.exp())

    return BCE + KLD
```

3.2 The test loop

Next to the training we have developed a test loop which will asses how similar the current trained distribution of the VAE is being with regards of the images unknown for the network.

```
def test():

    # set the evaluation mode
    model.eval()

    test_loss = 0

    # we don't need to track the gradients, since we are not updating the parameters during
        evaluation / testing
    with torch.no_grad():
        for i, (x, _) in enumerate(test_iterator):

            # reshape the data
            x = x.view(-1, 28 * 28)
            x = x.to(device)

            # forward pass
            x_sample, z_mu, z_var = model(x)

            # total loss
            loss = loss_function( x_sample, x, z_mu, z_var )
            test_loss += loss.item()

    return test_loss
```

One of the key factors in order to avoid the over-fitting is to stop the training after certain number of epochs. This is done considering as stop factor the amount of times the best_test_loss has not being improved.

```
best_test_loss = float('inf')
trainLoss = []
testLoss = []

for epoch in range(N_EPOCHS):

    train_loss = train()
    test_loss = test()
    loss = []
    train_loss /= len(train_dataset)
    test_loss /= len(test_dataset)

    if epoch % 5 == 0:
        print('>> epoch ', epoch)
        print('---> Train loss: >> ', train_loss)
        print('-----> Test loss: >> ', test_loss)
        print('')

    trainLoss.append( train_loss )
    testLoss.append( test_loss )

    if best_test_loss > test_loss:
        best_test_loss = test_loss
        patience_counter = 1
    else:
        patience_counter += 1

    if patience_counter > 3:
```

```
29          break
```

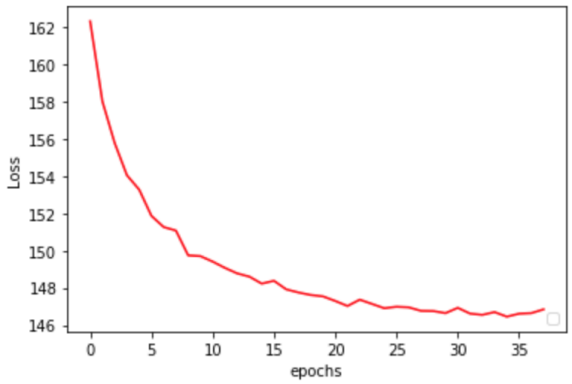
3.3 Gradient clipping

Also, we are implementing gradient clipping in the training process, so we can deal with numerical overflow or underflow (exploding gradients).

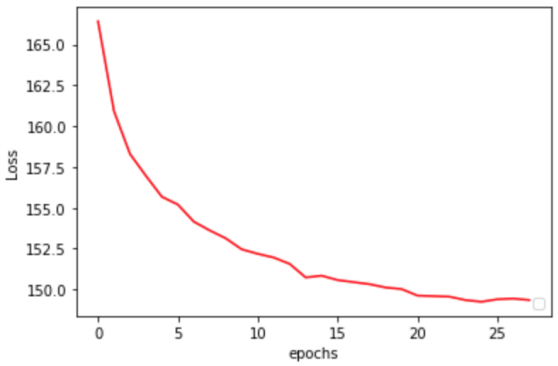
4 Experiments

4.1 Testing the batch-size

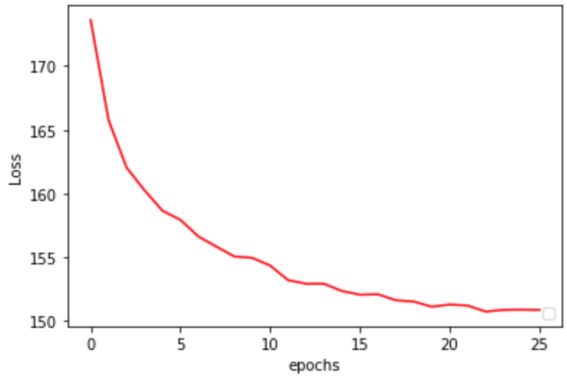
We trained the model several times with 4 different batch-sizes: 64, 128, 256 and 512. The hidden dimension was fixed to 784 so the encoding keeps all the pixel information. The variation of such dimension will be tested in further experiments. The latent dimension was arbitrary selected to be 2 for this experiment however in further test we are going to vary such value. The number of epochs was fixed to 50 to verify the behaviour of the batch_size, however as shown in the graphs of Figure 1 the test accuracy was worsen after 25-30 epochs so then the solution automatically stops the fitting phase.



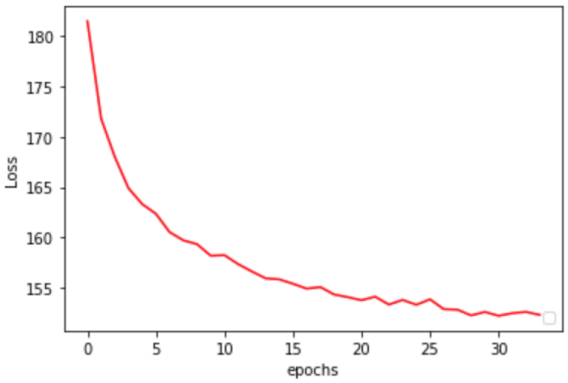
a. Batch_size = 64



b. Batch_size = 128



c. Batch_size = 256



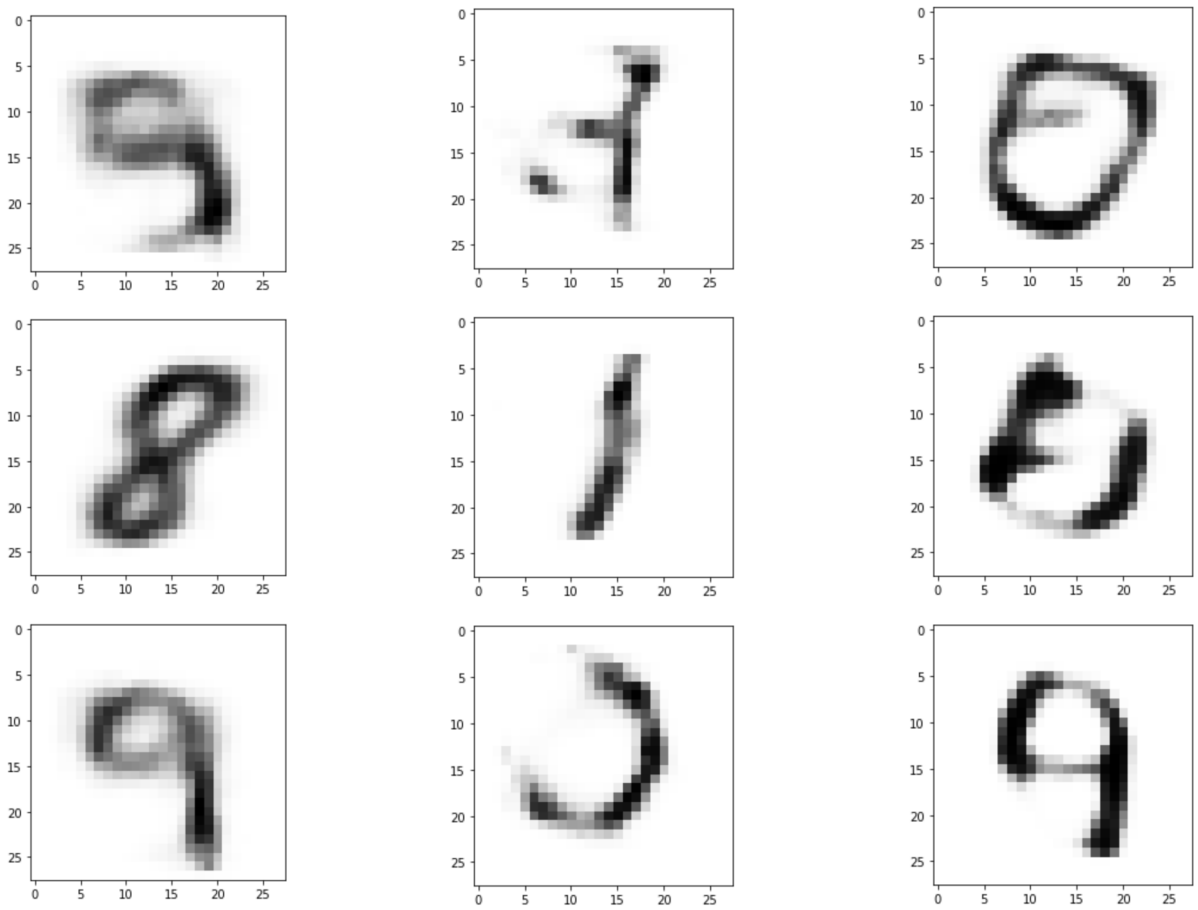
d. Batch_size = 512

Figure 1. Testing loss of the Variational Autoencoder using different batch_sizes

From the results shown above it is clear that the lower the batch_size the more reduction in loss throughout the epochs. Although the variation of such parameter is not impacting in a sever way, we identified the batch_size with value 256 as the one presenting more stability when reducing the test loss. Moreover it is obtaining a low value of loss in lower number of epochs with a bigger amount of samples to be trained. Therefore this will help to improve the performance time during the fitting phase. The following experiments during this lab will use 256 batch_size value. However this is not a conclusive value and more testing would have to be done.

4.2 Testing the latent dimension

We have trained the variational auto-encoder with 3 different values for the latent dimension: 2, 10, 20. For this purpose we decided to keep a hidden dimension of 748 so we don't lose any detail of the pixels. The goal of this test is to find the most appropriate value for such latent dimension.



a. Latent Dimension = 2 *b.* Latent Dimension = 10 *c.* Latent Dimension = 20

Figure 2. Generation of MNIST images from VAE with 784 nodes in hidden dimension and different values for latent dimensions

In Figure 2 we can appreciate that, when the latent dimension is bigger, the images generated from the VAE tend to produce representations that are not well defined or are far from the real representations of the original data-set. In the other hand when the latent dimension is 2 or minimal we found more meaningful images and with a better shape defined. Considering this experiment we have decided to fix the latent dimension in 2 for further experiments.

4.3 Testing the hidden dimension

We trained tree different architectures of Variational Autoencoders varying the hidden layer: (1000-2, 500-2 and 100-2) For all these architectures we used a ReLU as non linear function.

Each one of this architectures was trained with a fixed batch_size of 256 samples and run with a maximum value of 50 epochs. We must recall the epochs will be different depending on the values obtained for the *best_test_loss* during the train process.

4.3.1 Experiment with 1000 nodes in hidden dimension

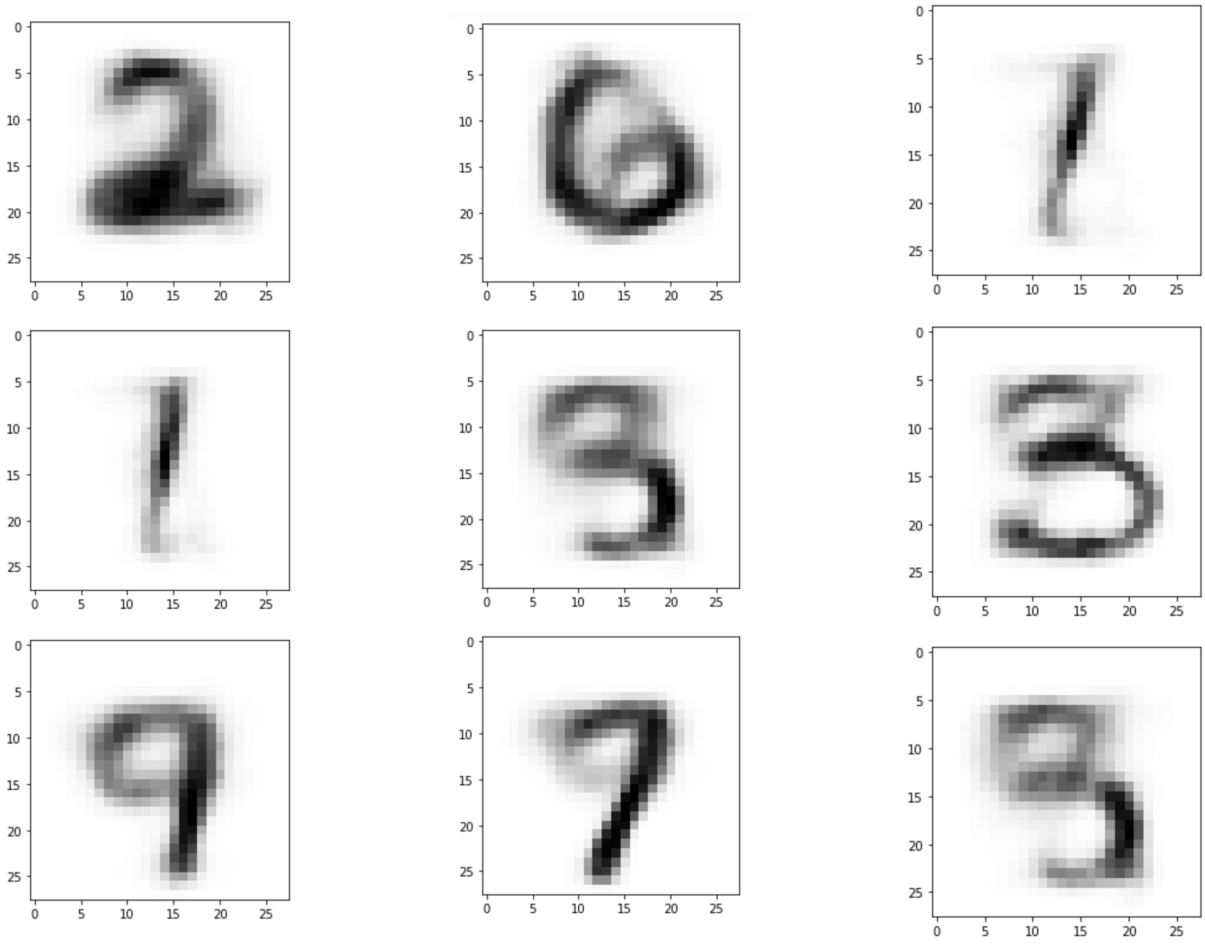


Figure 3. Generation of MNIST images from VAE with 1000 nodes in hidden dimension and 2 nodes in latent dimension

The nine images generated above from the Gaussian distribution trained within the variational auto-encoder are well defined. This satisfy the continuity (two close points in the latent space should not give two completely different contents once decoded) and completeness (for a chosen distribution, a point sampled from the latent space should give "meaningful" content once decoded) characteristics that a generative model must hold.

It is clear that our VAE is generating real values of digits. In a certain level it can be appreciated certain mixes among the digits distributions like the image with the number six where it shows a gray, blurry completion of the figure that could be trying to generate a zero, but such noise could be removed with help of other kind of models like the de-noising auto-encoder.

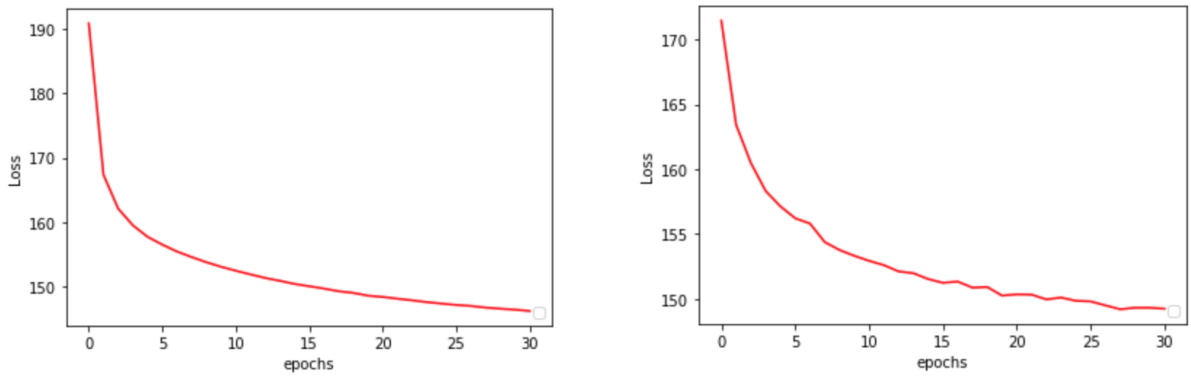


Figure 4. Training and Testing Loss functions for VAE 1000-2

4.3.2 Experiment with 500 nodes in hidden dimension

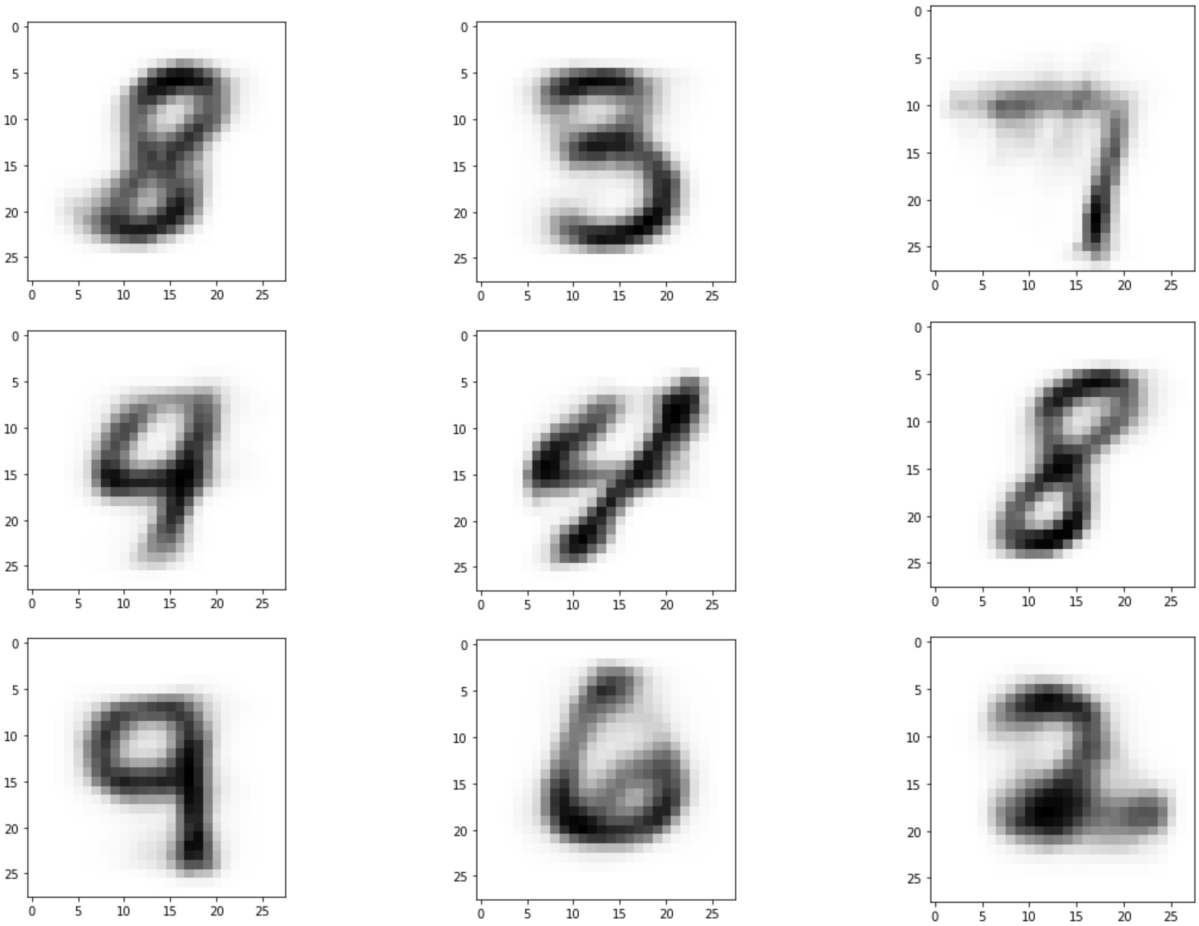


Figure 5. Generation of MNIST images from VAE with 500 nodes in hidden dimension and 2 nodes in latent dimension

The Figure 5 above show a meaningful generation of samples coming from the trained VAE. The hidden dimension of 500 was lower than the original representation of 784 but still close enough to the original dimensions in order to be able to keep the main characteristics of the data and generate proper samples than can be clearly identified.

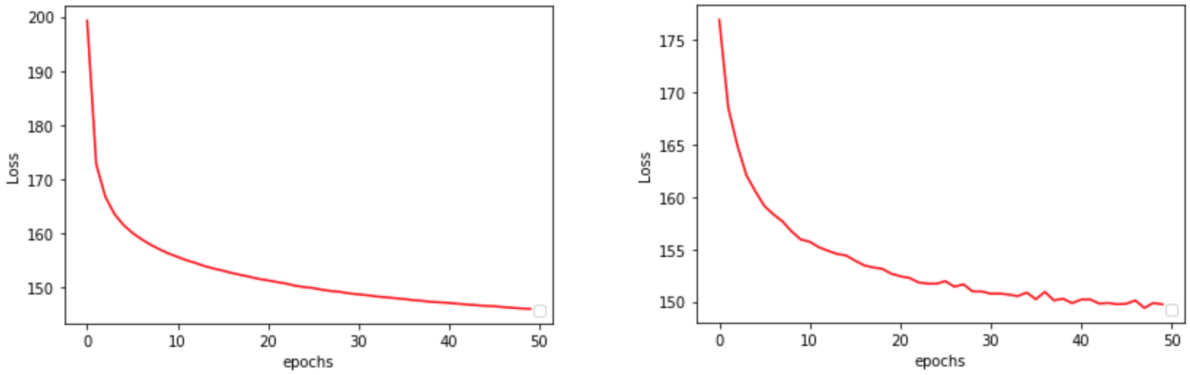


Figure 6. Training and Testing Loss functions for VAE 500-2

4.3.3 Experiment with 100 nodes in hidden dimension

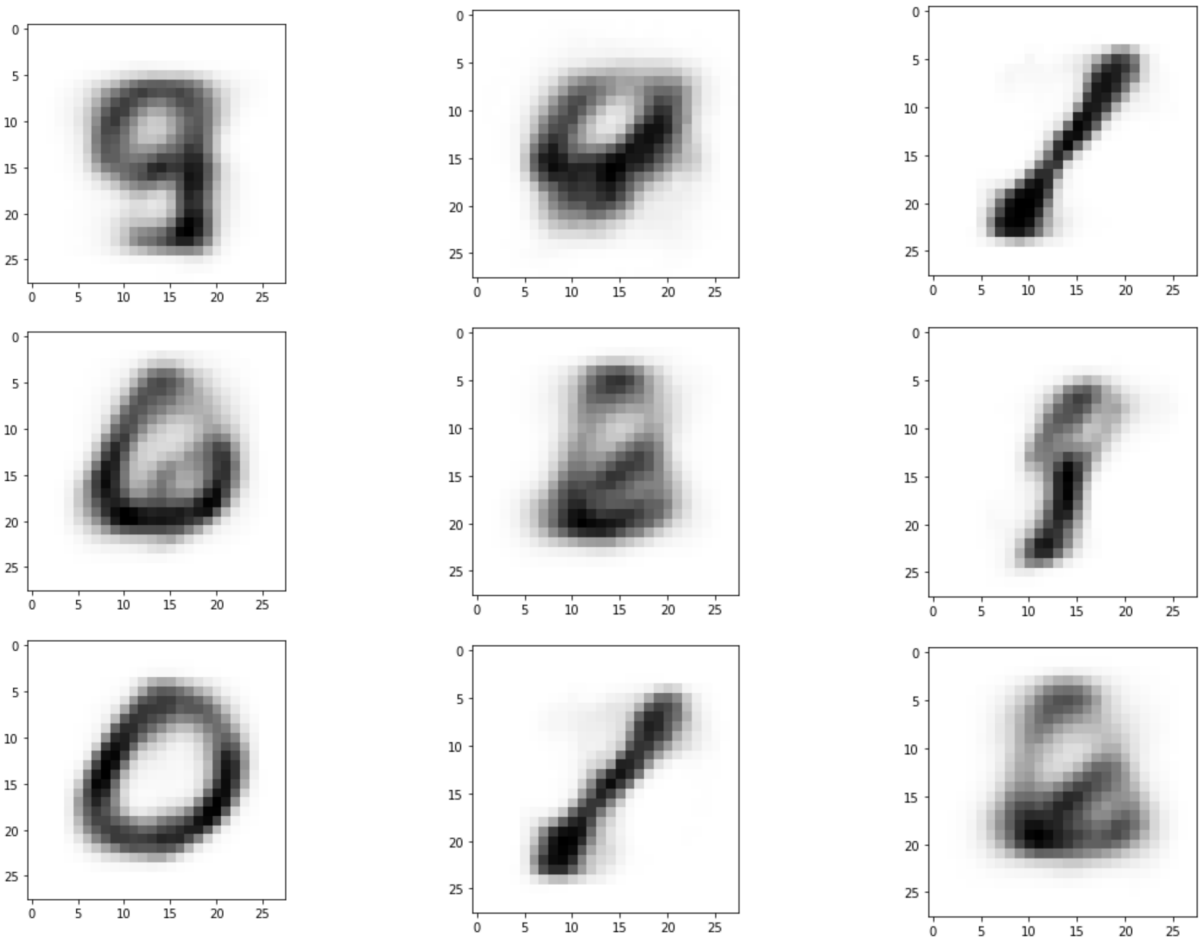


Figure 7. Generation of MNIST images from VAE with 100 nodes in hidden dimension and 2 nodes in latent dimension

The Figure 7 shows different samples generated from the VAE with 100 nodes of hidden dimension. We can clearly see that we have lost a big a mount of information in the encoding phase as the decoder is not able to generate meaningful content. Certain classes like zeros or ones are still useful, however it is not easy to visualize the nines or eights as clear as we did with the previous configurations.

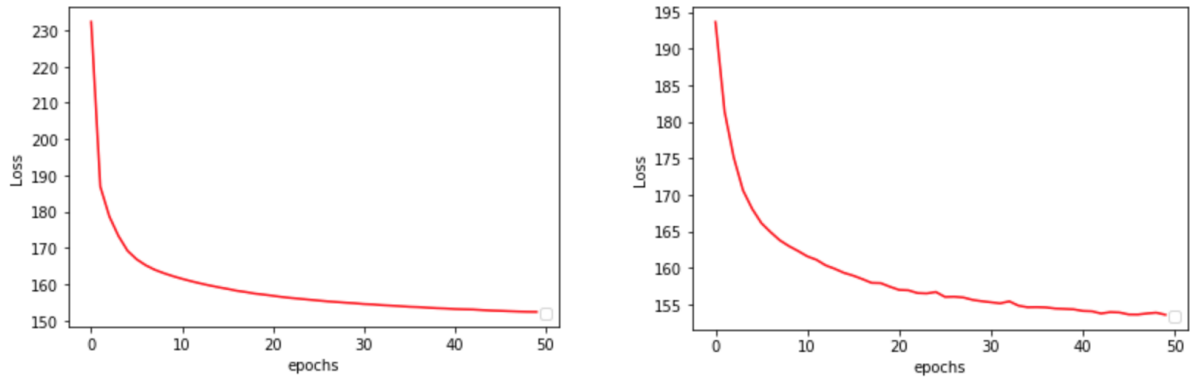


Figure 8. Training and Testing Loss functions for VAE 100-2

5 Final notes

- For the MNIST data-set when training deep learning models, a small batch_size is important to avoid impacts on the accuracy of the model. For variational auto-encoder the results show a similar behaviour: the lower the batch_size the better performance the model will have when reducing the loss function. However, it seems the impact in the performance of this generative model is not huge it is possible to use big batches like 128 or 256 without losing generative capabilities.
- For the MNIST data-set, if we consider a hidden dimension with dimensions similar than the original dimensions of an image, the fitting phase of the VAE converge after 25-30 epochs. The latter when using a very small step_size value of 0.001. However when we vary the number of nodes and generally we apply dimensionality reduction, then more epochs are necessary in order to converge (as expected). The number of epochs can be reduced with the variation of the step_size to a bigger one.
- The hidden dimension in the variational auto-encoder can be lower to the original if the purpose of the model is to reduce dimensions and then provide a distribution representation of the data. We must be aware that, the lower the dimension of the hidden layer, the less accurate are going to be the samples generated by the auto-encoder.

References

[1] Rocca, J. Understanding variational autoencoders (vaes).