
**UNIVERSITA' DEGLI STUDI DI PADOVA**
DEPARTMENT OF INFORMATION ENGINEERING

ICT for Internet and Multimedia

Neural Networks and Deep Learning Course

# Homework 2: Unsupervised Deep Learning Report

Fatıma Rabia YAPICIOĞLU
2049536

3 December 2021

# CONTENTS

**1.Convolutional Autoencoder (implementation details and results)…….…...3**

**1.1. Implementing and Testing the Convolutional Autoencoder, reconstruction loss and some examples of image reconstruction (structure & regularization methods)…….3**

**1.2. Experimenting the Optuna: Cross-Validation and Hyperparameter Tuning ……4**

**1.3. Reporting the results of the Optune Experiment and different Advanced Optimizers……..……...5**

**1.4.Visualizing the Latent Code( t-SNE)……..…….……5**

**2. Variational Convolutional Autoencoder(implementation details and results).....5**

**2.1. Setting up the Variational Convolutional Autoencoder…….6**

**2.2. Cross-validation & Hyperparameter Tuning by the Optuna ( implementation details & results).....6**

**Instructions and warnings about running the notebook**

*If you run this notebook on the Colab please make sure that you download all the necessary libraries. I've written the code and run the notebook in my local environment, and as you can see from the outputs it compiles with all the necessary libraries successfully. Run the notebook in a vertical manner starting from the first cell to the end. Runtime takes nearly 50-60 mins due to the Optuna and Variational Convolutional Autoencoder implementations. Each chapter in this report has been neatly designed as in the following structure: Introduction to the tasks that will be implemented, technical implementation details, and results concerning the experiments. For some very large tables, I only provided a preview on the report, you can see all of them in the *.ipynb. Thank you.*

## 1. Convolutional Autoencoder (implementation details and results)

### 1.1. Implementing and Testing the Convolutional Autoencoder, reconstruction loss and some examples of image reconstruction

In order to implement a convolutional autoencoder, again I preferred to use the FashionMNIST dataset. First of all I'll present the convolutional autoencoder structure which is capable of learning the encoded representations of the input data then it tries to construct the same.


**Figure 1.1.**

The following structure in the Figure 1.2 shows how I constructed the encoder and the Figure 1.3 shows how I constructed the decoder. Then, I prefered to create another class called Autoencoder() and combined the encoder and decoder that I created in this class. I also created other methods for the training and testing which also indicates and proves that we are working under unsupervised learning, therefore it does not need to have labels during the training. Also I added several Dropout layers to the encoder and decoder in different ratios as a regularization method.

```
#Convolutional section
self.encoder_cnn = nn.Sequential(
    nn.Conv2d(1, 8, 3, stride=2, padding=1),
    nn.ReLU(True),
    nn.Dropout(p=.5),
    nn.Conv2d(8, 16, 3, stride=2, padding=1),
    nn.BatchNorm2d(16),
    nn.ReLU(True),
    nn.Dropout(p=.5),
    nn.Conv2d(16, 32, 3, stride=2, padding=0),
    nn.ReLU(True)
)

#Flatten layer
self.flatten = nn.Flatten(start_dim=1)

#Linear section
self.encoder_lin = nn.Sequential(
    nn.Linear(3 * 3 * 32, 128),
    nn.ReLU(True),
    nn.Linear(128, encoded_space_dim)
)
```

**Figure 1.2.**

```
self.decoder_lin = nn.Sequential(
    nn.Linear(encoded_space_dim, 128),
    nn.ReLU(True),
    nn.Linear(128, 3 * 3 * 32),
    nn.ReLU(True)
)

self.unflatten = nn.Unflatten(dim=1, unflattened_size=(32, 3, 3))

### Convolutional section
self.decoder_conv = nn.Sequential(
    nn.ConvTranspose2d(32, 16, 3, stride=2, output_padding=0),
    nn.BatchNorm2d(16),
    nn.Dropout(p=.2),
    nn.ReLU(True),
    nn.ConvTranspose2d(16, 8, 3, stride=2, padding=1, output_padding=1),
    nn.BatchNorm2d(8),
    nn.Dropout(p=.1),
    nn.ReLU(True),
    nn.ConvTranspose2d(8, 1, 3, stride=2, padding=1, output_padding=1)
)
```

**Figure 1.3**

There is another method for plotting the reconstructed images after each epoch, on purpose to observe progress in each step. Since the resource and time limit I didn't experiment with a very high number of iterations, but if we had trained it with a higher number of iterations, we would be able to see better results, for sure. Despite running duration, I would like to show structure and show

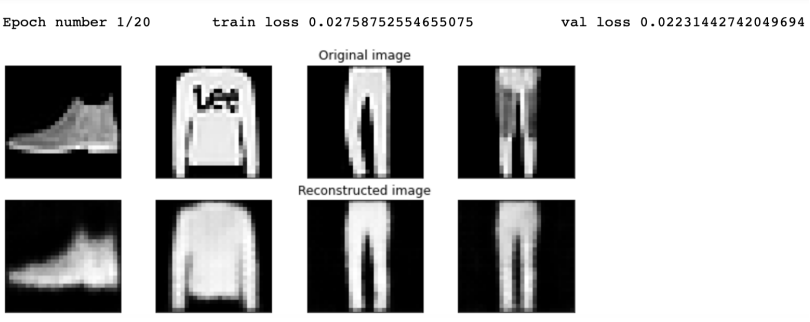



**Figure 1.4**

how it works, and how it can be implemented. As we can see in the Figure 1.4 after the first epoch the original image and the reconstructed image produces the following results. During the training I've also written a code for the Early Stopping as another regularization method.

```
# Early Stopping Regularization
plot_reconstructed_images(model,n=4)

if val_loss <= valid_loss_min:
   valid_loss_min = val_loss
else:
   print("Early Stopped, validation loss is getting high!")
   break;
```

**Figure 1.4.2**

### 1.2. Experimenting the  Optuna: Cross-Validation and Hyperparameter Tuning

In this section, I customized the implementation of the Optuna by creating two different objective functions in order to provide cross-validation with the hyperparameter tuning adjustments. Optuna's trial variable can be fed with the different types of parameters in different ranges. Unlike the Skorch that I've implemented in Homework 1, the Optuna takes a range (low,high,step) and tries the values one by one to find a better combination. This means that the computational complexity and running time is getting longer, therefore I only tried to see which learning rate is the best for the Convolutional Autoencoder that I've created.

```
lr = trial.suggest_float('lr', 0.01, 0.03 )
```

**Figure 1.5**

So, the Optuna tries all the values between 0.01 and 0.03 with all the parameters and after each trial produces the best combination, here in my case, it gives the best learning rate concerning that trial.

Also, I've written a specialized code for the cross-validation implementation in an objective method. I've used the K Fold method of the Sklearn library, and splitted the training set into 3 parts. The related code, iterates 3 epochs for each fold and tries to fine-tune the best learning rate after all the fold. In order to observe each fold's performance I've printed them in the following manner with the reconstructed images again in Figure 1.6.

```
[I 2021-12-03 20:58:18,757] Trial 0 finished with value: 0.9763428568840027 and parameters: {'lr': 0.01283334080493
1885}. Best is trial 0 with value: 0.9763428568840027.

In epoch 0, Training Error: 0.02413126640021801, Validation Loss: 0.026915134862065315
In epoch 1, Training Error: 0.02348710410296917, Validation Loss: 0.02558767795562744
In epoch 2, Training Error: 0.027937058359384537, Validation Loss: 0.02590901218354702
In epoch 3, Training Error: 0.030402593314647675, Validation Loss: 0.026840783655643463
In epoch 4, Training Error: 0.025353241711854935, Validation Loss: 0.025648104026913643
In fold 0, Mean Accuracy is : 0.9743518829345703
```

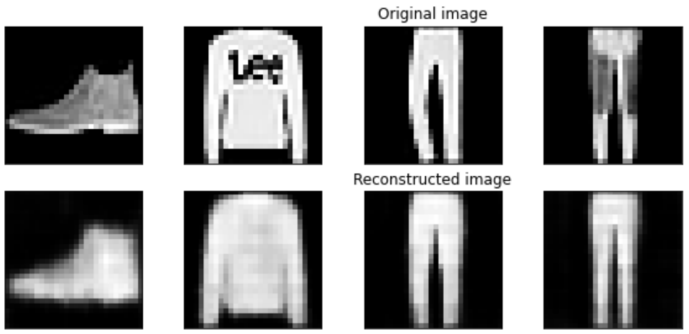
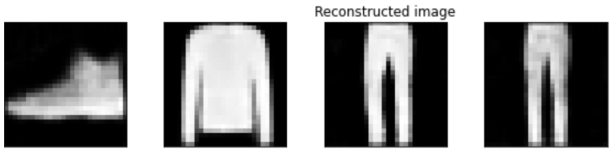



**Figure 1.6**

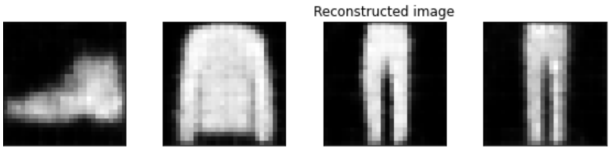
### 1.3. Reporting the results of the Optune Experiment and different Advanced Optimizers

The details of the specified Optuna objective method and the results can be seen longer in the *.ipynb. I also tested and computed the loss of the test dataset, it produced 0.021467363461852074 as a result. Also, I experimented with another advanced optimizer here called SGD, and observed the performance of the model, then finally compared these two models. In the following Figure 1.7, the left hand-side shows a sample which is reconstructed by the model with ADAM and the left hand-side shows a sample which is reconstructed by the model with SGD, and we can easily observe that the ADAM produces better reconstruction.

**ADAM Optimizer**




**SGD Optimizer**




| | |
|---|---|
| Trial 2 finished with value: 0.9738804697990417 and parameters: {'lr': 0.013811047610155013}. Best is trial 1 with value: 0.9743412137031555 optimizer had the learning rate as 0.01428. | Trial 2 finished with value: 0.970269501209259 and parameters: {'lr': 0.02848245434941231}. Best is trial 2 with value: 0.970269501209259. optimizer had the learning rate as 0.02428. |

**Figure 1.7**

### 1.4.Visualizing the Latent Code( t-SNE)

Now in this section, we can show the dynamic visualization in order to observe latent space learned by the convolutional autoencoder that we constructed. After creating the encoded samples by using the test set we prepared above, we can plot the latent space representation as in Figure 1.8.

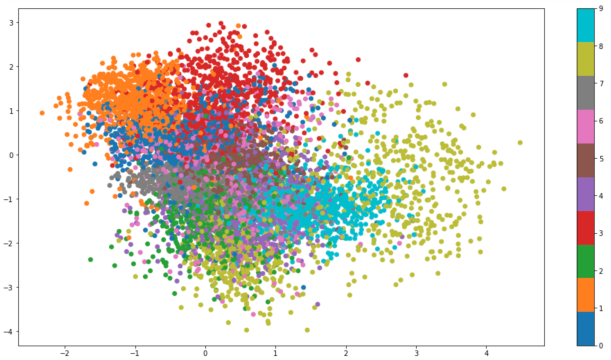


**Figure 1.8**

Since in this representation there are similar classes plotted together and overlaps, it is hard to read but we can use PCA(Principal Component Analysis or t-SNE) for dimensionality reduction. This way we'll be able to read clearer. In order to plot them in 2-dimension, we should set the number of components equal to 2. The following Figure 1.9 shows the latent space representation after applying the t-SNE dimensionality reduction.

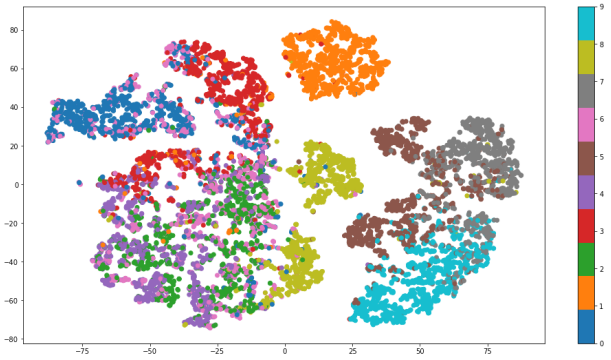


**Figure 1.9**

## 2. Variational Convolutional Autoencoder(implementation details and results)

### 2.1. Setting up the Variational Convolutional Autoencoder

In this section, I've implemented the variational autoencoder which aims to instead of encoding information into a vector, encoding information into a probability space. We are assuming the data is generated from a prior probability distribution and then trying to learn how to derive the data from that probability distribution. I defined two parameters in the method of parameterization in order to implement the parametrization trick. *"When training the model, we need to be able to calculate the relationship of each parameter in the network with respect to the final output loss using a technique known as backpropagation. However, we simply cannot do this for a random sampling process.*

```
def reparameterize(self, meanvec, variancevec):

    #Here we are introducing the standard deviation vector and mean vector,
    #specific to the variational autoencoders
    std = torch.exp(variancevec/2)
    eps = torch.randn_like(std)
    return meanvec + std * eps
```

**Figure 1.10**

*Fortunately, we can leverage a clever idea known as the "reparameterization trick" which suggests that we randomly sample ε from a unit Gaussian, and then shift the randomly sampled ε by the latent distribution's mean μ and scale it by the latent distribution's variance σ. With this reparameterization, we can now optimize the parameters of the distribution while still maintaining the ability to randomly sample from that distribution."* Also I defined the loss function as the sum of reconstruction loss and KL divergence measure which needs to be minimized and which is a measure between the learned distribution and the true prior distribution.

### 2.2. Cross-validation & Hyperparameter Tuning by the Optuna

Here, I used the Kfold method of the Sklearn library and wrote a custom code for the cross-validation by combining the Optuna hyper parameter tuning method. I would like to represent some sample random images after reconstruction via variational convolutional autoencoder. After training, one can observe the sample reconstructed image as in Figure 1.11.

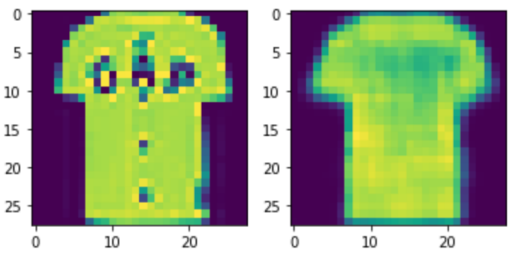


**Figure 1.11**

Due to the computational complexity and running time duration I had to keep epoch numbers small, but in order to show how one can implement the cross-validation and hyper-parameter tuning along with the variational convolutional autoencoders, I've written a customized objective method. The customized code is producing the output as in Figure 1.12.

```
In epoch 0, Training Error: 24720.62109375, Validation Loss: 0.021405979990959167
In epoch 1, Training Error: 24949.31640625, Validation Loss: 0.021405979990959167
In fold 1, Mean Accuracy is : 0.9785940051078796

In epoch 0, Training Error: 25311.2265625, Validation Loss: 0.021405979990959167
In epoch 1, Training Error: 25538.55859375, Validation Loss: 0.021405979990959167
In fold 2, Mean Accuracy is : 0.9785940051078796

[I 2021-12-04 20:00:13,217] Trial 1 finished with value: 0.9785940051078796 and parameters: {'lr': 0.0114306058051203
6}. Best is trial 0 with value: 0.9785940051078796.
```

**Figure 1.12**