EECS 445 PROJECT 2 REPORT

PART 1 Data Processing

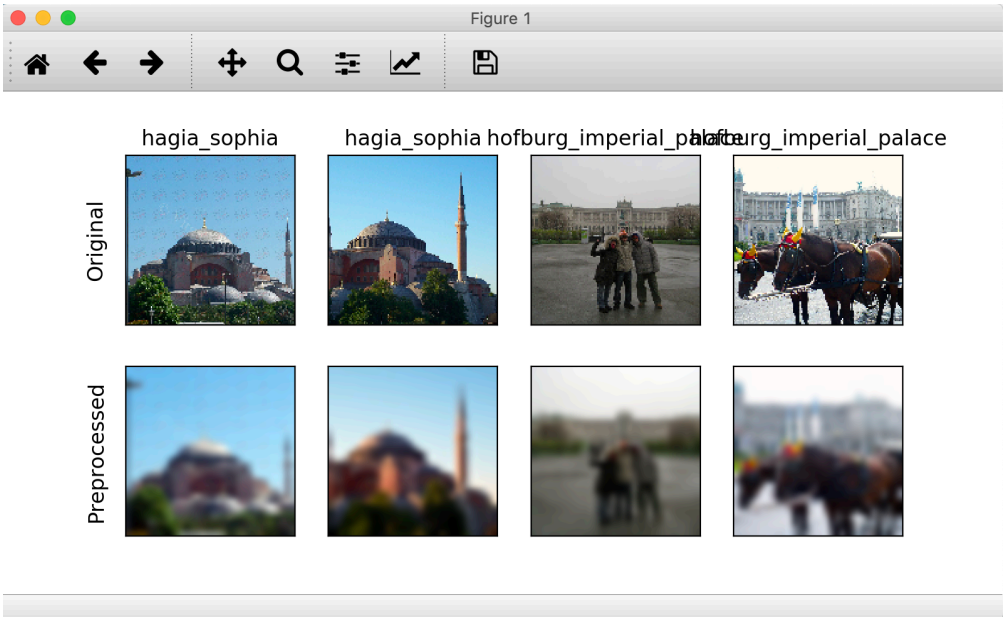
(a) Code Only

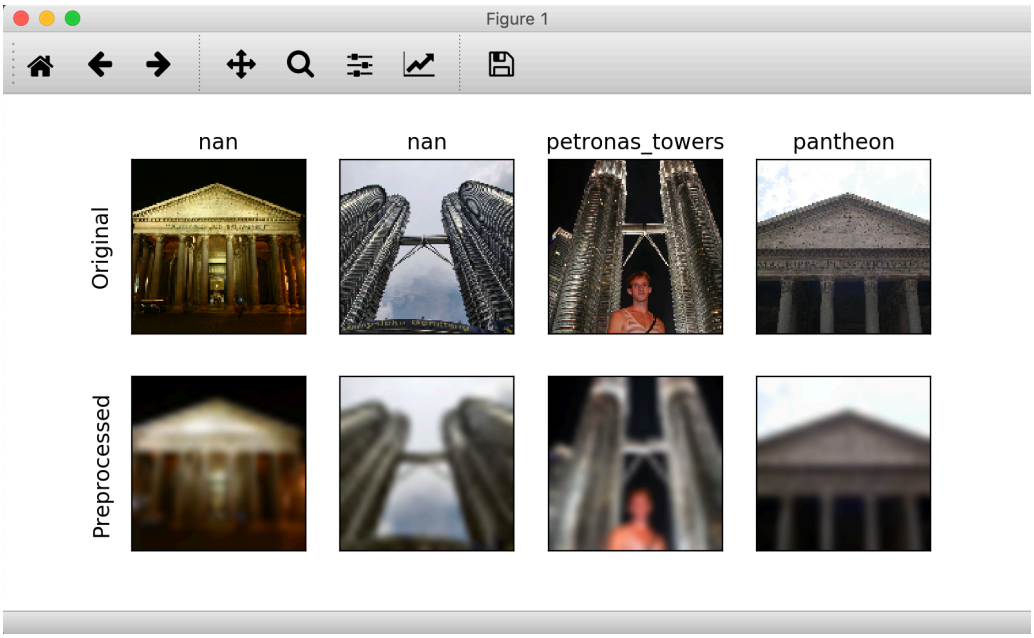
(b) The OUTPUT of CODE is:

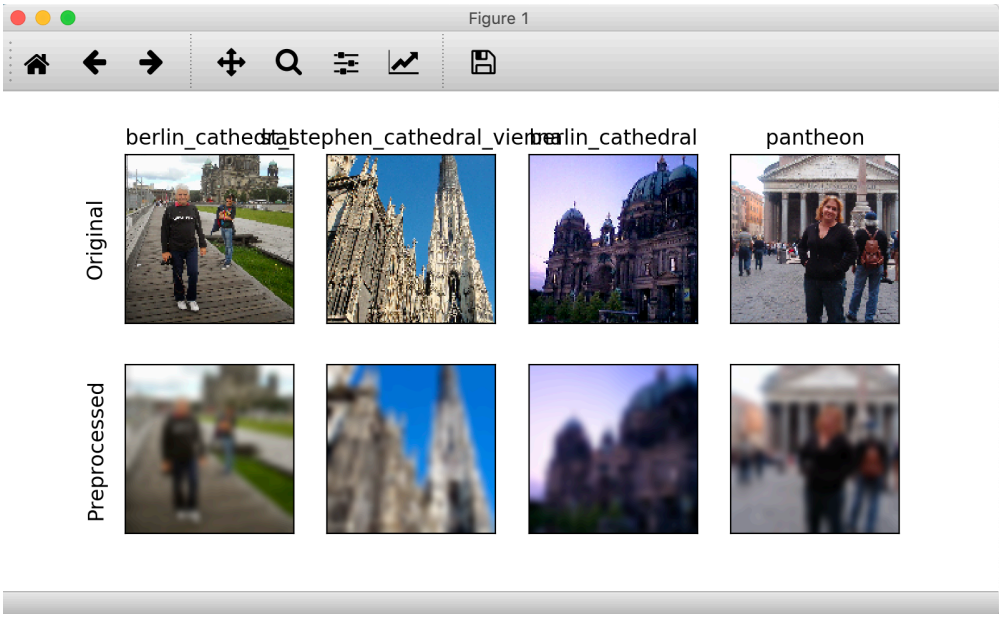
```
loading train...
loading val...
loading test...
Train:   7500
Val:     2500
Test:    2500
Mean: [116.456 118.613 118.74 ]
Std:  [65.083 67.571 74.561]
```

Because when we are doing training, usually we only have access to the training data, but not validation data or test data. So, while we are doing normalization, it's better for us to only normalize the training data but not the others.

(c) The OUTPUT IMAGE is:







It is clearly that after we resize the image data and then visualize them, they are becoming more blur and vague than before. However, it would not be bad things for our training because too many pixels would be too “noisy” for us to do training. We make this image blur and contain somehow less information, but we still can recognize the object and characteristic in this image. Than would be perfect for our future Convolutional Neural Network.

PART 2 Convolutional Neural Network

(a) Convolutional Layer 1 has 16 filters and each with filter size 5 by 5 with 3 channels:

$$\#\ of\ parameter = 16 \times (3\ \times\ 5\ \times\ 5 + 1) = \mathbf{1216}$$

Convolutional Layer 2 has 64 filters and each with filter size 5 by 5 with 16 inputs:

$$\#\ of\ parameter = 64 \times (16\ \times\ 5\ \times\ 5 + 1) = \mathbf{25664}$$

Convolutional Layer 3 has 32 filters and each with filter size 5 by 5 with 64 inputs:

$$\#\ of\ parameter = 32 \times (64\ \times\ 5\ \times\ 5 + 1) = \mathbf{51232}$$

Fully connected Layer 1 has 512 inputs with 64 outputs:

$$\#\ of\ parameter = (512 + 1)\ \times\ 64 = \mathbf{32832}$$

Fully connected Layer 2 has 64 inputs with 32 outputs:

$$\#\ of\ parameter = (64 + 1)\ \times\ 32 = \mathbf{2080}$$

Fully connected Layer 3 has 32 inputs with 5 outputs:

$$\#\ of\ parameter = (32 + 1)\ \times\ 5 = \mathbf{165}$$

Therefore, the total number of parameters is:

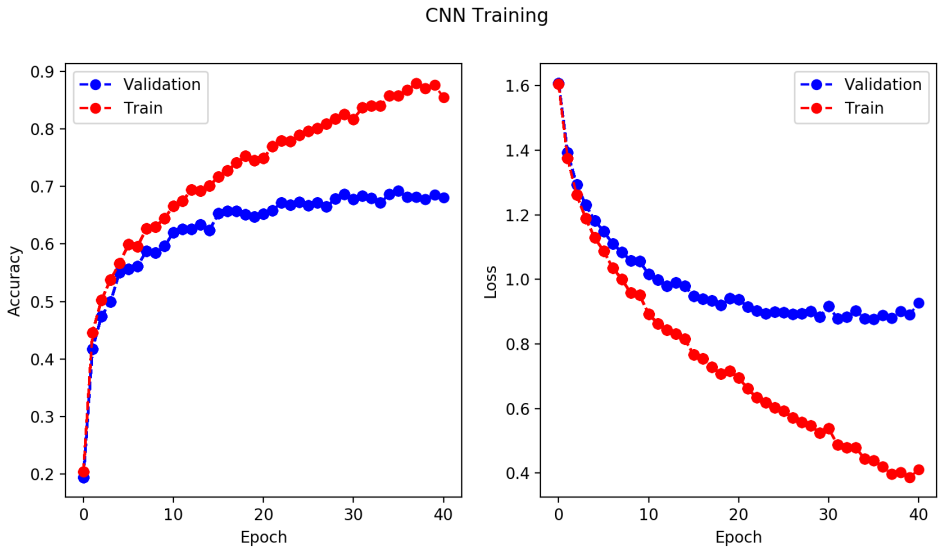
$$Total\ of\ parameter = 1216 + 25664 + 51232 + 32832 + 2080 + 165$$
$$= \mathbf{113189}$$

(b) Code Only

(c) Code Only

(d) Code Only

(e) Final graph is as below:



i. One important factor is that we are using similar algorithm to Stochastic Gradient Descent, which would not monotone decreasing. For our gradient descent, we update the weight only based on one point, thus, we will get a faster convergence and better optimal, but the trend would not be monotone decreasing.

Another important factor is that possibly our learning rate is too large so that we may overshoot, which result in a fluctuating pattern of our loss.

According to our algorithm, we are talking gradient and diminishing the loss so we should have a monotone decreasing loss if we take gradient respect to all data points. And possibly our training data has some noise or outliers.

ii. To Training Loss, it would continue decreasing with the increasing of epoch because we are still doing SGD and is on the way to reach the optimal parameter of our neural network on the training data. However, the decreasing rate would be lower because we are closer to the optimal situation and the gradient would become smaller. But in general, the more training data we have, the more SGD steps we do, we will get a set of parameters that closer to the optimal one, even with huge amount of ***overfitting***.

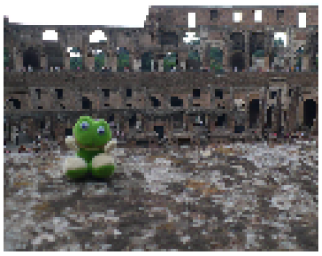
To Validation Loss, it would remain the same with the increase of epoch. Because we can find the trend from the graph that the performance and loss of the validation set almost remain the same after the epoch number of 25, which indicates that 25 is an appropriate number of epochs for us to get a good neural network parameter. The more epoch we have, we cannot get better performance because we are only increasing the amount of overfitting, which means that Training loss would decrease but not effect to Validation Loss.

Training Loss and Validation Loss are close related to each other when we have low number of epochs because a set of underfitting parameters would do bad on both Training and Validation dataset. However, with the increasing of epochs, they will become less correlated because while we are facing overfitting, Training Loss would still increase but Validation Loss would remain the same no matter how many epochs we add into the SGD.

iii. As I stated in (ii), I would choose Epoch = 25 for my future training model. Because by analyzing the trend of the validation loss, we can conclude that after Epoch = 25, the performance of the validation data remains the same, which indicates that after point of 25, the increase of epoch would only increase overfitting but has no effect on the performance of future prediction. Thus, I believe ***Epoch = 25*** is a good threshold, lower would result a model that underfit and higher would result in overfit.

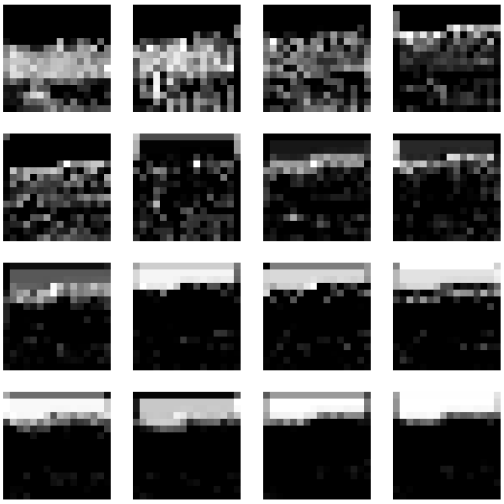
(f)

Original Image

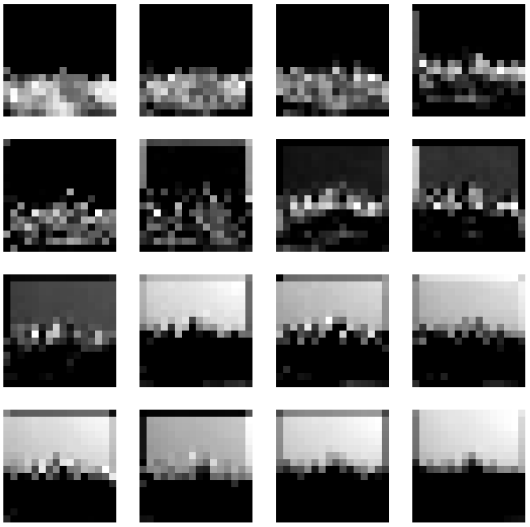


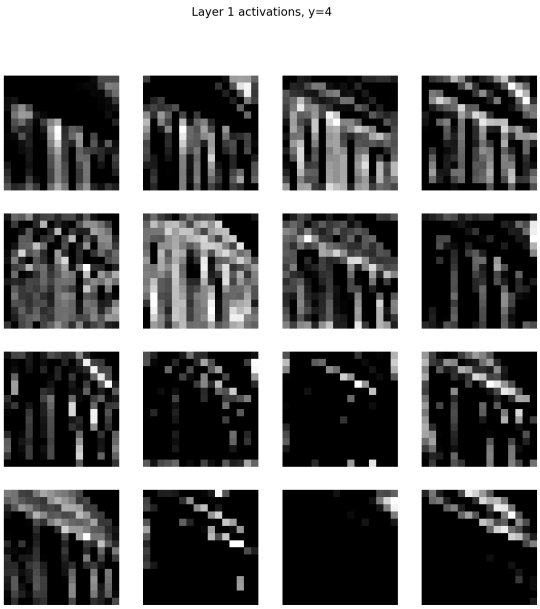
Layer Output

Layer 1 activations, y=0



Layer 1 activations, y=3





i. The most obvious difference between these images are that we can clearly recognize the boundary and even know what's the object in the image in the second and third one, with clear texture. However, in the first image, everything is like a mess after the layer and even the boundaries are pretty vague.

And the common characters are that the output images are seemed to separate the bright pixel and dark pixel in the original image. In all of the three cases, while the bright pixel becomes relatively white, the dark pixel turns totally black, vice versa. And all of the three images become vaguer, the sense of textures becomes larger, we can even see the little pixels in the image. But main object becomes obvious and noise disappears.

I think this is for the reason that the filter in the first layer is separating the bright pixel and dark pixel and point out the boundary of the object, cancel noise and lower the dimension of the image. Therefore, we can find out this bright-dark pattern and see the pixels.

ii. For the filters in the first row, the filter is looking for the dark part of the image. We can easily find the pattern that the relatively dark part of the original image becomes white in the image after the filter. The texture of the dark part is really clear, and we can somehow recognize what the object is in the image.

For the filters in the second row, the filter is looking for the brighter part of the image. The brighter the pixel is in the original image, the whiter it is in the filter image. Same as previous one, we can easily figure out the boundary of the object and recognize what the object is. Within the same row, the resulting image is become darker and darker. Texture is clear in these images.

For the image in the last two rows, they are similar to the second row, only focusing on the bright part. However, these filters are filtering vaguer that the previous two rows. We can hardly find the texture in these images. I believe these images are capturing the most important features or pixels so that the number of important bright feature would be pretty low.

PART 3 Classification by Autoencoder

(a)

i. Encoder: Layer 2, Layer 3. Because they are lowering the dimension.

Decoder: Layer 4, Layer 5. Because they are recovering the image from lower dimension.

And only there four layers have weight.

ii.

	Current Choice	**Alternative**	**How might the alternative choice affect…..**		
			Training Speed	Structure Error	Estimation Error
Initialization	Random Normal	Zero	↓	↑	↓
Activation	ELU	Sigmoid	↓	↑	↓
Depth	4	10	↓	↓	↑
Regularization	None	Dropout	↓	↑	↓

For initialization, we can take zero instead of random normalization. The training speed would absolutely go down because we do not need to take the time to generate random normal distribution and then apply to our data, and we would take more time to converge with 0's initialization. More importantly, if we take all 0 as initial point, actually we are losing randomness in our model, not so generalized, in other word, a simpler model. Therefore, we are facing more structure error because of simple model but lower estimation error.

For Activation, we can take sigmoid function instead of ELU. ELU would just take exponential of the loss is greater than 0 but sigmoid would take exponential value no matter the value of loss. And more importantly, sigmoid gives us a smoother curve than Sigmoid which would be better for our training but more time to converge. Therefore, we incur more structure error because we are inducing a more complex model and thus more estimation error.

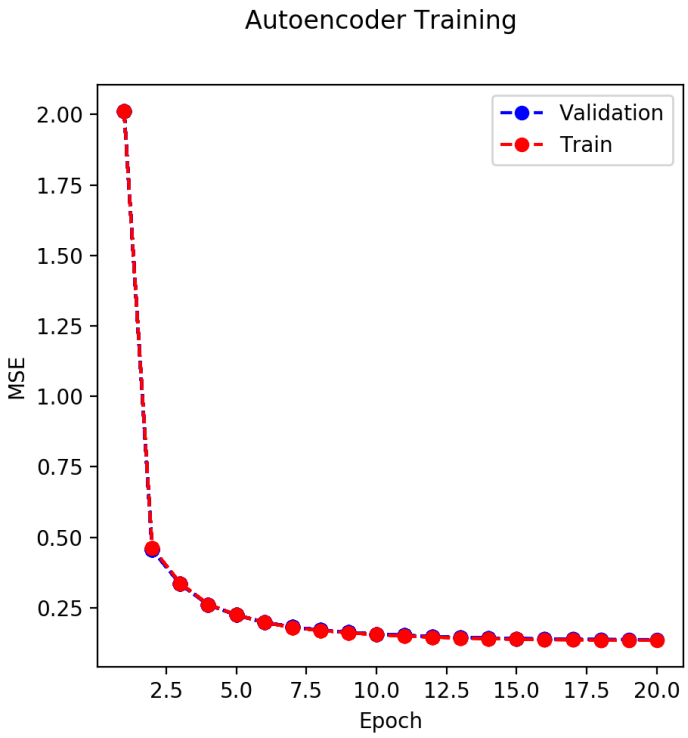
For Depth, we are calculating the total number of hidden layers. With more layers, we are inducing a more complex model and there will be more parameters for us to do training. Therefore, it would take more time. And relating to error, we are facing less structural error because of more layers but more estimation error.

For Regularization, we can take Dropout instead of NONE. We would take some time to determine what to drop out so the overall training speed would be lower. However, because we are doing dropping which reduce the complexity of the model. So different from activation and depth, our structural error would go up, but estimation error would go down because of regularization.

(b) Code Only

(c) Code Only

(d)



We would choose the stopping epoch number which is ***Epoch = 20.***

(e)

i. First, I would check the RGB pixel value for grey, which is [128, 128, 128]. For mean square error, the definition is $MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$. Because we will do normalization before we calculate everything, so if normalize value [128, 128, 128], we will get [0, 0, 0] as our $\hat{Y}_i$ values as 128 is the half of 256.

Then we can simplify the mean square error equation by plug in $\hat{Y}_i = 0$ and get:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - 0)^2 = \frac{1}{n}\sum_{i=1}^{n} {Y_i}^2$$

An interesting finding is that Y_i is just the original pixel value of each channel. Because for each channel the mean value is 0, so we can also see the expression as the variance of the pixel value, which is 1. Therefore, adding three MSE value together and dividing by number, we get the baseline is ***MSE = 1***

ii. The class MSE value is:

Class colosseum: 0.19536691904067993 MSE

Class petronas_towers: 0.2189786434173584 MSE

Class rialto_bridge: 0.15939229726791382 MSE

Class museu_nacional_dart_de_catalunya: 0.14018261432647705 MSE

Class pantheon: 0.19629919528961182 MSE

Class hofburg_imperial_palace: 0.15322504937648773 MSE

Class berlin_cathedral: 0.15212298929691315 MSE

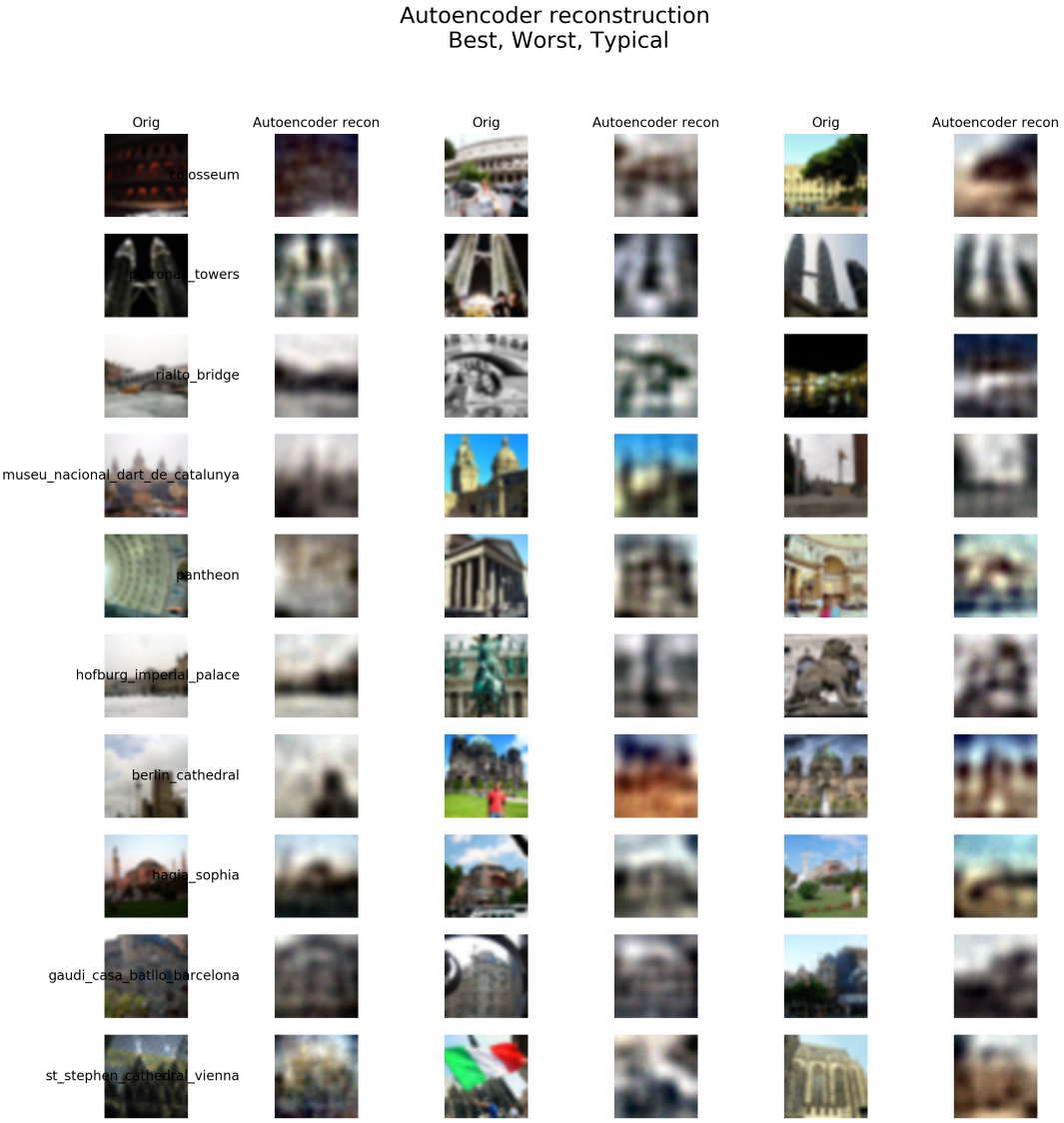
Class hagia_sophia: 0.14511488378047943 MSE

Class gaudi_casa_batllo_barcelona: 0.20037807524204254 MSE

Class st_stephen_cathedral_vienna: 0.14911891520023346 MSE

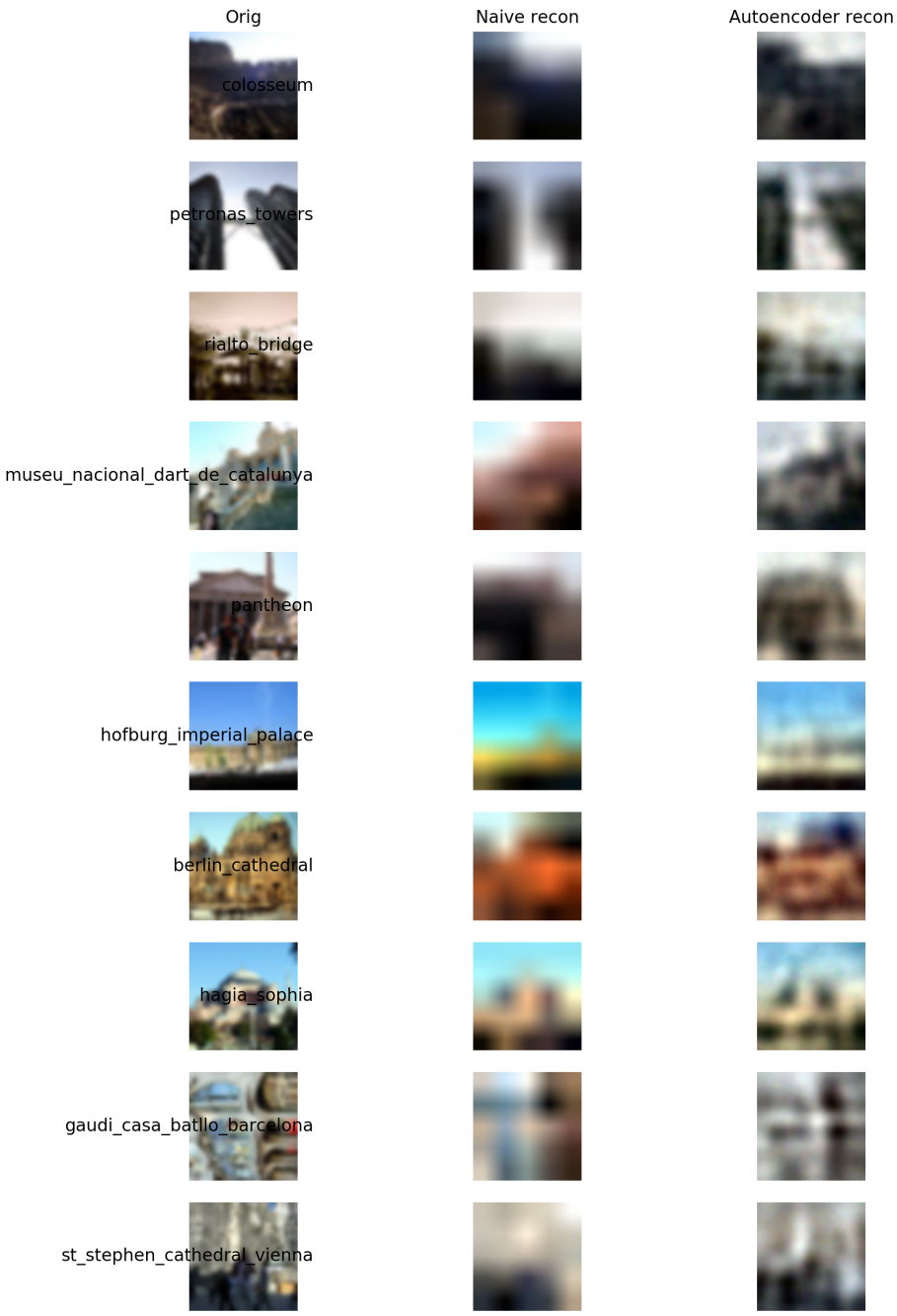
Therefore, the overall MSE which is the average of the above MSE's = ***0.17101795822***

iii. IMAGE as shown.



It seems that the reconstructions for images that is more colorful and detailed would be worse. We can see that the images in the left most column, which are being reconstructed best, somehow has less colors and less detailed than the middle column, which is the worst column. The middle row images always have more color such as the national flag and the man in red where the algorithm is very bad at. The right row is somehow in median, so it would be type TYPICAL.

(f)



Our Autoencoder learns the boundary of the landmarks and the approximate major color of some pixels best. I think we can somehow recognize what the actual landmark is and are able to classify the images by the reconstruction we have; thus, I think autoencoder is doing a good

job even though the reconstruction is not so perfect. The naïve one compresses the image more than the autoencoder one does. So the naïve one may have larger compression ratio.

Comparing to the naïve reconstruction, our Autoencoder one has more detailed color and better boundaries for all the objects. More importantly, the autoencoder reconstruct more things than the naïve one such as the bridge in the second image. There characteristics are very important for us to do the classification, which indicates that Autoencoder is performing better than the naïve one.

(g)

	CNN Accuracy	**Autoencoder Classifier Accuracy**
y = 0; Colosseum	0.376	0.32
y = 1; Petronas Towers	0.812	0.756
y = 2; Rialto Bridge	0.716	0.496
y = 3; MNAC	0.764	0.688
y = 4; Pantheon	0.736	0.548

This Autoencoder Classifier seems performing worse than CNN because all of the accuracy is lower than that of our CNN accuracy in Q2. However, considering training time, Autoencoder takes less time than CNN classifier.

PART 4 Challenge

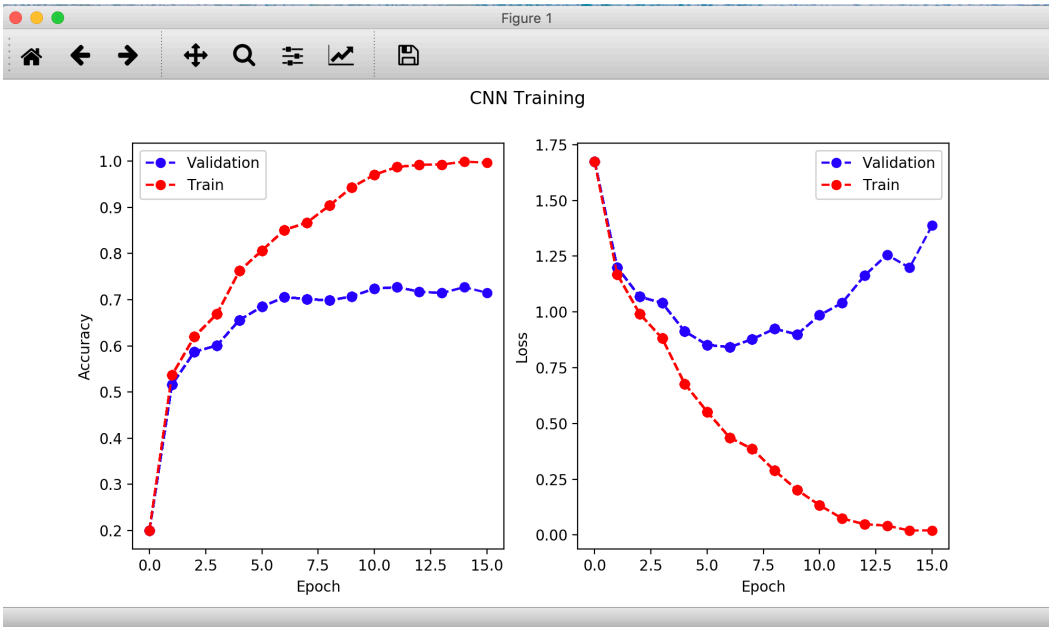
Note:

I copy-paste the implement of CNN structure in parameters in part 2 to set as BENCHMARK.

Any performance that worse than the BENCHMARK would be rejected.

Benchmark Validation Set Performance:

```
Finished Training
classes accuracy: [0.648 0.796 0.668 0.82  0.644]
overall accuracy: 0.7152
```



- Model evaluation

  The first and the most important rule for me to evaluate the performance of the model is accuracy. The higher the validation accuracy, the better the model. However, if the performances of models are similar to each other, then I'll look at the runtime and the overfitting problem. The less the runtime, the better the model. And from measuring the distance between the red and blue graph showing above, we can get the extent of overfitting we are facing. Generally, the less the overfitting, the better the model.

- Model Architecture

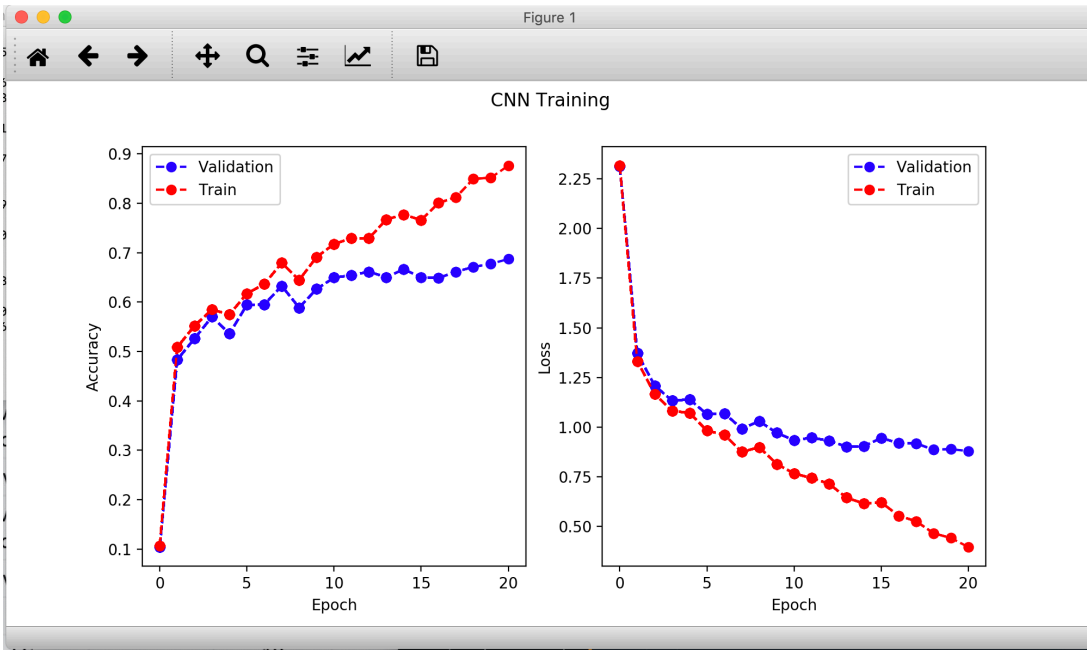
  Among all the hyperparameters that I need to determine, I believe the model architecture is the most important one that I need to determine before all of the others.

  MODEL TRAINING SET PARAMETERS:

  criterion = torch.nn.CrossEntropyLoss()

  optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay = 0.001)

  To get the starting rough architecture, I apply ***Lenet-5 Architecture*** as my starting point, I implement this architecture and then test on my data, the performance is:



  Which seems performing not so good on our image classification problem. Huge amount of overfitting even though I'm applying weight decay, around 0.9 on train data but below 0.7 for validation data. So, I'm going to change the structure parameter and add some layer to get better parameters good for our data.

  First, to avoid too much overfitting, I decide to apply ***dropout*** on my data structure after all the convolutional layers and before the fully connected layer. The dropout probability, which is a hyperparameter, will be tested in the hyperparameter part. I add this part

because the number of parameters reaches a max after we do apply convolutional layers while the overfitting problem would come a max. Therefore, adding a dropout there would be a good idea.

Then I'm going to change all the AvgPool2d in the Lenet-5 structure to ***MaxPool2d.*** Although this is a way to diminish overfitting, but it's also kind of feature selection. We are only counting the feature that is large enough, getting rid of many noises so that our classifier would be performing better. For the parameter of MaxPool2d, I apply the same parameter as Lenet-5, kernel size of 2 and stride 2, it seems performing best after my several tries.

For ***Convolutional Layer***, I add one layers in order to dug deeper into the image and extract some more useful information. I increase the kernel size for getting better feature selection, choosing more important features than just doing normal level of selection. But the increase of kernel size would result in decrease in final feature size, so I add some padding in each convolutional layer.

I try to add some ***fully connecting layers*** to the structure, however, because we are already suffering overfitting, this idea turns to not so good after I run many times. Therefore, I only add one layers for an intermediate step between one small and one large value of parameter. I lower the kernel size and increase some input and output parameters. Trying to get more features from the convolutional layer and select useful on in the fully connected layer. I increase some parameter in ***Conv2d*** and decreases some in ***Linear***.

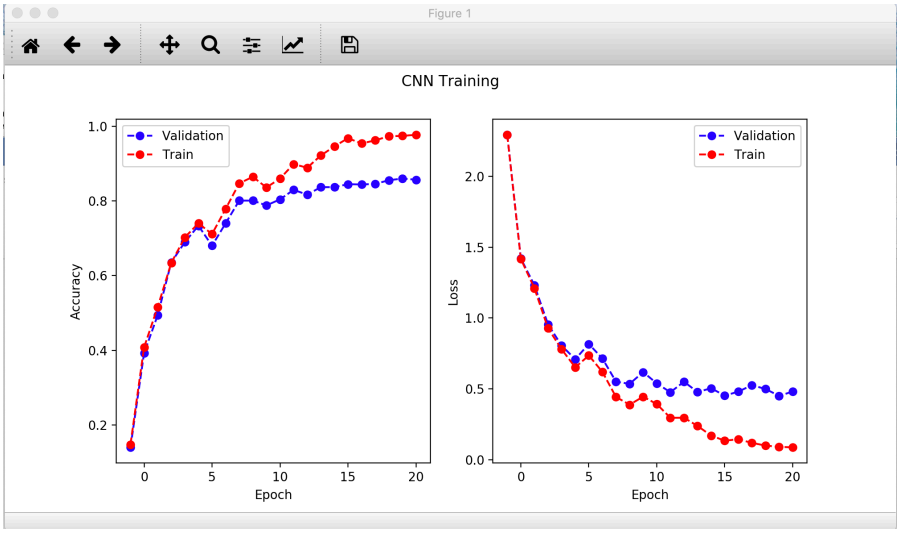
And for the ***loss function*** and ***optimizer***, I tried several optimizers like SGD and ADAGRAD but still, Adam keeps its rank 1 position so I'm keeping with Adam. Many of the loss functions are not available for our project and I try the limited available one like MSE,

same as optimizer, our original one performs best so I do not change our loss function and optimizer.

For the ***weight Initialization***, I try different normalization function like *Kaiming_normal_initialization* and Xavier but they seem performing worse on our dataset than our original setting. Therefore, I do not change anything in the weight initialization in this project.

There are tos of parameter tries and structure tries that turn out to be a failure during my training, but luckily, finally I come up with a relatively good structure that have performance around 0.85, which is pretty good to me.

```
Epoch 20
        Validation Loss: 0.4508930087089539
        Validation Accuracy: 0.86
        Train Loss: 0.09119183259705703
        Train Accuracy: 0.9741333333333333
Finished Training
classes accuracy: [0.768 0.884 0.804 0.952 0.868]
overall accuracy: 0.8552
```

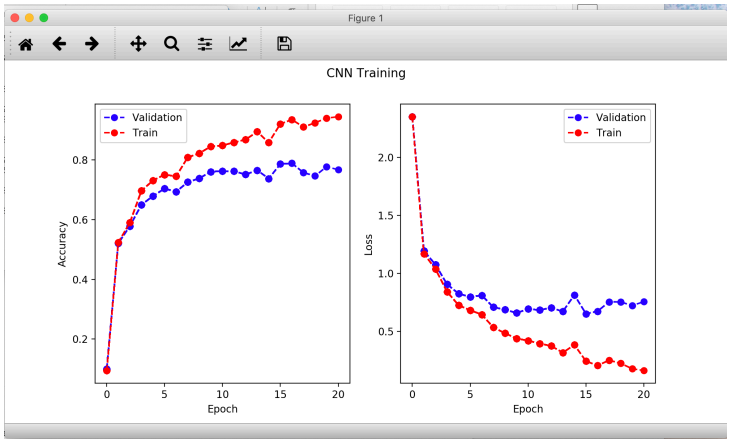


- Regularization
  - DROPOUT

    As you can see in Mode Selection, I add a dropout after the convolutional layers. Because DROPOUT would automatically drop by our given probability. We are now facing overfitting, so it's time for dropout! Thus, I'm going to test on the hyperparameter p on it. Performance and graph and parameters are shown below:

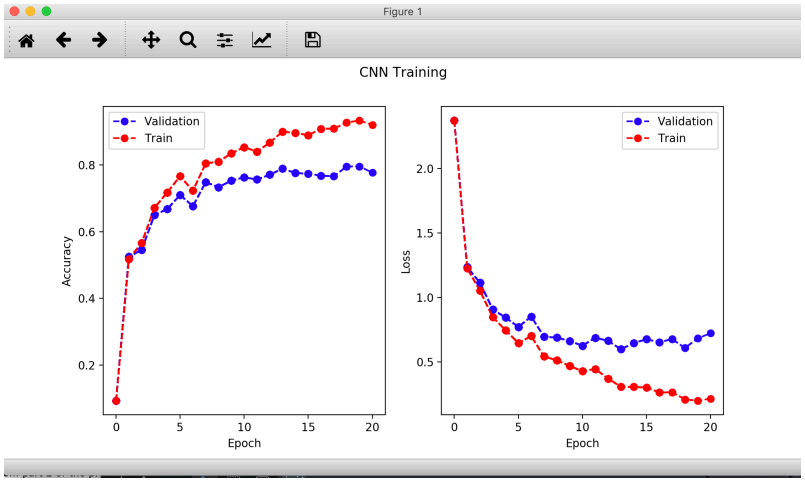
P = 0.2

```
Finished Training
classes accuracy: [0.496 0.84  0.852 0.904 0.772]
overall accuracy: 0.7728
```



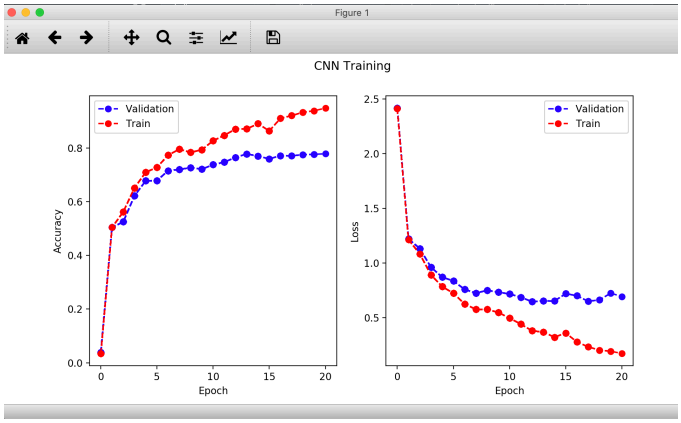
P = 0.3

```
Finished Training
classes accuracy: [0.584 0.892 0.844 0.92  0.648]
overall accuracy: 0.7776
```



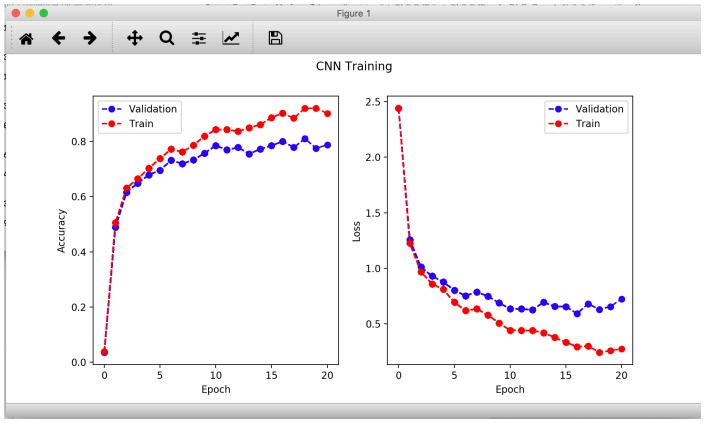
P = 0.25

```
Finished Training
classes accuracy: [0.692 0.84  0.652 0.868 0.808]
overall accuracy: 0.772
```



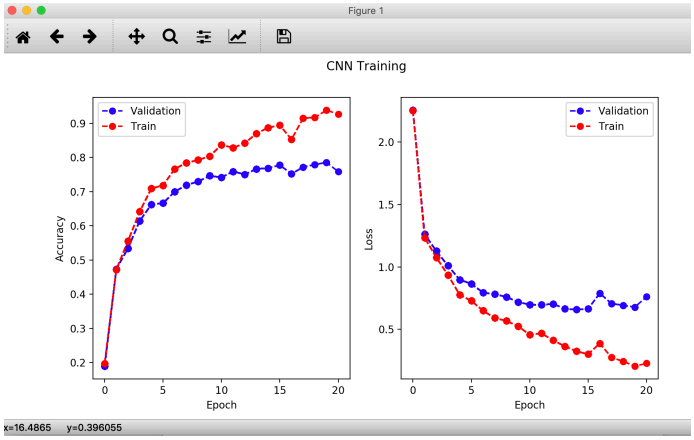
P = 0.5

```
Finished Training
classes accuracy: [0.804 0.812 0.608 0.928 0.796]
overall accuracy: 0.7896
```



P = 0.4

```
Train Accuracy: 0.9280000000000000
Finished Training
classes accuracy: [0.588 0.848 0.828 0.876 0.724]
overall accuracy: 0.7728
```



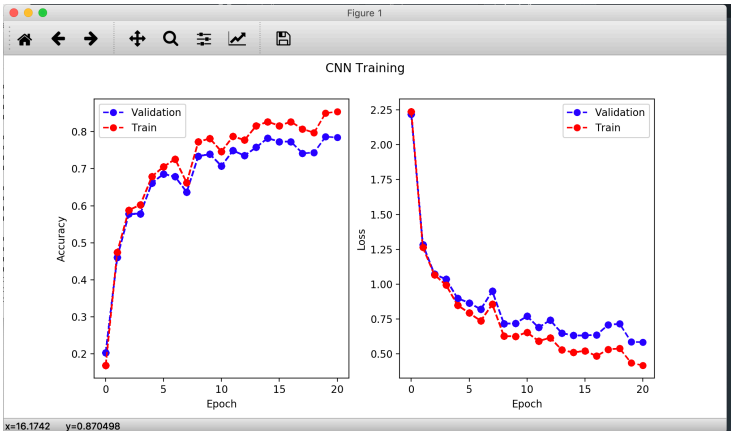
Therefore, among all these values, I'll choose the one with highest accuracy which is p = 0.5 for my following training.

- Weight Decay

  Obviously, we are facing great overfitting while we are evaluating the model. So, every way that would reduce overfitting would be good for us. I add so weight decay on the optimizer so that I can reduce the weight by a little proportion so that to drag the parameter away from overfitting. And there exist one hyperparameter, the coefficient, which I would call it c, to determine. I tried several values to test which one would be better.

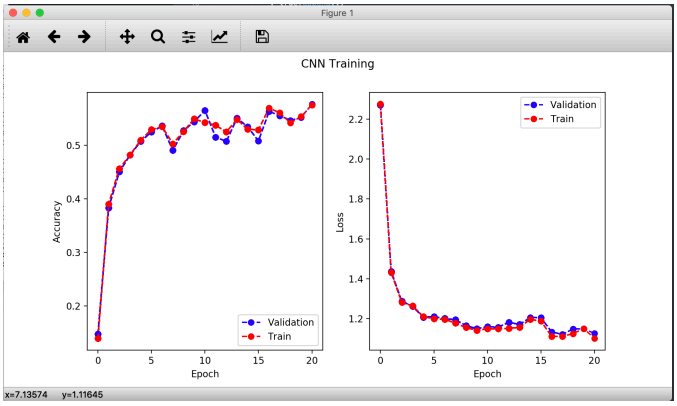
C = 0.01

```
Finished Training
classes accuracy: [0.632 0.812 0.796 0.912 0.748]
overall accuracy: 0.78
```



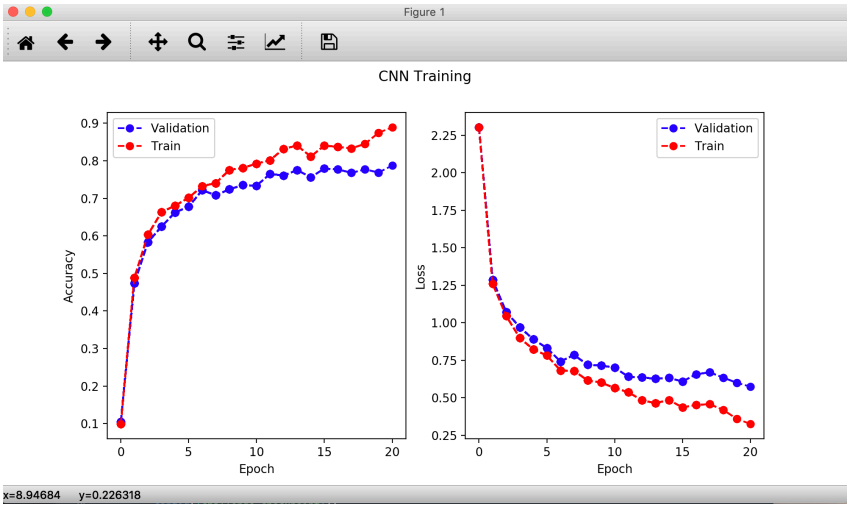
C = 0.1

```
Finished Training
classes accuracy: [0.508 0.752 0.404 0.76  0.42 ]
overall accuracy: 0.5688
```



C = 0.05

```
Finished Training
classes accuracy: [0.668 0.828 0.792 0.912 0.788]
overall accuracy: 0.7976
```



Among these values, we can easily find out that while c = 0.1, we are decaying too much from the parameter, so the accuracy remains low. While c = 0.05, we would reach a relatively stable and good parameter.

- Hyperparameter
  - Dropout rate and weight decay rate

    I've already tested these two in the previous ***Regularization*** part, please refer there to see the detailed testing.

  - Learning rate

    Learning rate is the most important hyperparameter in our training. If it's too low, it would take really long for us to reach the minimal loss point. If it's too large, we may face overshooting and never get to the optimal point. Therefore, good step size really decides our final result. According to the previous training, I try three value to compare on the learning rate and the performance are listing below.

lr = 1e-2

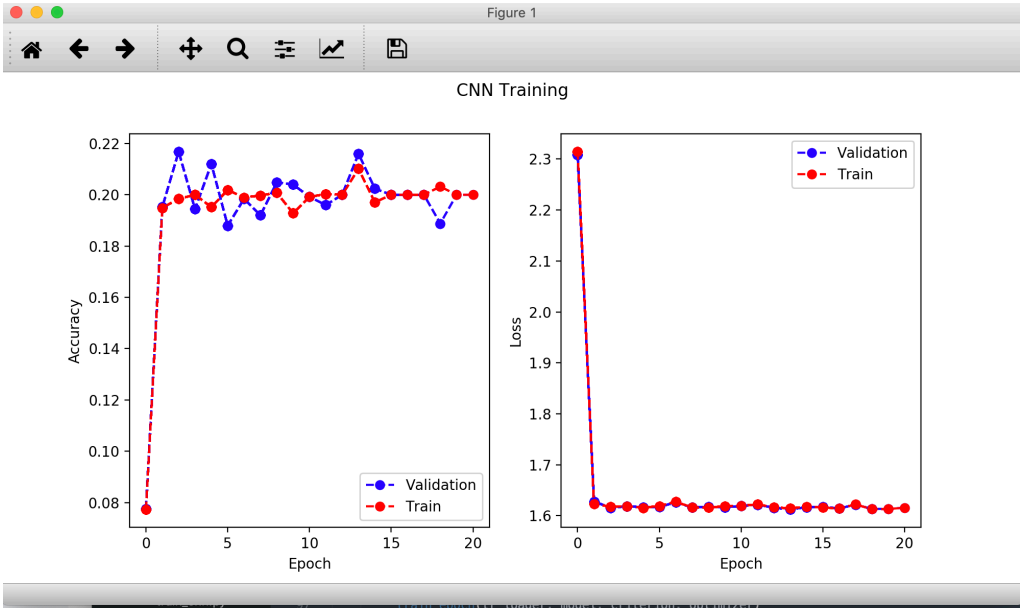


Figure 1
CNN Training
Accuracy
Epoch
Validation
Train
Loss
Epoch
Validation
Train

lr = 1e-3

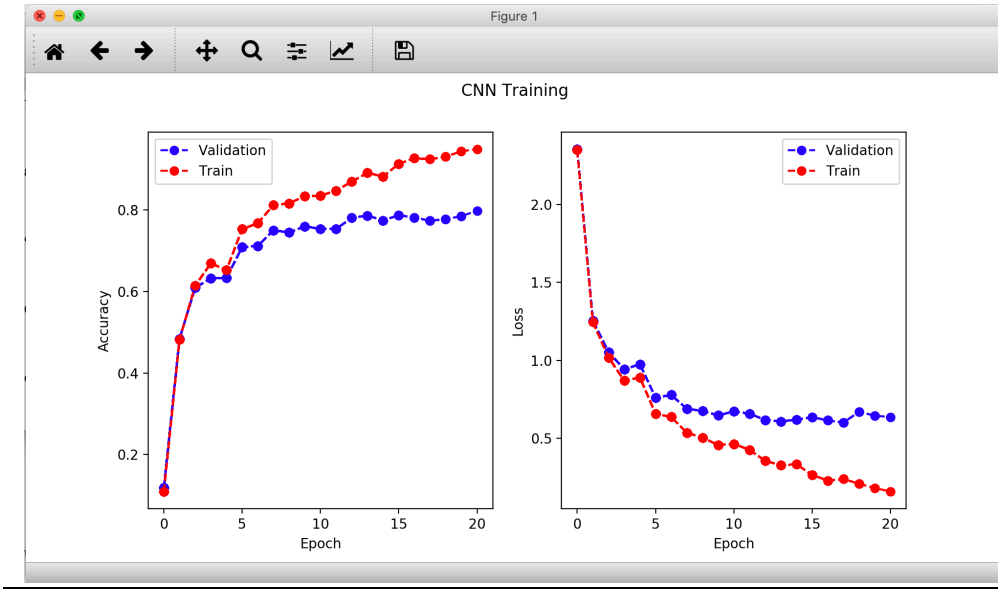
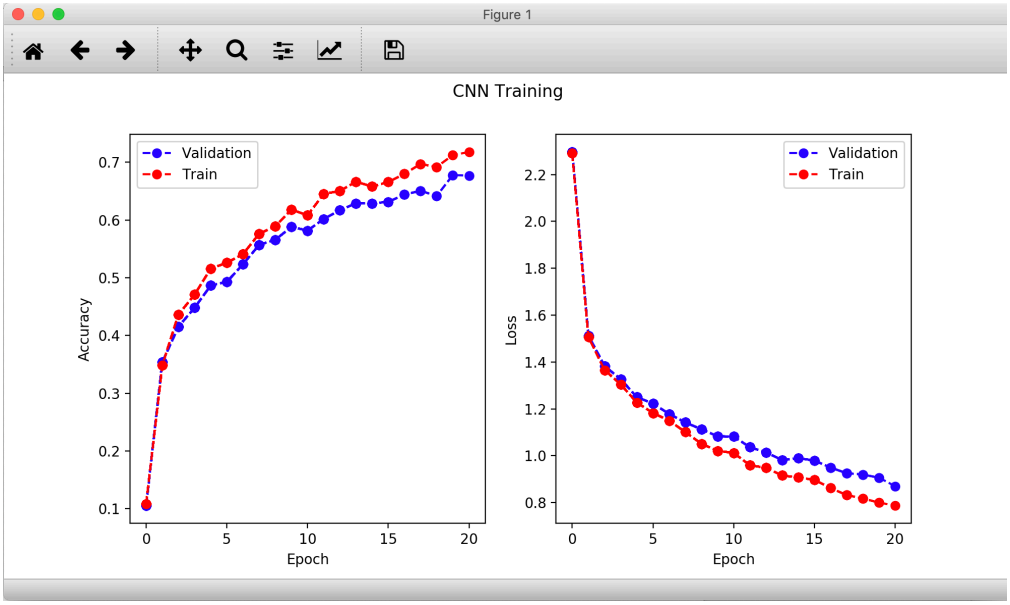


Figure 1
CNN Training
Validation
Train
Accuracy
Epoch
Validation
Train
Loss
Epoch

lr = 1e-4



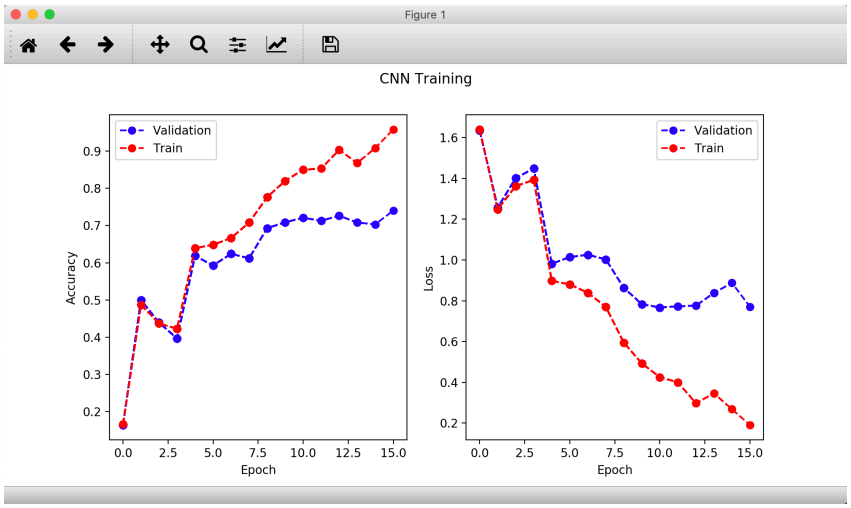
We can find that for lr = 1e-2, it would be too large so that our parameter is not moving towards the minimum loss point and is overshooting along the loss function. The accuracy remains at the level of 0.2 which is terribly low. While the situation comes to 1e-3 and 1e-4, it becomes much better. Although it seems like we are facing more overfitting in 1e-3 case than 1e-4, however, the performance of 1e-3 is better than that of 1e-4 so finally I would choose 1e-3 as my final learning rate.

- Activation function

  I have tried four types of common activation function, ***RELU***, ***ELU***, ***SIGMOID*** and ***TANH.*** And run these activation function on the same dataset, the performance is list following,

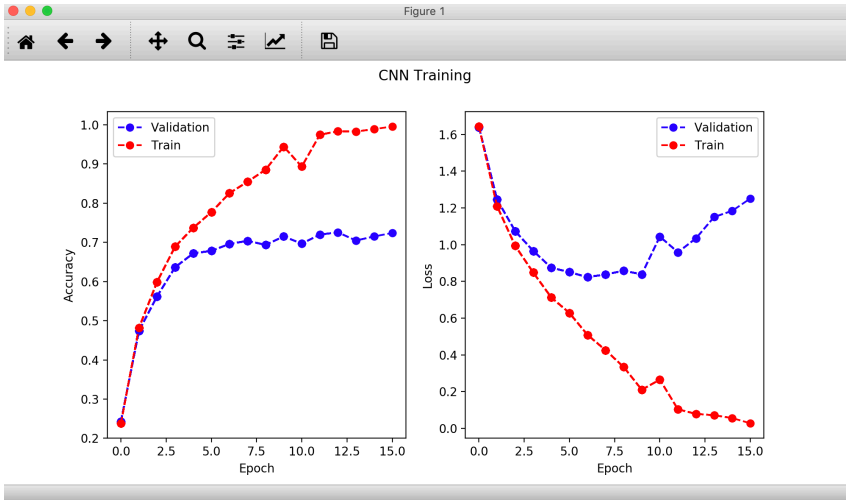
RELU

```
Finished Training
classes accuracy: [0.556 0.896 0.692 0.904 0.652]
overall accuracy: 0.74
```



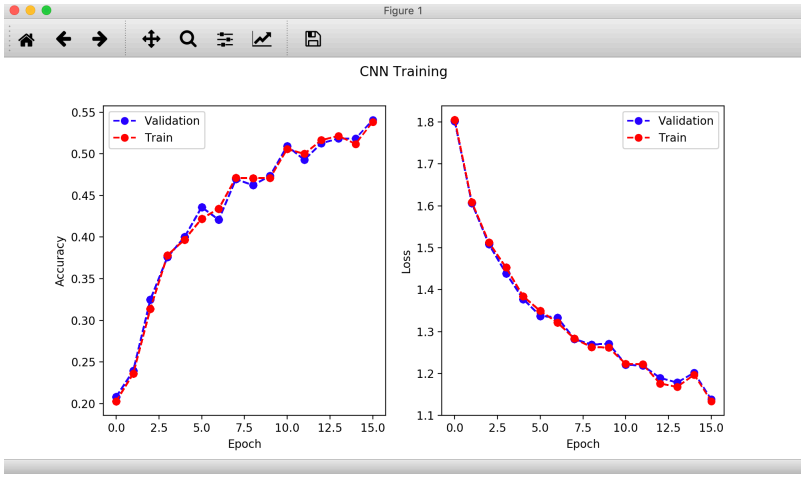
ELU

```
Finished Training
classes accuracy: [0.684 0.84  0.708 0.812 0.576]
overall accuracy: 0.724
```



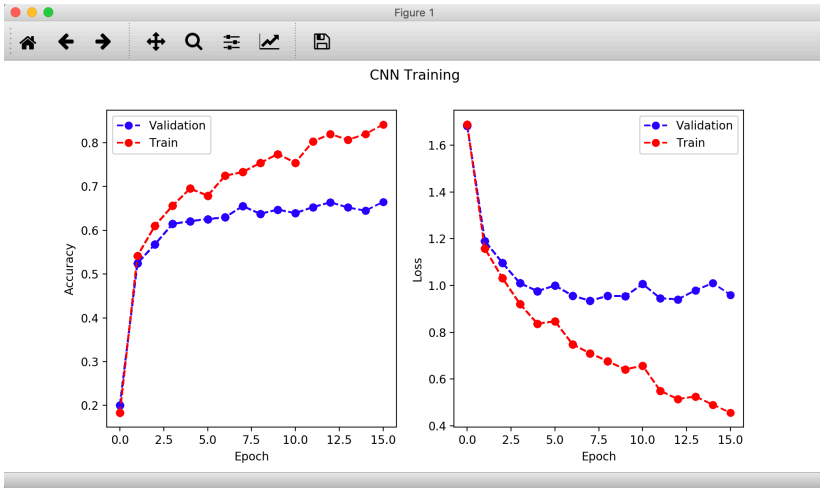
SIGMOID

```
Finished Training
classes accuracy: [0.236 0.792 0.312 0.796 0.56 ]
overall accuracy: 0.5392
```



TANH

```
Finished Training
classes accuracy: [0.408 0.768 0.6   0.824 0.708]
overall accuracy: 0.6616
```



An interesting trend is found that sigmoid really minimize our overfitting problem, the red like and blue line perfectly match. However, the performance is not so good maybe because of sigmoid is too smooth for gradient decent. Therefore, considering several factors like overfitting and performance, I'll choose ***RELU*** as my main activation function.

- Feature Selection & Extraction

  Feature selection is critical to our model. There are many ways to select features and I use the following strategies.

  - CONVOLUTIONAL LAYER (FILTER)

    This is the really normal way for us to do feature extraction and feature selection. While we are applying filter to our model, actually we are also doing feature selection on our model. With different filters and different values on the filter, we can extract various of feature of our image for the future training.

    In my model, for getting better feature extractions and remain our dimension for not losing information, I would use padding along with big size of the kernel. Therefore, while we are doing feature selection, we can have better considering of the features on the boarder and moreover, remaining the size of our feature may help us to do better on the following filters.

    I try to keep the result of my convolutional layers as x * 2 * 2 where x is a number around 100 to 200 because I believe this number can help me to perform better in the following fully connected layer. And other hyperparameters of the layers are all test by me on hand, running and waiting tons of time to finalize a best one.

  - MAXPOOL LAYER

    This layer has two main function – feature selection and regularization. It could choose the max value given the kernel size. Therefore, applying a maxpool layer can help us extract the max value from the features. And we know the features with large value after our filtering are usually the most important value to our training. Thus, taking a maxpool can get rid of some less important noise around, which is doing feature selection. And moreover, by cancelling these little noises, we are also doing regularization to avoid overfitting.

The two parameter, kernel and stride of MAXPOOL layer are determined by many tries on the same dataset and I finalize one with the least loss and best accuracy.

- GPU Usage
    - NO use of GPU hardware

      I do not want to use existing pretrained model and construct all layers by hand. The worst-case runtime is only around 10 mins. Therefore, I believe I do not need to use GPU for faster runtime, I can accept 10 mins per run while I'm training.

APPENDIX CODE

1 – a

```
def resize(X):
    """
    Resizes the data partition X to the size specified in the config file.
    Uses bicubic interpolation for resizing.

    Returns:
        the resized images as a numpy array.
    """
    # TODO: Complete this function
    image_dim = config('image_dim')
    resized = np.zeros((X.shape[0], image_dim, image_dim, 3))
    for i in range(X.shape[0]):
        resized[i] = imresize(X[i], (image_dim, image_dim, 3), 'bicubic')
    #

    return resized
```

1 – b

```
class ImageStandardizer(object):
    """
    Channel-wise standardization for batch of images to mean 0 and variance 1.
    The mean and standard deviation parameters are computed in `fit(X)` and
    applied using `transform(X)`.

    X has shape (N, image_height, image_width, color_channel)
    """
    def __init__(self):
        super().__init__()
        self.image_mean = None
        self.image_std = None

    def fit(self, X):
        # TODO: Complete this function
        self.image_mean = np.zeros((3), dtype = float)
        self.image_std = np.zeros((3), dtype = float)
        for i in range(3):
            self.image_mean[i] = X[:,:,:,i].mean()
            self.image_std[i] = X[:,:,:,i].std()

        #

    def transform(self, X):
        # TODO: Complete this function
        for i in range(3): X[:,:,:,i] = (X[:,:,:,i] - self.image_mean[i]) / self.image_std[i]
        return X
        #
```

2 – b

```python
class CNN(nn.Module):
    def __init__(self):
        super().__init__()

        # TODO: define each layer
        self.conv1 = nn.Conv2d(3, 16, (5 ,5), stride = (2, 2), padding = 2)
        self.conv2 = nn.Conv2d(16, 64, (5 ,5), stride = (2, 2), padding = 2)
        self.conv3 = nn.Conv2d(64, 32, (5 ,5), stride = (2, 2), padding = 2)
        self.fc1 = nn.Linear(512, 64, True)
        self.fc2 = nn.Linear(64, 32, True)
        self.fc3 = nn.Linear(32, 5, True)
        #

        self.init_weights()

    def init_weights(self):
        for conv in [self.conv1, self.conv2, self.conv3]:
            C_in = conv.weight.size(1)
            nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5*5*C_in))
            nn.init.constant_(conv.bias, 0.0)

        # TODO: initialize the parameters for [self.fc1, self.fc2, self.fc3]
        for fc in [self.fc1, self.fc2, self.fc3]:
            C_in = fc.weight.size(1)
            nn.init.normal_(fc.weight, 0.0, 1/ sqrt(C_in))
            nn.init.constant_(fc.bias, 0.0)
        #

    def forward(self, x):
        N, C, H, W = x.shape

        # TODO: forward pass
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))

        x = x.view(-1, 512)
        x = self.fc1(x)
        x = self.fc2(x)
        x = self.fc3(x)
        #

        return x
```

2 – c

```python
def predictions(logits):
    """
    Given the network output, determines the predicted class index

    Returns:
        the predicted class output as a PyTorch Tensor
    """
    # TODO: implement this function
    max_label = torch.max(logits, 1)[1]
    return max_label
    #
```

2 – d

```
# TODO: define loss function, and optimizer
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
#
```

3 – b

```
class Autoencoder(nn.Module):
    def __init__(self, repr_dim):
        super().__init__()
        self.repr_dim = repr_dim

        ## Solution: define each layer
        self.pool = nn.AvgPool2d(2, stride = 2)
        self.fc1 = nn.Linear(768, 128, True)
        self.fc2 = nn.Linear(128, 64, True)
        self.fc3 = nn.Linear(64, 20736, True)
        ##

        self.deconv = nn.ConvTranspose2d(repr_dim, 3, 5, stride=2, padding=2)
        self.init_weights()

    def init_weights(self):
        # TODO: initialize the parameters for
        #       [self.fc1, self.fc2, self.fc3, self.deconv]
        for mod in [self.fc1, self.fc2, self.fc3]:
            c_in = mod.weight.size(1)
            nn.init.normal_(mod.weight, 0.0, 0.1/sqrt(c_in))
            nn.init.constant_(mod.bias, 0.01)


        mod = self.deconv
        nn.init.normal_(mod.weight, 0.0, 0.01)
        nn.init.constant_(mod.bias, 0.00)
        #

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded

    def encoder(self, x):
        # TODO: encoder
        N, C, H, W = x.shape

        x = self.pool(x)
        x = x.view(-1, 768)
        x = F.elu(self.fc1(x))
        encoded = F.elu(self.fc2(x))
        #

        return encoded
```

```
def decoder(self, encoded):
    # TODO: decoder
    x = F.elu(self.fc3(encoded))
    z = x.view(-1 ,64, 18, 18)
    #

    decoded = self._grow_and_crop(z)
    decoded = _normalize(decoded)
    return decoded
```

3 – c

```
# TODO: define loss function, and optimizer
criterion = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)
#
```

3 – g

```
class AutoencoderClassifier(nn.Module):
    # skip connections
    def __init__(self, repr_dim, d_out, n_neurons=32):
        super().__init__()
        self.repr_dim = repr_dim

        # TODO: define each layer
        self.pool = nn.AvgPool2d(2, stride = 2)
        self.fc1 = nn.Linear(768, 128)
        self.fc2 = nn.Linear(128, 64)
        #

        self.fc_1 = nn.Linear(repr_dim, n_neurons)
        self.fc_2 = nn.Linear(n_neurons, n_neurons)
        self.fc_3 = nn.Linear(n_neurons, n_neurons)

        self.fc_last = nn.Linear(n_neurons, d_out)


    def forward(self, x):
        encoded = self.encoder(x)

        z1 = F.elu(self.fc_1(encoded))
        z2 = F.elu(self.fc_2(z1))
        z3 = F.elu(self.fc_3(z2))
        z = F.elu(self.fc_last(z1 + z3))
        return z

    def encoder(self, x):
        # TODO: encoder
        N, C, H, W = x.shape

        x = self.pool(x)
        x = x.view(-1, 768)
        x = F.elu(self.fc1(x))
        encoded = F.elu(self.fc2(x))

        #

        return encoded
```

```
def _train_epoch(data_loader, model, criterion, optimizer):
    """
    Train the `model` for one epoch of data from `data_loader`
    Use `optimizer` to optimize the specified `criterion`
    """

    # TODO: complete the training step
    for i, (X, y) in enumerate(data_loader):
        # clear parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        output = model(X)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()
```

GET ACCURACY

```
    with torch.no_grad():
        y_stat = np.zeros((2,5))
        for X, y in va_loader:
            output = ae_classifier(X)
            predicted = predictions(output.data)
            for i in range(y.size(0)):
                if (y[i] < 5):
                    y_stat[0][y[i]] = y_stat[0][y[i]] + 1
                    if (predicted[i] == y[i]):
                        y_stat[1][y[i]] = y_stat[1][y[i]] + 1
        y_final = np.zeros(5)
        for i in range(5):
            y_final[i] = y_stat[1][i] / y_stat[0][i]
    print(y_final)
```

CHALLENGE

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from math import sqrt

class Challenge(nn.Module):
    def __init__(self):
        super().__init__()

        # TODO:
        # self.maxp = nn.maxpool2d(3)
        # self.conv1 = nn.Conv2d(3, 6, (5 ,5), stride = (2, 2), padding = 2)
        # self.conv2 = nn.Conv2d(6, 16, (5 ,5), stride = (2, 2), padding = 2)
        # self.conv3 = nn.Conv2d(16, 240, (5 ,5), stride = (2, 2), padding = 2)
        # # self.drop = nn.dropout2d(p = 0.1)
        # self.fc1 = nn.Linear(240, 80, True)
        # self.fc2 = nn.Linear(80, 32, True)
        # self.fc3 = nn.Linear(32, 5, True)
        #


        self.conv1 = nn.Conv2d(3, 24, (4 ,4), padding = 2)
        self.conv2 = nn.Conv2d(24, 84, (4 ,4), padding = 2)
        self.conv3 = nn.Conv2d(84, 120, (3 ,3), padding = 2)
        self.conv4 = nn.Conv2d(120, 160, (5 ,5), stride = (2, 2), padding = 2)
        self.conv5 = nn.Conv2d(160, 200, (4, 4))
        self.fc1 = nn.Linear(200 * 2 * 2, 120, True)
        self.fc2 = nn.Linear(120, 84, True)
        self.fc3 = nn.Linear(84, 10, True)
        self.maxp = nn.MaxPool2d(2, stride = 2)
        self.drop1 = nn.Dropout2d(p = 0.5)
        self.drop2 = nn.Dropout2d(p = 0.3)

        self.init_weights()
```

```
def init_weights(self):
    # TODO:
    for conv in [self.conv1, self.conv2, self.conv3]:
        C_in = conv.weight.size(1)
        nn.init.normal_(conv.weight, 0.0, 1 / sqrt(5*5*C_in))
        nn.init.constant_(conv.bias, 0.0)

    # TODO: initialize the parameters for [self.fc1, self.fc2, self.fc3]
    for fc in [self.fc1, self.fc2, self.fc3]:
        C_in = fc.weight.size(1)
        nn.init.normal_(fc.weight, 0.0, 1/ sqrt(C_in))
        nn.init.constant_(fc.bias, 0.0)
    #
    #
```

```
def forward(self, x):
    N, C, H, W = x.shape

    # TODO:
    # TODO: forward pass
    # # x = self.maxp(x)s
    # x = F.relu(self.conv1(x))
    # # x = self.drop(x)
    # # appears not so effective on the accuracy
    # x = F.relu(self.conv2(x))
    # x = F.relu(self.conv3(x))
    # x = x.view(-1, 240)
    # x = self.fc1(x)
    # x = F.relu(x)
    # # important this relu, would a affect a lot during the training
    # x = self.fc2(x)
    # # x = f.relu(x)
    # x = self.fc3(x)
    # #

    x = F.relu(self.conv1(x))
    x = F.relu(self.conv2(x))
    x = self.maxp(x)
    x = F.relu(self.conv3(x))
    x = self.maxp(x)
    x = F.relu(self.conv4(x))
    x = F.relu(self.conv5(x))
    x = self.drop1(x)
    x = x.view(-1, 200 * 2 * 2)
    x = self.fc1(x)
    x = self.drop2(x)
    x = self.fc2(x)
    x = self.drop2(x)
    x = self.fc3(x)

    return x
```

```
# TODO: define model, loss function, and optimizer
# model = models.resnet18(pretrained = False)
model = Challenge()
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay = 0.005)
#
```

```
"challenge": {
    "checkpoint": "./checkpoints/challenge/",
    "learning_rate": 1e-3,
    "num_epochs": 30,
    "batch_size": 128,
    "num_classes": 5
}
```