Can Kocagil
21602218

Bilkent University

Department of Electric and Electrical Engineering

EEE443/543 Neural Networks

Mini Project 3

27.11.2020

TABLE OF CONTENTS

1 QUESTION 1

In this question, you will implement an autoencoder neural network with a single hidden layer for unsupervised feature extraction from natural images. The following cost function will be minimized:

$$J_{ae} = \frac{1}{2N}\sum_{i=1}^{N}\|d(m) - o(m)\|^2 + \frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}\left(W_{a,b}^{(1)}\right)^2 + \sum_{c=1}^{L_{out}}\sum_{b=1}^{L_{hid}}\left(W_{b,c}^{(2)}\right)^2\right] + \beta\sum_{b=1}^{L_{hid}} KL(\rho|\hat{\rho}_b)$$

1.1 PART A

In this part, questions ask to implement image preprocessing methods to prepare the input to feed to the network. To do that, firstly image will be converted into gray scale then will be normalized according to the rules that will be discussed in following parts.

Gray Scaling by Luminosity Model

In this part, input image should be converted into gray scale format by luminosity model. Then, our equation of luminosity is given as:

$$Y_{linear} = 0.2126 * R_{linear} + 0.7152 * G_{linear} + 0.0722 * B_{linear}$$

Therefore, the intuition behind this equation is that since red color has more wavelength of all the three colors, and green is the color that has not only less wavelength then red color but also green is the color that gives more soothing effect to the eyes. [1]

In other words, we need to increase the effect of the green, keep the effect of red in between blue and green while decreasing the effect of the blue color. [1]

Finally, this equation is yield that the contribution is coming from 21.26% of red, 71.52% of the green and 7.22% of blue. The visual example can be seen as:

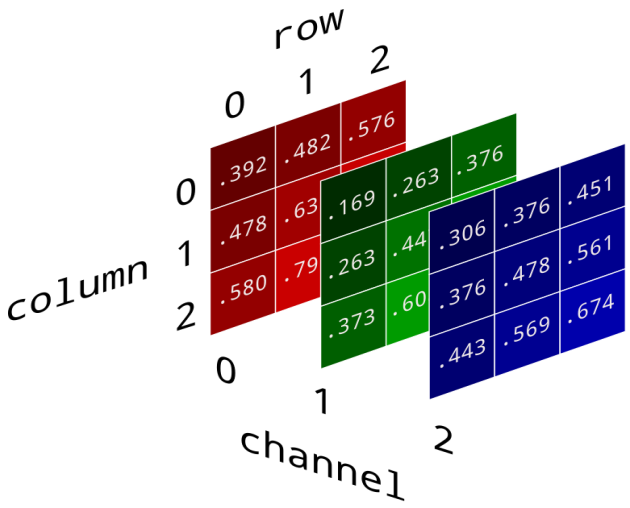


Fig. 1: The visualization of the RGB image data

This is the 3-D model of our training data. As I said, we multiply the contribution of each color by mentioned numbers that can be viewed as:

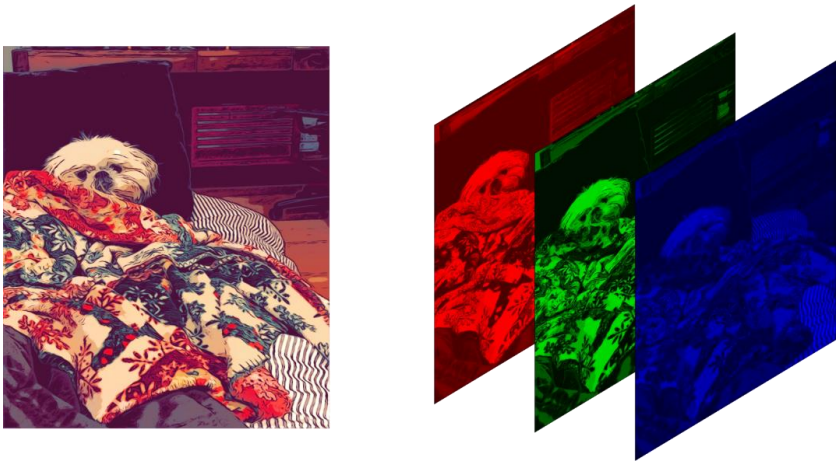


Fig. 2: The visualization of the RGB image data seperately

Here is the dog picture is just randomly selected picture through web to understand the concept of gray scaling.

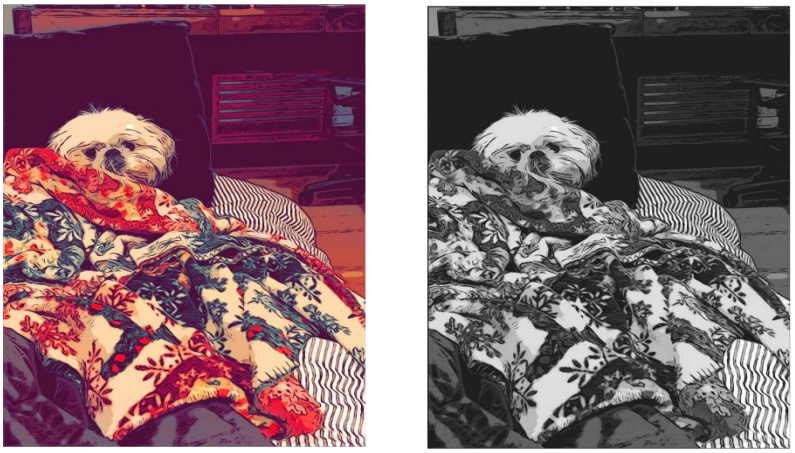


Fig. 3: The visualization of the RGB image data and gray scale image

This is the final version of the image after preprocessed by luminosity model. Hence, our training data has only one color channel now. The implementation of the luminosity model is provided by:

I firstly change the RGB axis of the data so that color channels are in last axes. (This is unnessary)

```
data = np.swapaxes(data,1,3)
```

```
The data has a shape: (10240, 16, 16, 3)
```

To construct an image preprocessing, I create a ImagePreprocessing class and luminosity model is created by:

```
class ImagePreprocessing:
  """
  _____Image preprocessor_____

  def ToGray(self,data):
    """
    Given the input image converting gray scale according to luminosity model
```

```
    """
    R_linear = 0.2126
    G_linear = 0.7152
    B_linear = 0.0722
    gray_data = (data[:,:,:,0] * R_linear) + (data[:,:,:,1] * G_linear) + (data[:
,:,:,2] * B_linear)

    return gray_data
```

Then, the data is converted from RGB to gray scale by:

```
# Defining preprocessor :
preprocessor = ImagePreprocessing()
# Converting gray scale :
gray_data = preprocessor.ToGray(data = data)
```

Therefore, I succesfully convert the data into expected format.

Image Normalization

In this part, I am going to normalize the data by first removing the mean pixel intensity of each image from itself, then clip the data range $\pm$ 3 standart deviation. To prevent saturations of the activation functions, I am going to map $\pm$ 3 std. Finally, the data range should become [0.1,0.9].

Fistly, I substract the mean of each image by itself by:

```
  def MeanRemoval(self,data):
    """
      Given the input image, substracking the mean of pixel intensity of each ima
ge
    """
    axis = (1,2)
    mean_pixel = np.mean(data,axis = axis)
    num_samples = data.shape[0]

    # Substracking means of each image seperately :
    for i in range(num_samples):
      data[i] -= mean_pixel[i]
    return data
```

```
# Mean removing :
mean_removed_data = preprocessor.MeanRemoval(data = gray_data)
```

Therefore, now the the mean of each pixel intensity is extracted from the patches.

Then I clipped the data by following conditions:

$$clip(X) = \begin{cases} 3 * std \ , if \ X > 3 * std \\ -3 * std \ , if \ X > 3 * std \ , \ \ for \ all \ X. \\ x \ else \end{cases}$$

The implementation is provided below:

```
  def ClipStd(self,data,std_scaler):
    """
    Given the data and range of standart deviation scaler,
    return clipped data
    """
    std_pixel = np.std(data)

    min_cond = - std_scaler * std_pixel
    max_cond =   std_scaler * std_pixel

    clipped_data = np.clip(data,min_cond,max_cond)

    return clipped_data
```

Hence, I clipped the data at $\pm$ 3 standart deviation range to prevent activations from saturation.

```
# Standart deviation clipping :
clipped_data = preprocessor.ClipStd(data = mean_removed_data,std_scaler = 3)
```

Then, I normalize the data by [0,1] as a first step of normalization in [0.1,0.9]:

$$\tilde{X} = \frac{X - min(X)}{max(X) - min(X)}$$

The code part is given by:

```
 def Normalize(self,data,min_scale,max_scale):
    """
    Given the data, normalize to given interval [min_val,max_val]
    """
    min = data.max()
    max = data.min()

    # First normalize in [0,1]
    norm_data = (data - min) / (max-min)
```

Then, I rearrange the range of function and normalized in between 0.1 and 0.9 by following equation:

$$\tilde{X}_{final} = \tilde{X} * (0.9 - 0.1) + 0.1$$

Implementation is shown below:

```
# Normalizing in [min_scale,max_scale]
    range = max_scale - min_scale
```

```
    interval_scaled_data = (norm_data * range) + min_scale

    return interval_scaled_data
```

Hence, I succesfully completed the image preprocessing part.

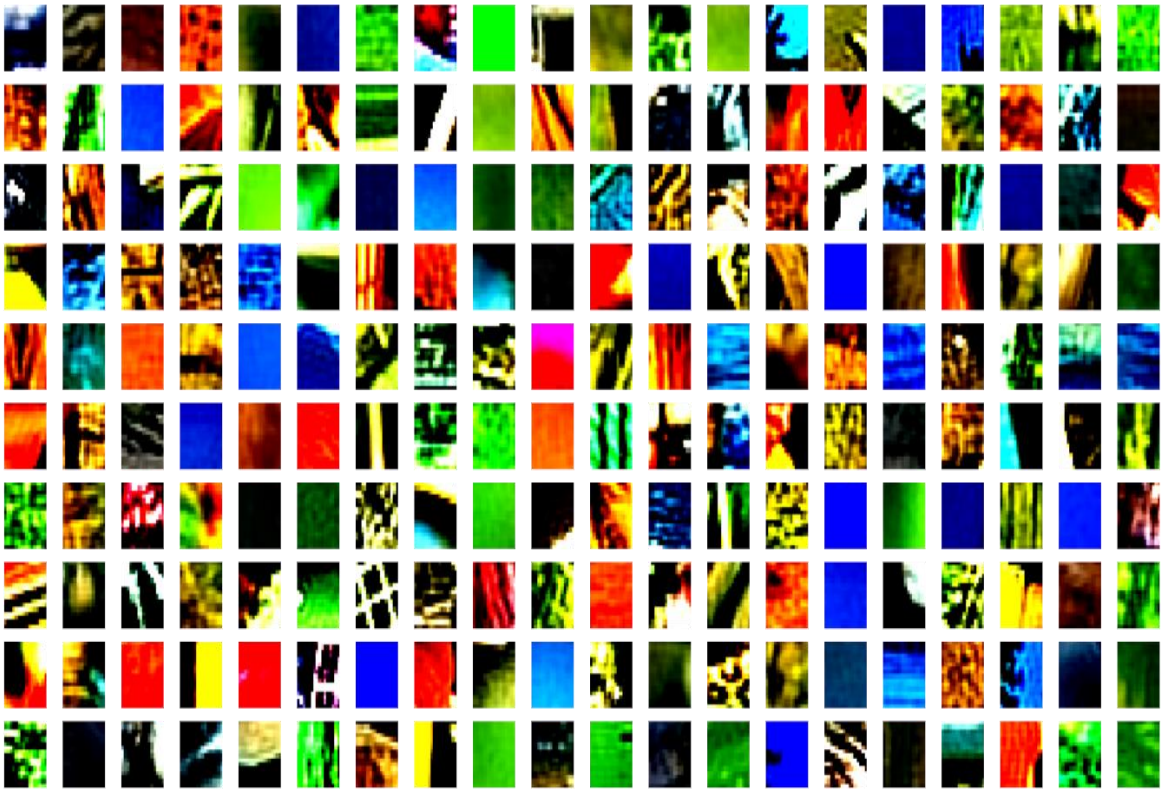
```
# Normalized data
data_processed = preprocessor.Normalize(data = clipped_data, min_scale = 0.1, max
_scale = 0.9)
```

Check Point:

```
print(f' Maximum val of data :  {data_processed.max()}')
print(f' Minimum val of data :  {data_processed.min()}')
```

```
Maximum val of data:  0.9000000357627869
Minimum val of data:  0.10000000149011612
```

Hence, the expectation is satisfied and here are the results of preprocessing:



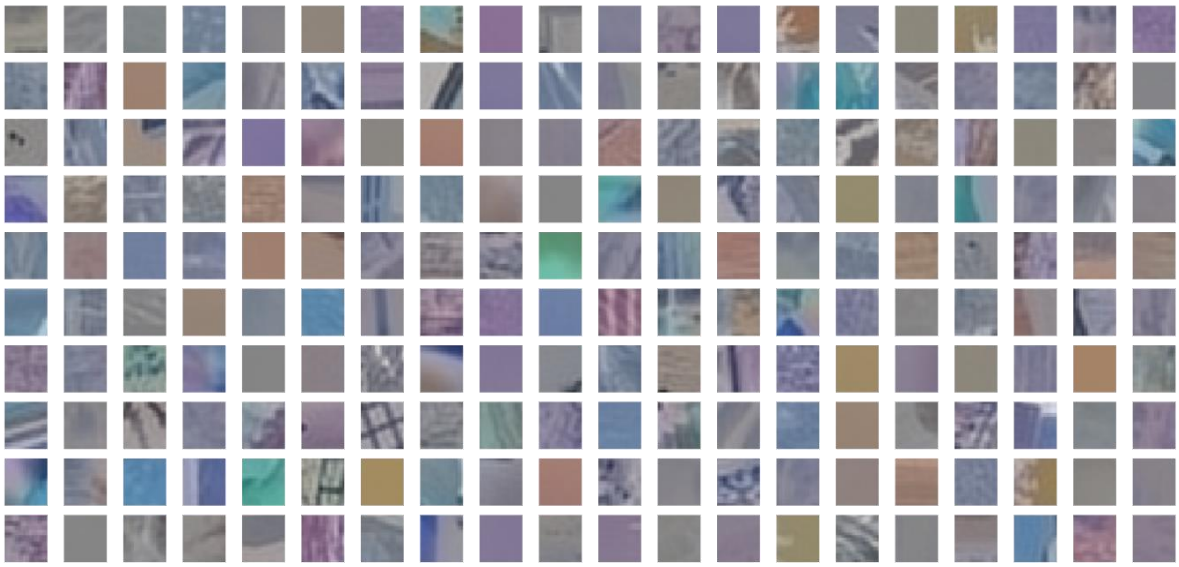


Fig. 4: The visualization of the RGB image data that are randomly selected in our dataset

Note that the first RGB image plot is the not clipped version of the data, i.e. the data given us varies between -7 and 7, when we plot it, imshow() method of the matplotlib package automatically clips given data in range between 0-1 or 0-255. Therefore, first plot is not actually our data but clipped version of the data. After that, the second RGB figure represents our data, I scaled the RGB data between 0-1 to plot. In below, the processed version of RGB data is shown.

Fig. 5: The visualization of the gray scale image data that are randomly selected in our dataset

Therefore, we see that gray scale images have 1 color channel while RGB images have 3. Gray scale versions of the images are representing the RGB versions in one color channel that will boost the neural network model performance. However, when we observe in a detailed way, we can see that there are some deviations from the RGB version of the image that is caused by the clipping the data at the ± 3 standart deviation range, but it is totally acceptable.

1.2 PART B

In this part, the question is divided into parts to analyze what I did in the detailed way.

Architecture

In this question, I am going to the construct the following architecture for the autoencoder.

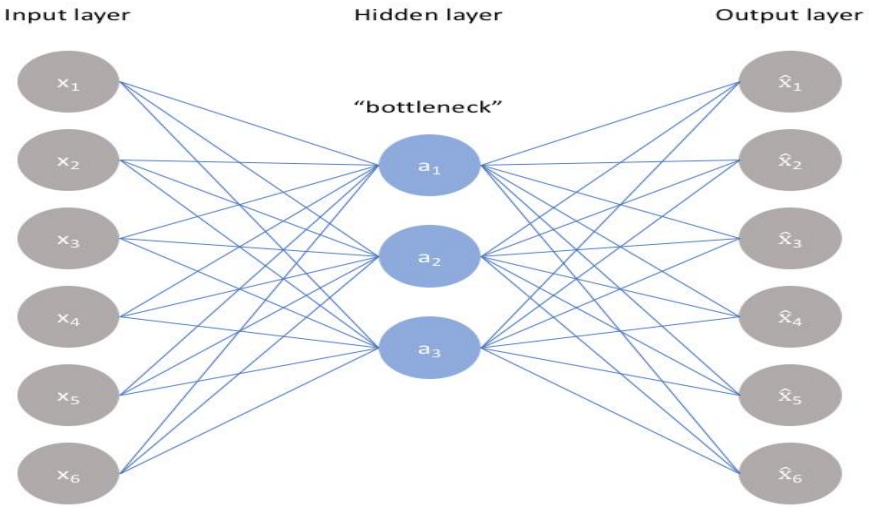


Fig. 6: The visualization of representative autoencoder architecture

Note that this figure is just a representative version of our model. In our case, we have 256 input nodes. The bottleneck layer is actually a hidden layer that represent the learned features. The weights of the bottleneck layer will imitate the data given. In nutshell, unsupervised feature extraction algorithm will be implemented via supervised learning concept, i.e., we will set the labels of the data to our training data so that the encoder-decoder model will learn features by itself. By doing so, the weights of the bottlencek layer will created a representation of the given features that we will see in the following parts of the paper.

Configurable Parameters

In this question, our hyperparameters are, learning rate η, lambda λ, beta β , rho ρ, hidden size. Along this question, I assumed a sigmoidal activation function for all neurons.

- Hidden layer unit $L_{hid} \in \{10,50,64,100\}$
- Lambda $\lambda \in \{0, 10^{-4}, 5x10^{-4}, 10^{-3}\}$
- Learning rate η $\in \{0.1,..., 0.9\}$
- Beta $\beta \in \{0.001, 0.01, 0.1\}$
- Rho ρ = 0.05
- Weight W_1, W_2 and B_1, B_2 are initialized where W_1, B_1 and W_2, B_2 corresponds to the hidden layers and output layer, respectively using following initialization techniques:

$$W_1, W_2, B_1, B_2 \sim Uniform[-\frac{6}{\sqrt{L_{pre}+L_{post}}}, \frac{6}{\sqrt{L_{pre}+L_{post}}}]$$

Therefore, I construct the network architecture by initializing weight in the Autoencoder class:

```
class Autoencoder:
    """
    ____Autoencoder____

    """

    def __init__(self,input_size,hidden_size,lambd):
        """
        Construction of the architecture of the autoencoder
        """
        np.random.seed(1500)
        self.lambd = lambd
        self.beta = 1e-1
        self.rho = 5e-2
        self.learning_rate = 9e-1

        self.params = {'L_in'     : input_size,
                       'L_hidden' : hidden_size,
                       'Lambda'   : self.lambd,
                       'Beta'     : self.beta,
                       'Rho'      : self.rho}

        self.W_e = self.InitParams(input_size,hidden_size)

        self.loss = []

    def InitParams(self,input_size,hidden_size):
        """
        Given the size of the input node and hidden node, initialize the weights
        drawn from uniform distribution ~ Uniform[- sqrt(6/(L_pre + L_post)) , sq
rt(6/(L_pre + L_post))]
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = input_size

        W1_high = self.w_o(input_size,hidden_size)
        W1_low = - W1_high
        W1_size = (input_size,hidden_size)
        self.W1 = np.random.uniform(W1_low,W1_high,size = W1_size)

        B1_size = (1,hidden_size)
        self.B1 = np.random.uniform(W1_low,W1_high,size = B1_size)

        W2_high = self.w_o(hidden_size,self.output_size)
```

```
        W2_low = - W2_high
        W2_size = (hidden_size,self.output_size)

        self.W2 = np.random.uniform(W2_low,W2_high,size = W2_size)

        B2_size = (1,self.output_size)
        self.B2 = np.random.uniform(W1_low,W1_high,size = B2_size)

        return  {'W1' : self.W1,
                 'W2' : self.W2,
                 'B1' : self.B1,
                 'B2' : self.B2}


    def w_o(self,L_pre,L_post):
        return np.sqrt(6/(L_pre + L_post))
```

Finally, I created the archtecture of the auto encoder in succesfull way.

Forward Pass

In forward propagation, we feed input to the network so that get predictions from the model. In our case, the predictions will be data itself. Hence, I will try to extract features of the given data as much as I can. In the following mathematical expressions, forward propagation is modeled. Note that L_{hidden} is the linear weighted sum of inputs with W_1 then adding bias term B_1. O_{hidden} is the output of the hidden layer. L_{output} is the linear weighted sum of weights W_2 with O_{hidden} then adding bias term B_2. Lastly, O_{output} is the output of the output layer.

$$L_{hidden} = \sum_{i=1}^{N}(W_1^{(i)} * X_1^{(i)}) + B_1$$
$$O_{hidden} = \text{sigmoid}(L_{hidden})$$
$$L_{output} = \sum_{i=1}^{N}(W_2^{(i)} * O_{hidden}^{(i)}) + B_2$$
$$O_{output} = \text{sigmoid}(L_{output})$$

where sigmoid(X) = $\frac{1}{1+e^{-X}}$. The following Python code shows the implementation of the forward propagation:

```
    def forward(self,X):
        """
        Forward propagation
        """
        W1 = self.W_e['W1']
        W2 = self.W_e['W2']
```

```
        B1 = self.W_e['B1']
        B2 = self.W_e['B2']

        Z1 = np.dot(X,W1) + B1
        A1 = self.sigmoid(Z1,grad = False)

        Z2 = np.dot(A1,W2) + B2
        A2 = self.sigmoid(Z2,grad = False)

        return {"Z1": Z1,"A1": A1,
                "Z2": Z2,"A2": A2}
```

Calculation of Loss

For this question, mean squared error (MSE) will be used with the KL-divergence and Tykhonow regularization. The following mathematical expressions describes the behavior of the MSE.

$$\text{MSE} = \frac{1}{2N}\sum_{m=1}^{N}\left\|d(m) - O_{output}(m)\right\|^2$$

Therefore, MSE function takes a prediction generated by the model and the label of training data, compare them by substracting each other and to eliminate negative values MSE function takes to sqaure of the difference. Taking square also helps us to punish outliers in the dataset. Finally, taking the mean of the differences will equate the MSE error.

Then, here is the Mean Squared Error gradient w.r.t. O_{output}:

$$\frac{\partial MSELoss}{\partial O_{output}} = \frac{1}{2N}\sum_{m=1}^{N}(d(m) - O_{output}(m))$$

For the notation, note that $Y = d(m)$ is the desired output that is training data itself. Python implementation of the MSE and it's derivative:

```
    def MSE(self,pred,label, grad = True):
        """
        Calculating Mean Sqaured Error and it's gradient w.r.t. output
        """
        return 1/2 *  (pred - label) if grad else 1/2 * np.sum((pred - la
bel)**2)/pred.shape[0]
```

Note that taking mean will be implemented in backprop.

Tykhonow regularization

This regularization is L2 type of regularization that have an equation of:

$$\text{Tykhonow Regularization}(W_1, W_2, \lambda) = \frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}\left(W_{a,b}^{(1)}\right)^2 + \sum_{c=1}^{L_{out}}\sum_{b=1}^{L_{hid}}\left(W_{b,c}^{(2)}\right)^2\right]$$

And here are the gradients of Tykhonow Regularization the w.r.t. each weight:

$$\frac{\partial \text{Tykhonow Regularization}(W_1,W_2,\lambda)}{\partial W_1} = \lambda W_1$$

$$\frac{\partial \text{Tykhonow Regularization}(W_1,W_2,\lambda)}{\partial W_1} = \lambda W_2$$

Note that $\frac{\partial \text{Tykhonow Regularization}(W_1,W_2,\lambda)}{\partial B_1} = \frac{\partial \text{Tykhonow Regularization}(W_1,W_2,\lambda)}{\partial B_2} = 0$

Here is the Python implementation of the Tykhonow regularization with it's gradient w.r.t. W_1, W_2.

```
 def TykhonowRegularization(self,W1,W2,lambd,grad = True):
        """
            L2 based regularization computing and its gradients
        """
        return {'dW1': lambd * W1, 'dW2': lambd * W2} if grad else (lambd
/2) * (np.sum(W1**2) + np.sum(W2**2))
```

KLterm (Kullback-Leibler divergence)

KL-divergence is a measure of the difference between two probability distributions. The mathematical equation is given below:

$$KL - divergence = \beta \sum_{b=1}^{L_{hid}} KL(\rho|\hat{\rho}_b)$$

Where ρ is the sparsity parameters (In our case, $\rho = 5\text{x}10^{-2}$, also noted in it is typical value used in the literature), $\hat{\rho}_b$ is the average hidden layer actions so that

$$\hat{\rho}_b = \frac{1}{m}\sum_{m=1}^{L_{hid}} O_{hidden}{}^{(m)}$$

Therefore, keeping $\rho = 5\text{x}10^{-2}$ means that most of the neurons will be inactive in the sigmoidal activation. The reason behind that we want put constraint on the network so that model can learn only limited features of the input. Therefore, our main intention is to:

$$\rho = \hat{\rho}_b$$

At that point, KL-divergence comes into play and add extra penalty to loss function J_{ae}. The intuition behind the KL-divergence is to penalize $\hat{\rho}_b$ for deviating significantly from the ρ. The mathematical expression for KL-divergence is given that:

$$KL - divergence = \beta \sum_{b=1}^{L_{hid}} KL(\rho|\hat{\rho}_b) = \beta \sum_{b=1}^{L_{hid}} \rho \log\left(\frac{\rho}{\hat{\rho}_b}\right) + (1-\rho)\log\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)$$

Hence, this is the divergence between Bernoulli random variable, let' say P, so that E[P] = ρ and Bernoulli random variable, let's say $\widehat{P}_b$, so that E[$\widehat{P}_b$] = $\hat{\rho}_b$.(E[X] is the expectation)

In detail, auto encoders can be separated as two versions that are under-complete auto encoder and sparse auto encoders(over-complete). In the under-complete auto encoders, since the number of the hidden size is less than input size, the data is 'compressed' so that only certain amount of information is passed through the network. Hence, the network is forced learn compressed representation of the input data. At the output layer, the network should reconstruct the input image by using the information coming from the hidden layer. In nutshell, in the under completed auto encoders the network is forced learn input representation in lower dimension so that this type of the auto encoders can be used to non-linear dimensionality reduction for inputs. Note that if we set L_{hid} = 64, our auto encoder can be classified into under completed.

In other case, sparse auto encoders, the hidden size is higher than the input size so that if we don't put constraint such as regularizations, KL-divergence and ext., the network is expected learn well. However, we don't want network to learn fully because it is meaningless, i.e., we have the same image input). What we want is some kind of representative learning to extract features from the input as well as decreasing the dimensionality of the input. To do that, some kind of constraints and penalties is added to loss function. In our case, KL-divergence is the penalty term. (Also, Tykhonow Regularization is also penalty term.)

However, in our case questions wants both compresses the data by setting hidden size lower than input size and putting constraints and penalties to the network such as Tykhonow Regularization and KL-divergence.

Lastly, the gradients of the KL-divergence term w.r.t. configurable parameters are given as:

$$\frac{\partial KL-divergence}{\partial W_1} = \beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right)+\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right] = \left[\beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right)+\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right] \quad \dots \quad \beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right)+\left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right]\right]_{(NxL_{Hid})}$$

Note that $\frac{\partial KL-divergence}{\partial B_1} = \frac{\partial KL-divergence}{\partial W_2} = \frac{\partial KL-divergence}{\partial B_2} = 0$.

The implementation in the Python is given by:

```
 def KL_divergence(self,rho,beta,expected,grad = True):
        """
        Computing KL-
divergence and it's gradients, note that gradients is only for W1
        """
        return np.tile(beta * (-(rho/expected) + (1-rho)/(1-
expected) ), reps = (10240,1)) if grad else beta * (np.sum((rho * np.log(rho/expe
cted)) + ((1-rho)*np.log((1-rho)/(1-expected)))))
```

Therefore, the total loss is calculated via:

```
  def total_loss(self,outs,label):
        W1 = self.W_e['W1']
        W2 = self.W_e['W2']
```

```
        Lambda = self.params['Lambda']
        beta = self.params['Beta']
        rho = self.params['Rho']

        J_mse = self.MSE(outs['A2'],label, grad = False)
        J_tykhonow = self.TykhonowRegularization(W1 = W1, W2 = W2,lambd = Lambda,
 grad = False)
        J_KL = self.KL_divergence(rho = rho,expected = np.mean(outs['A1']), beta
= beta, grad = False)

        return J_mse + J_tykhonow + J_KL
```

Backpropagation

In backpropagation part, the effect of the weights W_1 and W_2 and biases B_1, B_2 on the error (i.e., gradients of the W_1, W_2 and B_1, B_2) is calculated via chain rule. The following equations shows the implementation of the backprogation with single hidden layer auto encoder:

$$\frac{\partial J_{ae}}{\partial W_2} = \frac{\partial J_{ae}}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial W_2}$$

$$\frac{\partial J_{ae}}{\partial W_2} = \frac{1}{N}\left(O_{output} - Y\right) * \left(\frac{\partial sigmoid}{\partial L_{output}}\left(L_{output}\right)\right) * (W_2)^T + \lambda W_2$$

$$\frac{\partial J_{ae}}{\partial B_2} = \frac{\partial J_{ae}}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial B_2}$$

$$\frac{\partial J_{ae}}{\partial B_2} = \frac{1}{N}\sum_{i=1}^{N}\left(O_{output}{}^{(i)} - Y^{(i)}\right) * \left(\frac{\partial sigmoid}{\partial L_{output}}\left(L_{output}\right)\right)$$

$$\frac{\partial J_{ae}}{\partial W_1} = \frac{\partial J_{ae}}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial W_1}$$

$$\frac{\partial J_{ae}}{\partial W_1} = \left[\frac{1}{N}\left(O_{output} - Y\right)\left(\frac{\partial sigmoid}{\partial L_{output}}\left(L_{output}\right)\right)\left(W_2{}^T\right) * \frac{\partial sigmoid}{\partial L_{output}}\left(L_{hidden}\right)\right) * X^T\right] + \left[\frac{1}{N}\left[\beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right) + \left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right] \quad \dots \quad \beta\left[-\left(\frac{\rho}{\hat{\rho}_b}\right) + \left(\frac{1-\rho}{1-\hat{\rho}_b}\right)\right]\right]_{(NxL_{Hid})}\right] + [\lambda W_1]$$

$$\frac{\partial J_{ae}}{\partial B_1} = \frac{\partial J_{ae}}{\partial O_{output}} * \frac{\partial O_{output}}{\partial L_{output}} * \frac{\partial L_{output}}{\partial O_{hidden}} * \frac{\partial O_{hidden}}{\partial L_{hidden}} * \frac{\partial L_{hidden}}{\partial B_1}$$

$$\frac{\partial J_{ae}}{\partial B_1} = = \frac{1}{N}\sum_{i=1}^{N}\left(O_{output}{}^{(i)} - Y^{(i)}\right) * \left(\frac{\partial sigmoid}{\partial L_{output}}\left(L_{output}\right)\right) * \left(W_2{}^{(i)^T}\right) * \frac{\partial sigmoid}{\partial L_{output}}\left(L_{hidden}\right)$$

where X is the input, Y is the output(in our case, X = Y), $\frac{\partial sigmoid}{\partial x}$ = sigmoid(x) * (1- sigmoid(x))

and J_{ae} is the overall cost function. Python implementation of back propagation is shown below:

```
    def backward(self,outs,data):
        """
        Given the forward pass outputs, input and their labels,
        returning gradients w.r.t. loss functions
        """
        m = data.shape[0]

        Lambda = self.params['Lambda']
        beta = self.params['Beta']
        rho = self.params['Rho']

        W1 = self.W_e['W1']
        W2 = self.W_e['W2']
        B1 = self.W_e['B1']
        B2 = self.W_e['B2']

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']

        L2_grad = self.TykhonowRegularization(W1,W2,lambd = Lambda , grad = True)
        KL_grad_W1 = self.KL_divergence(rho,beta,expected = np.mean(A1),grad = Tr
ue)

        dZ2 = self.MSE(A2,data, grad = True) * self.sigmoid(Z2, grad = True)

        dW2 = (1/m) * (np.dot(A1.T,dZ2) + L2_grad['dW2'])
        dB2 = (1/m) * (np.sum(dZ2, axis=0, keepdims=True))

        dZ1 = (np.dot(dZ2,W2.T) + KL_grad_W1)  * self.sigmoid(Z1,grad = True)

        dW1 = (1/m) * (np.dot(data.T,dZ1) + L2_grad['dW1'])
        dB1 = (1/m) * (np.sum(dZ1, axis=0, keepdims=True))

        assert (dW1.shape == W1.shape and dW2.shape == W2.shape)

        return {"dW1": dW1, "dW2": dW2,
                "dB1": dB1, "dB2": dB2}
```

Stochastic Gradient Descent

Stochastic gradient descent on full batches is performed via the following equations.

$$W_2 := W_2 - \eta * \frac{\partial J_{ae}}{\partial W_2} \qquad W_1 := W_1 - \eta * \frac{\partial J_{ae}}{\partial W_1}$$

$$B_2 := B_2 - \eta * \frac{\partial J_{ae}}{\partial B_2} \qquad B_1 := B_1 - \eta * \frac{\partial J_{ae}}{\partial B_1}$$

The code implementation of stochastic gradient descent is shown below:

```
    def step(self,grads):
        """
        Updating configurable parameters according to full-
batch stochastic gradient update rule
        """
        self.W_e['W1'] += -self.learning_rate * grads['dW1']
        self.W_e['W2'] += -self.learning_rate * grads['dW2']
        self.W_e['B1'] += -
self.learning_rate * grads['dB1']
        self.W_e['B2'] += -self.learning_rate * grads['dB2']
```

Training

Finally, training loop is implemented by:

```
    def fit(self,data,epochs = 5000,verbose = True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """
        for epoch in range(epochs):

            loss_and_grads = self.aeCost(data)
            self.step(grads = loss_and_grads['J_grad'])

            if verbose:
                print(f"[{epoch}/{epochs}]     ----------
> Loss  :{loss_and_grads['J']}")

            self.loss.append(loss_and_grads['J'])
```

Hyperparamaters Tuning

I implemented manuel search algorithm to find best network parameters. I tried the mentioned configurable parameters. The best results are used in the network. The term “best” is considered by:

1) Model performance
2) Feature representation of the hidden layer

3) Reconstruction of the image
4) Computational cost

Therefore, final values calculated as a:

Hyperparameters	Results
Hidden size number $\boldsymbol{L_{Hid}}$	64
Initialization method	$Uniform[-\frac{6}{\sqrt{L_{pre}+L_{post}}},\frac{6}{\sqrt{L_{pre}+L_{post}}}]$
Epochs	5000
Batch size (Full batch)	10420
Learning rate η	9×10^{-1}
Lambda $\boldsymbol{\lambda}$	5×10^{-4}
Beta $\boldsymbol{\beta}$	1×10^{-1}
Rho $\boldsymbol{\rho}$	5×10^{-2}

aeCost($W_E, data, params$)

The function aeCost($W_E, data, params$) is imlemented via:

1) Forward propagation
2) Calculations of the loss
3) Calculation of gradients

The implementation in Python is:

```
    def aeCost(self,data):

        outs = self.forward(data)
        loss = self.total_loss(outs,data)
        grads = self.backward(outs,data)

        return {'J'      : loss,
                'J_grad' : grads}
```

Recall that used functions are desribed above so that the detailed explanations of the components of the aeCost is done in previous parts. Moreover, since I building the structure object-oriented based, it is not required the $W_E, params$, i.e., the W_E and $params$ are defined internally in the class structure.

1.3 PART C

The implementations and the discussing part are provided below.

Solver()

Solver concept is implemented manually, i.e., I trained the network by stoshastic gradient descent to optimize the network by minimizing cost. Note that since Python has not built-in solver method (In MATLAB, there are built-in solver methods like x = fmincon(fun, x_0,A,b) where fun is the function

the minimize, and the rest are data and the parameters), I am going to first create my Solver class in Python to train the network then check my results via using TensorFlow Keras API.

The Python implementation:

```
class Solver:
    """
    Given  as input, A Solver encapsulates all the logic necessary for tr
aining then
    implement gradients solver to minimize the cost.The Solver performs s
tochastic gradient descent.

    """
    def __init__(self, model,data):

        self.model = model
        self.data = data

    def train(self,epochs = 5000,verbose = False):
        """
        Optimization of the model by minimizing cost by solving gradients
        """
        self.model.fit(self.data,epochs,verbose)

    def parameters(self):
        """
        Returning configurable parameters of the network
        """
        return self.model.parameters()
```

Note that pre-builted functions in the Autoencoder class is used in the Solver class to prevent extra complexity.

Moreover, note that solver method is actually doing following minimization:

$$\min_{W_1,W_2,B_1,B_2} \frac{1}{2N}\sum_{i=1}^{N}\|d(m)-o(m)\|^2+\frac{\lambda}{2}\left[\sum_{b=1}^{L_{hid}}\sum_{a=1}^{L_{in}}\left(W_{a,b}^{(1)}\right)^2+\sum_{c=1}^{L_{out}}\sum_{b=1}^{L_{hid}}\left(W_{b,c}^{(2)}\right)^2\right]+\beta\sum_{b=1}^{L_{hid}}KL(\rho|\hat{\rho}_b)$$

Therefore, the Solver Class is handled this by fitting the model. The running part is:

```
data_feed = preprocessor.Flatten(data_processed)
input_size = data_feed.shape[1]
hidden_size = 64
autoencoder = Autoencoder(input_size = input_size, hidden_size = hidden_s
ize,lambd = 5e-4)
```

```
solver = Solver(model = autoencoder, data = data_feed)
solver.train()
```

We see that model is evolving:

```
[0/5000]      ----------> Loss  :2.794992968686732
[1/5000]      ----------> Loss  :2.077742786326408
[2/5000]      ----------> Loss  :2.068867287678074
[3/5000]      ----------> Loss  :2.064954256440666
[4/5000]      ----------> Loss  :2.062133504312842
[5/5000]      ----------> Loss  :2.059700146127778
[6/5000]      ----------> Loss  :2.057485639761386
[7/5000]      ----------> Loss  :2.055433041888980
[8/5000]      ----------> Loss  :2.053517247692437
[9/5000]      ----------> Loss  :2.051724210237005
[10/5000]     ----------> Loss  :2.050044252668423
```

Therefore, the concept of solver is implemented via minimizing the loss function by full-batch stochastic gradient descent.

Result/Discuss

In this part, I am going to discuss the results, outputs, plottings and so on. I implemented the TensorFlow Keras API functional model to compare my results and discuss the results. I am going to explain how I build TensorFlow model and its results:

To begin with, I created my custom MeanSquaredError() and KL_divergence(rho, beta) via:

```
def MeanSquaredError():
  def customMeanSquaredError(pred,label):
    return 1/2 * K.sum((pred - label)**2)/pred.shape[0]
  return customMeanSquaredError

def KL divergence(rho, beta):
  def customKL(out):
    kl1 = rho*K.log(rho/K.mean(out, axis=0))
    kl2 = (1-rho)*K.log((1-rho)/(1-K.mean(out, axis=0)))
    return beta * K.sum(kl1+kl2)
  return customKL
```

To give similar values returning from the training of the TensorFlow model. The initializaiton of the weights are:

```
W_scaler = lambda L_pre,L_post : np.sqrt(6/(L_pre + L_post))

def tf_weight_initializer(inp_dim,hidden_dim):
  initializer_1 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(inp_dim,hidden_dim), maxval=W_scaler(inp_dim,hidden_dim))
  values_2 = initializer_1(shape=(inp_dim,hidden_dim))
```

```
  initializer_2 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(hidden_dim,inp_dim), maxval=W_scaler(hidden_dim,inp_dim))
  values_2 = initializer_2(shape=(inp_dim,hidden_dim))

  initializer_3 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(inp_dim,hidden_dim), maxval=W_scaler(inp_dim,hidden_dim))
  values_3 = initializer_3(shape=(1,hidden_dim))

  initializer_4 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(hidden_dim,inp_dim), maxval=W_scaler(hidden_dim,inp_dim))
  values_4 = initializer_4(shape=(1,inp_dim))

  return {'W1':initializer_1,
          'W2':initializer_2,
          'B1':initializer_3,
          'B2':initializer_4}

tf_weights = tf_weight_initializer(inp_dim = inp_dim, hidden_dim = encodi
ng_dim)
```

```
encoding_dim = 64
rho,beta = 5e-1,1e-1
inp_dim = 256
lamb = 5e-4
```

Then, the model structure is implemented via Keras functional API:

```
input_img = keras.Input(shape=(inp_dim,))

encoded = layers.Dense(encoding_dim,activation='sigmoid',
                       kernel_regularizer=tf.keras.regularizers.l2(lamb),
                       activity_regularizer=KL_divergence(rho,beta),
                       kernel_initializer = tf_weights['W1'],
                       bias_initializer = tf_weights['B1'])(input_img)

decoded = layers.Dense(inp_dim,activation='sigmoid',
                       activity_regularizer=tf.keras.regularizers.l2(lamb
),
                       kernel_initializer = tf_weights['W2'],
                       bias_initializer = tf_weights['B2'])(encoded)

tf_autoencoder = keras.Model(input_img,decoded)
```

The model structure is same as self-written model and shown below:

```
Model: "functional_7"
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 256)]             0
_________________________________________________________________
dense_6 (Dense)              (None, 10)                2570
_________________________________________________________________
dense_7 (Dense)              (None, 256)               2816
=================================================================
Total params: 5,386
Trainable params: 5,386
Non-trainable params: 0
_________________________________________________________________
```

Fig. 7: TensorFlow Keras Model Summary

Then, I created stochastic gradient descent implementation:

```
optimizer = tf.keras.optimizers.SGD(learning_rate=0.9,momentum=0,nesterov
=False)
tf_autoencoder.compile(optimizer=optimizer,loss=MeanSquaredError())
```

Finally, I run the following code to fit the model.

```
tf_autoencoder.fit(data_feed, data_feed,
                   epochs=5000,
                   batch_size=data_feed.shape[0])
```

Therefore, the comparison model is created. Now, I am going to compare results. The results from the model I created from scratch and TensorFlow Keras is given below.

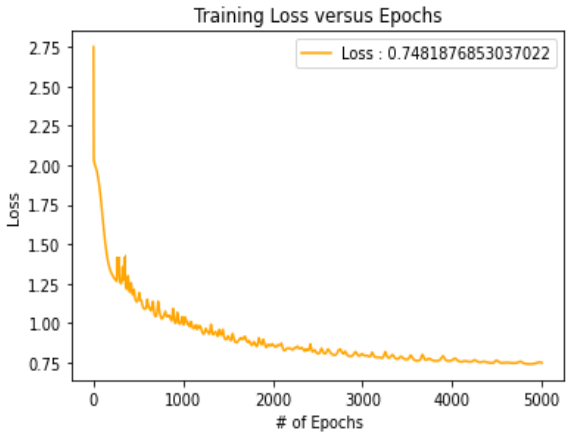


Fig. 8: My model's loss versus epochs

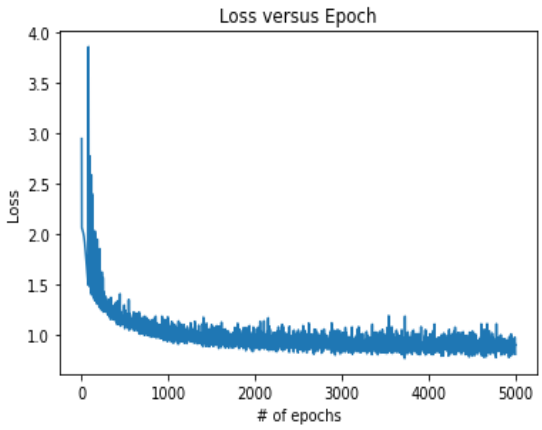


Fig. 9: Keras Model versus epochs

The left figure is my model and the other one is the model created by Keras. We see that my optimization works well. The reason why my model worked is that hyperparamaters are selected carefully so that the model is very stable.

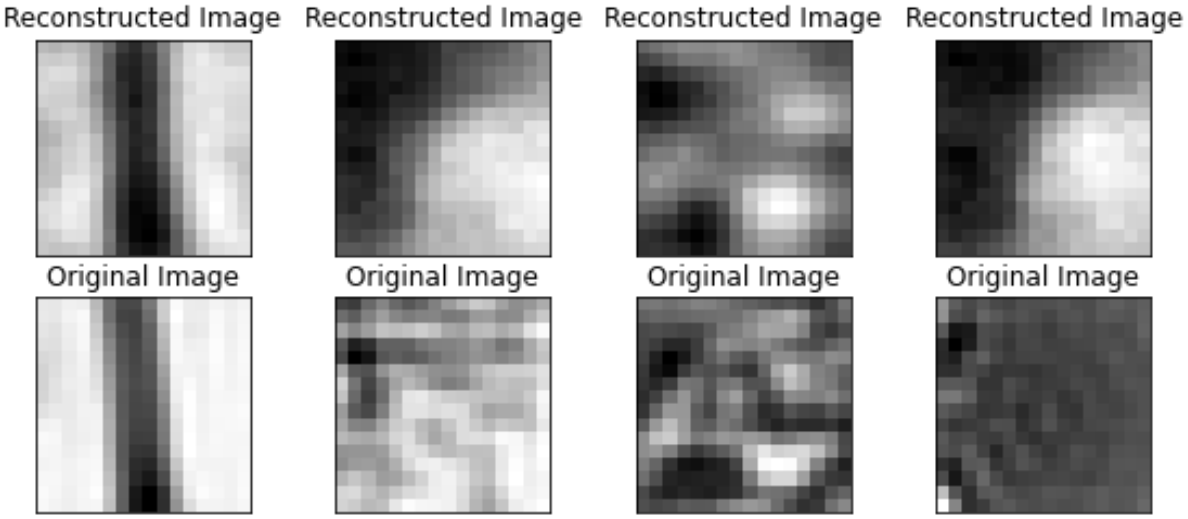


Fig.10 : My model's Outputs vs natural images with L_{hid} = 64, λ =5 x 10^{-5}, $\boldsymbol{\beta} = 10^{-1}$, $\boldsymbol{\rho} =$ 5x 10^{-2} and $\eta = 9$ x 10^{-1}

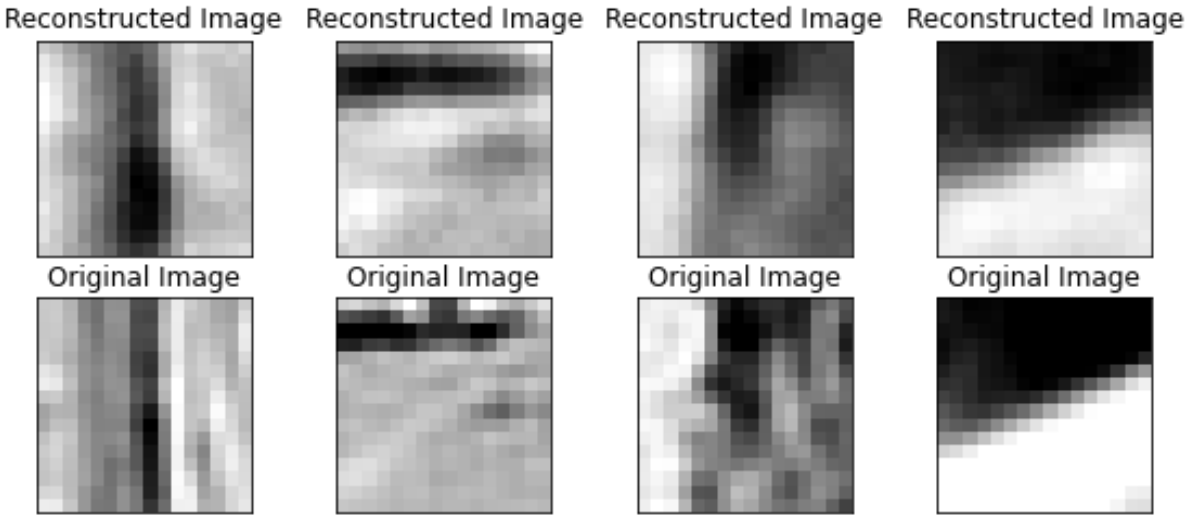


Fig.11 : Further on My model's Outputs vs natural images with L_{hid} = 64, λ =5 x 10^{-5}, $\boldsymbol{\beta} =$ 10^{-1}, $\boldsymbol{\rho} = 5$x 10^{-2}, $\eta = 9$ x 10^{-1}

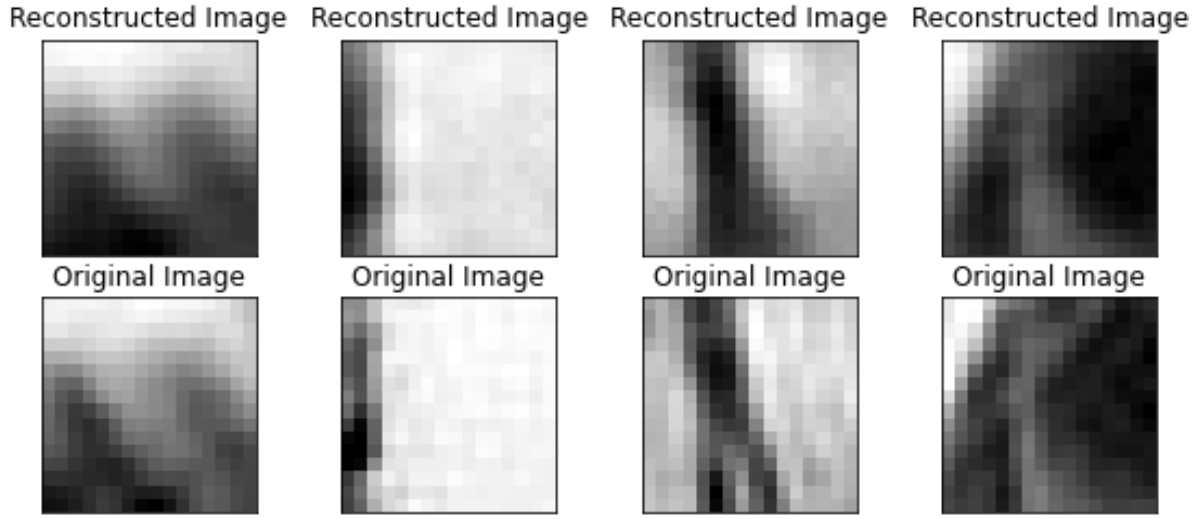


Fig.12 : Further on My model's Outputs vs natural images with L_{hid} = 64, λ =5 x 10^{-5}, $\boldsymbol{\beta}$ = 10^{-1}, $\boldsymbol{\rho} = 5\text{x } 10^{-2}$, $\eta = 9 \text{ x } 10^{-1}$

These images are randomly selected images in the training data. Actually, without checking the results with the Keras model, I can say that my model performed well because the original images and the Autoencoder outputs are nearly identical. The network is certainly learned well since it's reconstruction is quite well.

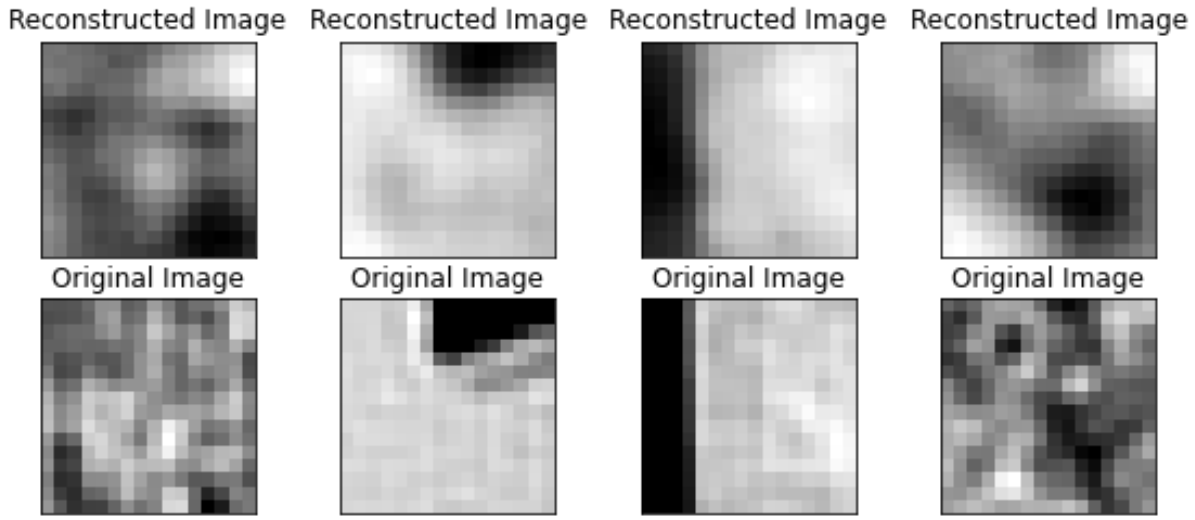


Fig.13 : Keras model's Outputs vs natural images with L_{hid} = 64, λ =5 x 10^{-5}, $\boldsymbol{\beta} = 10^{-1}$, $\boldsymbol{\rho} = 5\text{x } 10^{-2}$, $\eta = 9 \text{ x } 10^{-1}$

The figure above is the predictions come from the Keras model is placed. Hence, my feature extraction is performed well. Intentionally, I also randomly select the input images in the predictions to see Keras model's performance.

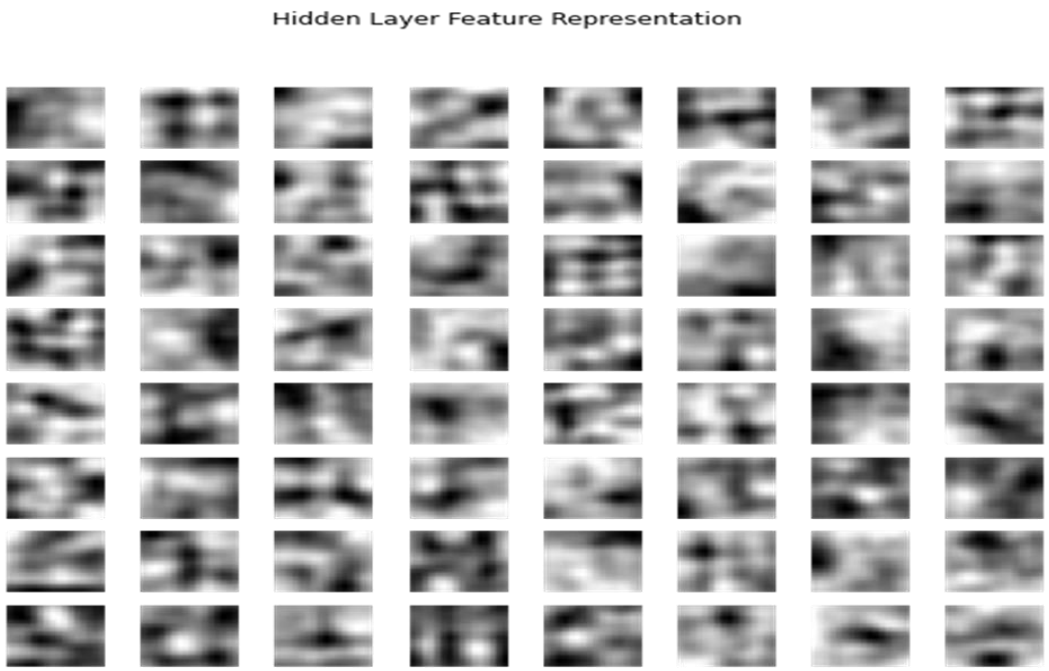


Fig. 14 : My model's hidden weight representation with L_{hid} = 64, λ =5 x 10^{-5}, $\boldsymbol{\beta} = 10^{-1}$, $\boldsymbol{\rho} =$ 5x 10^{-2}, $\eta = 9$ x 10^{-1}

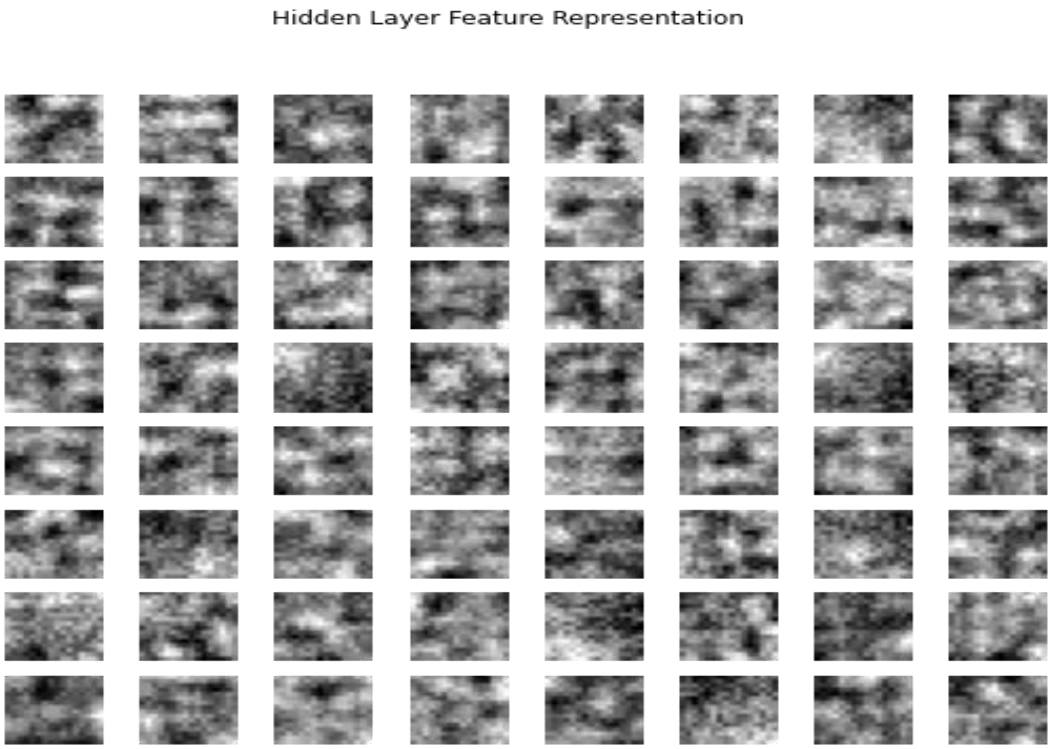


Fig. 15 : Keras model's hidden weight representation with L_{hid} = 64, λ =5 x 10^{-5}, $\boldsymbol{\beta} = 10^{-1}$, $\boldsymbol{\rho} =$ 5x 10^{-2} , $\eta = 9$ x 10^{-1}

The figure above shows my hidden layer's weights. As we expect, weights are representation of the input data since we forced learn from itself by encode-decode structure. Also, my network performed

well than the Keras model. One can see that from the hidden weights representation of the self-written model.

Training a Good Model

In this part, I trained a Keras model with Adam optimizer and Glorot initialization to see more optimal results and compare mine:

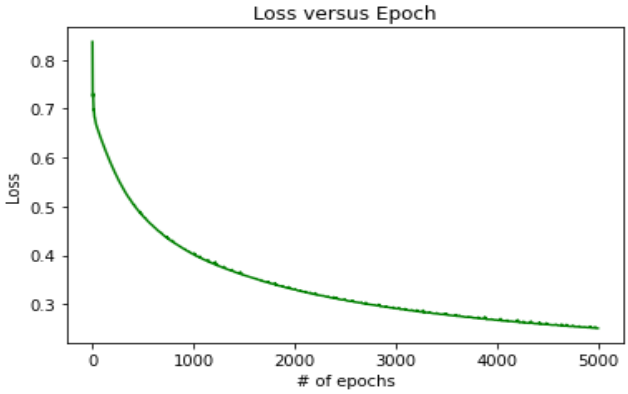


Fig.16: Keras optimized model's loss

With the more sophisticated optimization and weight initialization, our network reached 0.25 loss. Recall that my model reached 0.7 and initial Keras model reached 0.9.

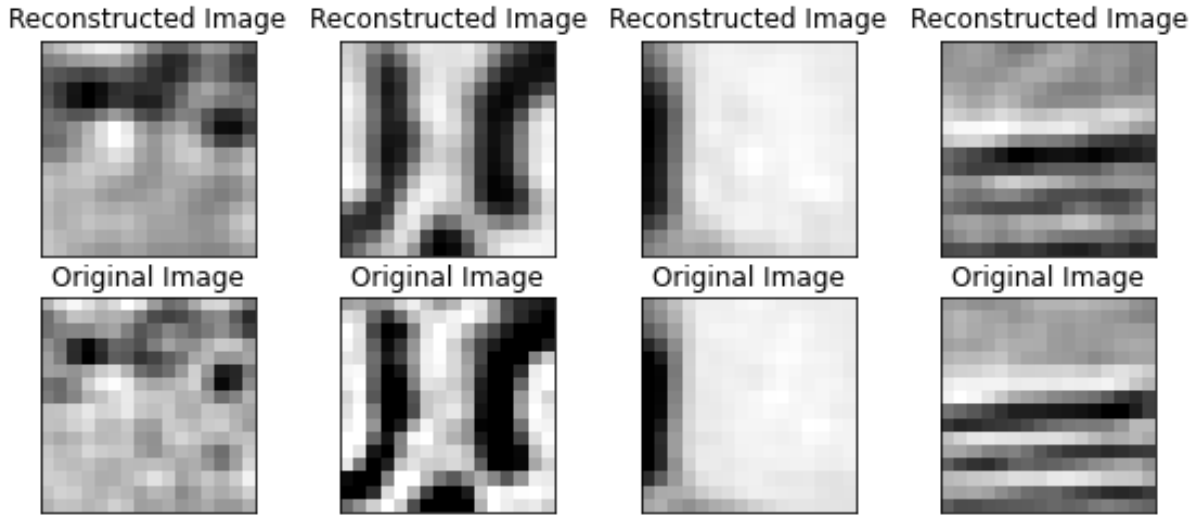


Fig.17: Keras model's reconstruction

The results are well because our optimized network is nearly identical to the original images.

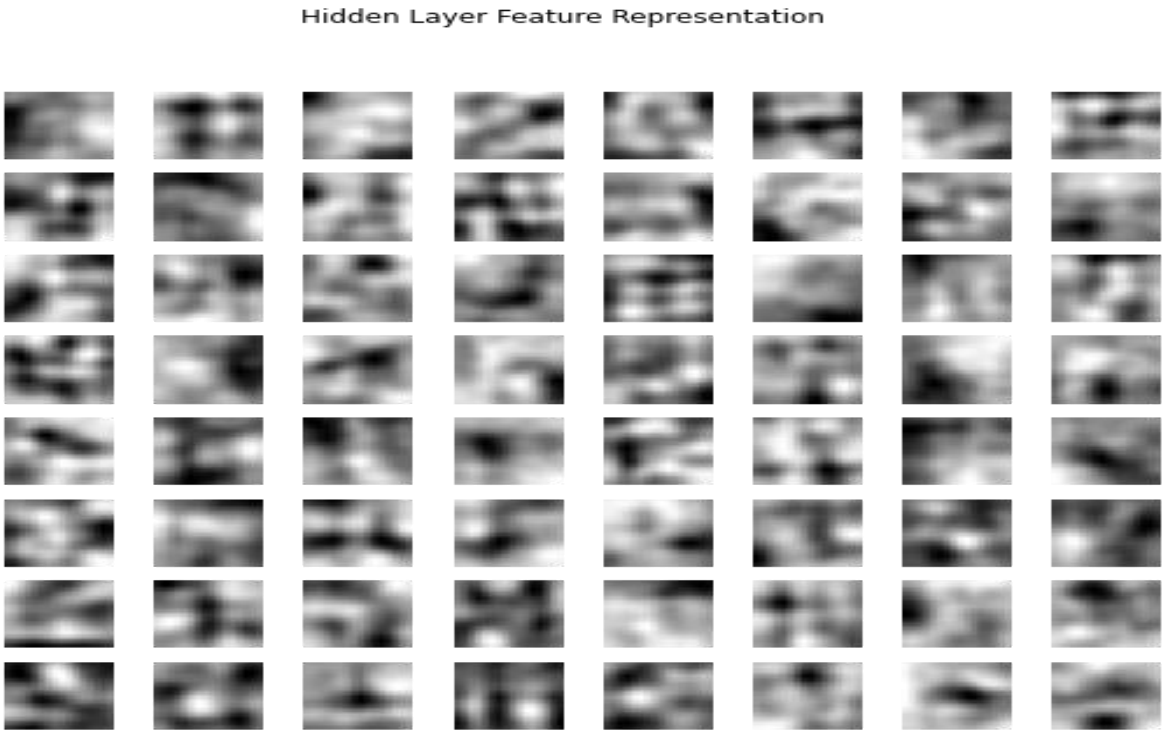


Fig.18: Keras model's hidden weight representation

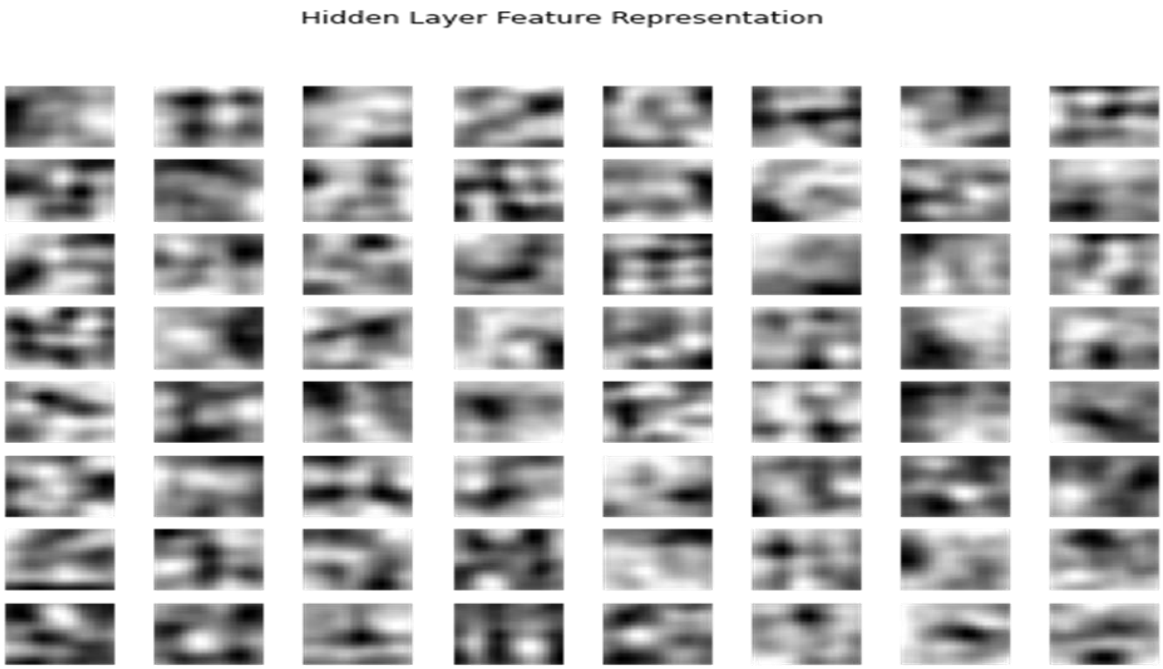


Fig.19: Self-written model's hidden weight representation

Finally, this is the hidden layer weights feature representation figure. When we compare the most optimized result between self-written model, I can say that weight representation is quite similar, even nearly same. Therefore, we can conclude that my model that I created from sctrach is performed well and it is solid proof that self-written Solver implementation works.

*1.4 **PART D***

In this part, question asks re-train the network created in the previous parts with 3 different hidden sizes L_{hid} = 10,50,100 and lambda λ = 0, 10^{-5}, 10^{-3}. Note that I am going to analyze self-written aeCost and Solver by:

1. Comparing training Losses between self-written and TensorFlow Keras
2. Comparing hidden layer weights representability between self-written and TensorFlow Keras
3. Comparing reconstruction ability of the images by auto encoders outputs between self-written and TensorFlow Keras. Note that I intentionally did not give the same indexes of the input in self-written model and Keras model to see variability of the models in another inputs and inference the features extracted by see the generalization capacity of the model.

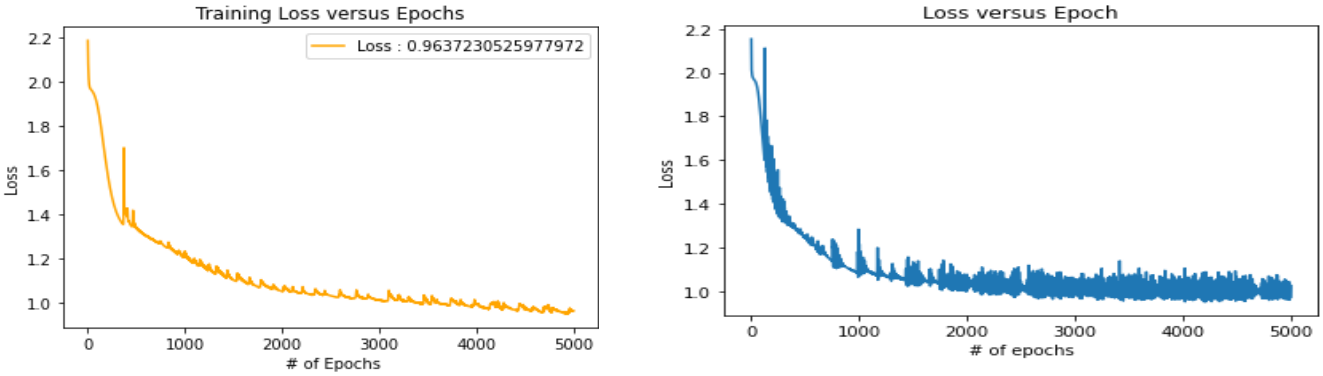


Fig.20,21 : Self-written model's and Keras model's loss over training respectively with L_{hid} = 10 and λ = 0

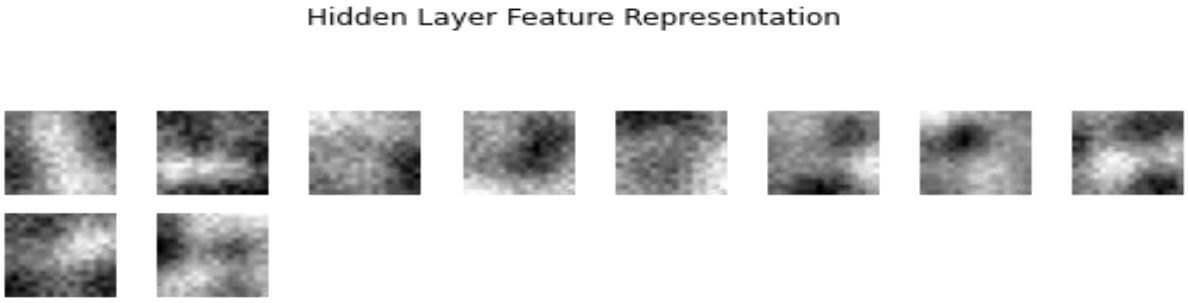


Fig.22 : Self-written model's hidden weights visualization with L_{hid} = 10 and λ = 0

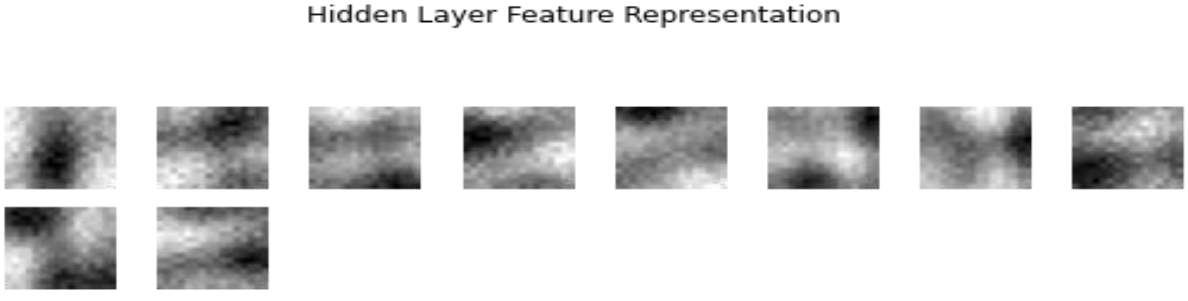


Fig.23 : Keras model's hidden weights visualization with L_{hid} = 10 and λ = 0

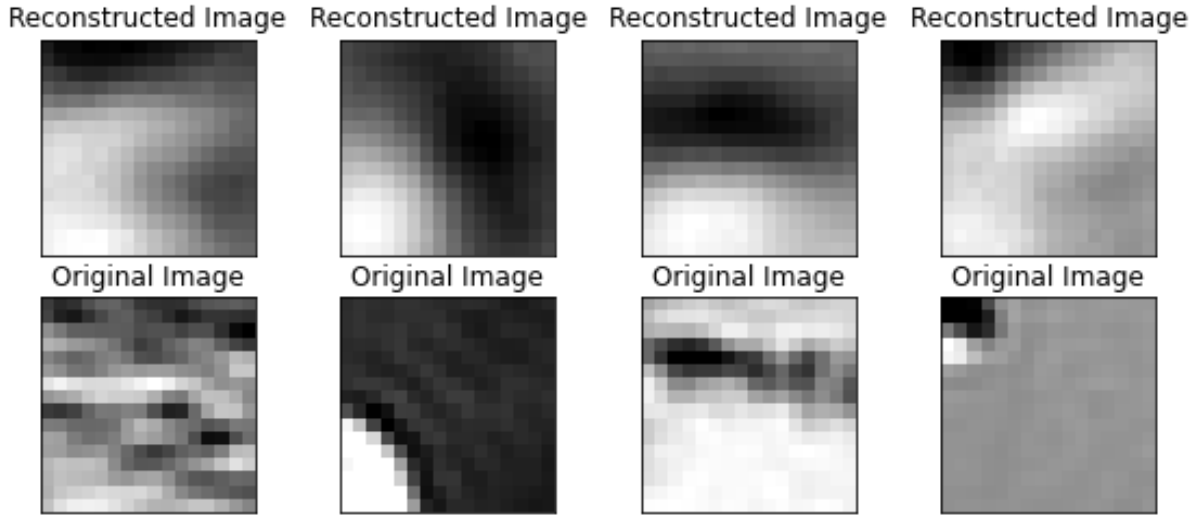


Fig.24 : Self-written model's predictions and original images with L_{hid} = 10 and λ = 0

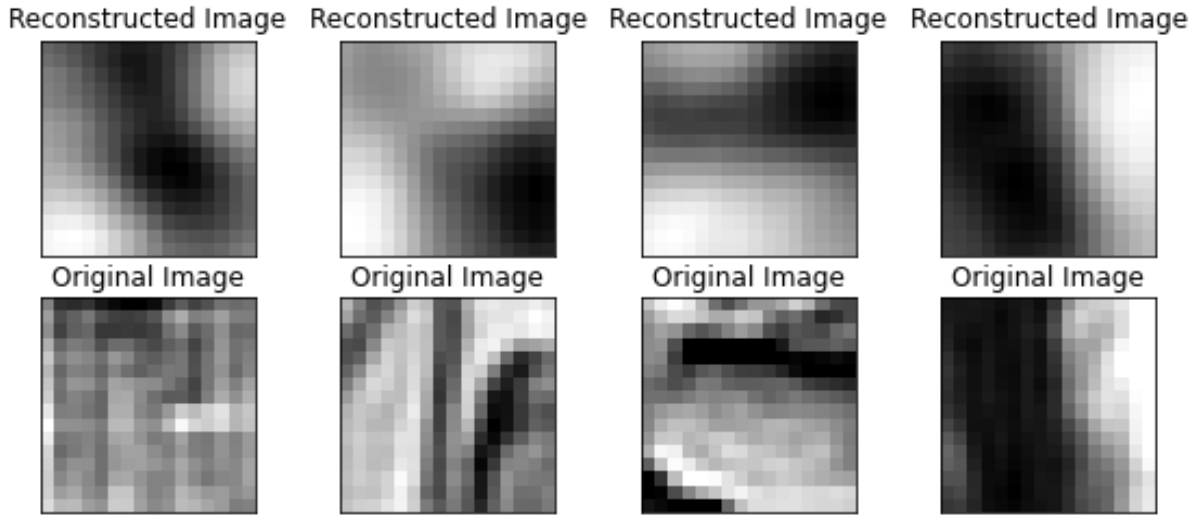


Fig. 25 : Keras model's predictions and original images with L_{hid} = 10 and λ = 0

When we compare the results, the outputs are quite similar as expected and self-written model reconstruction is nearly perfect. Then, note that when L_{hid} = 10 and λ = 0, we have 10 nodes to represent the input images so that's the reason why we have 10 separate images. Then, since the L_{hid} is decreased, we expect that the model learn the compressed version of the data. Hence, when we compare the part-b self-written model results (Recall that L_{hid} = 64) with current one we see that with the number of the hidden size decreased, the network are forced learn the representation of the data with 10 units instead of 64 units that result in variant in the reconstruction and feature representation. Note that I am going to discuss the effect of the λ in the end of each hidden size part.

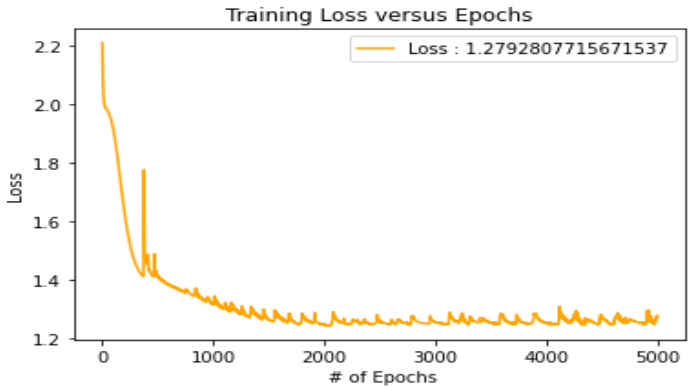


Fig. 26 : Self-written model's loss over training with L_{hid} = 10 and λ =10^{-3}

Hence, the loss is increased w.r.t. the previous model when we set λ =10^{-3}. Let's see the effect of the λ in the hidden representation:

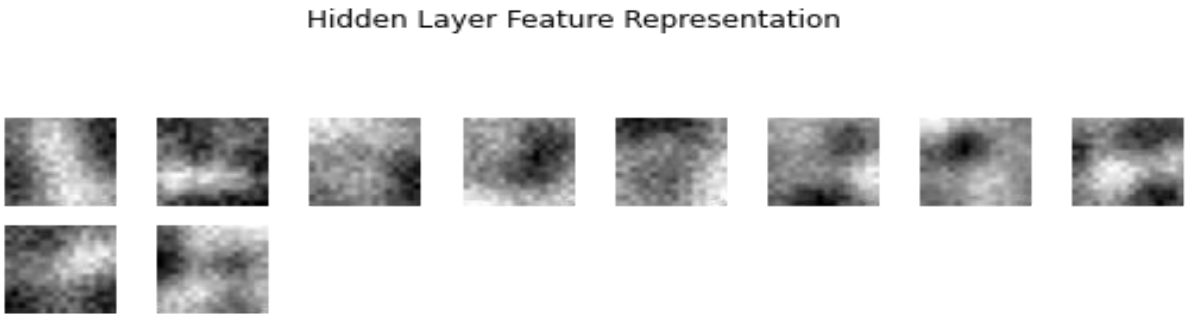


Fig. 27 : Self-written model's hidden representation with L_{hid} = 10 and λ =10^{-3}

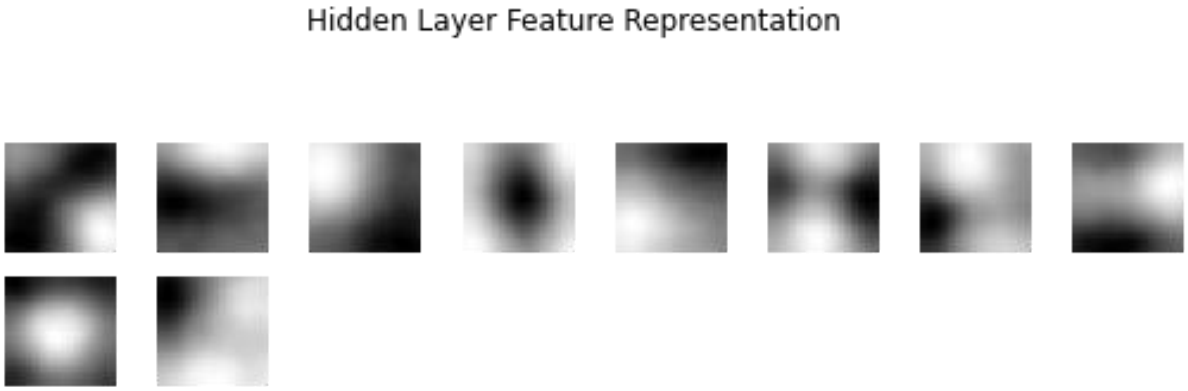


Fig. 28 : Keras model's hidden representation with L_{hid} = 10 and λ =10^{-3}

At that point, we see that there is little divergence between self-written and Keras model. It may depend on varius things such as the stability of the code, sensitivy of the code and much more. Anyway, when we set λ =10^{-3}, we see that the network's hidden representation is not as well as the model with $\lambda = 0$. It is very natural since the penalty effect is increased so that the extra constraint is putted that result in lower level feature extraction.

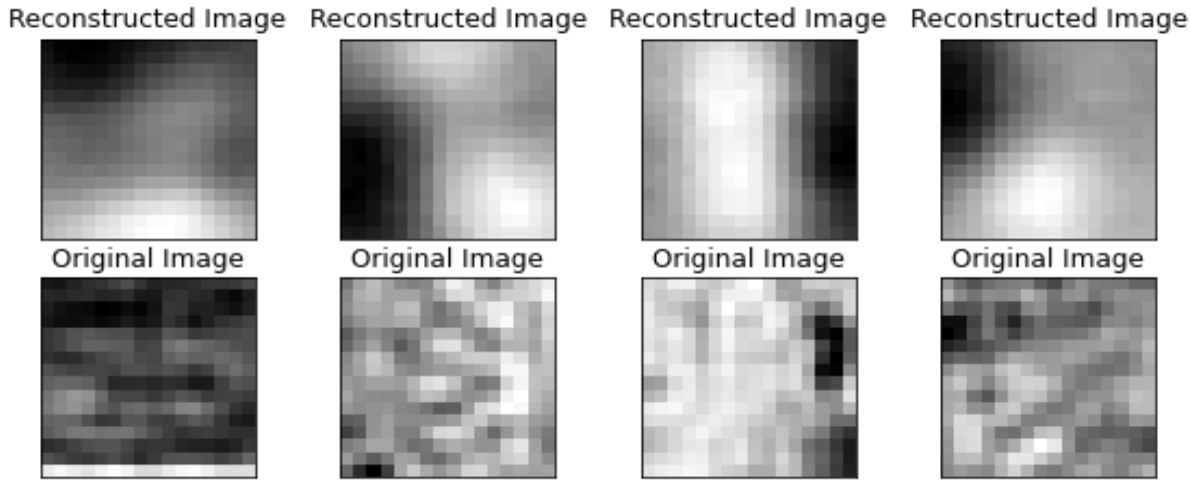


Fig. 29 : Self-written model's prediction and original images with L_{hid} = 10 and λ =10^{-3}

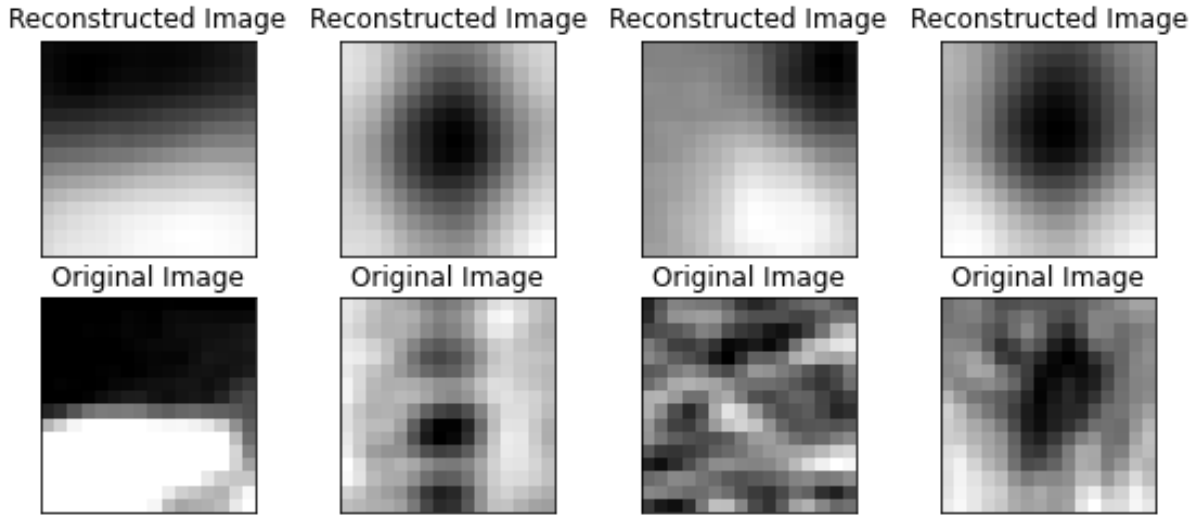


Fig. 30 : Keras model's prediction and original images with L_{hid} = 10 and λ =10^{-3}

Lastly, we can conclude that with the penalization term incrases, the model is forced to learn in compressed way that result in superficial feature representation.

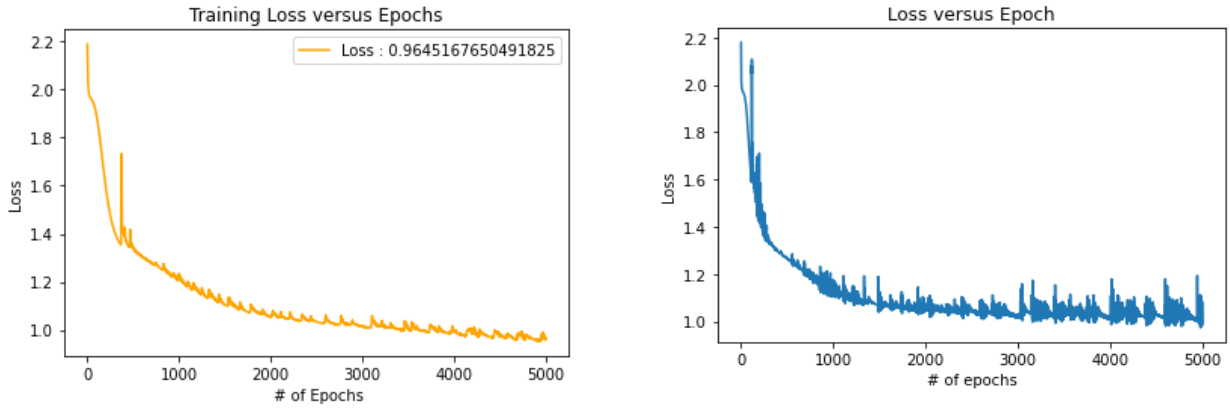


Fig. 31,32 : Self-written model's and Keras model's loss respectively with L_{hid} = 10 and λ = 10^{-5}

We can see that my model is perfomed similar with the Keras model.

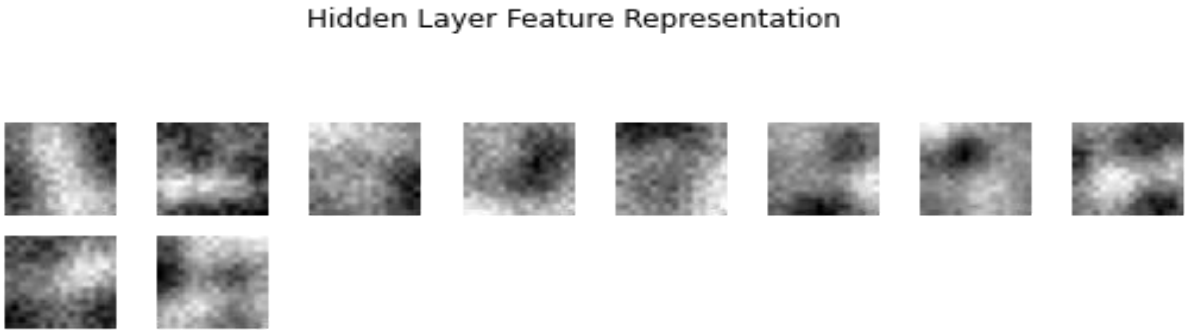


Fig. 32 : Self-written model's hidden representation with L_{hid} = 10 and $\lambda = 10^{-5}$

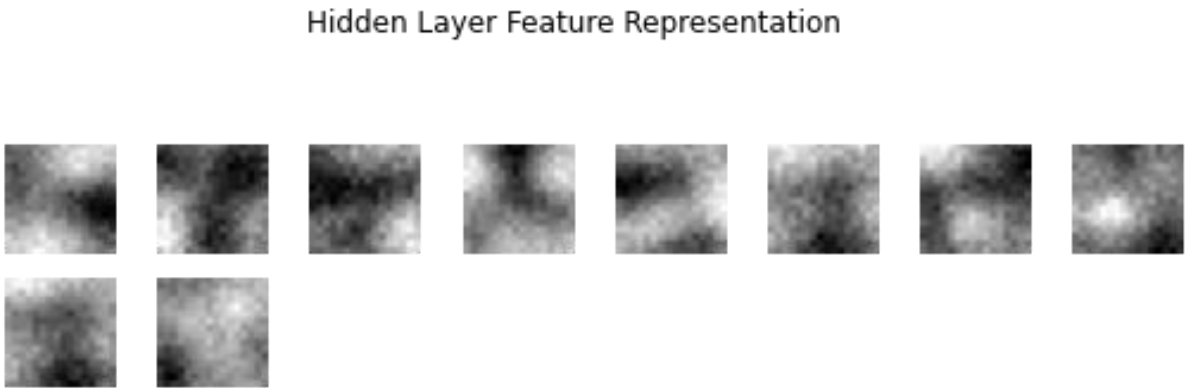


Fig. 33 : Keras model's hidden representation with L_{hid} = 10 and $\lambda = 10^{-5}$

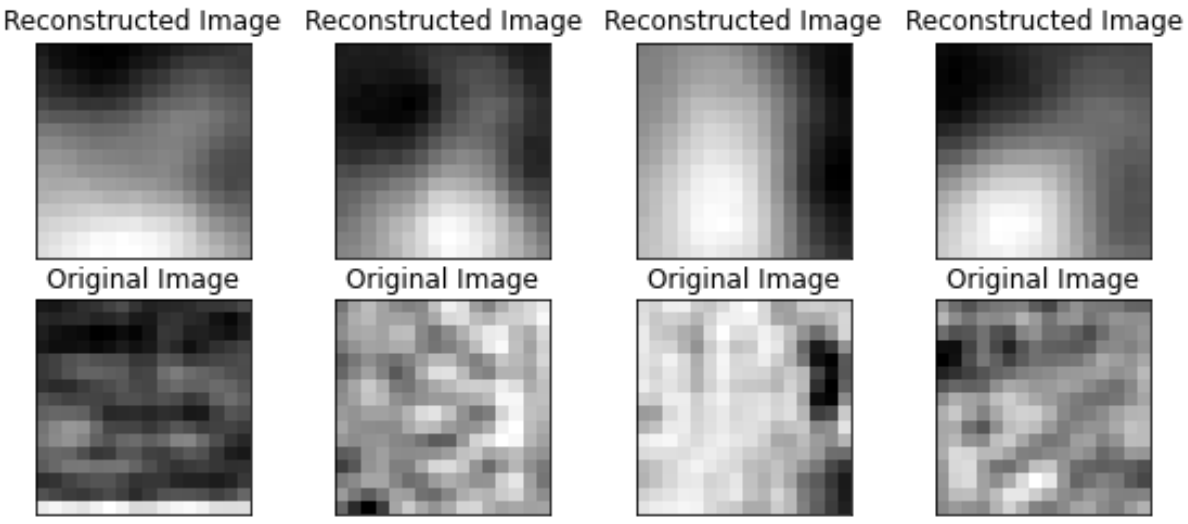


Fig. 34 : Self-written model's reconstruction with L_{hid} = 10 and $\lambda = 10^{-5}$

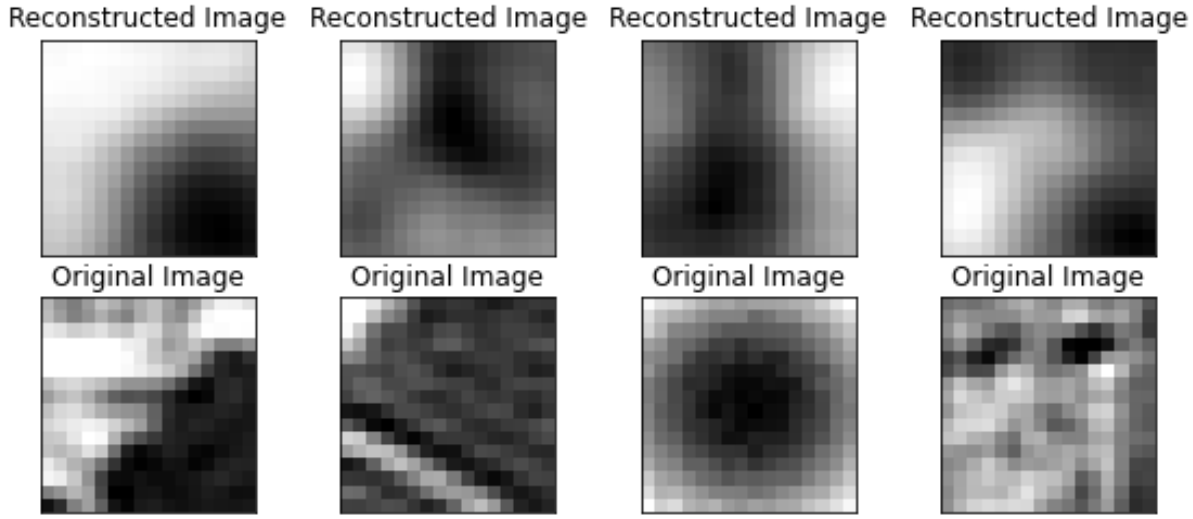


Fig. 35 : Keras model's hidden reconstruction with L_{hid} = 10 and $\lambda = 10^{-5}$

In this case, we see that the model with L_{hid} = 10 and $\lambda = 10^{-5}$ performed acceptable in hidden representation and the prediction part. However, when we decrease the $\lambda = 10^{-3}$ to 10^{-5}, we see that the model performed closer to reality since the penalty effect is decreased.

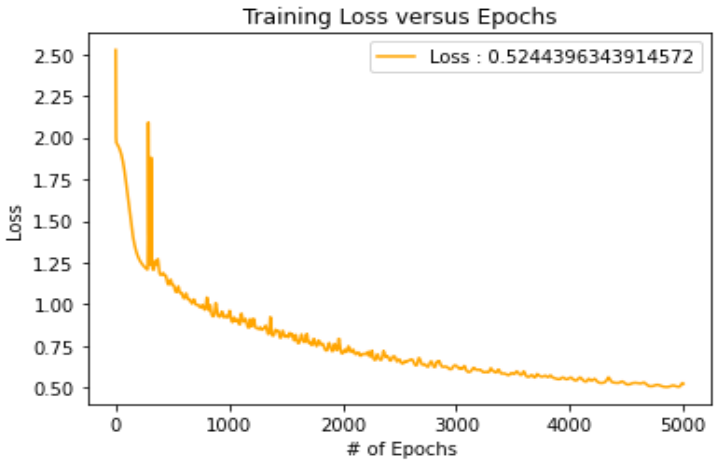


Fig. 36 : Self-written model's loss over training with L_{hid} = 50 and λ = 0

From now on, I did not give Keras implementations and results becaue I belive I proved that my model works well.

As we expect, with the number of hidden size is increased, the model performed better because we have more nodes to learn input features. When we compare the previous model with L_{hid} = 10 and λ = 0, we see that L_{hid} = 50 is performed better.

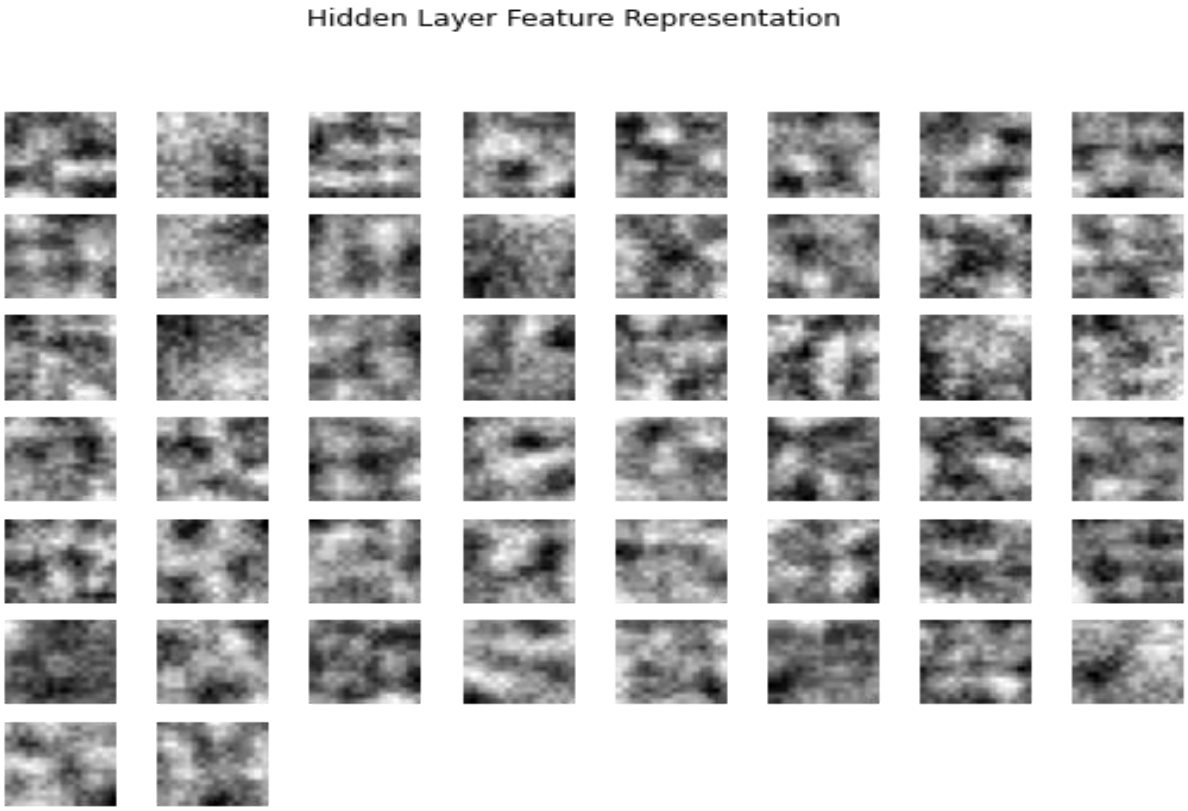


Fig. 37 : Self-written model's hidden representation with L_{hid} = 50 and λ = 0

From the figure above, we see that model with hidden size 50, performed also better than it's analogous. The extracted features look quite well.

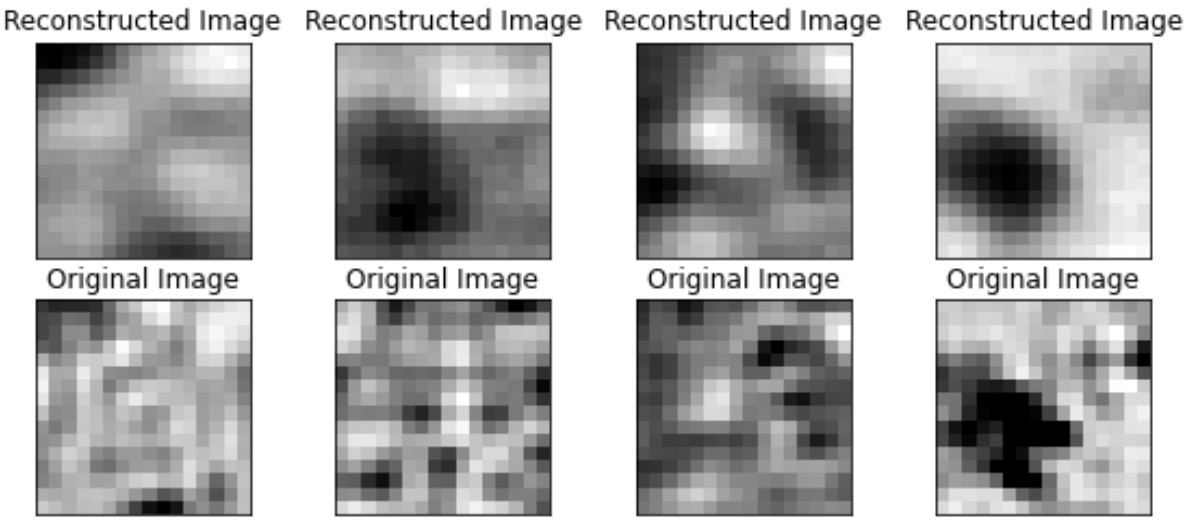


Fig. 38 : Self-written model's hidden representation with L_{hid} = 50 and λ = 0

To test this network, I tried to select the input samples that are comparatively harder to represent due to it's pixel distribution along height and width axis to analyze it. Here in the above figure, since there are little correlations in the input image, the representation of the features is much harder but the model actually performed acceptable in less correlated input images.

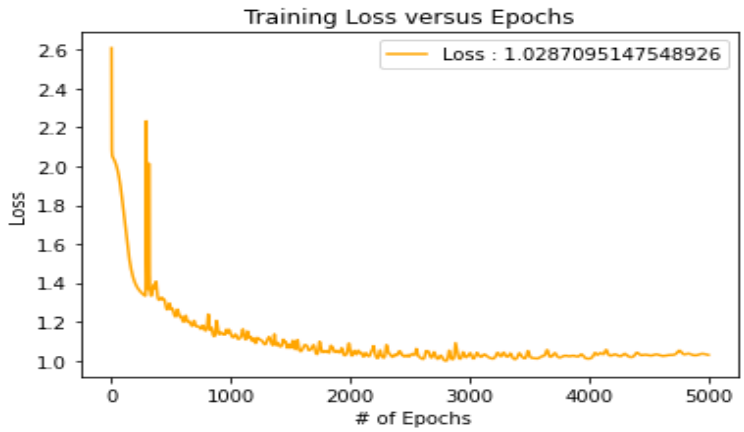


Fig. 39 : Self-written model's loss over training with L_{hid} = 50 and $\lambda = 10^{-3}$

As we expect, with the penalty term increases, loss is increased because the extra constraint is putted to network to learn more superficial features of the input.

Note that since they are all hyperparameters of the network and depens on the lots of conditions such as mean, variance and so on, the intuition may not hold in extreme cases.

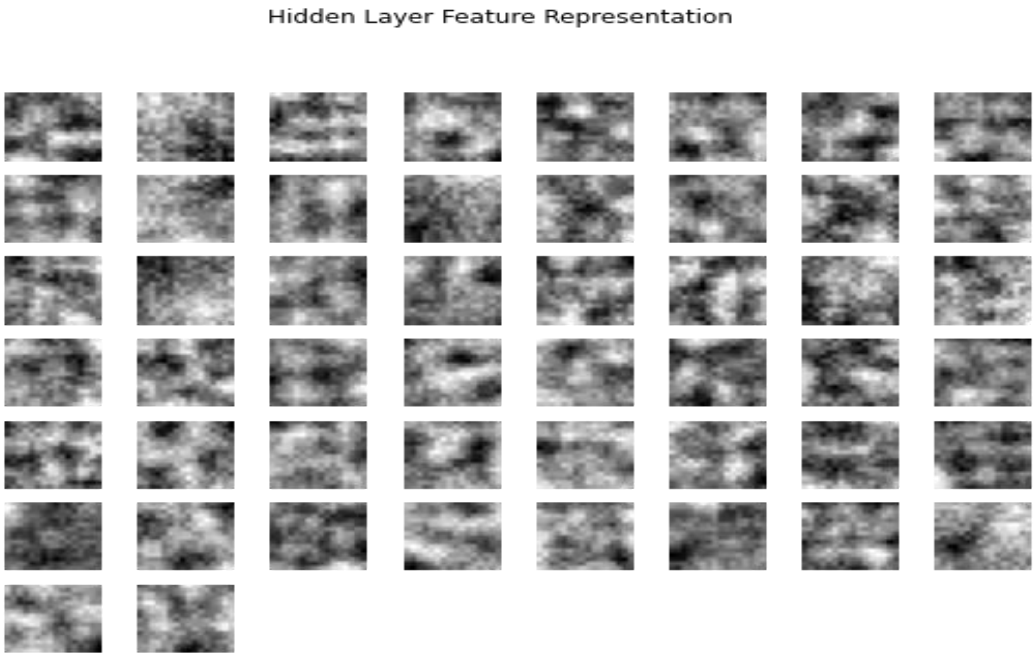


Fig. 40 : Self-written model's hidden representation with L_{hid} = 50 and $\lambda = 10^{-3}$

With the increase of the parameter λ, the model is extracted the more low level features like contrast.

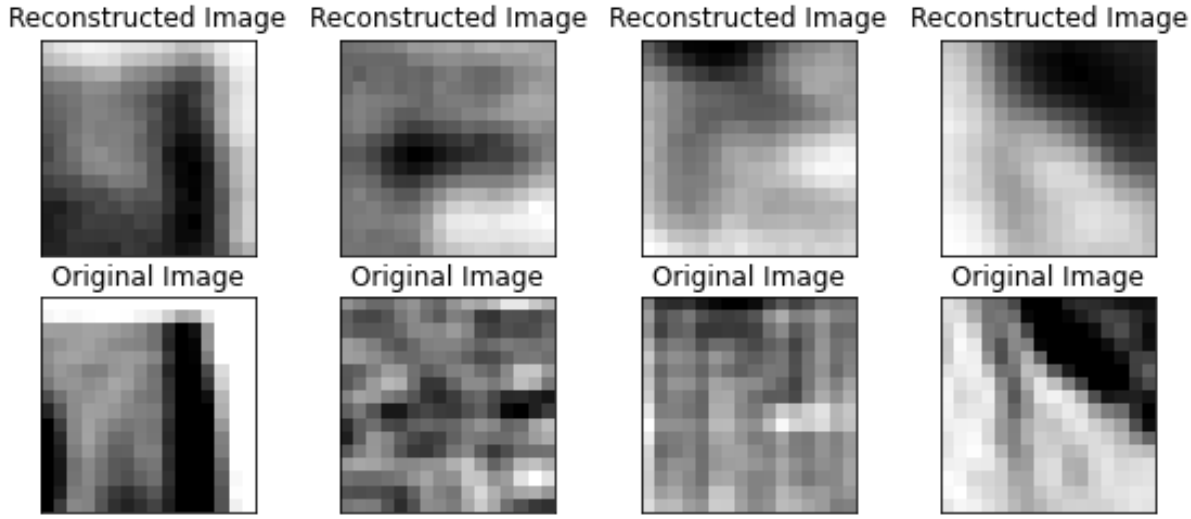


Fig. 41 : Self-written model's reconstruction with L_{hid} = 50 and $\lambda = 10^{-3}$

As I expect, the model with $\lambda = 10^{-3}$ is performed so that the low level features of the input image are reconstructed.

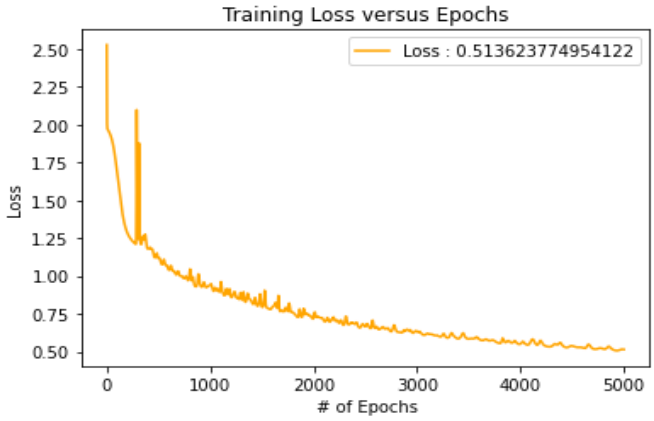


Fig. 42 : Self-written model's loss over training with L_{hid} = 50 and $\lambda = 10^{-5}$

I decreased the penalty effect on the loss function to release the network to be learn better. As expected, model is performed better.

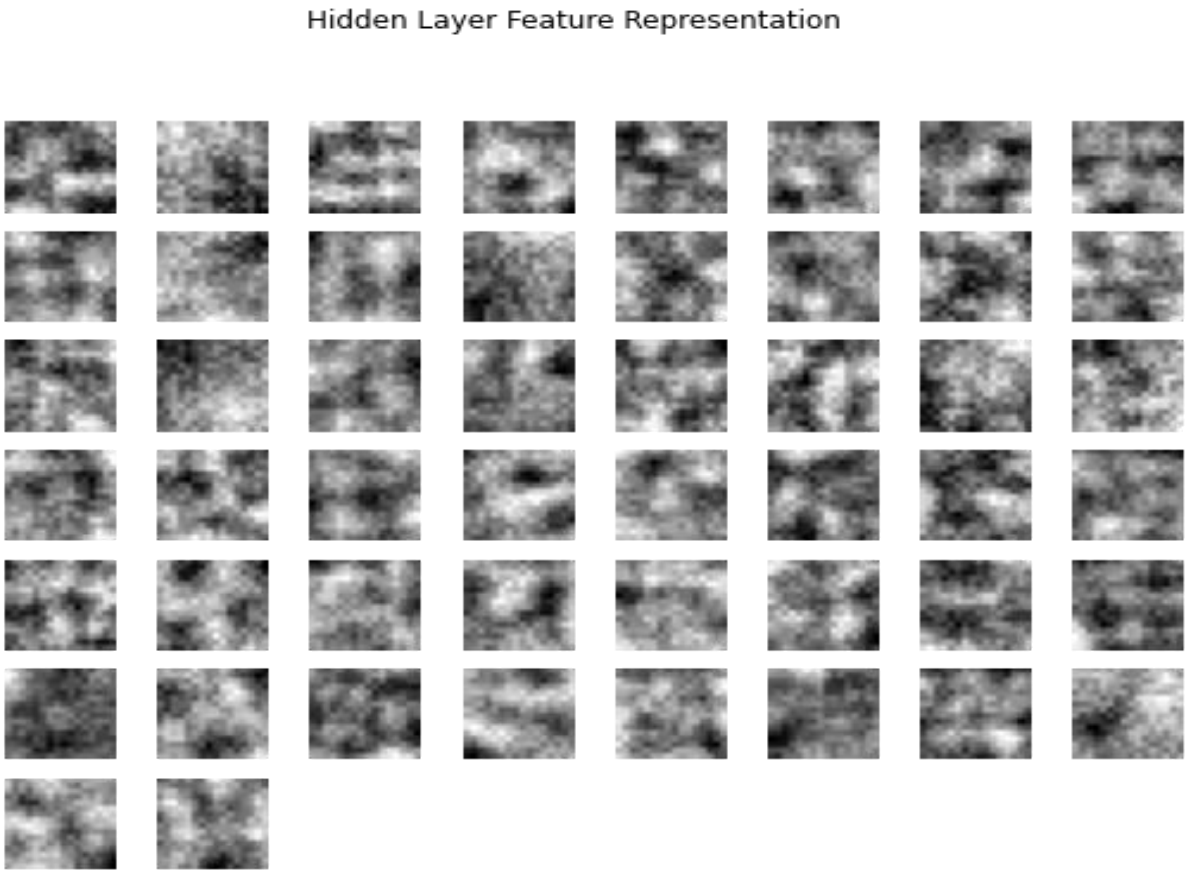


Fig. 43 : Self-written model's hidden representation with L_{hid} = 50 and $\lambda = 10^{-5}$

The hidden representation of the model is more evident as expected since it is learned better.

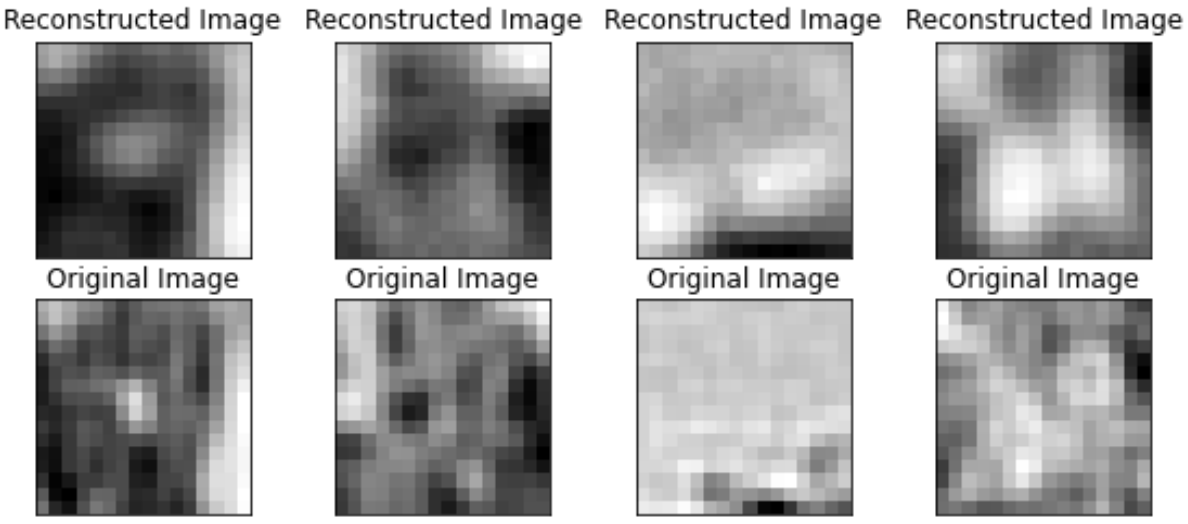


Fig. 44 : Self-written model's reconstruction with L_{hid} = 50 and $\lambda = 10^{-5}$

From the figure above, we can see that model with 50 and $\lambda = 10^{-5}$ is learned better w.r.t. the model with and $\lambda = 10^{-3}$ with the same hidden size.

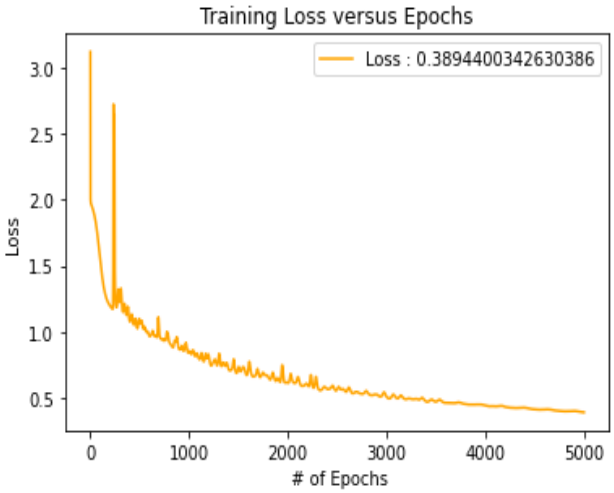


Fig. 45 : Self-written model's loss over training with L_{hid} = 100 and λ = 0

With the increment of the hidden size and giving no penalty to the network is resulted in best network among others. The reason is that we have more units to imitate the features of the given input and there is no obstactle learn in a deep way so that the model is performed best. When we compare this network with the model with Adam optimizer, we see that with the correct hyperparameters we can reach similar results with the model with sophisticated optimization algorithms.

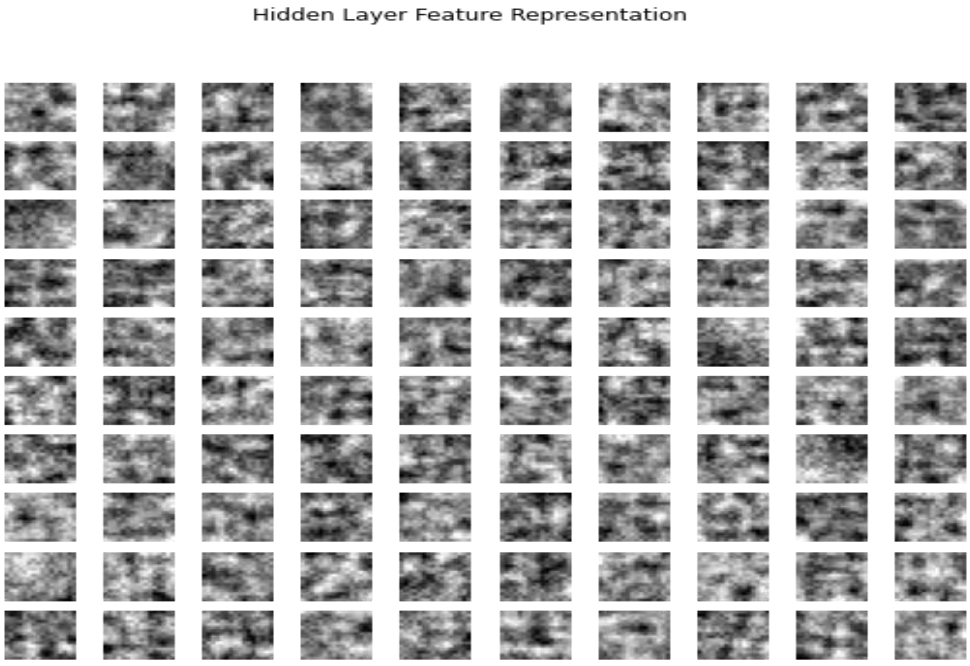


Fig. 46 : Self-written model's hidden representation with L_{hid} = 100 and λ = 0

As expected, most clear and prominent results are done with the model L_{hid} = 100 and λ = 0.

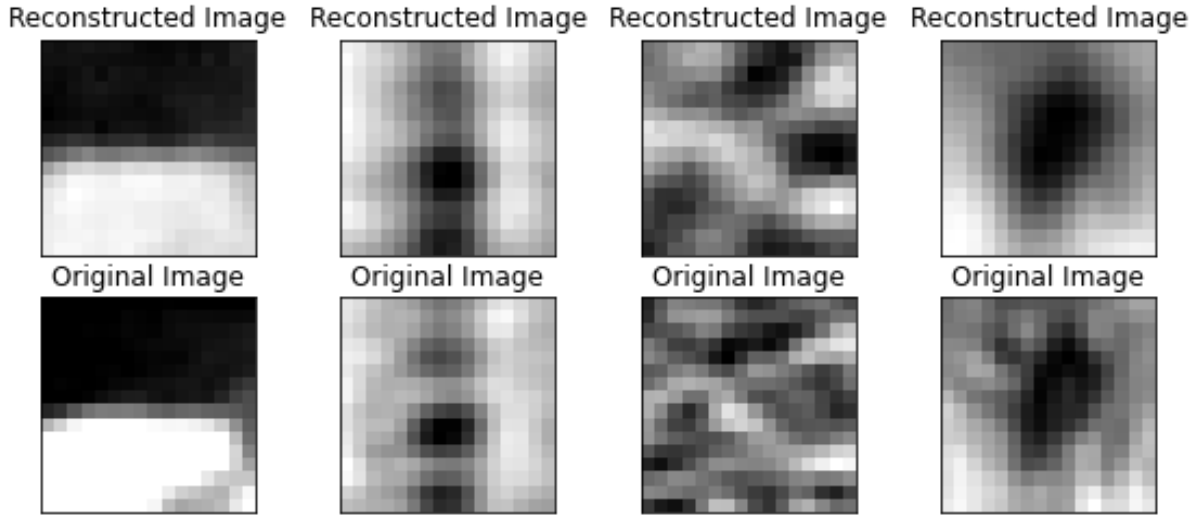


Fig. 47 : Self-written model's reconstruction with L_{hid} = 100 and λ = 0

When we compare the reconstructions and the original images we see that they are nearly identical to each other. This is what we expect for the reason provided above.

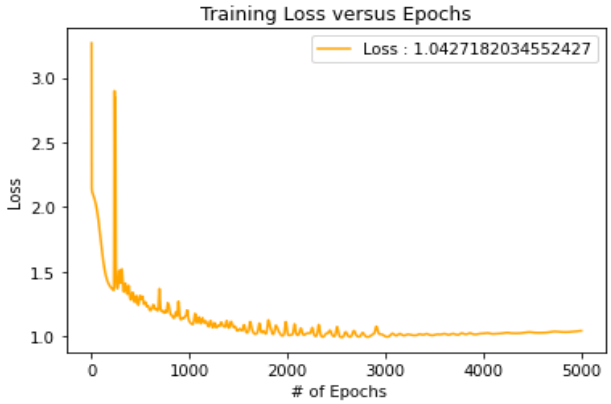


Fig. 48 : Self-written loss over training with L_{hid} = 100 and $\lambda = 10^{-3}$

With the increment of the penalty, model's loss is increased. The reason is provided in above plottings-results, so there is no need to repetetion.

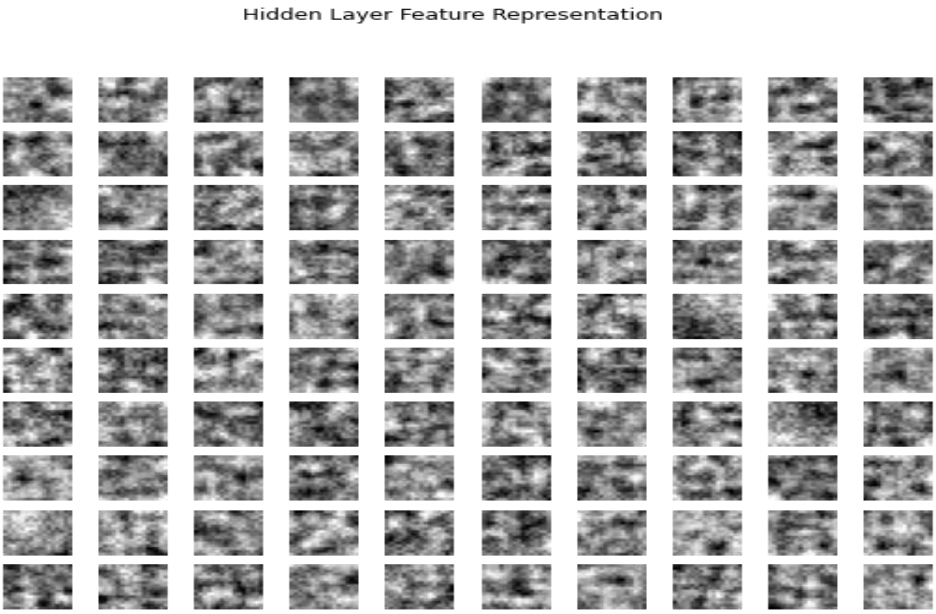


Fig. 49 : Self-written model's hidden representation with L_{hid} = 100 and $\lambda = 10^{-3}$

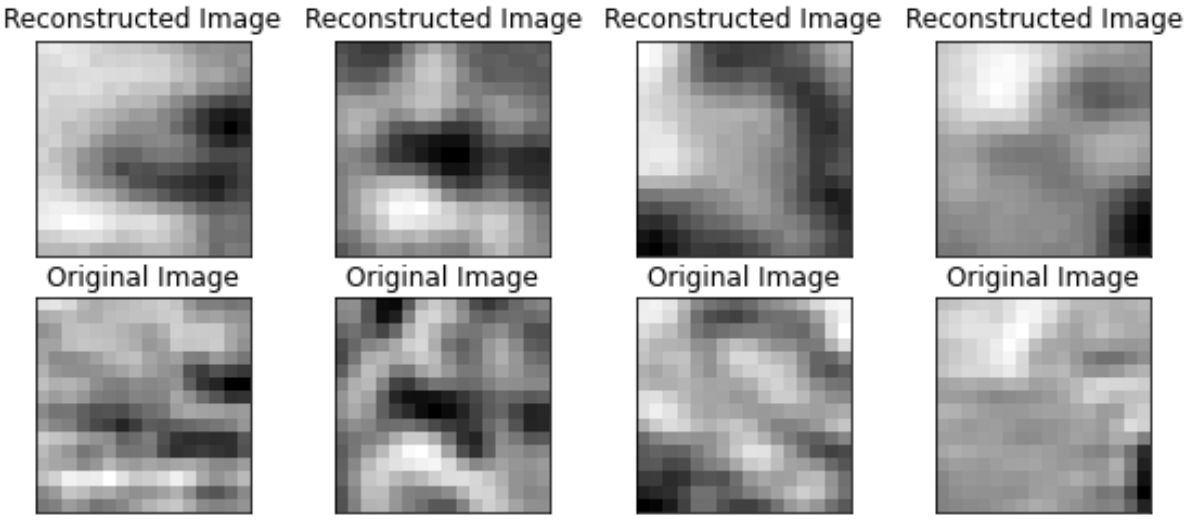


Fig. 50 : Self-written model's reconstruction with L_{hid} = 100 and $\lambda = 10^{-3}$

I explanied all possible combination of the results so there is no need to repetition.

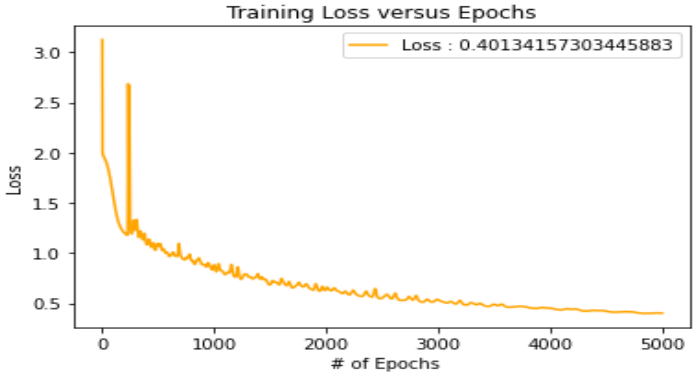


Fig. 51 : Self-written model's loss over training with L_{hid} = 100 and $\lambda = 10^{-5}$

As expected, with the decrement of the penalty term, the model's performance on the training loss is increased so that model is learned better.

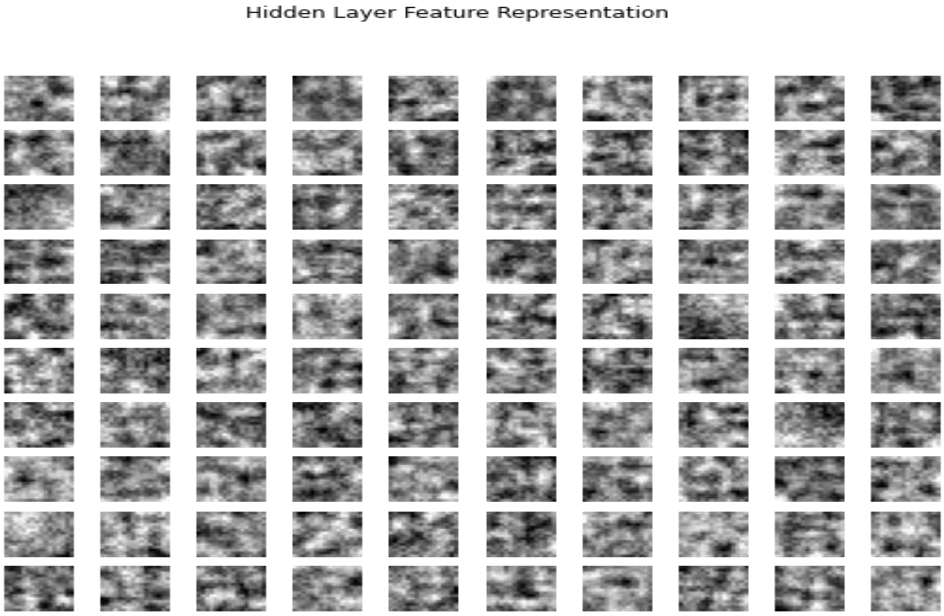


Fig. 52 : Self-written model's hidden representation with L_{hid} = 100 and $\lambda = 10^{-5}$

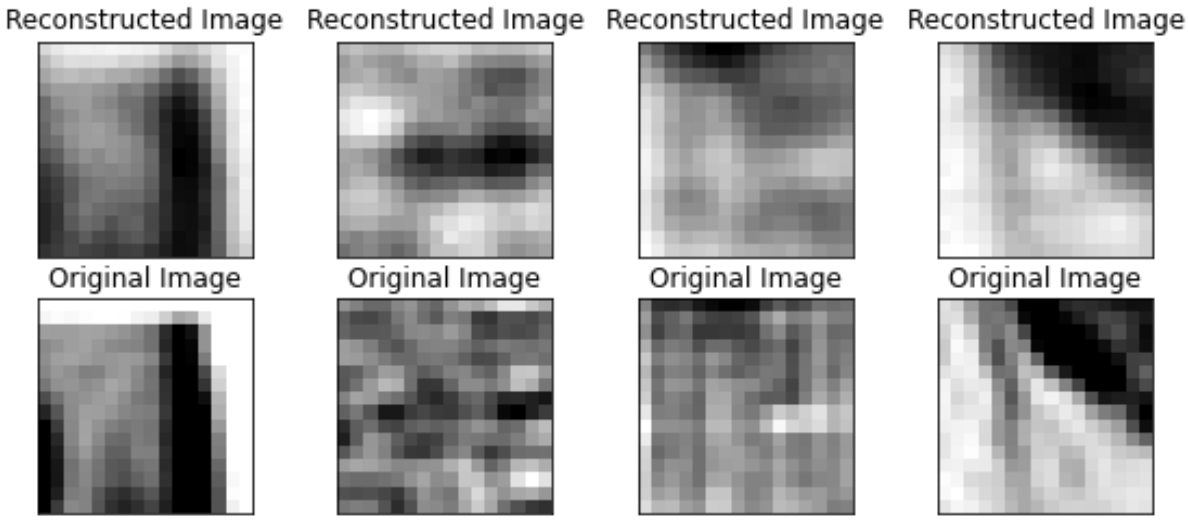


Fig. 53 : Self-written model's predictions versus original images with L_{hid} = 100 and $\lambda = 10^{-5}$

Finally, we can conclude that the hidden representation and the reconstruction capacity of the model is depending on:

1) Hidden layers size L_{hid}
   We can say that with the increment of the hidden layer's size of the network, the model adapts better to the input data because it can represent them by less compressed way so that the

model can learn better from this approach. This technique can be used in higher level feature extraction's.

2) Weight Decay/Penalty term
We can conclude that with the penalty term increases, the model is forced to penalize more so that certain constraints are putted to the network that result in extraction of low level features. Intuitionally, model is forced not to learn fully so that only it can extract primitive features on the input data. Actually, this can be useful approach when the input data has high dimension and you want apply some sort of dimensionality reduction techniques to reduce overfitting. You can apply some sort of linear dimensionality reduction techniques such as Principle Component Analysis (PCA), but the advantages of the auto-encoders are that they can serve non-linear dimensionality reduction so that information is kept in better way. The following figure explains the advantages of the auto-encoders over the linear dimensionality reduction techniques:

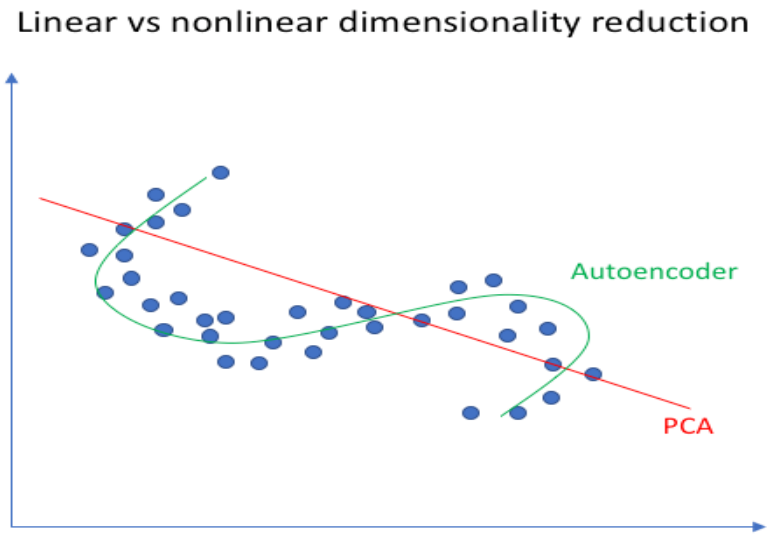


Fig. 54: The visualization of PCA(Linear) versus Autoencoders(Non-linear) in dimensionlity reduction

2 QUESTION 2

In this question, I am going to explore the convolutional neural networks by experimenting with two demos.

2.1 PART A

In this part, I am going to run the given code and analyze the forward, backward and other operations. The code below is the convolution operation:

```
def conv_forward_naive(x, w, b, conv_param):
    """
    A naive implementation of the forward pass for a convolutional layer.

    The input consists of N data points, each with C channels, height H and
```

```
    width W. We convolve each input with F different filters, where each filter
    spans all C channels and has height HH and width WW.

    Input:
    - x: Input data of shape (N, C, H, W)
    - w: Filter weights of shape (F, C, HH, WW)
    - b: Biases, of shape (F,)
    - conv_param: A dictionary with the following keys:
      - 'stride': The number of pixels between adjacent receptive fields in the
        horizontal and vertical directions.
      - 'pad': The number of pixels that will be used to zero-pad the input.
    Returns a tuple of:
    - out: Output data, of shape (N, F, H', W') where H' and W' are given by
      H' = 1 + (H + 2 * pad - HH) / stride
      W' = 1 + (W + 2 * pad - WW) / stride
    - cache: (x, w, b, conv_param)
    """
    out = None

    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    stride, pad = conv_param['stride'], conv_param['pad']
    H_out = 1 + (H + 2 * pad - HH) // stride  # Use `//` for python3
    W_out = 1 + (W + 2 * pad - WW) // stride
    out = np.zeros((N, F, H_out, W_out))

    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant', constant_val
ues=0)

    for n in range(N):
      for f in range(F):
        for h_out in range(H_out):
          for w_out in range(W_out):
            out[n, f, h_out, w_out] = np.sum(
              x_pad[n, :, h_out*stride:h_out*stride+HH, w_out*stride:w_out*stride
+WW]*w[f, :]) + b[f]
    cache = (x, w, b, conv_param)
    return out, cache
```

Note that we have N inputs with C channels and the height is represented as H and width W. We are going to convolve the input image with F different filters where each filter spans all C channel and has height and width HH and WW respectively. Finally, this functions returns output of the convolution and convolutions parameters. Note that at the output we have a shape of (N, F, H', W') and H' and W':

$$H' = \frac{H + 2 * Padding - H_W}{Stride} + 1 \text{ , and } W' = \frac{W + 2 * Padding - W_W}{Stride} + 1$$

To understand that, let ⊙ be the convolution operation and X be the input $\mathcal{F}$ be the filter(F is the number of filter) and $\hat{Y}$ be the output of the convolution operation the subscripts represents the dimension of the variables so that:

$$X_{(H\ x\ W\ x\ C)} \odot \mathcal{F}_{(H_W\ x\ W_W\ x\ C)} = \hat{Y}_{\left(\left(\frac{H+2*Padding-H_W}{Stride}+1\right)x\left(\frac{W+2*Padding-W_W}{Stride}+1\right)x\ (F)\right)}$$

Let's look at an example:

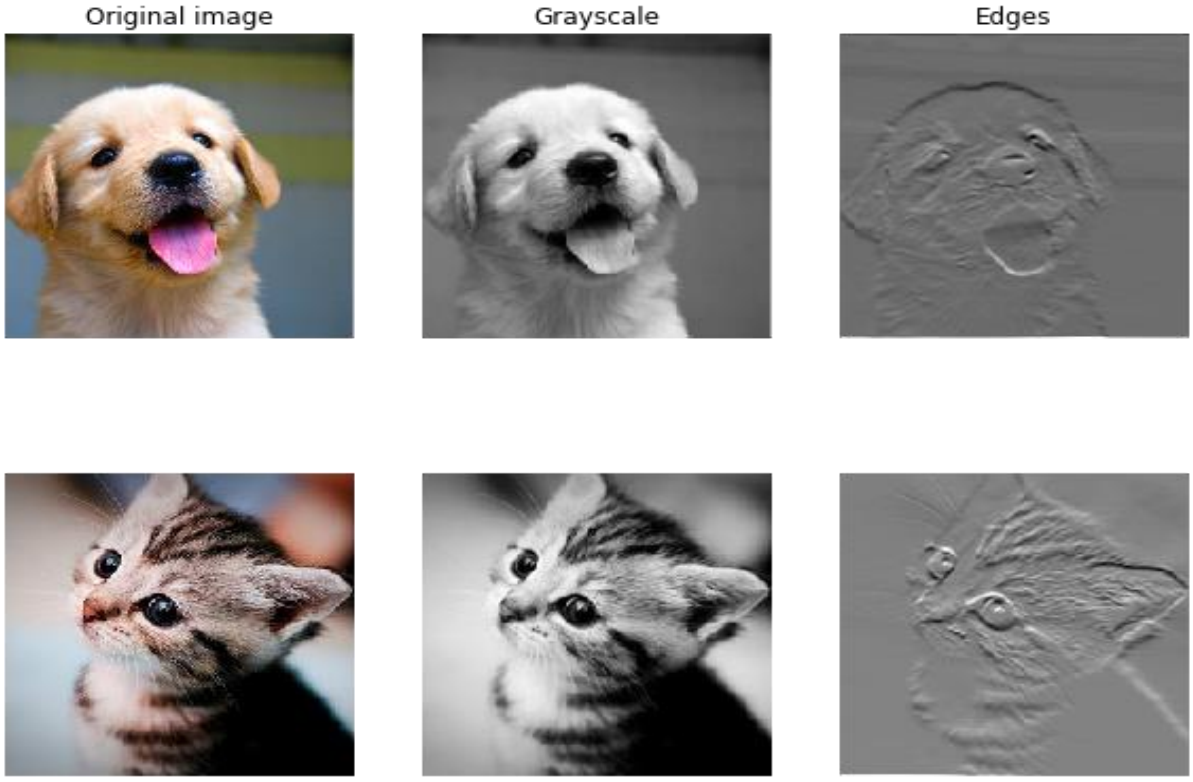


Fig. 55: Original images, gray scale versions and edges of the images

We see that with convolution operator we can extract different version of the natural images that is very useful in computer vision area. Let's see the filters that behave like a gray scaler and edge detector:

$$F_{Red} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.3 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad F_{Green} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.6 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad F_{Blue} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

These filter corresponds the RGB color channels respectively, they are 3x3 filters that are widely used in literature. Like the luminosity model, we distributed color contributions so that green color's contribution is increased while blue's decreased. Hence the convolution of the input image, in our case cat and dog, with specified filters outputs a gray scale version of it. Then let's see an edge detector filter:

$$F_{horizantal\ egde\ detector} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

This filter $F_{horizantal\ egde\ detector}$ is prepared for the extract edges in the blue color channels.

Then, let's move on backward propagation of the convolution layer of the neural networks:

```
def conv_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a convolutional layer
.

    Inputs:
    - dout: Upstream derivatives.
    - cache: A tuple of (x, w, b, conv_param) as in conv_forward_naive

    Returns a tuple of:
    - dx: Gradient with respect to x
    - dw: Gradient with respect to w
    - db: Gradient with respect to b
    """
    dx, dw, db = None, None, None

    x, w, b, conv_param = cache
    N, C, H, W = x.shape
    F, _, HH, WW = w.shape
    _, _, H_out, W_out = dout.shape
    stride, pad = conv_param['stride'], conv_param['pad']

    x_pad = np.pad(x, ((0,), (0,), (pad,), (pad,)), mode='constant', cons
tant_values=0)

    dx_pad = np.zeros_like(x_pad)
    dw = np.zeros_like(w)
    db = np.zeros_like(b)
    for n in range(N):
      for f in range(F):
        db[f] += np.sum(dout[n, f])
        for h_out in range(H_out):
          for w_out in range(W_out):
            dw[f] += x_pad[n, :, h_out*stride:h_out*stride+HH, w_out*stri
de:w_out*stride+WW] * \
            dout[n, f, h_out, w_out]
            dx_pad[n, :, h_out*stride:h_out*stride+HH, w_out*stride:w_out
*stride+WW] += w[f] * \
            dout[n, f, h_out, w_out]

    dx = dx_pad[:, :, pad:pad+H, pad:pad+W]
    return dx, dw, db
```

I am going to explain the backward operation in the convolutional neural network. To do that, let's take a look at that example

Let X be the input and Let W be the kernel so that:

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} \text{ and } W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

Then, lets stride = 1 and no padding just the sake of simplicity. Then the output Y should in the shape of 3x3. ($X_{4x4} \odot W_{2x2} = Y_{3x3}$). Therefore, Y becomes:

$$Y = \begin{bmatrix} y_{11} & y_{12} & y_{13} \\ y_{21} & y_{22} & y_{23} \\ y_{31} & y_{32} & y_{33} \end{bmatrix}$$

In the forward propagation we have:

$$Y_{ij} = (\sum_{k=1}^{2} \sum_{l=1}^{2} w_{kl}\, x_{i+k+1,j+l-1}) + b \quad \forall (i,j) \in \{1,2,3\}^2$$

Then, let's look at the backward propagation, Let L be loss so that:

$$dY_{ij} = \frac{\partial L}{\partial Y_{ij}}$$

Then, with a little trick we have:

$$db = dY_{ij} * \frac{\partial Y_{ij}}{\partial b}$$

Then, the summation on i and j gives:

$$\frac{\partial Y_{ij}}{\partial b} = 1 \ \ \forall (i,j) \text{ and } db = \sum_{i=1}^{3} \sum_{j=1}^{3} dY_{ij}$$

Then, let's look at the W term:

$$dW = \frac{\partial L}{\partial Y_{ij}} * \frac{\partial Y_{ij}}{\partial W} = dY * \frac{\partial Y}{\partial W}$$

$$dW_{mn} = dY_{ij} * \frac{\partial Y_{ij}}{\partial W_{mn}}$$

The need to compute $\frac{\partial Y_{ij}}{\partial W_{mn}}$. Then using the forward pass equation, we have

$$\frac{\partial \mathrm{Y}_{ij}}{\partial \mathrm{W}_{mn}} = (\sum_{k=1}^{2} \sum_{l=1}^{2} \frac{w_{kl}}{\mathrm{W}_{mn}} x_{i+k+1,j+l-1})$$

Note that $\frac{w_{kl}}{\mathrm{W}_{mn}}$ = 0 everywhere expect (k,l) = (m,n).Therefore, we have:

$$\frac{\partial \mathrm{Y}_{ij}}{\partial \mathrm{W}_{mn}} = x_{i+k+1,j+l-1}$$

This result yields:

$$\mathrm{dW}_{mn} = \mathrm{dY}_{ij} * x_{i+k+1,j+l-1}$$

Finally, we have:

$$\mathrm{dW}_{mn} = \sum_{i=1}^{3} \sum_{j=1}^{3} \mathrm{dY}_{ij} * \ x_{i+k+1,j+l-1}$$

Then we can calculate dX_{mn} by:

$$\mathrm{dX}_{mn} = \mathrm{dY}_{ij} * \frac{\partial \mathrm{Y}_{ij}}{\partial \mathrm{X}_{mn}}$$

Now, we need to calculate $\frac{\partial \mathrm{Y}_{ij}}{\partial \mathrm{X}_{mn}}$. Therefore, we can write is as a:

$$\frac{\partial \mathrm{Y}_{ij}}{\partial \mathrm{X}_{mn}} = \sum_{i=1}^{2} \sum_{j=1}^{2} w_{kl} \ * \frac{\partial \ x_{i+k+1,j+l-1}}{\mathrm{X}_{mn}}$$

Then, we have:

$$\frac{\partial \ x_{i+k+1,j+l-1}}{\mathrm{X}_{mn}} = \begin{cases} 1 \ , if \ m = i + k - 1 \ and \ n = j + l - 1 \\ 0 \ , \qquad\qquad else \end{cases}$$

With re-arranging the placements we have k = m –I +1 and l = n – j +1 and recall that m,n ∈ [1,4] inputs, k,l ∈ [1,2] filters and i,j ∈ [1,3] outputs.

However, when we set k = m – I +1, there is a problem with boundaries since (m – I +1) ∈ [-1,4]. To keep going, we choose to extend the definition of matrix W with zeros. Then, we have:

$$\frac{\partial \mathrm{Y}_{ij}}{\partial \mathrm{X}_{mn}} = W_{m-i+1,n-j+1}$$

Where $W_{m-i+1,n-j+1}$ is the extended version of the initial filter. Then,

$$\mathrm{dX}_{mn} = \sum_{i=1}^{3} \sum_{j=1}^{3} d\mathrm{Y}_{ij} * \ W_{2-i,2-j} \ = \sum_{i=1}^{3} d\mathrm{Y}_{i1} * \ W_{2-i,1} + \ + d\mathrm{Y}_{i2} * \ W_{2-i,0} + d\mathrm{Y}_{i3} * \ W_{2-i,-1}$$

Summary of the backpropagation equations:

$$db = \sum_{i=1}^{3}\sum_{j=1}^{3} dY_{ij}$$

$$dW = \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} \odot \begin{bmatrix} dy_{11} & dy_{12} & dy_{13} \\ dy_{21} & dy_{22} & dy_{23} \\ dy_{31} & yd_{32} & yd_{33} \end{bmatrix} = X \odot dY$$

$$dX = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & dy_{11} & dy_{12} & dy_{13} & 0 \\ 0 & dy_{21} & dy_{22} & dy_{23} & 0 \\ 0 & dy_{31} & dy_{32} & yd_{33} & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \odot \begin{bmatrix} w_{22} & w_{21} \\ w_{12} & w_{11} \end{bmatrix} = dY_0 \odot W^T$$

Note that backpropagation parts are retrieved by the course CS231n: Convolutional Neural Networks for Visual Recognition course notes. The needed citation is placed in reference part.

Max-Pooling part

Max-pooling implementation of the Python code is:

```
def max_pool_forward_naive(x, pool_param):
    """
    A naive implementation of the forward pass for a max-pooling layer.

    Inputs:
    - x: Input data, of shape (N, C, H, W)
    - pool_param: dictionary with the following keys:
      - 'pool_height': The height of each pooling region
      - 'pool_width': The width of each pooling region
      - 'stride': The distance between adjacent pooling regions

    No padding is necessary here. Output size is given by

    Returns a tuple of:
    - out: Output data, of shape (N, C, H', W') where H' and W' are given
 by
      H' = 1 + (H - pool_height) / stride
      W' = 1 + (W - pool_width) / stride
    - cache: (x, pool_param)
    """
    out = None

    N, C, H, W = x.shape
    pool_height = pool_param['pool_height']
    pool_width = pool_param['pool_width']
    stride = pool_param['stride']
    H_out = 1 + (H - pool_height) // stride
```

```
    W_out = 1 + (W - pool_width) // stride
    out = np.zeros((N, C, H_out, W_out))

    for n in range(N):
      for h_out in range(H_out):
        for w_out in range(W_out):
          out[n, :, h_out, w_out] = np.max(x[n, :, h_out*stride:h_out*str
ide+pool_height,
            w_out*stride:w_out*stride+pool_width], axis=(-1, -
2)) # axis can also be (1, 2)

    cache = (x, pool_param)
    return out, cache
```

In this part, I am going to explain max-pooling operation.

Max-pooling is a sample-based discretization process. [2] The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned. [2]

Since max-pooling operation decreases the size of the dimensionality, it helps model to prevent overfitting. Moreover, it decreases the computational cost by reducing the configurable parameters while trying to keep same amount of necessary information. We can think max-pool operation is like a filtering, e.g., let's think of 4x4 matrix with 2x2 filter stride of 2(step size 2, i.e., no overlap) and the filter is doing max operation. The visual representation is:

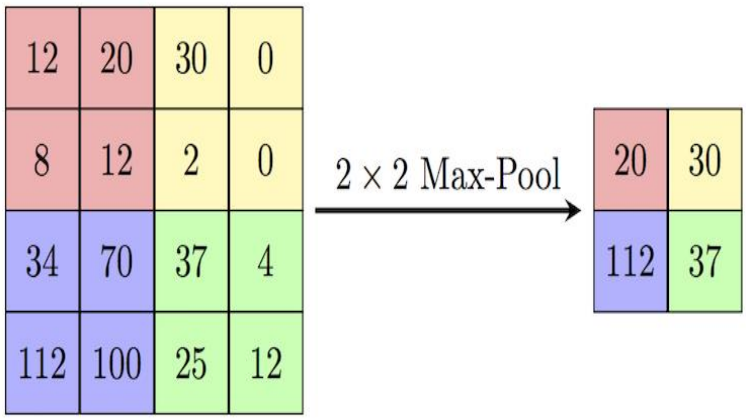


Fig. 56: Max-pooling example [2]

Intutionally, it is very easy since you take first upper left 2x2 matrix and extract the maximum value (in this case 20) then paste the 2x2 matrix and so on.

Then, let's look at real case and after the convolution layer with 64 filter we have an input to the max-pooling layer with a dimension of 224x224x64, the 2x2 max-pooling operation yields the output with 112x112x64 dimensions.

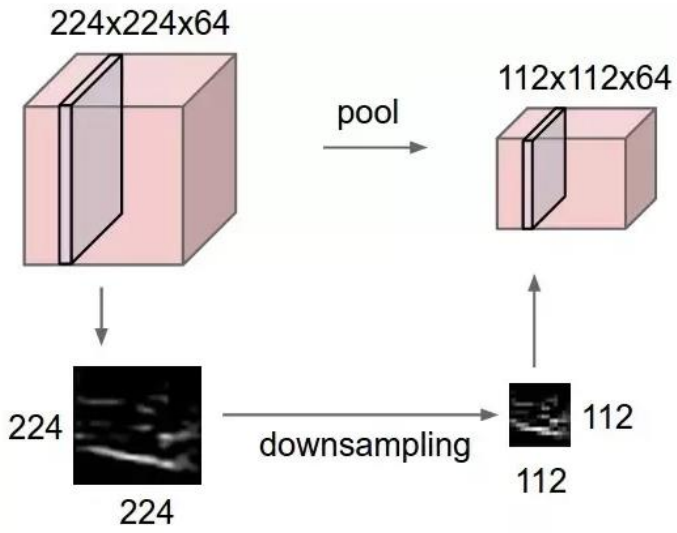


Fig. 57: Max-pooling example in real-life [2]

The below part of the figure shows the real example in the dataset with 224x224 2-D image and down sampling operation.

Then, let's look at the backward operation is the max-pooling layer.

```
def max_pool_backward_naive(dout, cache):
    """
    A naive implementation of the backward pass for a max-pooling layer.

    Inputs:
    - dout: Upstream derivatives
    - cache: A tuple of (x, pool_param) as in the forward pass.

    Returns:
    - dx: Gradient with respect to x
    """
    dx = None

    x, pool_param = cache
    N, C, H, W = x.shape
    pool_height = pool_param['pool_height']
    pool_width = pool_param['pool_width']
    stride = pool_param['stride']
    H_out = 1 + (H - pool_height) // stride
    W_out = 1 + (W - pool_width) // stride
    dx = np.zeros_like(x)

    for n in range(N):
      for c in range(C):
        for h in range(H_out):
```

```
          for w in range(W_out):
            # Find the index (row, col) of the max value
            # Ref: examples of https://docs.scipy.org/doc/numpy-
1.14.0/reference/generated/numpy.argmax.html
            ind = np.unravel_index(np.argmax(x[n, c, h*stride:h*stride+po
ol_height,
              w*stride:w*stride+pool_width], axis=None), (pool_height, po
ol_width))

            dx[n, c, h*stride:h*stride+pool_height, w*stride:w*stride+poo
l_width][ind] = \
            dout[n, c, h, w]
    return dx
```

In the backward propagation, there is no gradient w.r.t. non-maximum values since the change w.r.t. corresponding inputs does not affect the outputs.

Then, suppose we are in layer L that comes on top of a layer P. The usual forward pass is:

$$L_i = f(\sum_j Wij * Pj)$$

Where L_i is the activation is i^{th} node of the layer L, f is the activations and W is the connection weights. Then, by chain rule we have:

$$\nabla Pj = \sum_j \nabla L_i * f'(Wij)$$

Since in this case, activation function f is identity function for the maximum neuron and f zero for all other neurons. Therefore, $f'(.) = 1$ for maximum neuron and $f'(.) = 0$ for others. That yields:

$$\nabla P\text{max neuron} = \sum_j \nabla L_i * Wi\,\max neuron$$

$$\nabla Prest = 0.$$

Fast Layers

Makin the convolution and pooling can be slow and challenging since it's a little bit complicated and computationally costly. (Run time complexity $O(n^4)$ for fully-functional convolution and $O(n^3)$ for fully-functional max-pooling). In this part, the techniques and algorithms used in naïve versions are nearly same in fast layers except this layers are faster with certain conditions.

Convolutional "sandwich" layers

Sandwich layers are referred to the concept of combining multiple layers/operations such as convolution-pooling or weighted summation/activation. In this part, nothing is new but the new layer that is the combination of convolution operation followed max-pooling as a single layer.

Three-layer ConvNet

In this part, the implemented layers are used to classify CIFAR-10 dataset.

Overfitting Small data

In this part, we are using 100 training samples to see overfitting. Overfitting the model means that we reached high training accuracy but low validation/testing accuracy so that model can be generalized.

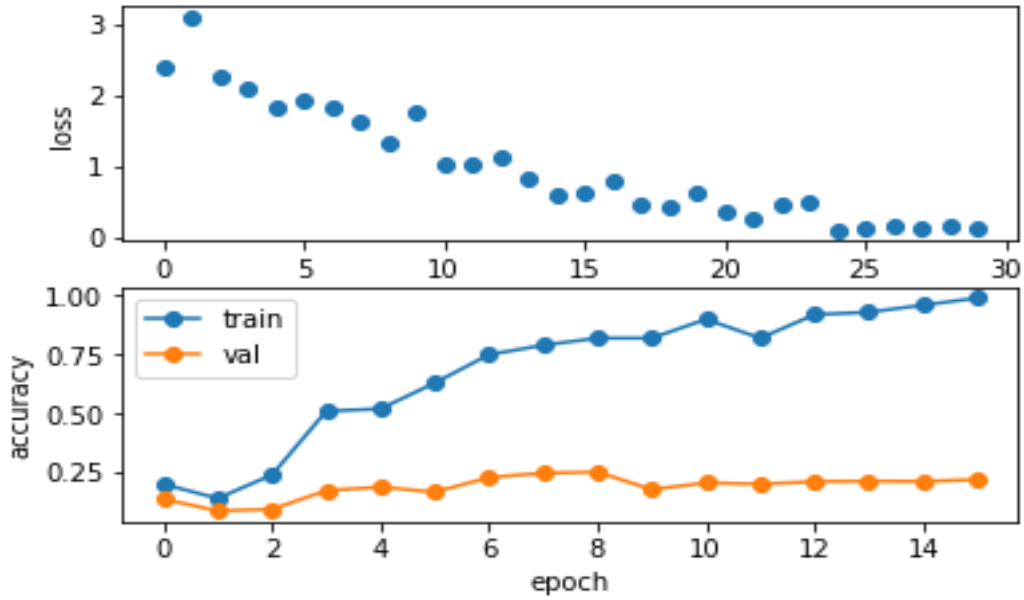


Fig. 58: The visualization of overfitting in ThreeLayerConvNet

Then, let's look the following model:

```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

Training CIFAR-10 dataset over 1 epochs gives: %50 training and validation accuracy. So, let's look at the trained filters:

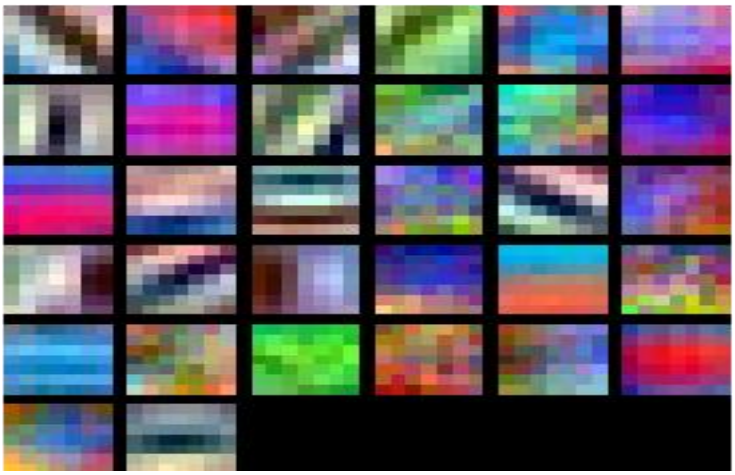


Fig. 59: The visualization of trained model's first layer's filter

We see that the they are representation of the inputs.

Spatial Batch Normalization

Batch normalization is useful method to prevent overfitting and accelerate the training phase of the model. In general, before normalizing the inputs, they can vary in continuous range means that it can be large. Then, if the model's weights are large, weighted summation gives large outputs to be feed the activation. This result in big gradients and big updates in stochastic gradient descent. Since the large updates in the configurable parameters, the learning curve starts oscillations and have trouble finding the global minima. Therefore, we need to prevent the learning phase of the model from the large oscillations caused by large inputs.

Batch normalization comes into play to avoid mentioned problems by normalizing the inputs. Let $h_i(x) = g(a_i(x))$ where g is the activation, $h_i(x)$ is the result of i^{th} neurons firing. We know that $h_i(x)$ will be used as a input in next layer. In order to bring all i's (all activations) in the same scale, we need to normalize them by:

$$h_{ij}(x)^{norm} = \frac{h_{ij} - \mu_{ij}}{\sigma_{ij}}$$

Then we need to arrange gamma γ $and\ beta\ \beta$ to maintain the representative power of the hidden layers. Therefore, after normalizing the input by mentioned equation we need to implement one more step that is:

$$h_{ij}(x)^{final} = \gamma_{ij} * h_{ij}(x)^{norm} + \beta_j$$

Note that γ and β are new hyperparamaters of the network that should be learned together.

In this special case of batch normalization, spatial batch normalization, we have higher dimensional data to be normalized. But the main logic is same. The mathematical expression described the spatial batch normalization:

$$h_{ij}(x)^{final}{}_{spatial} = \frac{\gamma_{ij} * (h_{ij} - \mu_{ij})}{\sigma_{ij} + \beta_j}$$

It is actually the same expect for μ_{ij} σ_{ij}, γ_{ij}, β_j has a dimension of 1x C x 1 x1.

Spatial Batch Normalization: forward

Forward propagation of the spatial batch normalization layers can be coded as:

```
def spatial_batchnorm_forward(x, gamma, beta, bn_param):
    """
    Computes the forward pass for spatial batch normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (C,)
    - beta: Shift parameter, of shape (C,)
    - bn_param: Dictionary with the following keys:
      - mode: 'train' or 'test'; required
      - eps: Constant for numeric stability
      - momentum: Constant for running mean / variance. momentum=0 means
that
        old information is discarded completely at every time step, while
        momentum=1 means that new information is never incorporated. The
        default of momentum=0.9 should work well in most situations.
      - running_mean: Array of shape (D,) giving running mean of features
      - running_var Array of shape (D,) giving running variance of featur
es

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None

    N, C, H, W = x.shape

    # Reshape x to N*H*W * C to call batch normalization
    x_new = np.reshape(np.transpose(x, (0, 2, 3, 1)), (-1, C))

    out, cache = batchnorm_forward(x_new, gamma, beta, bn_param)

    # Reshape out to (N, C, H, W)
    out = np.transpose(np.reshape(out, (N, H, W, C)), (0, 3, 1, 2))

    return out, cache
```

Spatial Batch Normalization: backward

Backprop of spatial batch norm can be coded as:

```
def spatial_batchnorm_backward(dout, cache):
```

```
    """
    Computes the backward pass for spatial batch normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None

    ###########################################################################
######

    N, C, H, W = dout.shape

    # Reshape dout to N*H*W * C to call batch normalization
    dout_new = np.reshape(np.transpose(dout, (0, 2, 3, 1)), (-1, C))

    dx, dgamma, dbeta = batchnorm_backward_alt(dout_new, cache)

    # Reshape dx to (N, C, H, W)
    dx = np.transpose(np.reshape(dx, (N, H, W, C)), (0, 3, 1, 2))

    return dx, dgamma, dbeta
```

Group Normalization

Group normalization is another method to normalize the inputs in neural networks. Let's visualize the group normalization to see difference between others:

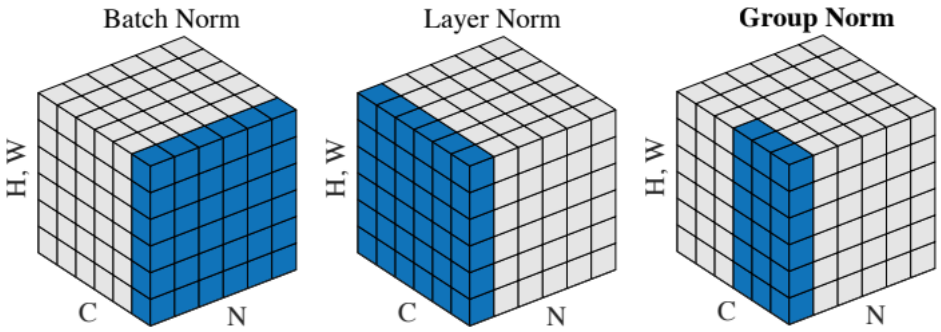


Fig. 60: The visualization of normalization techniques

In this figure, N is the batch size, C is the color channel and H, W and hieght and width respectively. Therefore, we see that group normalization is the partial version of the layer normalization. In layer normalization, we normalize the inputs in color channel axis. The expression for that:

$$h_{ij}(x)^{final}{}_{layer} = \frac{\gamma_{ij} * (h_{ij} - \mu_{ij})}{\sigma_{ij} + \beta_j}$$

Where μ_{ij} σ_{ij}, has a dimension of N x1 x 1 x1. γ_{ij}, β_j has a dimension of 1 x 1 x H x W. In group normalization, we split each data point into G groups and per-group per-data normalization is applied.

Group Normalization: forward

The forward propagation can be coded as:

```
def spatial_groupnorm_forward(x, gamma, beta, G, gn_param):
    """
    Computes the forward pass for spatial group normalization.
    In contrast to layer normalization, group normalization splits each e
ntry
    in the data into G contiguous pieces, which it then normalizes indepe
ndently.
    Per feature shifting and scaling are then applied to the data, in a m
anner
    identical to that of batch normalization and layer normalization.

    Inputs:
    - x: Input data of shape (N, C, H, W)
    - gamma: Scale parameter, of shape (C,)
    - beta: Shift parameter, of shape (C,)
    - G: Integer mumber of groups to split into, should be a divisor of C
    - gn_param: Dictionary with the following keys:
      - eps: Constant for numeric stability

    Returns a tuple of:
    - out: Output data, of shape (N, C, H, W)
    - cache: Values needed for the backward pass
    """
    out, cache = None, None
    eps = gn_param.get('eps',1e-5)

    N, C, H, W = x.shape
    x = np.reshape(x, (N*G, C//G*H*W))

    # Transpose x to use batchnorm code
    x = x.T

    # Just copy from batch normalization cdoe
    mu = np.mean(x, axis=0)

    xmu = x - mu
    sq = xmu ** 2
```

```
    var = np.var(x, axis=0)

    sqrtvar = np.sqrt(var + eps)
    ivar = 1./sqrtvar
    xhat = xmu * ivar

    # Transform xhat and reshape
    xhat = np.reshape(xhat.T, (N, C, H, W))
    out = gamma[np.newaxis, :, np.newaxis, np.newaxis] * xhat + beta[np.n
ewaxis, :, np.newaxis, np.newaxis]

    cache = (xhat, gamma, xmu, ivar, sqrtvar, var, eps, G)

    return out, cache
```

Group Normalization: backward

Backprop implementation of the group normalization is given by:

```
def spatial_groupnorm_backward(dout, cache):
    """
    Computes the backward pass for spatial group normalization.

    Inputs:
    - dout: Upstream derivatives, of shape (N, C, H, W)
    - cache: Values from the forward pass

    Returns a tuple of:
    - dx: Gradient with respect to inputs, of shape (N, C, H, W)
    - dgamma: Gradient with respect to scale parameter, of shape (C,)
    - dbeta: Gradient with respect to shift parameter, of shape (C,)
    """
    dx, dgamma, dbeta = None, None, None
    N, C, H, W = dout.shape

    xhat, gamma, xmu, ivar, sqrtvar, var, eps, G = cache

    dxhat = dout * gamma[np.newaxis, :, np.newaxis, np.newaxis]

    # Set keepdims=True to make dbeta and dgamma's shape be (1, C, 1, 1)
    dbeta = np.sum(dout, axis=(0, 2, 3), keepdims=True)
    dgamma = np.sum(dout*xhat, axis=(0, 2, 3), keepdims=True)

    # Reshape and transpose back
    dxhat = np.reshape(dxhat, (N*G, C//G*H*W)).T
    xhat = np.reshape(xhat, (N*G, C//G*H*W)).T
```

```
    Nprime, Dprime = dxhat.shape

    dx = 1.0/Nprime * ivar * (Nprime*dxhat - np.sum(dxhat, axis=0) - xhat
*np.sum(dxhat*xhat, axis=0))

    dx = np.reshape(dx.T, (N, C, H, W))
    return dx, dgamma, dbeta
```

In summary, different normalization techniques can be applied to the input to accelerate training and decrease the computationally cost. They can be summarized as a following visualization:

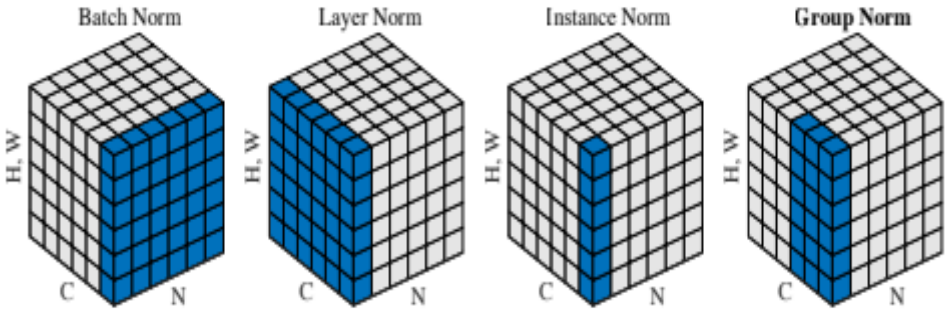


Fig. 61: The visualization of different normalization techniques used in neural network

2.2 PART B

In this part of the question, the question asks pick a single frame work (TensorFlow or PyTorch) and experiment with the given code and discuss the results. To do that, I picked the PyTorch framework to continue this question. I am going to copy and paste the code and explanation snippets, then discuss the results. Note that I also added this part in Appendix II-part B. Let's start with introduction to PyTorch.

What's this PyTorch business?

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you switch over to that notebook).

What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).

- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

PyTorch is Python-based scientific computing package targeted at two sets of audiences [3]:

- A replacement for NumPy to use the power of GPUs [3]
- a deep learning research platform that provides maximum flexibility and speed [3]

Moreover, Pytorch serve lots of built-in function to be used in deep learning area. One of the most powerful feature of the PyTorch is that automatic differentiation package autograd. Thanks to computional graph based differentiation, autograd is super useful in backpropagation parts of the training a neural model.

Then, the other main advantage of PyTorch is that since it uses tensor based computations, we can utilize the power of GPU in matrix computations. The following figure summarizes the advantage of GPU over CPU.

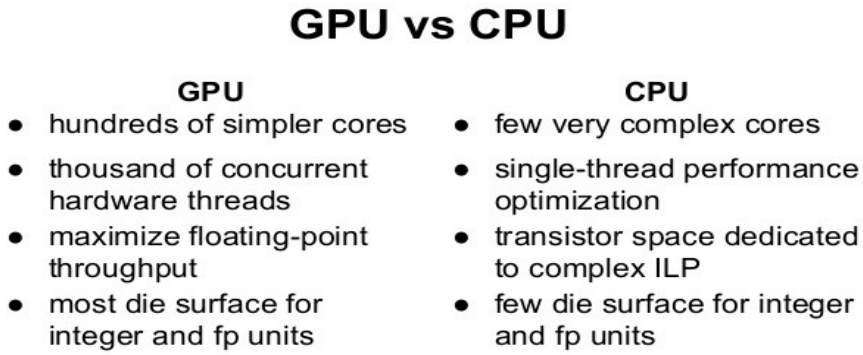


Fig. 62 : The differences between GPU and CPU[4]

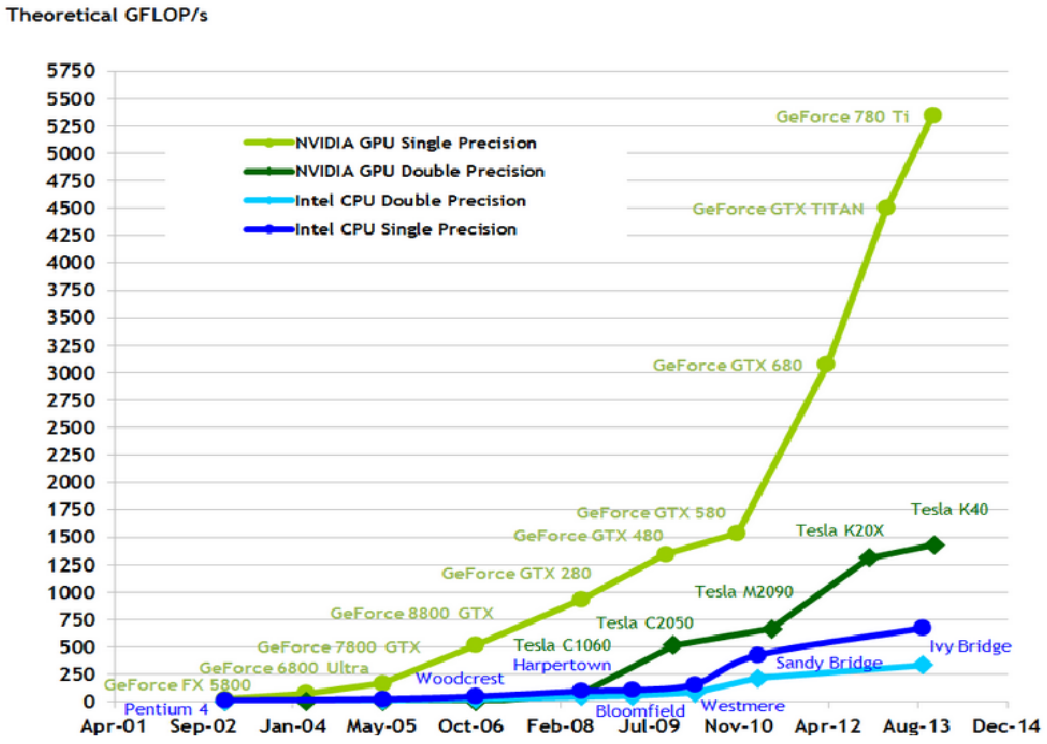


Fig. 62 : The effects on computation when use of GPU and CPU [5]

In PyTorch, if we have a GPU accelerator, we can simply select to computation device like:

```
cpu=torch.device("cpu")
gpu=torch.device("cuda:0") # GPU 0

# Create tensor with CPU
x=torch.ones(3,3, device=cpu)
print("CPU:",x.device)
x=torch.ones(3,3, device=gpu)
print("GPU:",x.device)
x=torch.ones(3,3).cuda(0)
print("CPU to GPU:",x.device)
x=torch.ones(3,3, device=gpu).cpu()
print("GPU to CPU:",x.device)
```

```
CPU: cpu
GPU: cuda:0
CPU to GPU: cuda:0
GPU to CPU: cpu
```

Then, in this question, the table of content is given by:

Table of Contents

This assignment has 5 parts. You will learn PyTorch on different levels of abstractions, which will help you understand it better and prepare you for the final project.

1. Preparation: we will use CIFAR-10 dataset.
2. Barebones PyTorch: we will work directly with the lowest-level PyTorch Tensors.
3. PyTorch Module API: we will use nn.Module to define arbitrary neural network architecture.
4. PyTorch Sequential API: we will use nn.Sequential to define a linear feed-forward network very conveniently.
5. CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparamaters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebones	High	Low
nn.Module	High	Medium
nn.Sequential	Low	High

Part I. Preparation

First, we load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

PyTorch has built-in transformation and dataset loading functions that are very useful since we can apply some kind of transformation such as numpy ndarray to tensor, normalization and so on while loading dataset into our environment. The following code describes the mentioned features:

```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing dat
a
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by t
he
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.
2010))
            ])

# We set up a Dataset object for each split (train / val / test); Dataset
s load
# training examples one at a time, so we wrap each Dataset in a DataLoade
r which
```

```
# iterates through the Dataset and forms minibatches. We divide the CIFAR
-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=Tr
ue,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_T
RAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True
,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRA
IN, 50000)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=Tr
ue,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

Therefore, we uploaded the necessary dataset into `'./cs231n/datasets'` relative path. Then, I check the whether GPU exists or not by:

```
USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100

print('using device:', device)
```

```
using device: cuda
```

Since I am running this notebook on Google Colab, I enable the GPU acceleration for the sake of fastness of computations.

Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the scalar loss at the end.

PyTorch come with an option requires_grad that enables the computational graph so that start tracking the operation on the variable.

PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of data points
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector -- it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

Therefore, we can flatten to necessary input while feeding to the fully-connected denses by following codes:

```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
```

```
    return x.view(N, -
1)  # "flatten" the C * H * W values into a single vector per image

def test_flatten():
    x = torch.arange(12).view(2, 1, 3, 2)
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[  0.,   1.],
          [  2.,   3.],
          [  4.,   5.]]],


        [[[  6.,   7.],
          [  8.,   9.],
          [ 10.,  11.]]]])
After flattening:  tensor([[  0.,   1.,   2.,   3.,   4.,   5.],
        [  6.,   7.,   8.,   9.,  10.,  11.]])
```

The results are given above. Note that view functions operate like a reshape function in NumPy.

Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

Here is the code:

```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have
H units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
```

```
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the n
etwork;
      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification scor
es for
      the input data x.
    """
    # first we flatten the image
    x = flatten(x)  # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Sinc
e w1 and
    # w2 have requires_grad=True, operations involving these Tensors will
 cause
    # PyTorch to build a computational graph, allowing automatic computat
ion of
    # gradients. Since we are no longer implementing the backward pass by
 hand we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
    x = F.relu(x.mm(w1))
    x = x.mm(w2)
    return x


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature
dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

What we do is simply flatten the coming input then implement matmul operation as a mm (we can think like a dot product with little differences). Then we pass the weighted into ReLU activation. It is just that simple because PyTorch will calculate the gradients automatically when we call backward () function. Here is the forward pass example:

```
def two_layer_fc_test():
    hidden_layer_size = 42
```

```
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature
dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

Then, let's look at the how to implement ConvNet. Here is the architecture.

Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

The following code describe the architecture:

```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-
layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of ima
ges
    - params: A list of PyTorch Tensors giving the weights and biases for
 the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving
weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for t
he first
        convolutional layer
```

```
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2)
 giving
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for t
he second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-
connected layer. Can you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-
connected layer. Can you
        figure out what the shape should be?

    Returns:
    - scores: PyTorch Tensor of shape (N, C) giving classification scores
 for x
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None

    conv1 = F.conv2d(x, weight=conv_w1, bias=conv_b1, padding=2)
    relu1 = F.relu(conv1)
    conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, padding=1)
    relu2 = F.relu(conv2)
    relu2_flat = flatten(relu2)
    scores = relu2_flat.mm(fc_w) + fc_b

    return scores
```

We simply use the built-in function in functional package of the PyTorch. In this package, all necessary operations are defined such as conv2d for convolution in 2-D, relu for ReLU activation an so on. Let's test the code by:

```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, i
mage size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel, in_
channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel, in_
channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, b
efore the fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
```

```
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2,
fc_w, fc_b])
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

Therefore, the code works because what we want is to see 64 x10 dimension. Let's look at the random distribution side of the PyTorch.

Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The code for that:

```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means t
hat we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channe
l, kH, kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan
_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=T
rue)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

```
tensor([[-0.0183, -0.0282,  0.5041, -0.6743,  0.1662],
        [-0.3404,  0.4510,  0.3055, -0.3067, -0.4398],
        [-0.5025, -0.4530, -0.2349,  2.3406,  1.0483]], device='cuda:0')
```

Randn is standard normal distribution is always being and we tried the kamier initialization.

When testing the performance of the model, there is no need to compute gradients so that with torch.no_grad() session should de call to performed computations without calculation of gradients. Here is the example:

```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g.
GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples,
 100 * acc))
```

This was the accuracy calculation, first we feed inputs to the model, then get predictions without computation of the gradients.

After that, the training loop can be implemented by the following code:

```
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CIFAR-10.
```

```
    Inputs:
    - model_fn: A Python function that performs the forward pass of the m
odel.
      It should have the signature scores = model_fn(x, params) where x i
s a
      PyTorch Tensor of image data, params is a list of PyTorch Tensors g
iving
      model weights, and scores is a PyTorch Tensor of shape (N, C) givin
g
      scores for the elements in x.
    - params: List of PyTorch Tensors giving weights for the model
    - learning_rate: Python scalar giving the learning rate to use for SG
D

    Returns: Nothing
    """
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)
```

In this part, the training loop is implemented where x is the input and y is the corresponding label for that. We turned x,y to CUDA version to accelerate the process. Then, we need to implement forward pass, note that we already create the model:

```
# Forward pass: compute scores and loss
scores = model_fn(x, params)
```

Forward propagation is done by calling the model, then we need to calculate the loss:

```
loss = F.cross_entropy(scores, y)
```

In the functional package, there are lots of built-in loss functions, in our case cross-entropy is used.

Then, what should we do is to compute gradients. Since PyTorch has an automatic differentiation thanks to computational graph concept, we can compute all gradients just by calling the function backward:

```
loss.backward()
```

Then, last thing we should do is the update parameters. Note that while updating we should call the torch.no_grad() function since we don't want these gradients. Also note that since PyTorch accumulates all gradient over time we need to call grad.zero_() to stop accumulation and start over.

```
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad
```

```
                # Manually zero the gradients after running the backward
pass
                w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

Then, let's get into activation and train ta two-layer network.

BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 40% after training for one epoch.

The code for that is given by:

```
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

Then, let's train the ConvNet using created training loop by following conditions:

BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

The code for training CIFAR-10 in created CNN:

```
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None


conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight((channel_1,))
conv_w2 = random_weight((channel_2, 32, 3, 3))
conv_b2 = zero_weight((channel_2,))
fc_w = random_weight((channel_2*32*32, 10))
fc_b = zero_weight((10,))

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

It is just that simple. This part is over. In the following part, I am going to explain more generic and easy way to create neural network architectures using Module API.

Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

PyTorch comes with torch.nn.Module to enable sub classing to create neural networks in flexible way. One such example is given by:

```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores
```

```
def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64,
feature dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

In this part, we create TwoLayerFC class and inherint the nn.Module class. In the constructor function, we need the create layer type to be called in the forward pass part. Therefore, we call nn.Linear to operate linear weighted sum. Note that when we call the TwoLayerFC network in the training loop, forward is automatically called. In the forward pass, we need to determine the sequence of the operations. In our case, we first flattened the input then feed in first layer followed by ReLu non-linearity. Lastly, we compute linear weighted sum operation with the input coming from previous layer. Then, let's look at the ConvNet implementation.

Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

This code explains the mentioned architecture:

```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padd
ing=2, bias=True)
        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.constant_(self.conv1.bias, 0)

        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, paddi
ng=1, bias=True)
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.fc = nn.Linear(channel_2*32*32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        nn.init.constant_(self.fc.bias, 0)
```

```
    def forward(self, x):
        scores = None


        relu1 = F.relu(self.conv1(x))
        relu2 = F.relu(self.conv2(relu1))
        scores = self.fc(flatten(relu2))
        return scores


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, i
mage size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, nu
m_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

It is the same logic, we implement the following architecture

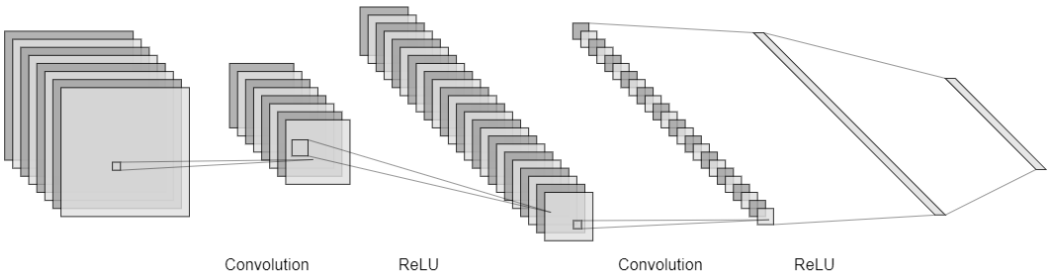


Fig. 63: The visualization of the ConvNet Architecture

Note that it is just representative figure, please do not stick to dimensions of the figure's ext.

Since the checking accuracy is nearly same, I am not going to provide the same thing here. Let's move on training loop and the differences:

```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to
train for
```

```
    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/G
PU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g.
GPU
            y = y.to(device=device, dtype=torch.long)
```

This above part is the same as before.

```
scores = model(x)
loss = F.cross_entropy(scores, y)
```

Again, we implement forward propagation by calling the model and calculate cross-entropy.

```
# Zero out all of the gradients for the variables which the optimizer
            # will update.
optimizer.zero_grad()

# This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.loss.backward()# Ac
tually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()
```

The difference between current one and the before is now we are calling zero_grad() function to stop accumulation of the gradients then call the step() function the update necessary gradients according to the rule described in the optimizer.

Here is the running part:

```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

Then, let's look at the same logic in CNN model.

```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16
```

```
model = None
optimizer = None

model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)


train_part34(model, optimizer)
```

Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

In this part, I am going to explore nn.Sequential() functionalities. In this sequential API, (like in the TensorFlow), we create the layers by sequential order. The example for that is:

```
model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)
```

We simply create model object and describe the necessary operations in sequential order. It is very easy and convenient to implement but there are several drawbacks such as not allowing layer sharing ext.

Let’s train the TwoLayerFC by sequential API:

```
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
```

```
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2
# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

In the same way, training of ConvNet:

```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None


model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, 10),
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

# Weight initialization
# Ref: http://pytorch.org/docs/stable/nn.html#torch.nn.Module.apply
def init_weights(m):
    # print(m)
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        random_weight(m.weight.size())
        zero_weight(m.bias.size())

model.apply(init_weights)

train_part34(model, optimizer)
```

Therefore, you simply determined the order of computation by sequencing the sequential API.

Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

In this part, the custom model for training of CIFAR-10 is created by following code:

```
model = None
optimizer = None

# A 4-layer convolutional network
# (conv -> batchnorm -> relu -> maxpool) * 3 -> fc
layer1 = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=5, padding=2),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.MaxPool2d(2)
)

layer2 = nn.Sequential(
    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.MaxPool2d(2)
)

layer3 = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Dropout(0.25),
    nn.MaxPool2d(2)
)

fc = nn.Sequential(
    nn.Linear(64*4*4, 128),
    nn.ReLU(),
    nn.Linear(128, 10)
)


model = nn.Sequential(
    layer1,
    layer2,
    layer3,
    Flatten(),
    fc
)

learning_rate = 1e-3

optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
# Print training status every epoch: set print_every to a large number
print_every = 10000

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

Note that I changed the code given in the Jupyter Notebook by adding dropout layers and extra fully-connected layers to see results. Finally, my results give 75% accuracy that is a little greather than the notebook's itself.

3 QUESTION 3

In this question, we are going to classify the human activity (downstairs=1, jogging=2, sitting=3, standing=4, upstairs=5, walking=6) from the movement signals measured with three sensors simultaneously. Then, we are going to explore basic recurrent neural networks architecures that is trained by back propagation through time algorithms to solve multi-class time series classfication problem.

3.1 PART A

In this part, I am going to create single hidden layer RNN architecture by assuming the following identities:

- Hidden layer unit $L_{hid} = 128$
- Optimizer = Stochastic gradient descent on mini batches with momentum
- Learning rate $\eta = 10^{-1}$ (I am going to adjust if necessary)
- Momentum coefficient α = 0.85 (Adjustable also)
- Batch size = 32
- Weights/biases initialization = Xavier Uniform

Architecture

I am going build following architectures to experiment with. The results will be discussed in the last part. Moreover, rectified linear unit activation (ReLU) is used whenever needed.

1. Vanilla RNN
   In this achitecture, I am going to create a model with single hidden layer recurrent neural network followed by softmax classifier. As I said, the hidden state vector's dimension is 128.

Hyperparameters	Results
Hidden size number $\boldsymbol{L_{Hid}}$	128

Initialization method	$Uniform[-\frac{6}{\sqrt{L_{pre}+L_{post}}}, \frac{6}{\sqrt{L_{pre}+L_{post}}}]$
Epochs	40
Batch size (Full batch)	32
Learning rate η	10^{-1}
Momentum coefficien $\boldsymbol{\alpha}$	0.85

2. Multi Layer RNN
   In this achitecture, I am going to create a model with single hidden layer recurrent neural network followed by hidden dense layer then softmax classifier. Hidden dense layer's dimension L_{Hid_2} will be 64.

3. Three Hidden Layer RNN

   In this achitecture, I am going to create a model with single hidden layer recurrent neural network followed by two hidden dense layers then softmax classifier. Hidden dimensions are $L_{Hid_2} = 64 \; and \; L_{Hid_3} = 32$. The representative schematic is given below:

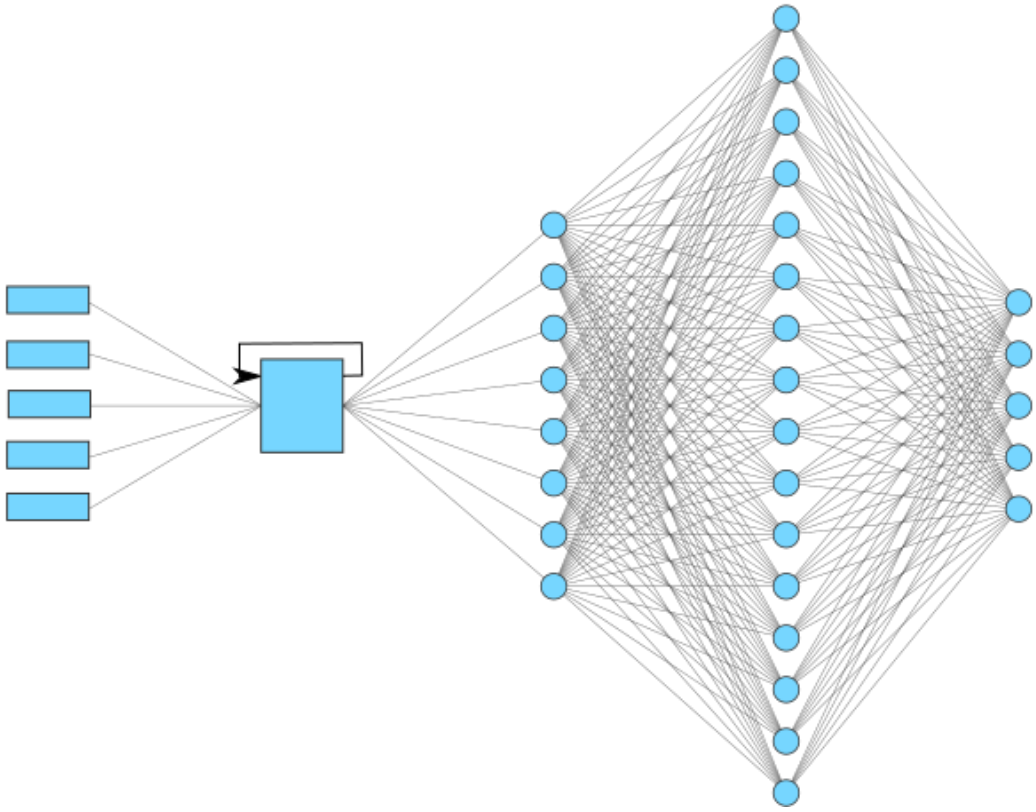


Fig. 64: The visualization of the RNN Architecture

Note that it is just representative schematic that is not reflecting our hidden size, ext. The first is input layer, in our case we have 3000 number of recording of human activity each has 150 time units and lastly datas are collected from three sensors simultaneously. Therefore, we have sequence length of 150 and input size 3.

4. Five hidden Layer RNN
   Fianlly, I am going to create a model with single hidden layer recurrent neural network followed by 4 hidden dense layer then softmax classifier. The effects/results will be discussed in final part of the question.

The code given below is belong to single hidden layer RNN model:

```
class RNN(object):
    """
    Recurrent Neural Network for classifying human activity.
    RNN encapsulates all necessary logic for training the network.

    """
    def __init__(self,input_dim = 3,hidden_dim = 128,
seq_len = 150, learning_rate = 1e-1, mom_coeff = 0.85, batch_size = 32,
 output_class = 6):

        """

        Initialization of weights/biases and other configurable parameter
s.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim_inp2hid = Xavier(self.input_dim,self.hidden_dim)
        self.W1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(self.input_dim,self.hidden_dim))
        self.B1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.hidden_dim))

        lim_hid2hid = Xavier(self.hidden_dim,self.hidden_dim)
        self.W1_rec= np.random.uniform(-
lim_hid2hid,lim_hid2hid,(self.hidden_dim,self.hidden_dim))

        lim_hid2out = Xavier(self.hidden_dim,self.output_class)
```

```
        self.W2 = np.random.uniform(-
lim_hid2out,lim_hid2out,(self.hidden_dim,self.output_class))
        self.B2 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.output_class))
```

Forward pass

In the forward propagation, we feed input to the recurrent network and go along the time and keep states of the activations, the matmematical expressions for that:

$$L_{hidden} = \sum_{i=1}^{N}({W_1}^{(i)} * {X_1}_{(t)}^{(i)} + {W_1}_{recurrent}^{(i)} * {O_{hidden}}_{(t-1)}^{(i)}) + B_1$$

$$O_{hidden} = \tanh(L_{hidden})$$

$$L_{output} = \sum_{i=1}^{N}({W_2}^{(i)} * {O_{hidden}}_{(t)}^{(i)}) + B_2$$

$$O_{output} = \text{softmax}(L_{output})$$

Note that just the recurrent part and output part is considered just for the sake of simplicity of the equations and the remaining part is just feed forward dense layers. The hyperbolic tangent and Softmax classifier's equations are given by:

$$\tanh(X) = \frac{e^{X} - e^{-X}}{e^{X} + e^{-X}} \text{ and soft-max}(Z_i) = \frac{e^{Z_i}}{\sum_j e^{Z_j}}$$

Therefore, we can visualize this operation as a:

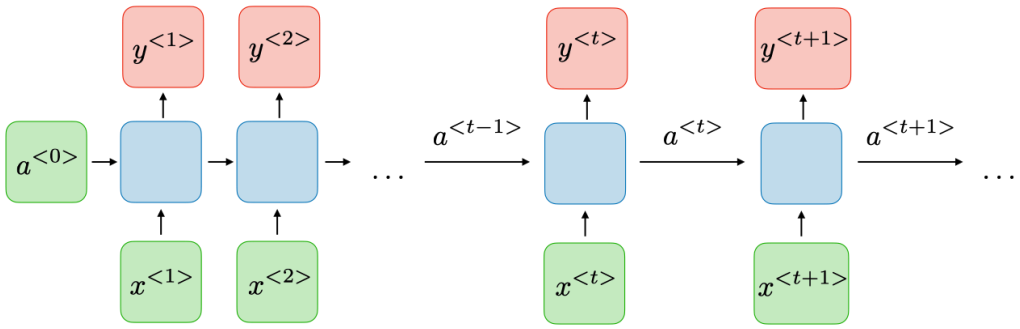


Fig. 64: The visualization of the RNN forwards pass [4]

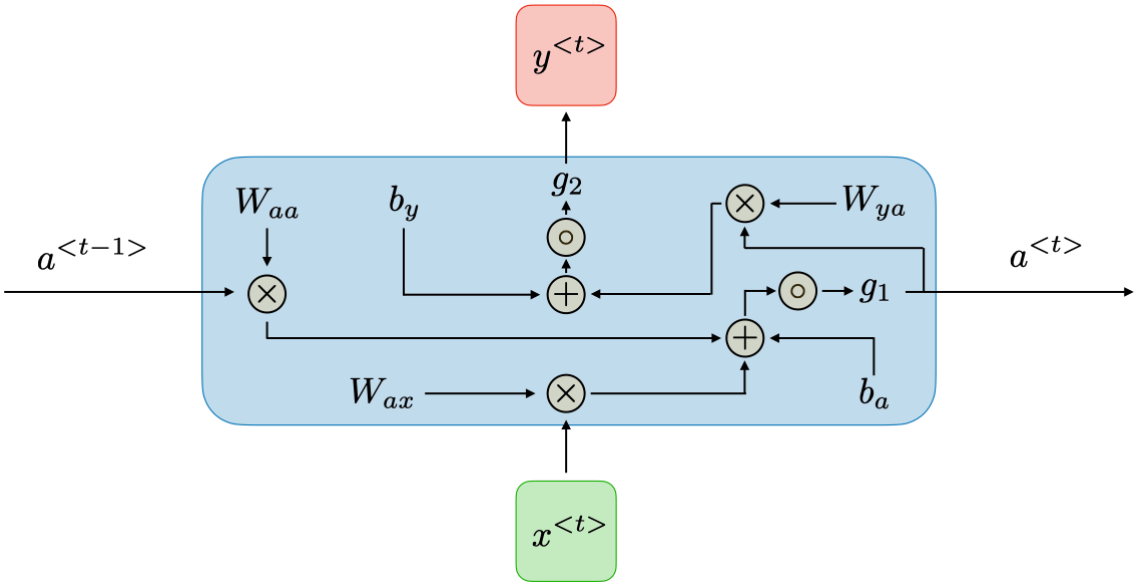


Fig. 65: The visualization of the RNN cell [4]

In the figure, $a^{<t-1>}$ *is the previous hidden state*, $x^{<t>}$ is the input at time t. W_{aa} , W_{ya} and W_{ax} are the weights connected hidden to hidden, hidden to output, input to hidden, respectively. Then, b_a and b_y are the biased of hidden layer and output layer respectively. Note that since the weights/biases are shared along the time, there is no need to add time subscript. After that, in our case, we are going to predict human activity after passing through the all-time points so that the type of RNN is many-to-one case. The following figure expressed the many-to-one approach:

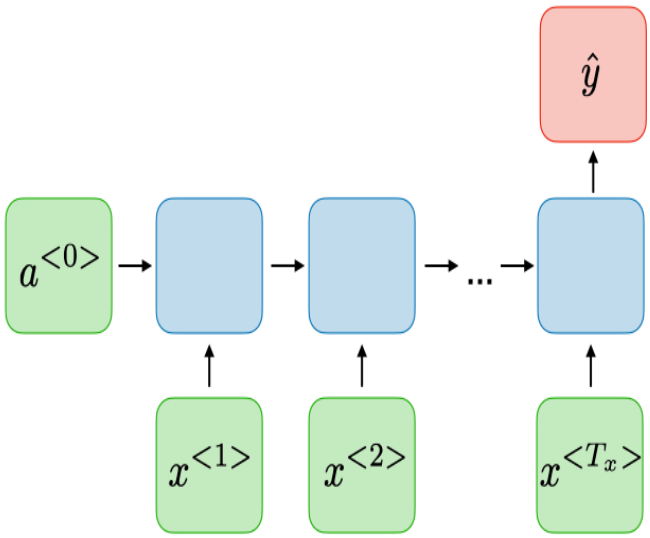


Fig. 66: The visualization of the many-to-one RNN cell [4]

Therefore, we are going to predict after the all-time steps. The Python code for forward pass of this architecture is given by:

```
    def forward(self,X) -> tuple:
        """
        Forward propagation of the RNN through time.

        Inputs:
        --- X is the bacth.
        --- h_prev_state is the previous state of the hidden layer.

        Returns:
        --- (X_state,hidden_state,probs) as a tuple.
        ------ 1) X_state is the input across all time steps
        ------ 2) hidden_state is the hidden stages across time
        -----
- 3) probs is the probabilities of each outputs, i.e. outputs of softm
ax
               """
        X_state = dict()
        hidden_state = dict()
        output_state = dict()
        probs = dict()


        self.h_prev_state = np.zeros((1,self.hidden_dim))
        hidden_state[-1] = np.copy(self.h_prev_state)

        # Loop over time T = 150 :
        for t in range(self.seq_len):

            # Selecting first record with 3 inputs, dimension = (batch
_size,input_size)
            X_state[t] = X[:,t]

            # Recurrent hidden layer :
            hidden_state[t] = np.tanh(np.dot(X_state[t],self.W1) + np.
dot(hidden_state[t-1],self.W1_rec) + self.B1)
            output_state[t] = np.dot(hidden_state[t],self.W2) + self.B
2

            # Per class probabilites : (Note that this computation and
the output state computation can be implemented outside of the loop
since we are interesting in many-to-one type of the problem.
            probs[t] = activations.softmax(output_state[t])

        return (X_state,hidden_state,probs)
```

Categorical Cross Entropy

In this question, we are going to use categorical cross entropy and the equation is given by:

$$\text{Categorical Cross-Entropy Loss } Ł(O_{output_i}, Y_i) = \sum_{i=1}^{N} Y_i * \log(O_{output_i})$$

The code for cross-entropy is:

```
def CategoricalCrossEntropy(self,labels,preds):
    """
    Computes cross entropy between labels and model's predictions
    """
    predictions = np.clip(preds, 1e-12, 1. - 1e-12)
    N = predictions.shape[0]
    return -np.sum(labels * np.log(predictions + 1e-9)) / N
```

Backpropagation through time

Since we are interested in training recurrent neural networks that has a memory (i.e., hidden state vector), we need to go backward along the time. Then, let the T be the sequence length, in our case T = 150 that means we need to backward 150 time steps. The following algorithm explain the BPTT:

For t from T down to 1:

$\nabla L_{output} = (O_{output} - Y)$

$\nabla B2 \mathrel{+}= \sum_m \nabla L_{output}$

$\nabla W2 \mathrel{+}= \nabla L_{output} * O^T{}_{output}$

$\nabla O_{hidden} \mathrel{+}= (W_2)^T * \nabla L_{output}$

$\nabla L_{hidden} = \frac{\partial tanh}{\partial L_{hidden}}(L_{hidden}) * \nabla O_{hidden}$

$\nabla W1 \mathrel{+}= \nabla L_{hidden} * X^T$

$\nabla B2 \mathrel{+}= \sum_m \nabla L_{hidden}$

$\nabla W1_{recurrent} \mathrel{+}= \nabla L_{hidden} * O^T{}_{output}{}^{(t-1)}$

$\nabla O_{hidden}{}^{(t-1)} = W_{1_{recurrent}}{}^T * \nabla L_{hidden}$

end

Therefore, it is very similar to just feed-forward models when we unfoll the parameters in time. The visualization of what I did is:

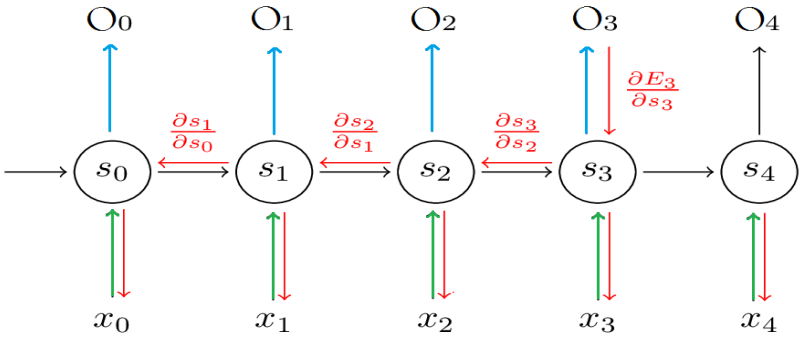


Fig. 67: The visualization of backprop through time

Then, the next issue is gradient clipping. Since we are accumulating each time steps while BPTT, large gradients grow exponentially that results in exploiding gradient problem while small gradients shrink exponentially that results in vanishing pradient problem. To overcome this situation, we need to clip gradient with some value C so that when gradients passes to value of C (both in positive and negative direction), we clip that gradients and equate them to value C. The figure explains the logic:

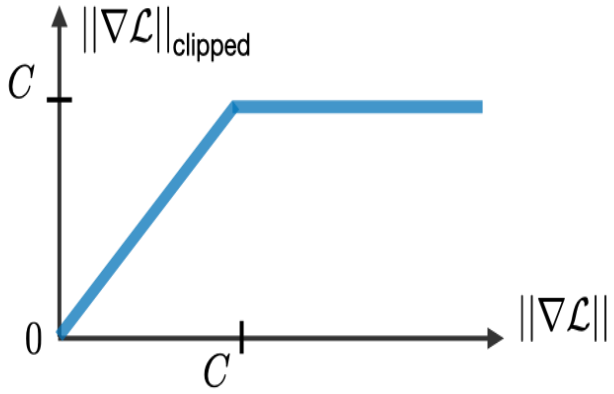


Fig. 68: Gradient clipping logic

The code for that is given by:

```
 def BPTT(self,cache,Y):
        """

        Back propagation through time algorihm.
        Inputs:
        -- Cache = (X_state,hidden_state,probs)
        -- Y = desired output

        Returns:
        -- Gradients w.r.t. all configurable elements
        """

        X_state,hidden_state,probs = cache

        # backward pass: compute gradients going backwards
        dW1, dW1_rec, dW2 = np.zeros_like(self.W1), np.zeros_like(self.W1
_rec), np.zeros_like(self.W2)

        dB1, dB2 = np.zeros_like(self.B1), np.zeros_like(self.B2)

        dhnext = np.zeros_like(hidden_state[0])

        dy = np.copy(probs[149])
        dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1

        dB2 = np.sum(dy,axis = 0, keepdims = True)
```

```
        dW2 = np.dot(hidden_state[149].T,dy)


        for t in reversed(range(1,self.seq_len)):

            dh = np.dot(dy,self.W2.T) + dhnext

            dhrec = (1 - (hidden_state[t] * hidden_state[t])) * dh

            dB1 += np.sum(dhrec,axis = 0, keepdims = True)

            dW1 += np.dot(X_state[t].T,dhrec)

            dW1_rec += np.dot(hidden_state[t-1].T,dhrec)

            dhnext = np.dot(dhrec,self.W1_rec.T)


        for grad in [dW1,dB1,dW1_rec,dW2,dB2]:
            np.clip(grad, -10, 10, out = grad)


        return [dW1,dB1,dW1_rec,dW2,dB2]
```

Stochastic Gradient Descent on mini-batches

The expression for SGD on mini batches:

$$\Delta {W_2}^N = -\eta * \frac{\partial Error}{\partial W_2} + \alpha * \Delta {W_2}^{N-1}$$

$$W_2 \mathrel{+}= \Delta {W_2}^N$$

$$\Delta {B_2}^N = -\eta * \frac{\partial Error}{\partial B_2} + \alpha * \Delta {B_2}^{N-1}$$

$$B_2 \mathrel{+}= \Delta {B_2}^N$$

$$\Delta {W_1}^N{}_{recurrence} = -\eta * \frac{\partial Error}{\partial W_{1_{recurrence}}} + \alpha * \Delta {W_1}^N{}_{recurrence}$$

$$W_{1_{recurrence}} \mathrel{+}= \Delta {W_1}^N{}_{recurrence}$$

$$\Delta {B_1}^N = -\eta * \frac{\partial Error}{\partial B_1} + \alpha * \Delta {B_1}^{N-1}$$

$$B_1 \mathrel{+}= \eta * \frac{\partial Error}{\partial B_1}$$

$$\Delta {W_1}^N = -\eta * \frac{\partial Error}{\partial W_1} + \alpha * \Delta {W_1}^{N-1}$$

$$W_1 \mathrel{+}= \Delta {W_1}^N$$

All are same with previous assignments, therefore there is no need to repeat myself. The code for SGD:

```
def step(self,grads,momentum = True):
        """
        SGD on mini batches
        """

        if momentum:

            delta_W1 = -
self.learning_rate * grads[0] +  self.mom_coeff * self.prev_updates['W1']
            delta_B1 = -
self.learning_rate * grads[1] +  self.mom_coeff * self.prev_updates['B1']

            delta_W1_rec = -
self.learning_rate * grads[2] +  self.mom_coeff * self.prev_updates['W1_r
ec']
            delta_W2 = -
self.learning_rate * grads[3] +  self.mom_coeff * self.prev_updates['W2']

            delta_B2 = -
self.learning_rate * grads[4] +  self.mom_coeff * self.prev_updates['B2']


            self.W1 += delta_W1
            self.W1_rec += delta_W1_rec
            self.W2 += delta_W2
            self.B1 += delta_B1
            self.B2 += delta_B2

            self.prev_updates['W1'] = delta_W1
            self.prev_updates['W1_rec'] = delta_W1_rec
            self.prev_updates['W2'] = delta_W2
            self.prev_updates['B1'] = delta_B1
            self.prev_updates['B2'] = delta_B2
```

Model fitting

Finally, the training loop is implemented via:

```
   def fit(self,X,Y,X_val,y_val,epochs = 50 ,verbose = True, earlystoppin
g = False):
        """
        Given the traning dataset, their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """
```

```
        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)

            for i in range(round(X.shape[0]/self.batch_size)):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]

                cache_train = self.forward(X_feed)

                grads = self.BPTT(cache_train,y_feed)
                self.step(grads)

                # Here is the early stopping based on both accuracy and
categorical cross entropy loss, the inputs to that function are both ce
loss and accuracy, therefore, just the comparison part is done in this
function. The selection of the 10% training samples should be done in the
above. However, the logic is simple and straightforward.
                if earlystopping:
                    if self.earlyStopping(ce_train = cross_loss_train,c
e_val = cross_loss_val,ce_threshold = 3.0,acc_train = acc_train,acc_va
l = acc_val,acc_threshold = 15):
                        break
```

Furthermore, I implemented early stopping algorithm based on both categorical cross entropy and the accuracy. The code for that:

```
def earlyStopping(self,ce_train,ce_val,ce_threshold,acc_train,acc_val,acc_thresho
ld):
        if ce_train - ce_val < ce_threshold or acc_train - acc_val > acc_threshol
d:
            return True
        else:
            return False
```

Therefore, if categorical cross entropy of the training is not going stable with the validation, I stopped the learning. Also, I implemented these for the accuracy metric. Note that the inputs to the early stopping function is 10% of the randomly selected training data's and the testing data's results on both cross entropy and the accuracy.

Results/Discuss

In this part, I am going to evaluate my implementation by using the cross-entropy error on training and validation data, accuracy score and confusion matrix of the training/testing elements. Furthermore, I am going discuss and compare the effects of number of hidden layer.

Vanilla RNN

This is the simplest architecure I created but the results are pretty good in both training and testing phase. The results and discuss are given below:

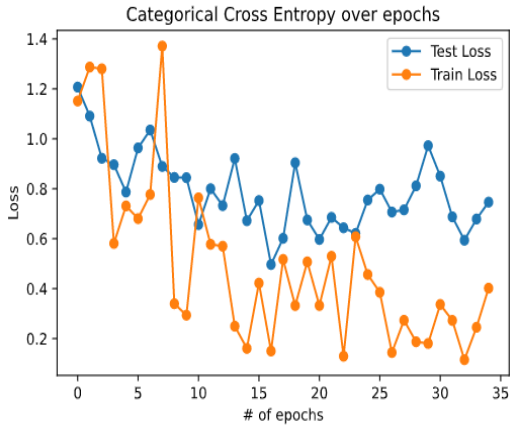


Fig. 69: Cross-entropy error over training in Vanilla RNN

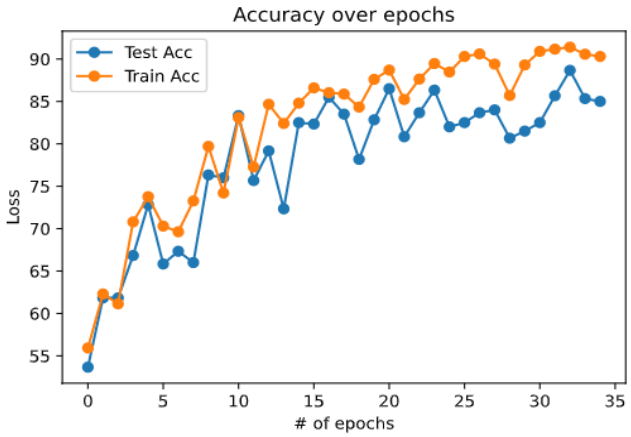


Fig. 70: Accuracy over training in Vanilla RNN

From the cross-entropy and accuracy plottings, we see that Vanilla RNN model performed well on both training and testing data. The accuracy reached 90% in training and 85% in testing that is pretty good for this simple architecture. Let's have a look at the confusion matrix to assess performance in deeper way.

	downstairs	jogging	sitting	standing	upstairs	walking
downstairs	356	16	0	1	119	8
jogging	3	488	0	0	5	4
sitting	2	1	468	27	2	0
standing	2	1	6	476	13	2

	downstairs	jogging	sitting	standing	upstairs	walking
upstairs	16	25	2	0	453	4
walking	10	3	0	0	19	468

Here is the confusion matrix. I labeled the axis so that it can we easily understood. Note that rows represent actual classes and columns represents the predictions. We see that our model is predictions are sensible, especially for jogging, standing and sitting activities. To be more concrete, I analyze the precision, recall and F-1 score for our training predictions that will be dissucced below.

	downstairs	jogging	sitting	standing	upstairs	walking
downstairs	71	1	0	0	26	2
jogging	5	92	0	0	2	1
sitting	3	0	94	1	2	0
standing	0	0	2	73	24	1
upstairs	9	0	0	0	88	3
walking	6	1	0	0	1	92

Here is the confusion matrix for testing Vanilla RNN. As expected, our Vanilla RNN predictions also sensible since we reached 85% accuracy. Furthermore, we predicted the jogging,sitting and upstairs classes very well that shows that our model is generalized.

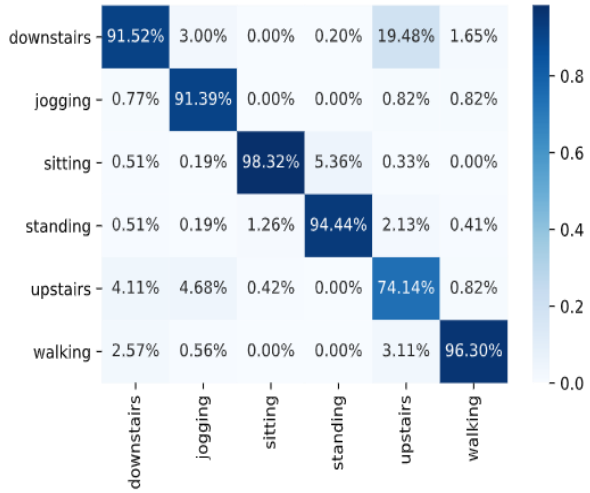


Fig. 71: Confusion matrix visualization for training Vanilla RNN

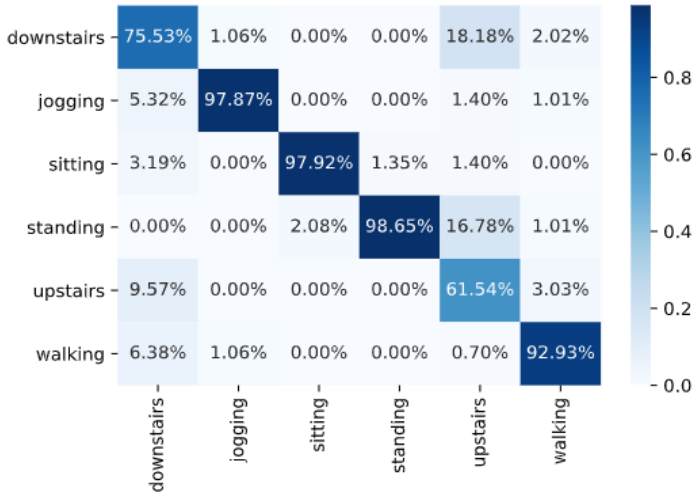


Fig. 72: Confusion matrix visualization for testing Vanilla RNN

	precision	recall	f1-score	support
0	0.92	0.71	0.80	500
1	0.91	0.98	0.94	500
2	0.98	0.94	0.96	500
3	0.94	0.95	0.95	500
4	0.74	0.91	0.82	500
5	0.96	0.94	0.95	500
accuracy			0.90	3000
macro avg	0.91	0.90	0.90	3000
weighted avg	0.91	0.90	0.90	3000

Fig. 73: Classification report for training Vanilla RNN

	precision	recall	f1-score	support
0	0.76	0.71	0.73	100
1	0.98	0.92	0.95	100
2	0.98	0.94	0.96	100
3	0.99	0.73	0.84	100
4	0.62	0.88	0.72	100
5	0.93	0.92	0.92	100
accuracy			0.85	600
macro avg	0.87	0.85	0.85	600
weighted avg	0.87	0.85	0.85	600

Fig. 74: Classification report for testing Vanilla RNN

I also printed the classification report to discuss model's performance. Here are the visualizations of the confusion matrix and classification report. The percentages show that how our predictions are distributed along classes. For example, let's look at the training confusion matrix. In the case of we should predict jogging, we predicted jogging with 91.39% accuracy. Then, let's talk about the precison, recall and F-1 score. Precision for jogging is the number of correctly classified jogging activity out all predicted jogging activity. Wheras the recall for jogging is the number of correctly predicted jogging activities out of the number of actual jogging activity. F-1 score is just the harmonic average of the recall and precision. In this way, we can inference from the confusion matrix. Let's dive into more complex models.

Multi Layer RNN

I gradually increased to hidden layer of the network while decrease the size of hidden layers (i.e., number of neurons) to see results. In this architecture, we have input layer → hidden recurrent layer → hidden dense layer → output layer. Hence, the results are given below.

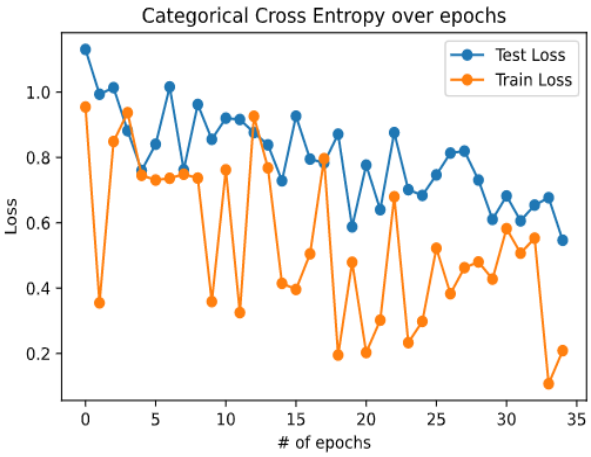


Fig. 75: Cross-entropy error over training in Multi Layer RNN

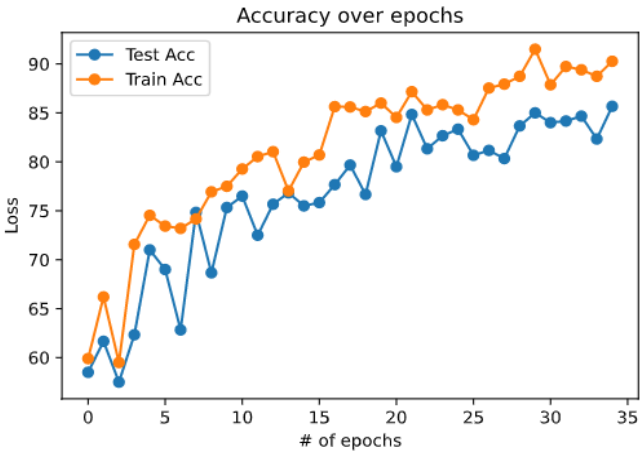


Fig. 76: Accuracy over training in Multi Layer RNN

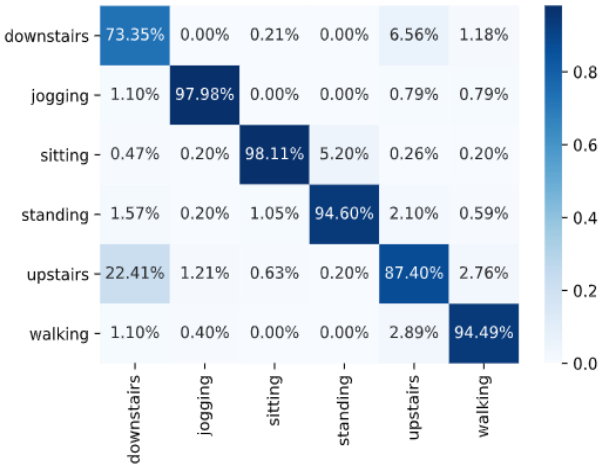


Fig. 77: Confusion matrix visualization for training Multi Layer RNN

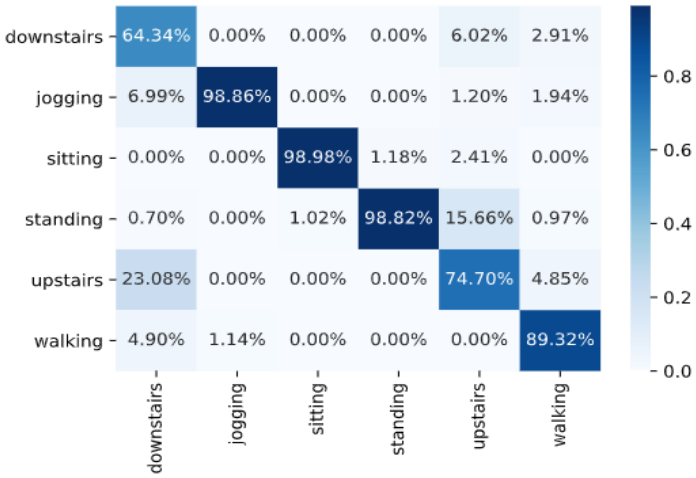


Fig. 78: Confusion matrix visualization for testing Multi Layer RNN

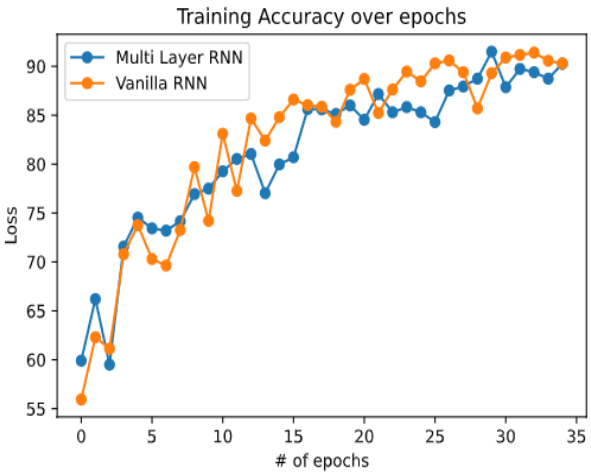


Fig. 79: Comparison between Vanilla RNN and Multi Layer RNN

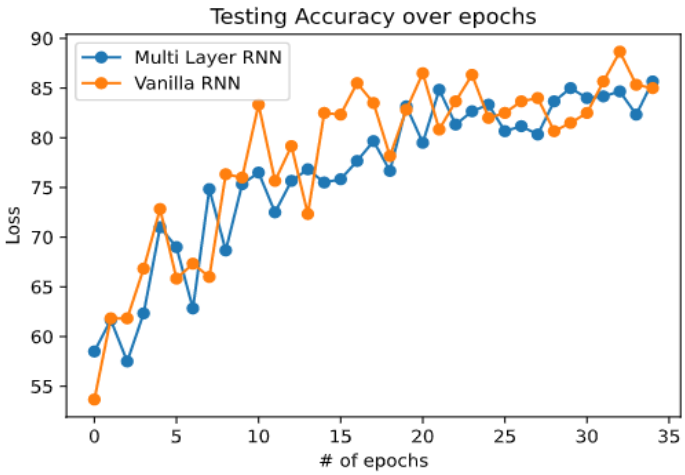


Fig. 80: Comparison between Vanilla RNN and Multi Layer RNN

From the provided figures, we can see that multi-layer recurrent neural network with ReLU activation performed better on testing set while performed similar in the training set. Therefore, the gap between the training and testing in decreased so that our model's variance decreased. This model can be used as a predicter of a human activity in general since result's are heartwarming.

Three hidden layer RNN

In this architecture, we have input layer → hidden recurrent layer → hidden dense layer → hidden dense layer → output layer. Hence, the results are given below.

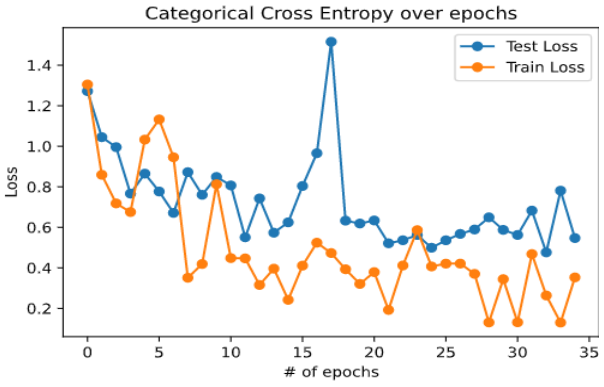


Fig. 81: Cross-entropy error over training in Three hidden layer RNN

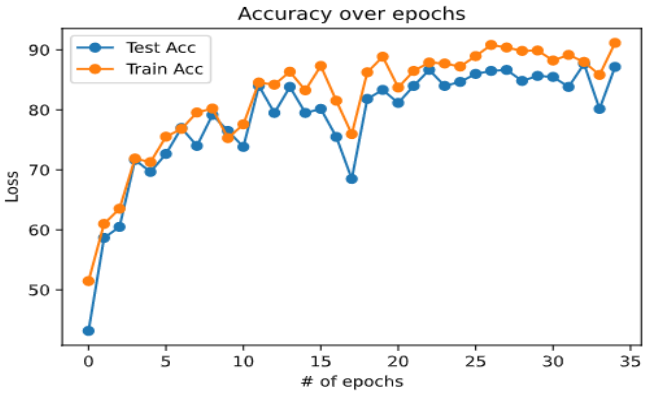


Fig. 82: Accuracy over training in Three hidden layer RNN

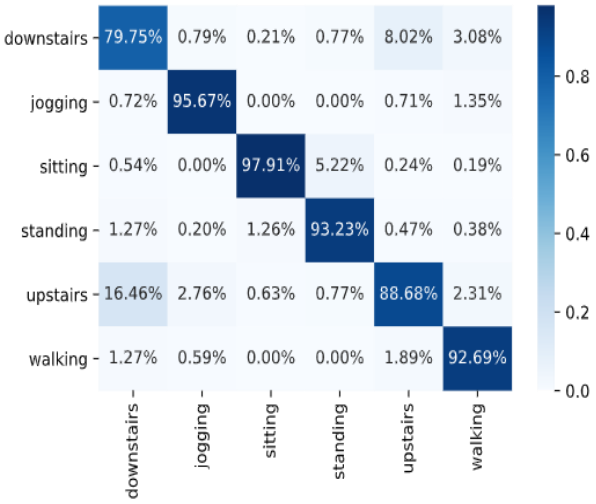


Fig. 83: Confusion matrix visualization for training Three hidden layer RNN

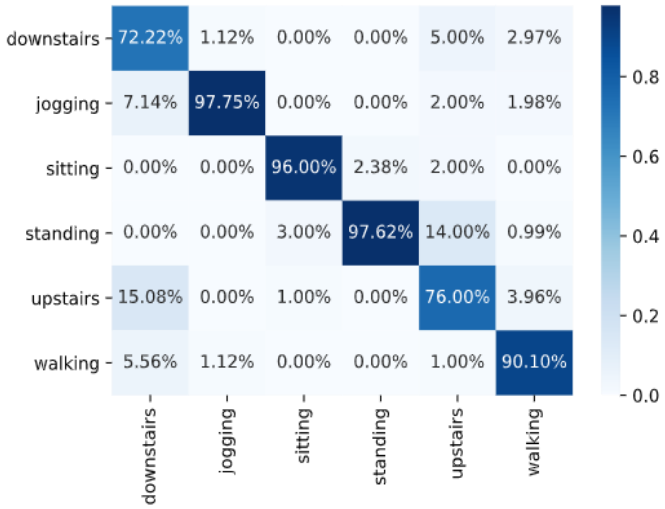


Fig. 84: Confusion matrix visualization for testing Multi Layer RNN

Fig. 85: Comparison between Vanilla RNN, Multi Layer RNN and Three hidden layer RNN

Fig. 86: Comparison between Vanilla RNN, Multi Layer RNN and Three hidden layer RNN

Along the models, the model with 3 hidden layer (including recurrent layer) is performed best on both training and testing. Furthermore, the confusion matrix's results are more balanced, i.e. each class's prediction can be used a predictor for that class. Note that this model can performed better with few more epochs, get 97% training and 94.43% testing accuracy but to not pass the 35 epochs, I stopped training. Anyway, this is best predictor along mentioned models. Let's look at a deeper model.

Five hidden Layer RNN

In this architecture, we have input layer → hidden recurrent layer → hidden dense layer → hidden dense layer → hidden dense layer → hidden dense layer → output layer. Hence, the results are given below.

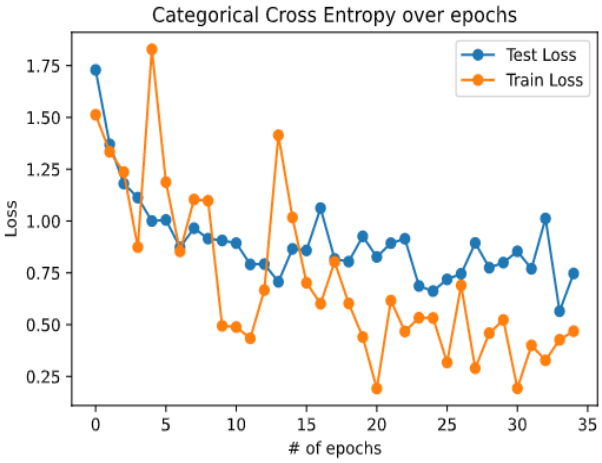


Fig. 83: Cross-entropy error over training in Five hidden layer RNN

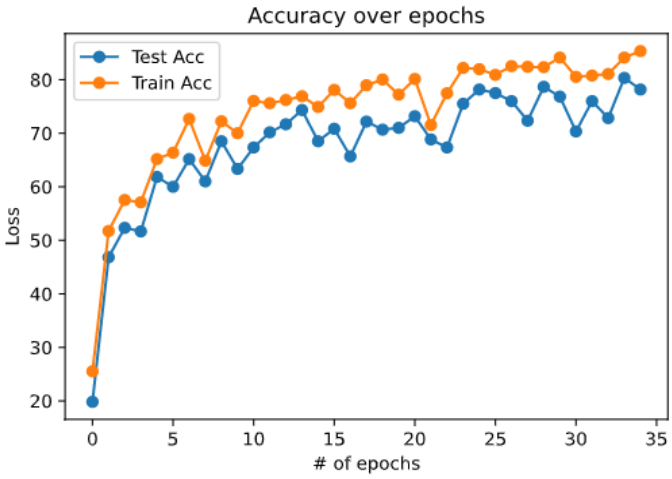


Fig. 84: Accuracy over training in Five hidden layer RNN

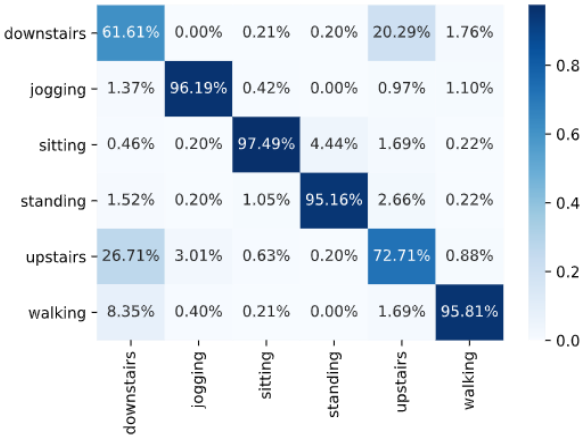


Fig. 85: Confusion matrix visualization for training Five hidden layer RNN

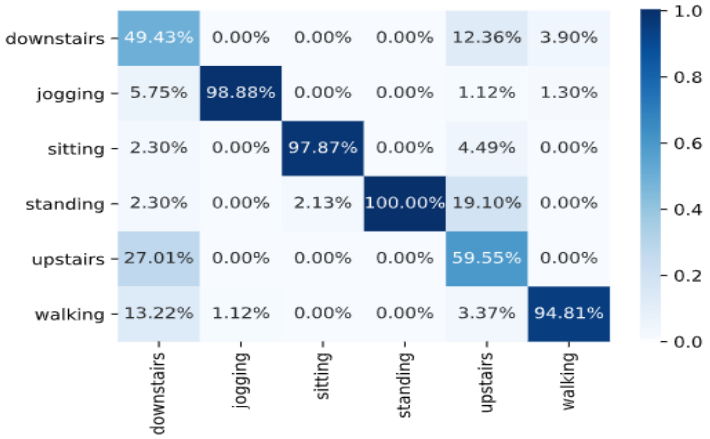


Fig. 86: Confusion matrix visualization for testing Five hidden layer RNN

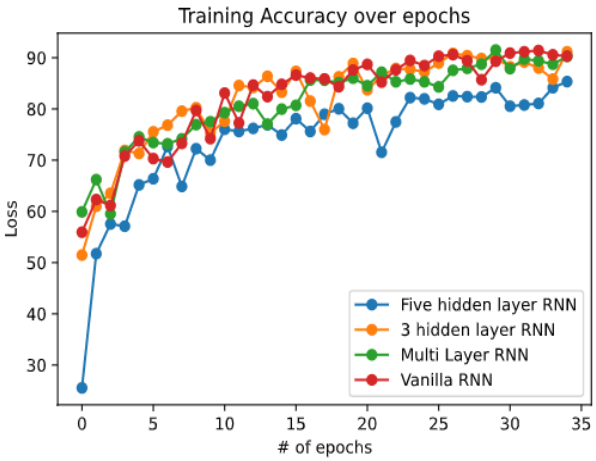


Fig. 87: Comparison between Vanilla RNN, Multi Layer RNN, Three hidden layer RNN and Five hidden layer RNN

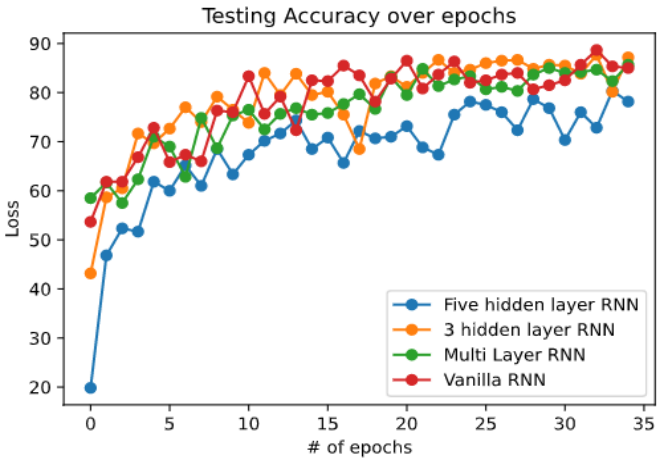


Fig. 87: Comparison between Vanilla RNN, Multi Layer RNN, Three hidden layer RNN and Five hidden layer RNN

In this case, I experiment with 5 hidden layers (including recurrent layer) , we can inference that this model is worst along the other models. There are possibly few reasons, since the model complexity is increased a lot, model find it hard to learn with limited data. Furthermore, let's look at the confusion matrix for testing, we see that our model predictions are unbalanced. We failed to predict the downstairs and upstairs class while we predicted 100% accurately the standing class. Finally, the winner out of 4 models is the model with 3 hidden layer. In a nutshell, the simplest model Vanilla RNN perfomed very well even the size of the model is low, it shows the recurrent layer's power to predict sensible predictions. After that, with the model complexity increases, we start to get better results until a certain point so that for this data, the model with 3 hidden layer is prefable among others. Note that it can be depend on lots of things, we cannot generalize this discussion because with the right hyperparameters we can performe much better than other models but the main goal of the discussion and experiments are trying to keep hyperparamters similar/same and only increased the hidden layer number to see more independant results.

3.2 PART B

In this part of the question, LSTM (Long-Short Term Memory) recurrent neural network will be implemented. Let's start with why LSTM's are much more preferable rather straight RNN models. In RNN models, one is likely to encounter the problem of exploding and vanishing gradients. The reason behind that is to difficulty of capturing long term dependencies. In RNN models, gradients are grows/shrinks exponentially that causes model to stop learning. When the gradients are growing exponentially, the updates terms also growing exponentially so that model cannot converge global minima and oscillations between some points in the learning curve. On the other hand, when gradients shrink exponentially, the updates terms are closer to 0 that results in no/little update in the optimization phase so that model is stopped to learn. LSTM comes into play to handle long-term dependencies and avoiding vanishing/exploding gradient problems.

In this section, I am going to discuss only differences between previous part because many parts are similar.

In LSTM, we have controlling gates and states to determine the behavior of the cell. Due to its broadcast capabilities, LSTM's have ability to remove or add information that is controlled by gates. Then, gates are actually a layers with sigmoidal activation which compresses the input between 0 and 1 and multiplications. The output from sigmoid determines the how much we remove/add information to current cell state. Therefore, 0 means forget completely while 1 means keep completely. Let's dive into gates and states of the LSTM. Furthermore, tanh activation is used to regulate the values between -1 and 1.

In LSTM's we have 3 gates to determine the behavior of the cell state and hidden states which are:

- Forget Gate
- Input Gate
- Output Gate

In the forget gate, we determine how much information we are going to keep/throw away, this decision is made by sigmoid layer. Hence, forget gate takes input and hidden state vector and determine the degree of flow of the information. The mathematical expression for that:

$$\mathrm{F}_t = \sigma\left(W_f \, \Theta \, [\mathrm{h}_{t-1}, x_t] + \, b_f \right)$$

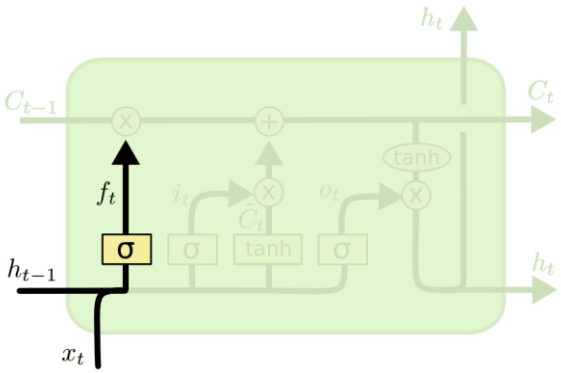


Fig. 88: LSTM cells forget gate

Note that F_t is the forget gate's output, σ is sigmoidal activation, W_f is the weight of the forget gate, $[\mathrm{h}_{t-1}, x_t]$ is the concatenation of the previous hidden state vector and current output and b_f is bias of the forget gate.

Then, we need to determine how much new information should we add. To do that, we utilize the concept of the input gate. This phase has two parts. The combination of the two stages results the output of the input gate so that expressions for that:

$$\mathrm{i}_t = \sigma\left(W_i \, \Theta \, [\mathrm{h}_{t-1}, x_t] + \, b_i \right)$$

$$\tilde{\mathrm{C}}_t = tanh\left(W_c \, \Theta \, [\mathrm{h}_{t-1}, x_t] + \, b_c \right)$$

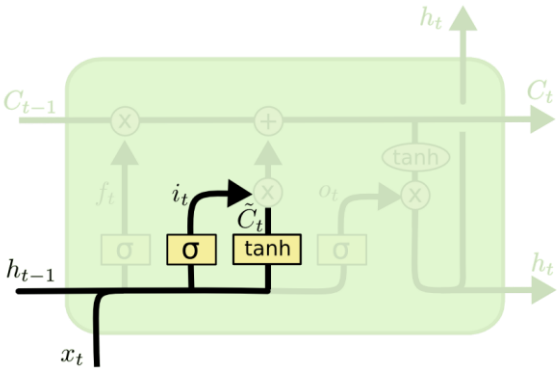


Fig. 89: LSTM cell's input gate

Hence, i_t is the first phase of the inputs gate and determines how much new information should we add, to do that it squeezes the values between 0 and 1 and multiplying by $\tilde{C}_t$. $\tilde{C}_t$ is the new candidate to the cell state. Hence, to determine the new cell state, we multiply forget gate output by previous cell state and add them to input gate's outputs.

$$C_t = F_t * C_{t-1} + i_t + \tilde{C}_t$$

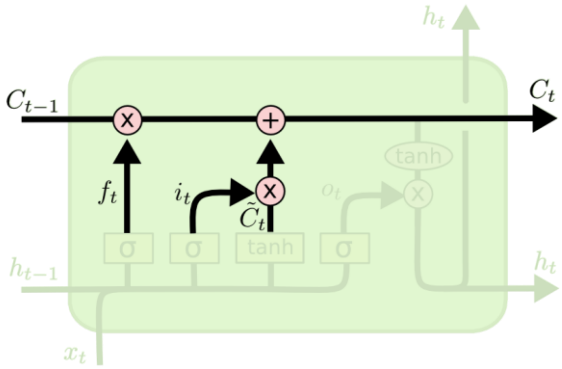


Fig. 90: LSTM cell's cell state

Last step is to calculation of the output. We are going to calculate the output of the LSTM cell based on the current cell state. We are going to determine how much information is needed from the current cell state by scaling it by sigmoidal activation.

$$o_t = \sigma\,(W_o \Theta [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t * \tanh(C_t)$$

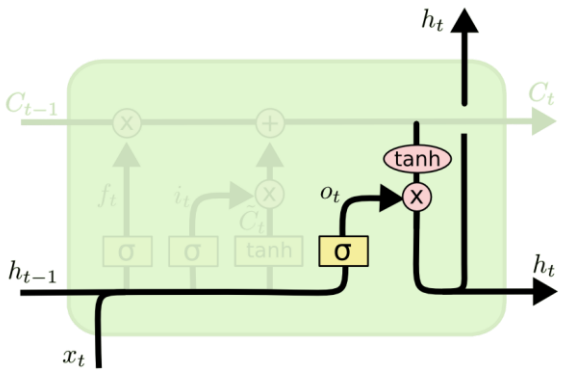


Fig. 91: LSTM cell's output gate

Backprop part of the LSTM is quite similar to basic RNN's after unfolding the time steps. It is just chain rule back warding in the time so that I am not going to give equations for that. Let's dive into results/discussion part.

To optimize LSTM, I implemented different optimizations technique to experiment with. Here is the list of optimization methods I tried:

- Stochastic Gradients Descent
- Stochastic Gradients Descent with Momentum
- AdaGrad
- RMSprop
- Adam (both bias corrected version and vanilla version)

AdaGrad, RMSprop and Adam are adaptive optimization algorithms to minimize the loss function. Since these are not the topic of these paper. I am going to just give expression for that and move on.

AdaGrad

$$\delta_i = \delta_{i-1} + \nabla^2\theta_i$$

$$\theta_i := \theta_{i-1} - \frac{\eta}{\sqrt{\delta_i}+\varepsilon} * \nabla\theta_i$$

RMSprop

$$\delta_i = \alpha * \delta_{i-1} + (1-\alpha) * \nabla^2\theta_i$$

$$\theta_i := \theta_{i-1} - \frac{\eta}{\sqrt{\delta_i}+\varepsilon} * \nabla\theta_i$$

Adam

$$\delta_{M_i} = \beta_1 * \delta_{M_i} + (1-\beta_1) * \nabla\theta_i \,, \quad \delta_{V_i} = \beta_2 * \delta_{V_i} + (1-\beta_2) * \nabla^2\theta_i$$

$$\widetilde{\delta_{M_i}} = \frac{\delta_{M_i}}{1-\beta_1} \,, \quad \widetilde{\delta_{V_i}} = \frac{\delta_{V_i}}{1-\beta_2} \,, \theta_i := \theta_{i-1} - \frac{\eta}{\sqrt{\widetilde{\delta_{V_i}}}+\varepsilon} * \widetilde{\delta_{M_i}}$$

Where δ is the accumulated sum of squares in general and θ is the parameters to be updated.

These are all accumulation gradients with different techniques to keep track of the history of gradients.

Important Notes:

- Various learning rates and other hyperparameters are used to see which one does better jobs and epoch number is limited by 15 epochs since it is expensive to train and takes a lot of time to get full performance. However, as I said, I limited the epoch number so that I am going to compare the results accordingly. Also, note that the results are heartwarming even in 15 epochs.
- All LSTM models perfomed better in 35 epochs when comparing RNN's.
- All LSTM models in 50 epoch reached more than 93% accuracy and more than 88% accuracy.
- Different optimization algorithms are used since training LSTM models are not easy.

Here are results:

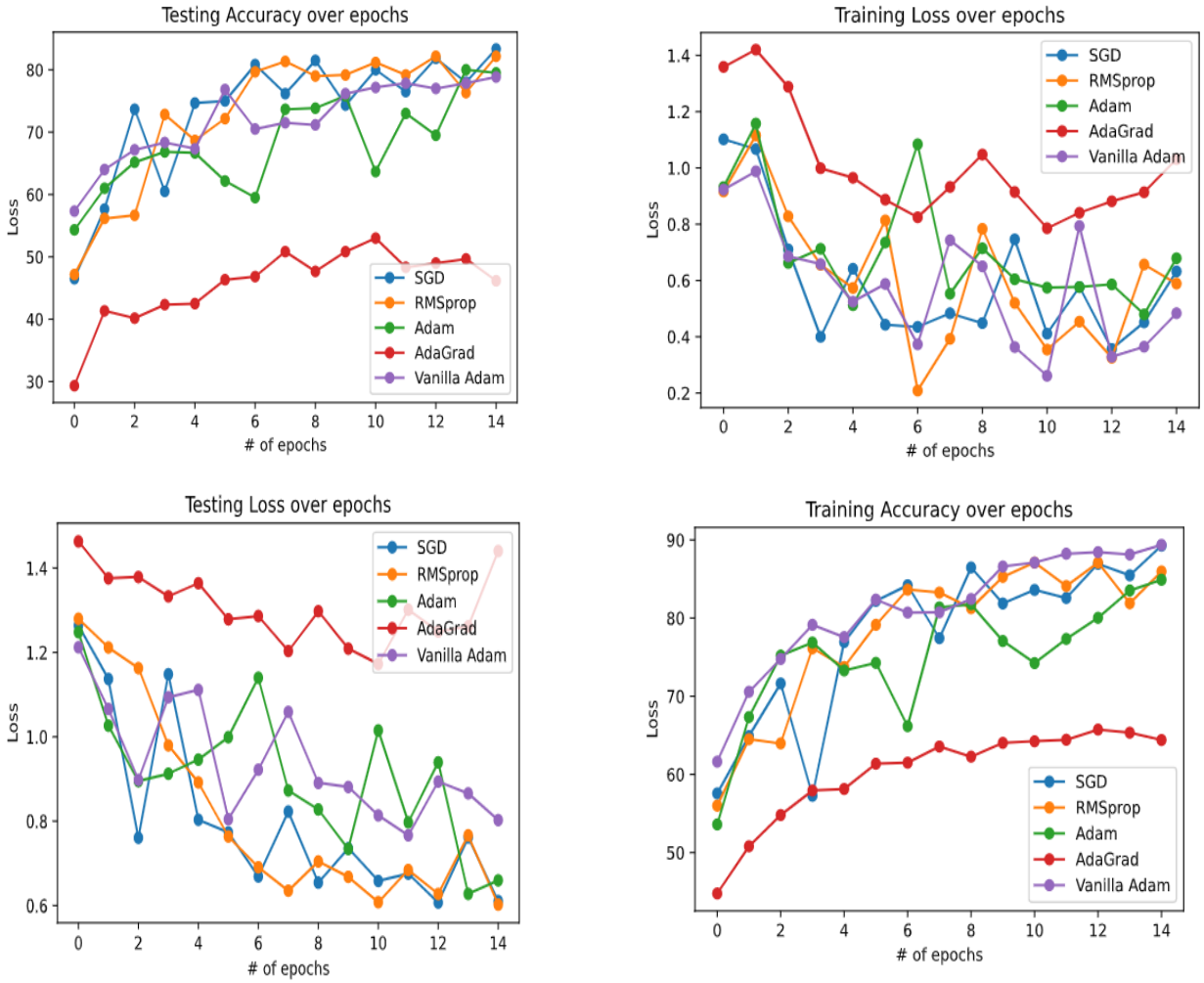


Fig. 92: Comparison plots for LSTM with optimizer SGD, AdaGrad, RMSprop, Adam, Vanilla Adam

Hence, we see that RMSprop, SGD and Vanilla Adam optimization algorithm performed similar.

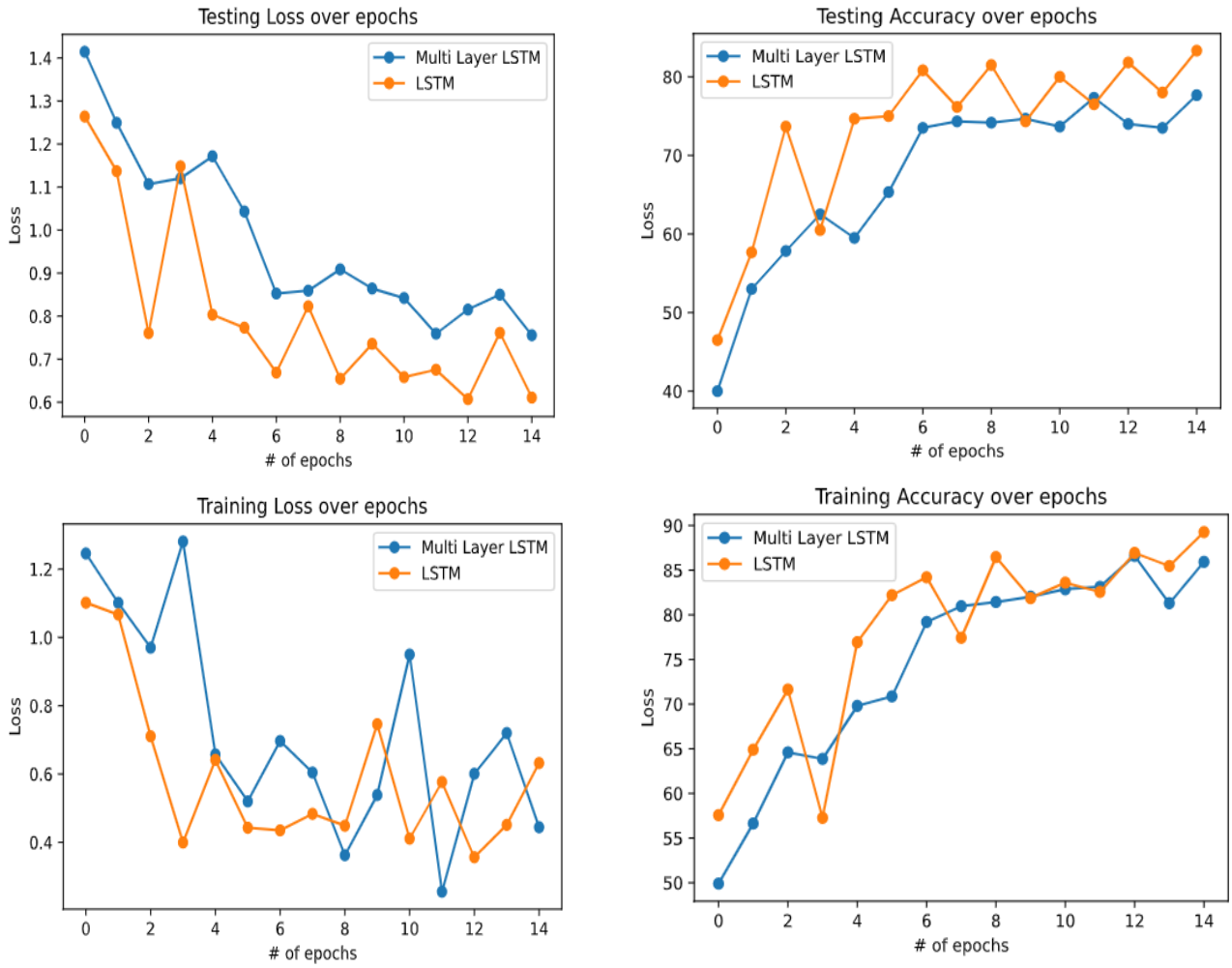


Fig. 93: Comparison plots for Vanilla LSTM and Multi Layer LSTM

Then, I compared the multi-layer LSTM and single layer LSTM. From the figures, we can see that our vanilla LSTM performed better. Multi-layer LSTM get overfitted a bit in 15 epochs since this model is too complicated for our data. However, when we run the mutli-layer LSTM model with 30-40 epochs, the learning curves became similar. Since I limited the epoch number with 15, the single hidden layer LSTM model perfomed better.

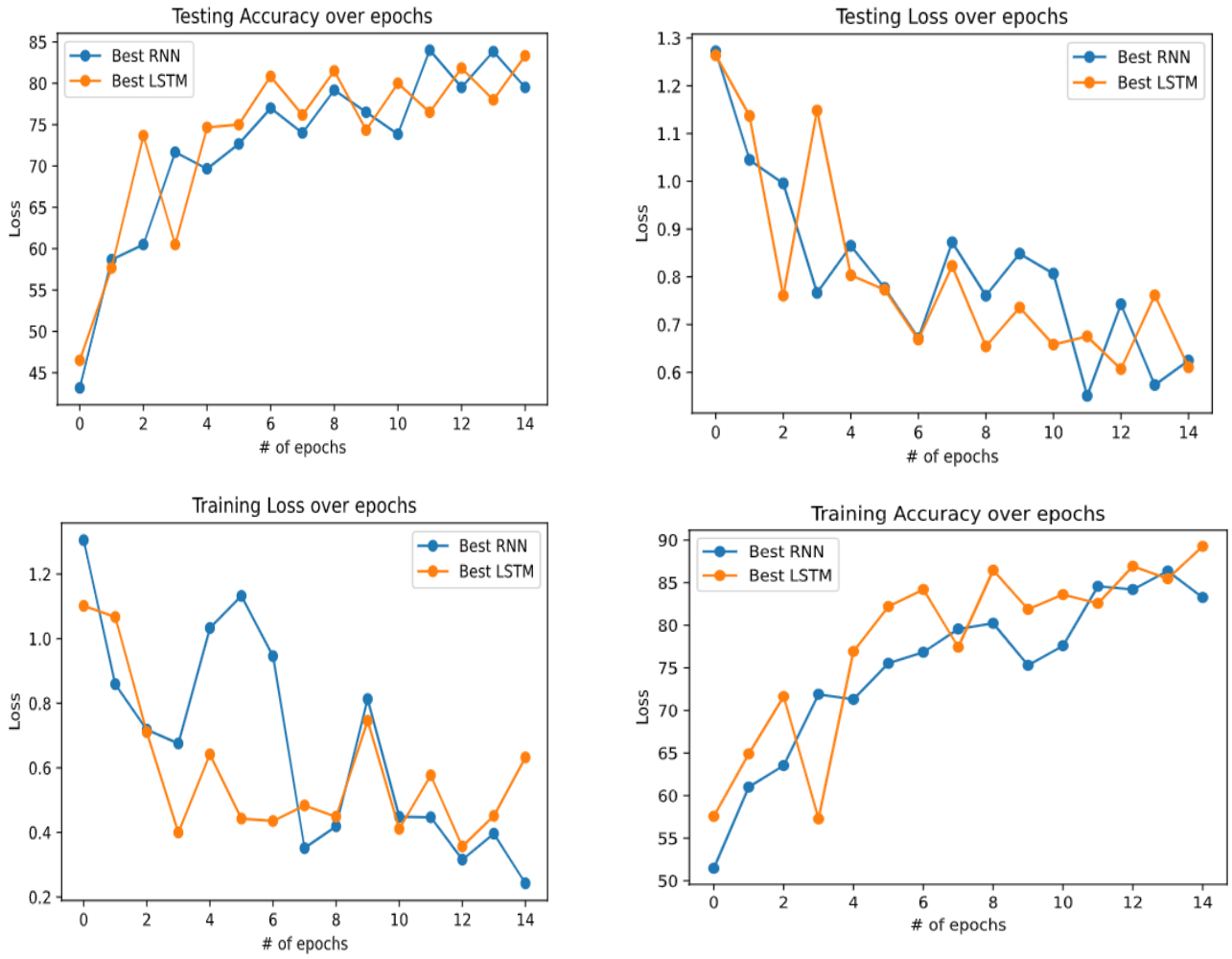


Fig. 94: Best LSTM and best RNN

From the figure above, we see that both LSTM and RNN's performed similar in 15 epochs but LSTM is the winner. Also, note that LSTM model is performed much better in 50 epochs and see the training accuracy 95% accuracy and 90% testing accuracy. However, as I said, the epoch number is limited by 15.

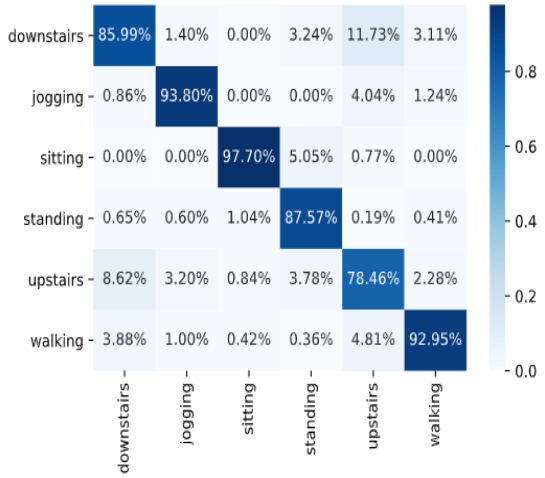


Fig. 95: LSTM training confusion matrix

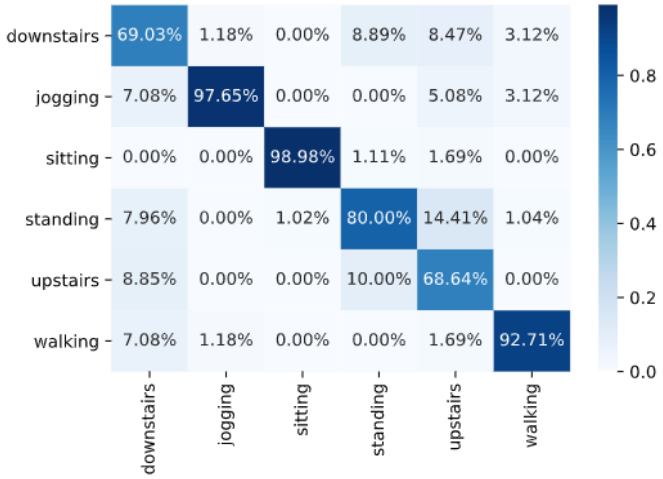


Fig. 96: LSTM testing confusion matrix

Finally, here are the confusion matrix's for vailla LSTM trained with 15 epochs. The results are stable and good. In both testing and training, we predicted the jogging, sitting and walking class nearly perfect.

Here is the code for gate's implementation:

```
def cell_forward(self,X,h_prev,C_prev):
        """

        Takes input, previous hidden state and previous cell state, compu
te:
        --
- Forget gate + Input gate + New candidate input + New cell state +
            output gate + hidden state. Then, classify by softmax.
        """
        #print(X.shape,h_prev.shape)
        # Stacking previous hidden state vector with inputs:
        stack = np.column_stack([X,h_prev])

        # Forget gate:
        forget_gate = activations.sigmoid(np.dot(stack,self.W_f) + self.B
_f)

        # İnput gate:
        input_gate = activations.sigmoid(np.dot(stack,self.W_i) + self.B_
i)

        # New candidate:
        cell_bar = np.tanh(np.dot(stack,self.W_c) + self.B_c)

        # New Cell state:
        cell_state = forget_gate * C_prev + input_gate * cell_bar
```

```
        # Output fate:
        output_gate = activations.sigmoid(np.dot(stack,self.W_o) + self.B
_o)

        # Hidden state:
        hidden_state = output_gate * np.tanh(cell_state)

        # Classifiers (Softmax) :
        dense = np.dot(hidden_state,self.W) + self.B
        probs = activations.softmax(dense)

        return (stack,forget_gate,input_gate,cell_bar,cell_state,output_g
ate,hidden_state,dense,probs)
```

3.3 PART C

In this part of the question, I am going to implement GRU (Gated Recurrent Unit) as an alternative of LSTM. They are similar to LSTM networks but GRU is simplied version of the LSTM's. Let's discuss the GRU.

In GRU networks, vanishing/exploiding gradient problem in basic RNN is solved by the gate structure. Different from LSTM, GRU has 2 gates;

- Update Gate
- Reset Gate

Let's discuss step by step. In the update gate controls the how much of the past information needs to be passed into future implementation also this gate learns to update/modify the old one. The mathmetical expression for that:

$$z_t = \sigma\ (W_z \Theta\ x_t + U_z \Theta\ h_{t-1} + \ b_z)$$

Note that z_t is the update gate's output, σ is sigmoidal activation, W_z is the weight of the input at time t, h_{t-1} the previous hidden state vector and b_z is bias term of the update gate.

Then, we have a reset gate to decide how much of the past information to forget.

$$r_t = \sigma\ (W_r \Theta\ x_t + U_r \Theta\ h_{t-1} + \ b_r)$$

Then, like in the LSTM, we have current memory cell to determined final output state.

$$\tilde{h}_t = tanh\ (\ W_h\ \Theta\ x_t + U_h\ \Theta\ (\ h_{t-1} * \ r_t) + \ b_h\ )$$

Finally, we have hidden state vector h_t

$$h_t = (1\text{-}z_t) * \tilde{h}_t \ + z_t * h_{t-1}$$

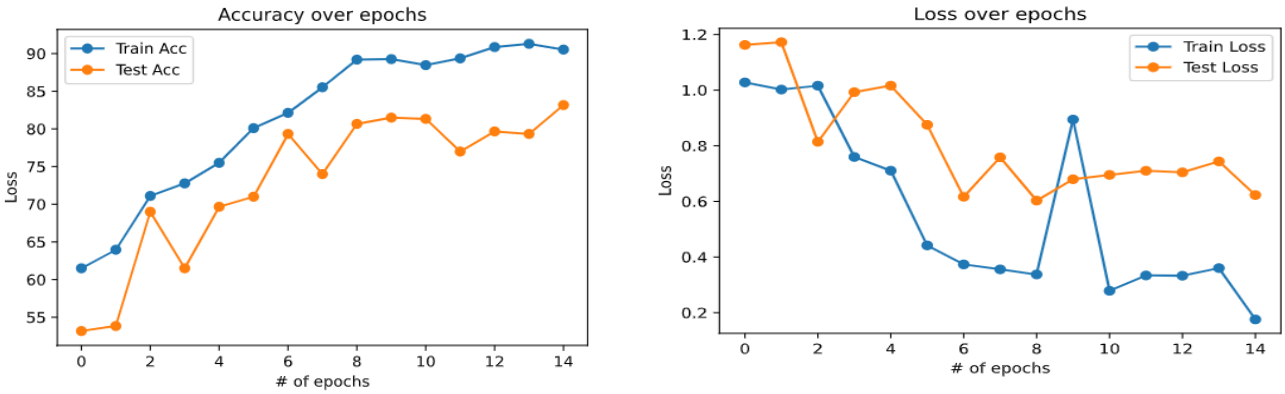


Fig. 97: GRU performance

In this part of the question, I also limited the epoch size and compare the results with 15 epochs. Here are the the results of vanilla GRU model with training accuracy 91% and testing accuracy 83.2%. Therefore, GRU does good job in both phases of learning.

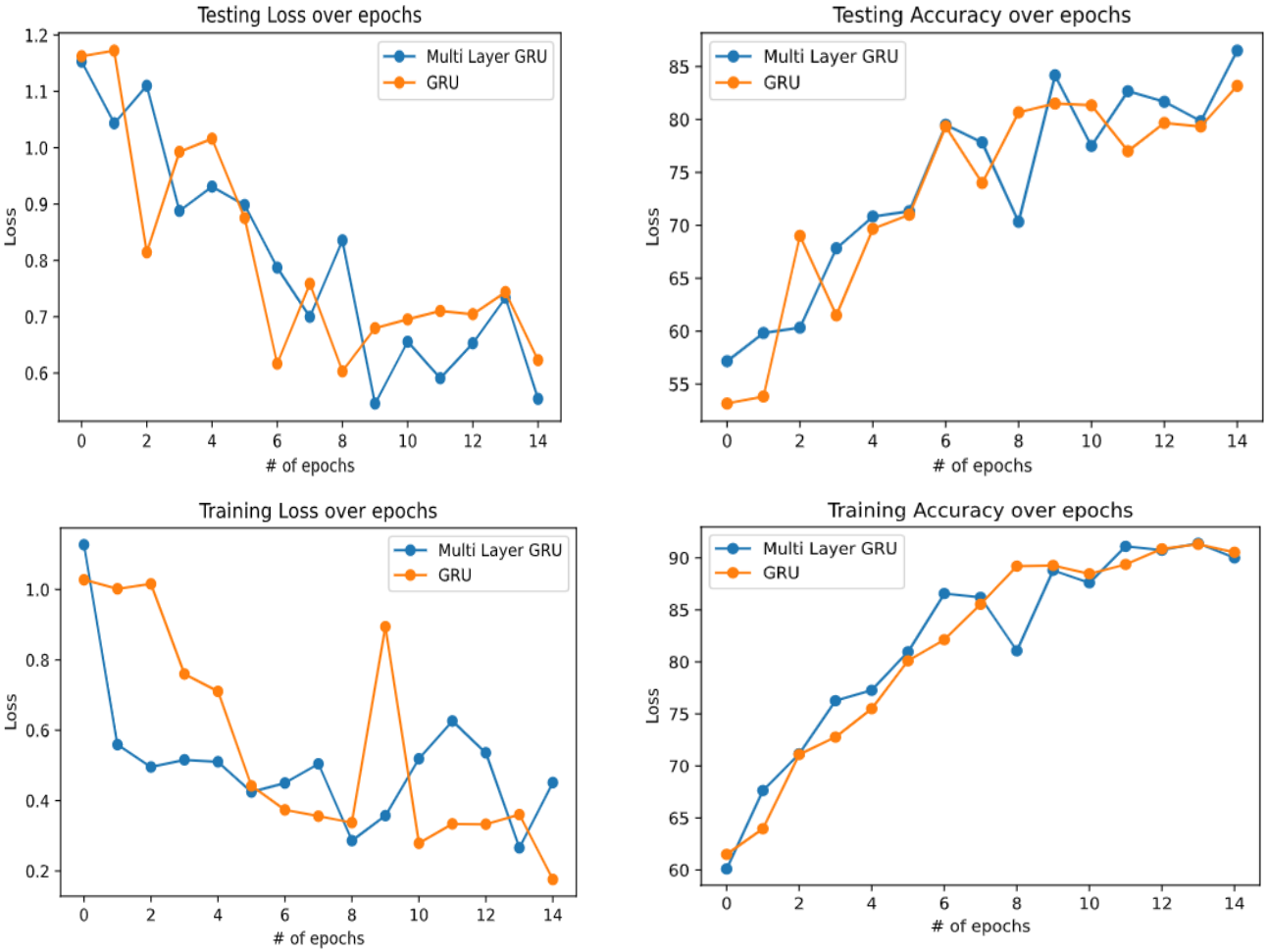


Fig. 98: Single Hidden Layer GRU versus Multi Layer GRU

I implemented multi-layer GRU to experiment with. The results are pretty good. We reached 92% training accuracy and more than 87% testing accuracy.

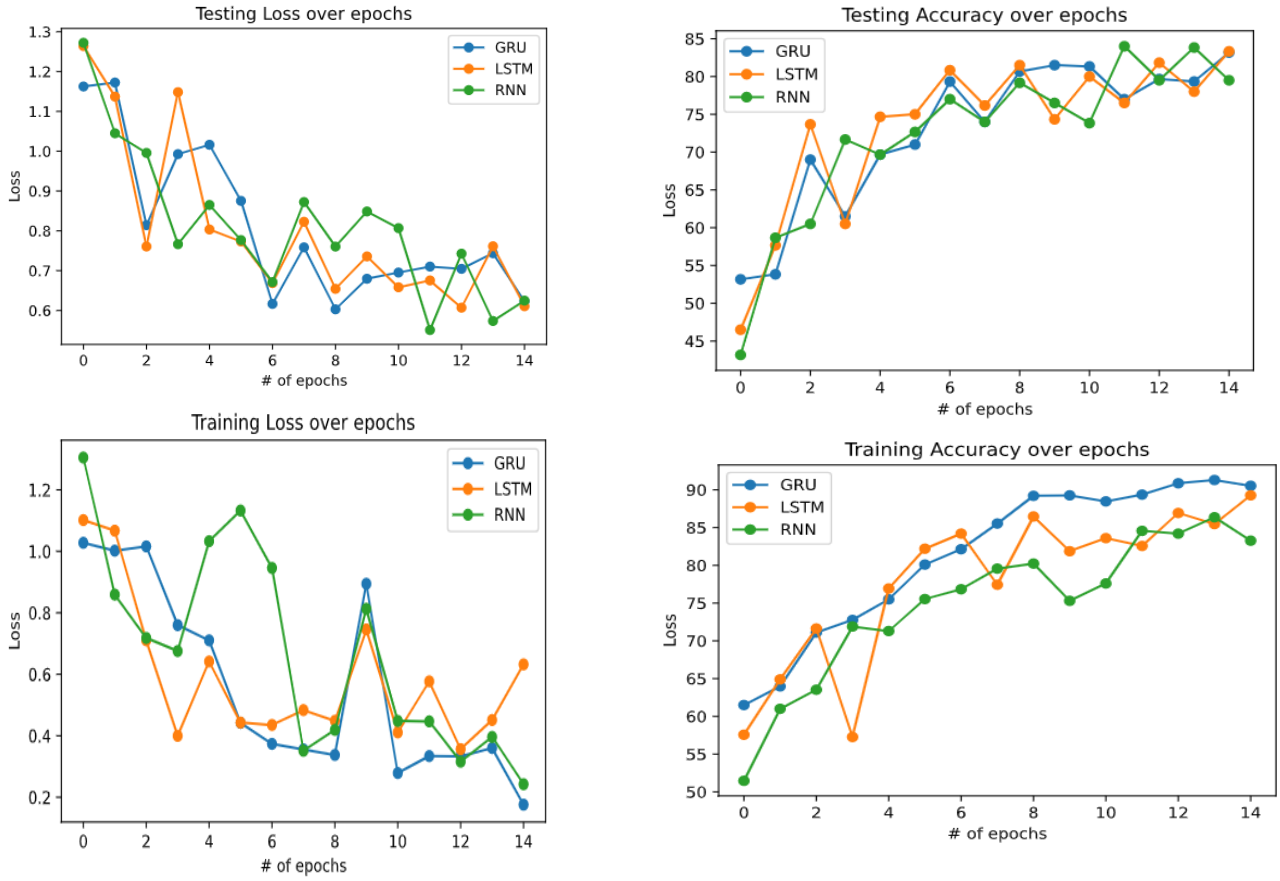


Fig. 99: RNN, LSTM and GRU comparison

Then, I compared the all networks I implemented. GRU is the winner among others. Furthermore, multi-layer GRU reaches 98% training accuracy and 94% testing accuracy in 55 epochs.

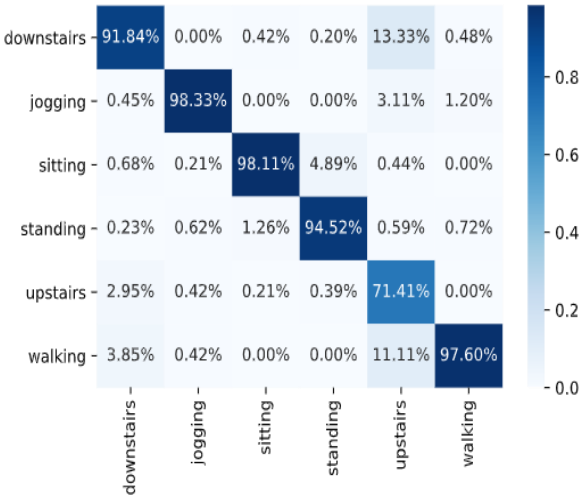


Fig. 100: GRU training confusion matrix

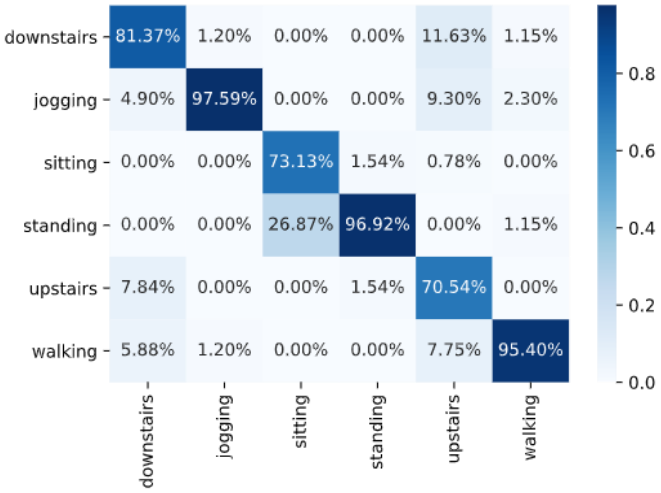


Fig. 101: GRU testing confusion matrix

Here are the results of the single-layer GRU model trained with 15 epochs. The predicted classes are more or less stable and balanced expect the upstairs class. We could not perfomed well on upstairs class in GRU network.

Here is the Python implementation of GRU forward propagation:

```
    def cell_forward(self,X,h_prev):
        """

        Takes input, previous hidden state , compute:
        --- Update gate + reset gate + Cell memory content +
hidden state. Then, classify by softmax.
        """



        # Update gate:
        update_gate = activations.sigmoid(np.dot(X,self.W_z) + np.dot(h_p
rev,self.U_z) + self.B_z)

        # Reset gate:
        reset_gate = activations.sigmoid(np.dot(X,self.W_r) + np.dot(h_pr
ev,self.U_r) + self.B_r)

        # Current memory content:
        h_hat = np.tanh(np.dot(X,self.W_h) + np.dot(np.multiply(reset_gat
e,h_prev),self.U_h) + self.B_h)

        # Hidden state:
        hidden_state = np.multiply(update_gate,h_prev) + np.multiply((1-
update_gate),h_hat)
```

```
        # Classifiers (Softmax) :
        dense = np.dot(hidden_state,self.W) + self.B
        probs = activations.softmax(dense)

        return (update_gate,reset_gate,h_hat,hidden_state,dense,probs)
```

4 APPENDIX

4.1 APPENDIX I

Part A

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'

# %%
# Necessary imports :
import numpy as np
import matplotlib.pyplot as plt
import h5py


# %%
def get_data(path) -> tuple :
    """
    Given the path of the dataset, return
    training and testing images with respective
    labels.
    """
    with h5py.File(path,'r') as F:
        # Names variable contains the names of training and testing file
        names = list(F.keys())

        data = np.array(F[names[0]][()])
        invXForm = np.array(F[names[1]][()])
        xForm = np.array(F[names[2]][()])

    return {'data'    : data,
            'invXForm': invXForm,
            'xForm'   : xForm}
```

```
path = 'assign3_data1.h5'
data_h5 = get_data(path)


# %%
data = data_h5['data']
invXForm = data_h5['invXForm']
xForm = data_h5['xForm']


# %%
print(f'The data has a shape: {data.shape}')


# %%
data = np.swapaxes(data,1,3)


# %%
print(f'The data has a shape: {data.shape}')


# %%
class ImagePreprocessing:
  """
  _____Image preprocessor_____

  Functions :
  --- ToGray(data)
      -Takes an input image then converts to gray scale by Luminosity Model

  --- MeanRemoval(data)
      -Extracking the mean of each image themselves

  --- ClipStd(data)
      - Clipping the input image within given condition

  --- Normalize(data,min_scale,max_scale)
      - Normalizing input image to [min_scale,max_scale]

  --- Flatten(data)
      - Flattening input image
  """
  def __init__(self):
    pass

  def ToGray(self,data):
    """
    Given the input image converting gray scale according to luminosity model
```

```
    """
    R_linear = 0.2126
    G_linear = 0.7152
    B_linear = 0.0722
    gray_data = (data[:,:,:,0] * R_linear) + (data[:,:,:,1] * G_linear) + (dat
a[:,:,:,2] * B_linear)

    return gray_data

  def MeanRemoval(self,data):
    """
      Given the input image, substracking the mean of pixel intensity of each
image
    """
    axis = (1,2)
    mean_pixel = np.mean(data,axis = axis)
    num_samples = data.shape[0]

    # Substracking means of each image seperately :
    for i in range(num_samples):
      data[i] -= mean_pixel[i]
    return data

  def ClipStd(self,data,std_scaler):
    """
    Given the data and range of standart deviation scaler,
    return clipped data
    """
    std_pixel = np.std(data)

    min_cond = - std_scaler * std_pixel
    max_cond =   std_scaler * std_pixel

    clipped_data = np.clip(data,min_cond,max_cond)

    return clipped_data

  def Normalize(self,data,min_scale,max_scale):
    """
    Given the data, normalize to given interval [min_val,max_val]
    """
    min = data.max()
    max = data.min()

    # First normalize in [0,1]
    norm_data = (data - min) / (max-min)

    # Normalizing in [min_scale,max_scale]
```

```
    range = max_scale - min_scale
    interval_scaled_data = (norm_data * range) + min_scale

    return interval_scaled_data

  def Flatten(self,data):
    """
    Given the input image data returning flattened version of the data
    """
    num_samples = data.shape[0]
    flatten  = data.reshape(num_samples,-1)

    return flatten


# %%
# Defining preprocessor :
preprocessor = ImagePreprocessing()


# %%
# Converting gray scale :
gray_data = preprocessor.ToGray(data = data)


# %%
# Mean removing :
mean_removed_data = preprocessor.MeanRemoval(data = gray_data)


# %%
# Standart deviation clipping :
clipped_data = preprocessor.ClipStd(data = mean_removed_data,std_scaler = 3)


# %%
# Normalized data
data_processed = preprocessor.Normalize(data = clipped_data, min_scale = 0.1,
max_scale = 0.9)


# %%
print(f' Maximum val of data :  {data_processed.max()}')
print(f' Minimum val of data :  {data_processed.min()}')


# %%
def plot_patches(data,num_patches, cmap = 'viridis'):
    num_samples = data.shape[0]
```

```
    random_indexes = np.random.randint(num_samples,size = num_patches)

    plt.figure(figsize = (18,16))
    for i in range(num_patches):
        plt.subplot(20,20,i+1)
        plt.imshow(data[random_indexes[i]],cmap = cmap)
        plt.axis('off')
    plt.show()


# %%
plot_patches(data,num_patches = 200)


# %%
plot_patches(data_processed,num_patches = 200, cmap = 'gray')


# %%
class Autoencoder:
    """
    ____Autoencoder____

    Functions :
    --- __init__(input_size,hidden_size)
        - Building overall architecture of the model

    --- InitParams(input_size,hidden_size)
        - Initializing configurable parameters

    --- aeCost(W,data,params)
        - Calculating cost and it's derivatives

    --- Forward(X)
        - Forward pass

    --- Backward(X)
        - Calculation of gradients w.r.t. loss function

    --- KL_divergence()
        - Calculate KL divergence and it's gradients

    --- TykhonowRegulator(X,grad)
        - Computing Tykhonow regularization term and it's gradient

    --- Predict(X)
        - To make predictions

    --- Sigmoid(X, grad)
```

```
        - Compute sigmoidal activation and it's gradients

    --- History()
        - To keep track history of the model
    """

    def __init__(self,input_size,hidden_size,lambd):
        """
        Construction of the architecture of the autoencoder
        """
        np.random.seed(1500)
        self.lambd = lambd
        self.beta = 1e-1
        self.rho = 5e-2
        self.learning_rate = 9e-1

        self.params = {'L_in'     : input_size,
                       'L_hidden' : hidden_size,
                       'Lambda'   : self.lambd,
                       'Beta'     : self.beta,
                       'Rho'      : self.rho}

        self.W_e = self.InitParams(input_size,hidden_size)

        self.loss = []

    def InitParams(self,input_size,hidden_size):
        """
        Given the size of the input node and hidden node, initialize the weigh
ts
        drawn from uniform distribution ~ Uniform[- sqrt(6/(L_pre + L_post)) ,
 sqrt(6/(L_pre + L_post))]
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = input_size

        W1_high = self.w_o(input_size,hidden_size)
        W1_low = - W1_high
        W1_size = (input_size,hidden_size)
        self.W1 = np.random.uniform(W1_low,W1_high,size = W1_size)

        B1_size = (1,hidden_size)
        self.B1 = np.random.uniform(W1_low,W1_high,size = B1_size)

        W2_high = self.w_o(hidden_size,self.output_size)
        W2_low = - W2_high
        W2_size = (hidden_size,self.output_size)
```

```
        self.W2 = np.random.uniform(W2_low,W2_high,size = W2_size)

        B2_size = (1,self.output_size)
        self.B2 = np.random.uniform(W1_low,W1_high,size = B2_size)

        return  {'W1' : self.W1,
                 'W2' : self.W2,
                 'B1' : self.B1,
                 'B2' : self.B2}


    def w_o(self,L_pre,L_post):
        return np.sqrt(6/(L_pre + L_post))


    def sigmoid(self,X, grad = True):
        """
        Computing sigmoid and it's gradient w.r.t. it's input

        """
        sig = 1/(1 + np.exp(-X))

        return sig * (1-sig) if grad else sig


    def forward(self,X):
        """
        Forward propagation
        """
        W1 = self.W_e['W1']
        W2 = self.W_e['W2']
        B1 = self.W_e['B1']
        B2 = self.W_e['B2']

        Z1 = np.dot(X,W1) + B1
        A1 = self.sigmoid(Z1,grad = False)

        Z2 = np.dot(A1,W2) + B2
        A2 = self.sigmoid(Z2,grad = False)

        return {"Z1": Z1,"A1": A1,
                "Z2": Z2,"A2": A2}

    def total_loss(self,outs,label):
        W1 = self.W_e['W1']
        W2 = self.W_e['W2']

        Lambda = self.params['Lambda']
```

```
        beta = self.params['Beta']
        rho = self.params['Rho']

        J_mse = self.MSE(outs['A2'],label, grad = False)
        J_tykhonow = self.TykhonowRegularization(W1 = W1, W2 = W2,lambd = Lamb
da, grad = False)
        J_KL = self.KL_divergence(rho = rho,expected = np.mean(outs['A1']), be
ta = beta, grad = False)

        return J_mse + J_tykhonow + J_KL

    def MSE(self,pred,label, grad = True):
        """
        Calculating Mean Sqaured Error and it's gradient w.r.t. output
        """
        return 1/2 *  (pred - label) if grad else 1/2 * np.sum((pred - label)*
*2)/pred.shape[0]

    def aeCost(self,data):

        outs = self.forward(data)
        loss = self.total_loss(outs,data)
        grads = self.backward(outs,data)

        return {'J'      : loss,
                'J_grad' : grads}

    def KL_divergence(self,rho,beta,expected,grad = True):
        """
        Computing KL-
divergence and it's gradients, note that gradients is only for W1
        """
        return np.tile(beta * (-(rho/expected) + (1-rho)/(1-
expected) ), reps = (10240,1)) if grad else beta * (np.sum((rho * np.log(rho/e
xpected)) + ((1-rho)*np.log((1-rho)/(1-expected)))))

    def TykhonowRegularization(self,W1,W2,lambd,grad = True):
        """
            L2 based regularization computing and it's gradients
        """
        return {'dW1': lambd * W1, 'dW2': lambd * W2} if grad else (lambd/2) *
 (np.sum(W1**2) + np.sum(W2**2))


    def backward(self,outs,data):
        """
        Given the forward pass outputs, input and their labels,
        returning gradients w.r.t. loss functions
```

```
        """
        m = data.shape[0]

        Lambda = self.params['Lambda']
        beta = self.params['Beta']
        rho = self.params['Rho']

        W1 = self.W_e['W1']
        W2 = self.W_e['W2']
        B1 = self.W_e['B1']
        B2 = self.W_e['B2']

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']

        L2_grad = self.TykhonowRegularization(W1,W2,lambd = Lambda , grad = Tr
ue)
        KL_grad_W1 = self.KL_divergence(rho,beta,expected = np.mean(A1),grad =
 True)

        dZ2 = self.MSE(A2,data, grad = True) * self.sigmoid(Z2, grad = True)

        dW2 = (1/m) * (np.dot(A1.T,dZ2) + L2_grad['dW2'])
        dB2 = (1/m) * (np.sum(dZ2, axis=0, keepdims=True))

        dZ1 = (np.dot(dZ2,W2.T) + KL_grad_W1)  * self.sigmoid(Z1,grad = True)

        dW1 = (1/m) * (np.dot(data.T,dZ1) + L2_grad['dW1'])
        dB1 = (1/m) * (np.sum(dZ1, axis=0, keepdims=True))

        assert (dW1.shape == W1.shape and dW2.shape == W2.shape)

        return {"dW1": dW1, "dW2": dW2,
                "dB1": dB1, "dB2": dB2}


    def fit(self,data,epochs = 5000,verbose = True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """
        for epoch in range(epochs):

            loss_and_grads = self.aeCost(data)
            self.step(grads = loss_and_grads['J_grad'])
```

```
            if verbose:
                print(f"[{epoch}/{epochs}]     ----------
> Loss  :{loss_and_grads['J']}")

            self.loss.append(loss_and_grads['J'])

    def step(self,grads):
        """
        Updating configurable parameters according to full-
batch stochastic gradient update rule
        """
        self.W_e['W1'] += -self.learning_rate * grads['dW1']
        self.W_e['W2'] += -self.learning_rate * grads['dW2']
        self.W_e['B1'] += -self.learning_rate * grads['dB1']
        self.W_e['B2'] += -self.learning_rate * grads['dB2']
        self.learning_rate *= 0.9999


    def evaluate(self):
        plt.plot(self.loss, color = 'orange')
        plt.xlabel(' # of Epochs')
        plt.ylabel('Loss')
        plt.title('Training Loss versus Epochs')
        plt.legend([f'Loss : {self.loss[-1]}'])

    def display_weights(self):
        """
        Display weights as a image for feature representation
        """
        W1 = self.W_e['W1']
        num_disp = W1.shape[1]
        fig = plt.figure(figsize = (9,8))
        for i in range(num_disp):
            plt.subplot(8,8,i+1)
            plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
            plt.axis('off')
        fig.suptitle('Hidden Layer Feature Representation')
        plt.show()

    def display_outputs(self,output,data,num = 4):

        """
        Displaying outputs, please give only sqaured values, i.e., 1,4,16,...
        """
        random_indexes = np.random.randint(output.shape[0],size = num)
        plt.figure(figsize=(12, 4))
        for i in range(len(random_indexes)):
```

```
            ax = plt.subplot(2,5,i+1)
            plt.imshow(output[random_indexes[i]].reshape(16,16),cmap = 'gray')
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)
            plt.title("Reconstructed Image")
            #plt.axis('off')
            ax = plt.subplot(2, 5, i + 1 + 5)
            plt.imshow(data[random_indexes[i]].reshape(16,16),cmap = 'gray')
            plt.title("Original Image")
            plt.gray()
            ax.get_xaxis().set_visible(False)
            ax.get_yaxis().set_visible(False)

        plt.show()


    def parameters(self):
        """
        Returns configurable parameters

        """
        return self.W_e

    def history(self):
        return {'Loss' : self.loss}




# %%
class Solver:
    """
    Given  as input, A Solver encapsulates all the logic necessary for trainin
g then
    implement gradients solver to minimize the cost.The Solver performs stocha
stic gradient descent.

    """
    def __init__(self, model,data):

        self.model = model
        self.data = data

    def train(self,epochs = 5000,verbose = False):
        """
        Optimization of the model by minimizing cost by solving gradients
        """
        self.model.fit(self.data,epochs,verbose)
```

```
    def parameters(self):
        """
        Returning configurable parameters of the network
        """
        return self.model.parameters()


# %%
data_feed = preprocessor.Flatten(data_processed)
input_size = data_feed.shape[1]
hidden_size = 64
autoencoder = Autoencoder(input_size = input_size, hidden_size = hidden_size,l
ambd = 5e-4)


# %%
solver = Solver(model = autoencoder, data = data_feed)
solver.train(verbose = True)


# %%
net_params = solver.parameters()
net_history = autoencoder.history()


# %%
autoencoder.evaluate()


# %%
autoencoder.display_weights()


# %%
preds = autoencoder.forward(data_feed)
autoencoder.display_outputs(preds['A2'],data_feed)


# %%
hidden_size_1 = 10
lambd_1 = 0
autoencoder_1 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_1, lambd = lambd_1)
solver_1 = Solver(model = autoencoder_1, data = data_feed)
solver_1.train()
autoencoder_1.evaluate()
autoencoder_1.display_weights()
preds_1 = autoencoder_1.forward(data_feed)
```

```
autoencoder_1.display_outputs(preds_1['A2'],data_feed)}


# %%
hidden_size_2 = 10
lambd_2 = 1e-3
autoencoder_2 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_2, lambd = lambd_2)
solver_2 = Solver(model = autoencoder_2, data = data_feed)
solver_2.train()
autoencoder_2.evaluate()
autoencoder_2.display_weights()
preds_2 = autoencoder_2.forward(data_feed)
autoencoder_2.display_outputs(preds_2['A2'],data_feed)


# %%
hidden_size_3 = 10
lambd_3 = 1e-5
autoencoder_3 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_3, lambd = lambd_3)
solver_3 = Solver(model = autoencoder_3, data = data_feed)
solver_3.train()
autoencoder_3.evaluate()
autoencoder_3.display_weights()
preds_3 = autoencoder_3.forward(data_feed)
autoencoder_3.display_outputs(preds_3['A2'],data_feed)


# %%
hidden_size_4 = 50
lambd_4 = 0
autoencoder_4 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_4, lambd = lambd_4)
solver_4 = Solver(model = autoencoder_4, data = data_feed)
solver_4.train()
autoencoder_4.evaluate()
autoencoder_4.display_weights()
preds_4 = autoencoder_4.forward(data_feed)
autoencoder_4.display_outputs(preds_4['A2'],data_feed)


# %%
hidden_size_5 = 50
lambd_5 = 1e-3
autoencoder_5 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_5, lambd = lambd_5)
solver_5 = Solver(model = autoencoder_5, data = data_feed)
solver_5.train()
```

```
autoencoder_5.evaluate()
autoencoder_5.display_weights()
preds_5 = autoencoder_5.forward(data_feed)
autoencoder_5.display_outputs(preds_5['A2'],data_feed)


# %%
autoencoder_5.display_outputs(preds_5['A2'],data_feed)


# %%
hidden_size_6 = 50
lambd_6 = 1e-5
autoencoder_6 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_6, lambd = lambd_6)
solver_6 = Solver(model = autoencoder_6, data = data_feed)
solver_6.train()
autoencoder_6.evaluate()
autoencoder_6.display_weights()
preds_6 = autoencoder_6.forward(data_feed)


# %%
autoencoder_6.display_outputs(preds_6['A2'],data_feed)


# %%
hidden_size_7 = 100
lambd_7 = 0
autoencoder_7 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_7, lambd = lambd_7)
solver_7 = Solver(model = autoencoder_7, data = data_feed)
solver_7.train()
autoencoder_7.evaluate()
#autoencoder_7.display_weights()
preds_7 = autoencoder_7.forward(data_feed)
autoencoder_7.display_outputs(preds_7['A2'],data_feed)


# %%
#autoencoder_7.display_weights()
W1 = autoencoder_7.W_e['W1']
num_disp = W1.shape[1]
fig = plt.figure(figsize = (9,8))
for i in range(num_disp):
    plt.subplot(10,10,i+1)
    plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
    plt.axis('off')
fig.suptitle('Hidden Layer Feature Representation')
```

```
plt.show()
preds_7 = autoencoder_7.forward(data_feed)
autoencoder_7.display_outputs(preds_7['A2'],data_feed)


# %%
autoencoder_7.display_outputs(preds_7['A2'],data_feed)


# %%
hidden_size_8 = 100
lambd_8 = 1e-3
autoencoder_8 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_8, lambd = lambd_8)
solver_8 = Solver(model = autoencoder_8, data = data_feed)
solver_8.train()
autoencoder_8.evaluate()
W1 = autoencoder_8.W_e['W1']
num_disp = W1.shape[1]
fig = plt.figure(figsize = (9,8))
for i in range(num_disp):
    plt.subplot(10,10,i+1)
    plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
    plt.axis('off')
fig.suptitle('Hidden Layer Feature Representation')
plt.show()
preds_8 = autoencoder_8.forward(data_feed)
autoencoder_8.display_outputs(preds_8['A2'],data_feed)


# %%
hidden_size_9 = 100
lambd_9 = 1e-5
autoencoder_9 = Autoencoder(input_size = input_size, hidden_size = hidden_size
_9, lambd = lambd_9)
solver_9 = Solver(model = autoencoder_9, data = data_feed)
solver_9.train()
autoencoder_9.evaluate()
W1 = autoencoder_9.W_e['W1']
num_disp = W1.shape[1]
fig = plt.figure(figsize = (9,8))
for i in range(num_disp):
    plt.subplot(10,10,i+1)
    plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
    plt.axis('off')
fig.suptitle('Hidden Layer Feature Representation')
plt.show()
preds_9 = autoencoder_9.forward(data_feed)
```

```
# %%
autoencoder_9.display_outputs(preds_9['A2'],data_feed)


# %%
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow import keras
import tensorflow.keras.backend as K


# %%
if tf.test.gpu_device_name():
    print('Default GPU Device:{}'.format(tf.test.gpu_device_name()))
else:
   print("Please install GPU version of TF")


# %%
def MeanSquaredError():
  def customMeanSquaredError(pred,label):
    return 1/2 * K.sum((pred - label)**2)/pred.shape[0]
  return customMeanSquaredError

def KL_divergence(rho, beta):
  def customKL(out):
    kl1 = rho*K.log(rho/K.mean(out, axis=0))
    kl2 = (1-rho)*K.log((1-rho)/(1-K.mean(out, axis=0)))
    return beta*K.sum(kl1+kl2)
  return customKL


# %%
def create_model(hidden_size,lambd):
  tf_weights = tf_weight_initializer(inp_dim = inp_dim, hidden_dim = hidden_si
ze)
  input_img = keras.Input(shape=(inp_dim,))
  encoded = layers.Dense(encoding_dim,activation='sigmoid',
                         kernel_regularizer=tf.keras.regularizers.l2(lambd),
                         activity_regularizer=KL_divergence(rho,beta),
                         kernel_initializer = tf_weights['W1'],
                         bias_initializer = tf_weights['B1'])(input_img)

  decoded = layers.Dense(inp_dim,activation='sigmoid',
                         activity_regularizer=tf.keras.regularizers.l2(lambd),
                         kernel_initializer = tf_weights['W2'],
                         bias_initializer = tf_weights['B2'])(encoded)

  tf_autoencoder = keras.Model(input_img,decoded)
```

```
  optimizer = tf.keras.optimizers.SGD(learning_rate=0.9,momentum=0,nesterov=Fa
lse)
  tf_autoencoder.compile(optimizer=optimizer,loss=MeanSquaredError())




  return tf_autoencoder

def plot_tf_weights(W1):
  num_disp = W1.shape[1]
  fig = plt.figure(figsize = (9,8))
  for i in range(num_disp):
      plt.subplot(10,10,i+1)
      plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
      plt.axis('off')
  fig.suptitle('Hidden Layer Feature Representation')
  plt.show()


# %%
tf_model_1 = create_model(hidden_size = 10,lambd = 0)
tf_model_1.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_1 = tf_model_1.history.history
plt.plot(tf_history_1['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')
plt.legend([f'Loss : {tf_history_1['loss'][-1]}'])

tf_preds_1 = tf_model_1.predict(data_feed)
autoencoder.display_outputs(tf_preds_1,data_feed)
tf_weights_1 = tf_model_1.get_weights()
plot_tf_weights(tf_weights_1[0])



# %%
tf_model_2 = create_model(hidden_size = 10,lambd = 1e-3)
tf_model_2.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_2 = tf_model_2.history.history
plt.plot(tf_history_2['loss'])
plt.xlabel('# of epochs')
```

```
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_2 = tf_model_2.predict(data_feed)
autoencoder.display_outputs(tf_preds_2,data_feed)
tf_weights_2 = tf_model_2.get_weights()
plot_tf_weights(tf_weights_2[0])


# %%
tf_model_3 = create_model(hidden_size = 10,lambd = 1e-5)
tf_model_3.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_3 = tf_model_3.history.history
plt.plot(tf_history_3['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_3 = tf_model_3.predict(data_feed)
autoencoder.display_outputs(tf_preds_3,data_feed)
tf_weights_3 = tf_model_3.get_weights()
plot_tf_weights(tf_weights_3[0])


# %%
tf_model_4 = create_model(hidden_size = 50,lambd = 0)
tf_model_4.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_4 = tf_model_4.history.history
plt.plot(tf_history_4['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_4 = tf_model_4.predict(data_feed)
autoencoder.display_outputs(tf_preds_4,data_feed)
tf_weights_4 = tf_model_4.get_weights()
plot_tf_weights(tf_weights_4[0])


# %%
tf_model_5 = create_model(hidden_size = 50,lambd = 1e-3)
tf_model_5.fit(data_feed, data_feed,
                epochs=5000,
```

```
                batch_size=data_feed.shape[0])

tf_history_5 = tf_model_5.history.history
plt.plot(tf_history_5['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_5 = tf_model_5.predict(data_feed)
autoencoder.display_outputs(tf_preds_5,data_feed)
tf_weights_5 = tf_model_5.get_weights()
plot_tf_weights(tf_weights_5[0])


# %%
tf_model_6 = create_model(hidden_size = 50,lambd = 1e-5)
tf_model_6.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_6 = tf_model_6.history.history
plt.plot(tf_history_6['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_6 = tf_model_6.predict(data_feed)
autoencoder.display_outputs(tf_preds_6,data_feed)
tf_weights_6 = tf_model_1.get_weights()
plot_tf_weights(tf_weights_6[0])


# %%
tf_model_7 = create_model(hidden_size = 100,lambd = 0)
tf_model_7.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_7 = tf_model_7.history.history
plt.plot(tf_history_7['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_7 = tf_model_7.predict(data_feed)
autoencoder.display_outputs(tf_preds_7,data_feed)
tf_weights_7 = tf_model_7.get_weights()
plot_tf_weights(tf_weights_7[0])
```

```
# %%
tf_model_8 = create_model(hidden_size = 100,lambd = 1e-3)
tf_model_8.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_8 = tf_model_8.history.history
plt.plot(tf_history_8['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_8 = tf_model_8.predict(data_feed)
autoencoder.display_outputs(tf_preds_8,data_feed)
tf_weights_8 = tf_model_8.get_weights()
plot_tf_weights(tf_weights_8[0])


# %%
tf_model_9 = create_model(hidden_size = 100,lambd = 1e-5)
tf_model_9.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])

tf_history_9 = tf_model_9.history.history
plt.plot(tf_history_9['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')

tf_preds_9 = tf_model_9.predict(data_feed)
autoencoder.display_outputs(tf_preds_9,data_feed)
tf_weights_9 = tf_model_9.get_weights()
plot_tf_weights(tf_weights_9[0])


# %%
#encoding_dim = 10
rho,beta = 5e-1,1e-1
inp_dim = 256
#lamb = 0

W_scaler = lambda L_pre,L_post : np.sqrt(6/(L_pre + L_post))

def tf_weight_initializer(inp_dim,hidden_dim):
  initializer_1 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(inp_dim,hidden_dim), maxval=W_scaler(inp_dim,hidden_dim))
  values_2 = initializer_1(shape=(inp_dim,hidden_dim))
```

```
  initializer_2 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(hidden_dim,inp_dim), maxval=W_scaler(hidden_dim,inp_dim))
  values_2 = initializer_2(shape=(inp_dim,hidden_dim))

  initializer_3 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(inp_dim,hidden_dim), maxval=W_scaler(inp_dim,hidden_dim))
  values_3 = initializer_3(shape=(1,hidden_dim))

  initializer_4 = tf.keras.initializers.RandomUniform(minval=-
W_scaler(hidden_dim,inp_dim), maxval=W_scaler(hidden_dim,inp_dim))
  values_4 = initializer_4(shape=(1,inp_dim))

  return {'W1':initializer_1,
          'W2':initializer_2,
          'B1':initializer_3,
          'B2':initializer_4}

tf_weights = tf_weight_initializer(inp_dim = inp_dim, hidden_dim = encoding_di
m)


# %%
input_img = keras.Input(shape=(inp_dim,))

encoded = layers.Dense(encoding_dim,activation='sigmoid',
                       kernel_regularizer=tf.keras.regularizers.l2(lamb),
                       activity_regularizer=KL_divergence(rho,beta),
                       kernel_initializer = tf_weights['W1'],
                       bias_initializer = tf_weights['B1'])(input_img)

decoded = layers.Dense(inp_dim,activation='sigmoid',
                       activity_regularizer=tf.keras.regularizers.l2(lamb),
                       kernel_initializer = tf_weights['W2'],
                       bias_initializer = tf_weights['B2'])(encoded)

tf_autoencoder = keras.Model(input_img,decoded)


# %%
tf_autoencoder.summary()


# %%
optimizer = tf.keras.optimizers.SGD(learning_rate=0.9,momentum=0,nesterov=Fals
e)
tf_autoencoder.compile(optimizer=optimizer,loss=MeanSquaredError())


# %%
```

```
tf_autoencoder.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])


# %%
tf_history = tf_autoencoder.history.history
plt.plot(tf_history['loss'])
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')


# %%
tf_preds = tf_autoencoder.predict(data_feed)


# %%
autoencoder.display_outputs(tf_preds,data_feed)


# %%
tf_weights = tf_autoencoder.get_weights()


# %%
tf_weights = tf_autoencoder.get_weights()
W1 = tf_weights[0]
num_disp = W1.shape[1]
fig = plt.figure(figsize = (9,8))
for i in range(num_disp):
    plt.subplot(8,8,i+1)
    plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
    plt.axis('off')
fig.suptitle('Hidden Layer Feature Representation')
plt.show()


# %%



# %%
input_img_optim = keras.Input(shape=(inp_dim,))

encoded_optim = layers.Dense(encoding_dim,activation='sigmoid',
                      kernel_regularizer=tf.keras.regularizers.l2(5e-4),
                      activity_regularizer=KL_divergence(rho,beta))(input_img
_optim)
```

```
decoded_optim = layers.Dense(inp_dim,activation='sigmoid',
                      activity_regularizer=tf.keras.regularizers.l2(5e-
4))(encoded_optim)

tf_autoencoder_optim = keras.Model(input_img_optim,decoded_optim)

tf_autoencoder.compile(optimizer='adam',loss=MeanSquaredError())
tf_autoencoder.summary()


# %%
tf_autoencoder.fit(data_feed, data_feed,
                epochs=5000,
                batch_size=data_feed.shape[0])


# %%
tf_history_optim = tf_autoencoder.history.history
plt.plot(tf_history_optim['loss'],color = 'green')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss versus Epoch')


# %%
tf_preds_optim = tf_autoencoder.predict(data_feed)
autoencoder.display_outputs(tf_preds_optim,data_feed)


# %%
tf_weights_optim = tf_autoencoder_optim.get_weights()
W1 = tf_weights[0]
num_disp = W1.shape[1]
fig = plt.figure(figsize = (9,8))
for i in range(num_disp):
    plt.subplot(8,8,i+1)
    plt.imshow(W1.T[i].reshape(16,16),cmap = 'gray')
    plt.axis('off')
fig.suptitle('Hidden Layer Feature Representation')
plt.show()


# %%
```

Part C

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'

# %%
# Necessary imports :
import numpy as np
import matplotlib.pyplot as plt
import h5py
import math
import pandas as pd
import seaborn as sns


# %%
def sigmoid(x):
    c = np.clip(x,-700,700)
    return 1 / (1 + np.exp(-c))
def dsigmoid(y):
    return y * (1 - y)
def tanh(x):
    return np.tanh(x)
def dtanh(y):
    return 1 - y * y


# %%
with h5py.File('assign3_data3.h5','r') as F:
# Names variable contains the names of training and testing file
    names = list(F.keys())
    X_train = np.array(F[names[0]][()])
    y_train = np.array(F[names[1]][()])
    X_test = np.array(F[names[2]][()])
    y_test = np.array(F[names[3]][()])


# %%
class Metrics:
    """
    Necessary metrics to evaluate the model.
        Functions(labels,preds):
        --- confusion_matrix
        --- accuracy_score
    """
    def confusion_matrix(self,labels,preds):
        """
```

```
        Takes desireds/labels and softmax predictions,
        return a confusion matrix.
        """
        label = pd.Series(labels,name='Actual')
        pred = pd.Series(preds,name='Predicted')
        return pd.crosstab(label,pred)

    def accuracy_score(self,labels,preds):
        """
        Takes desireds/labels and softmax predictions,
        return a accuracy_score.
        """
        count = 0
        size = labels.shape[0]
        for i in range(size):
            if preds[i] == labels[i]:
                count +=1
        return  100 * (count/size)

    def accuracy(self,labels,preds):
        """
        Takes desireds/labels and softmax predictions,
        return a accuracy.
        """
        return 100 * (labels == preds).mean()


# %%
class Activations:
    """
    Necessary activation functions for recurrent neural network(RNN,LSTM,GRU).
    """
    def relu_alternative(self,X):
        """
            Rectified linear unit activation(ReLU).
        """
        return np.maximum(X, 0)

    def ReLU(self,X):
        """
            Rectified linear unit activation(ReLU).
            Most time efficient version.
        """
        return (abs(X) + X) / 2

    def relu_another(self,X):
        """
            Rectified linear unit activation(ReLU).
```

```
        """
        return X * (X > 0)

    def tanh(self,X):
        return np.tanh(X)

    def tanh_manuel(self,X):
        """
            Hyperbolic tangent activation(tanh).
        """
        return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))

    def sigmoid(self,X):
        """
            Sigmoidal activation.
        """
        c = np.clip(X,-700,700)
        return 1/(1 + np.exp(-c))

    def softmax(self,X):
        """
            Stable version of softmax classifier, note that column sum is equa
l to 1.
        """
        e_x = np.exp(X - np.max(X, axis=-1, keepdims=True))
        return e_x / np.sum(e_x, axis=-1, keepdims=True)

    def softmax_stable(self,X):
        """
        Less stable version of softmax activation

        """
        e_x = np.exp(X - np.max(X))
        return e_x / np.sum(e_x)

    def ReLUDerivative(self,X):
        """
        The derivative of the ReLU function w.r.t. given input.
        """
        return 1 * (X > 0)

    def ReLU_grad(self,X):
        """
        The derivative of the ReLU function w.r.t. given input.
        """
        X[X<=0] = 0
        X[X>1] = 1
        return X
```

```
    def dReLU(self,X):
        """
        The derivative of the ReLU function w.r.t. given input.
        """
        return np.where(X <= 0, 0, 1)

    def dtanh(self,X):
        """
        The derivative of the tanh function w.r.t. given input.
        """
        return  1-(np.tanh(X)**2)

    def dsigmoid(self,X):
        """
        The derivative of the sigmoid function w.r.t. given input.
        """
        return self.sigmoid(X) * (1-self.sigmoid(X))

    def softmax_stable_gradient(self,soft_out):
        return soft_out * (1 - soft_out)

    def softmax_grad(self,softmax):
        s = softmax.reshape(-1,1)
        return np.diagflat(s) - np.dot(s, s.T)

    def softmax_gradient(self,Sz):
        """Computes the gradient of the softmax function.
        z: (T, 1) array of input values where the gradient is computed. T is t
he
        number of output classes.
        Returns D (T, T) the Jacobian matrix of softmax(z) at the given z. D[i
, j]
        is DjSi - the partial derivative of Si w.r.t. input j.
        """

        # -
SjSi can be computed using an outer product between Sz and itself. Then
        # we add back Si for the i=j cases by adding a diagonal matrix with th
e
        # values of Si on its diagonal.
        D = -np.outer(Sz, Sz) + np.diag(Sz.flatten())
        return D


# %%
class RNN(object):
    """
```

```
    Recurrent Neural Network for classifying human activity.
    RNN encapsulates all necessary logic for training the network.

    """
    def __init__(self,input_dim = 3,hidden_dim = 128, seq_len = 150, learning_
rate = 1e-1, mom_coeff = 0.85, batch_size = 32, output_class = 6):

        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim_inp2hid = Xavier(self.input_dim,self.hidden_dim)
        self.W1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(self.input_dim,self.hidden_dim))
        self.B1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.hidden_dim))

        lim_hid2hid = Xavier(self.hidden_dim,self.hidden_dim)
        self.W1_rec= np.random.uniform(-
lim_hid2hid,lim_hid2hid,(self.hidden_dim,self.hidden_dim))

        lim_hid2out = Xavier(self.hidden_dim,self.output_class)
        self.W2 = np.random.uniform(-
lim_hid2out,lim_hid2out,(self.hidden_dim,self.output_class))
        self.B2 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # Storing previous momentum updates :
        self.prev_updates = {'W1'       : 0,
```

```
                            'B1'       : 0,
                            'W1_rec'   : 0,
                            'W2'       : 0,
                            'B2'       : 0}


    def forward(self,X) -> tuple:
        """
        Forward propagation of the RNN through time.
        ____________________________________________________________

        Inputs:
        --- X is the bacth.
        --- h_prev_state is the previous state of the hidden layer.
        ____________________________________________________________

        Returns:
        --- (X_state,hidden_state,probs) as a tuple.
        ------ 1) X_state is the input across all time steps
        ------ 2) hidden_state is the hidden stages across time
        -----
- 3) probs is the probabilities of each outputs, i.e. outputs of softmax
        ____________________________________________________________
        """
        X_state = dict()
        hidden_state = dict()
        output_state = dict()
        probs = dict()


        self.h_prev_state = np.zeros((1,self.hidden_dim))
        hidden_state[-1] = np.copy(self.h_prev_state)

        # Loop over time T = 150 :
        for t in range(self.seq_len):

            # Selecting first record with 3 inputs, dimension = (batch_size,in
put_size)
            X_state[t] = X[:,t]

            # Recurrent hidden layer :
            hidden_state[t] = np.tanh(np.dot(X_state[t],self.W1) + np.dot(hidd
en_state[t-1],self.W1_rec) + self.B1)
            output_state[t] = np.dot(hidden_state[t],self.W2) + self.B2

            # Per class probabilites :
            probs[t] = activations.softmax(output_state[t])
```

```
        return (X_state,hidden_state,probs)


    def BPTT(self,cache,Y):
        """

        Back propagation through time algorihm.
        Inputs:
        -- Cache = (X_state,hidden_state,probs)
        -- Y = desired output

        Returns:
        -- Gradients w.r.t. all configurable elements
        """

        X_state,hidden_state,probs = cache

        # backward pass: compute gradients going backwards
        dW1, dW1_rec, dW2 = np.zeros_like(self.W1), np.zeros_like(self.W1_rec)
, np.zeros_like(self.W2)

        dB1, dB2 = np.zeros_like(self.B1), np.zeros_like(self.B2)

        dhnext = np.zeros_like(hidden_state[0])

        dy = np.copy(probs[149])
        dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1

        dB2 += np.sum(dy,axis = 0, keepdims = True)
        dW2 += np.dot(hidden_state[149].T,dy)

        for t in reversed(range(1,self.seq_len)):



            dh = np.dot(dy,self.W2.T) + dhnext

            dhrec = (1 - (hidden_state[t] * hidden_state[t])) * dh

            dB1 += np.sum(dhrec,axis = 0, keepdims = True)

            dW1 += np.dot(X_state[t].T,dhrec)

            dW1_rec += np.dot(hidden_state[t-1].T,dhrec)

            dhnext = np.dot(dhrec,self.W1_rec.T)
```

```
        for grad in [dW1,dB1,dW1_rec,dW2,dB2]:
            np.clip(grad, -10, 10, out = grad)


        return [dW1,dB1,dW1_rec,dW2,dB2]

    def earlyStopping(self,ce_train,ce_val,ce_threshold,acc_train,acc_val,acc_
threshold):
        if ce_train - ce_val < ce_threshold or acc_train - acc_val > acc_thres
hold:
            return True
        else:
            return False


    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def step(self,grads,momentum = True):
        """
        SGD on mini batches
        """



        #for config_param,grad in zip([self.W1,self.B1,self.W1_rec,self.W2,sel
f.B2],grads):
            #config_param -= self.learning_rate * grad

        if momentum:

            delta_W1 = -
self.learning_rate * grads[0] +  self.mom_coeff * self.prev_updates['W1']
            delta_B1 = -
self.learning_rate * grads[1] +  self.mom_coeff * self.prev_updates['B1']
            delta_W1_rec = -
self.learning_rate * grads[2] +  self.mom_coeff * self.prev_updates['W1_rec']
            delta_W2 = -
self.learning_rate * grads[3] +  self.mom_coeff * self.prev_updates['W2']

            delta_B2 = -
self.learning_rate * grads[4] +  self.mom_coeff * self.prev_updates['B2']
```

```
            self.W1 += delta_W1
            self.W1_rec += delta_W1_rec
            self.W2 += delta_W2
            self.B1 += delta_B1
            self.B2 += delta_B2


            self.prev_updates['W1'] = delta_W1
            self.prev_updates['W1_rec'] = delta_W1_rec
            self.prev_updates['W2'] = delta_W2
            self.prev_updates['B1'] = delta_B1
            self.prev_updates['B2'] = delta_B2

            self.learning_rate *= 0.9999

    def fit(self,X,Y,X_val,y_val,epochs = 50 ,verbose = True, earlystopping =
False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)

            for i in range(round(X.shape[0]/self.batch_size)):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]

                cache_train = self.forward(X_feed)

                grads = self.BPTT(cache_train,y_feed)
                self.step(grads)


            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[2][149])
            predictions_train = self.predict(X)
```

```
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            _,__,probs_test = self.forward(X_val)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if earlystopping:
                if self.earlyStopping(ce_train = cross_loss_train,ce_val = cro
ss_loss_val,ce_threshold = 3.0,acc_train = acc_train,acc_val = acc_val,acc_thr
eshold = 15):
                    break



            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)

    def predict(self,X):
        _,__,probs = self.forward(X)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}



# %%
```

```
input_dim = 3
activations = Activations()
metrics = Metrics()
model = RNN(input_dim = input_dim,learning_rate = 1e-
4, mom_coeff = 0.0, hidden_dim = 128)


# %%
model.fit(X_train,y_train,X_test,y_test,epochs = 35)


# %%
history = model.history()


# %%
plt.figure()
plt.plot(history['TestLoss'],'-o')
plt.plot(history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Categorical Cross Entropy over epochs')
plt.legend(['Test Loss','Train Loss'])
plt.show()


# %%
plt.figure()
plt.plot(history['TestAcc'],'-o')
plt.plot(history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Accuracy over epochs')
plt.legend(['Test Acc','Train Acc'])
plt.show()


# %%
train_preds = model.predict(X_train)
test_preds = model.predict(X_test)


# %%
confusion_mat_train = metrics.confusion_matrix(np.argmax(y_train,1),train_pred
s)
confusion_mat_test = metrics.confusion_matrix(np.argmax(y_test,1),test_preds)


# %%
```

```
body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_,confusion_mat_test_ = confusion_mat_train,confusion_mat_t
est
confusion_mat_train.columns = body_movements
confusion_mat_train.index = body_movements
confusion_mat_train


# %%
confusion_mat_test.columns = body_movements
confusion_mat_test.index = body_movements
confusion_mat_train


# %%
sns.heatmap(confusion_mat_train/np.sum(confusion_mat_train), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
confusion_mat_test


# %%
sns.heatmap(confusion_mat_test/np.sum(confusion_mat_test), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
print(classification_report(np.argmax(y_train,1),train_preds))


# %%
print(classification_report(np.argmax(y_test,1),test_preds))


# %%
plt.matshow(confusion_mat_test, cmap=plt.cm.gray_r)
plt.title('Testing Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(len(confusion_mat_test.columns))
plt.xticks(tick_marks, confusion_mat_test.columns, rotation=45)
plt.yticks(tick_marks, confusion_mat_test.index)
plt.tight_layout()
plt.ylabel(confusion_mat_test.index.name)
plt.xlabel(confusion_mat_test.columns.name)
```

```
plt.show()


# %%
plt.matshow(confusion_mat_train, cmap=plt.cm.gray_r)
plt.title('Training Confusion Matrix')
plt.colorbar()
tick_marks = np.arange(len(confusion_mat_train.columns))
plt.xticks(tick_marks, confusion_mat_train.columns, rotation=45)
plt.yticks(tick_marks, confusion_mat_train.index)
plt.tight_layout()
plt.ylabel(confusion_mat_train.index.name)
plt.xlabel(confusion_mat_train.columns.name)
plt.show()


# %%
sns.heatmap(confusion_mat_test/np.sum(confusion_mat_test), annot=True,
            fmt='.2%',cmap = 'Greens')
plt.show()


# %%
sns.heatmap(confusion_mat_test/np.sum(confusion_mat_test), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
class Multi_Layer_RNN(object):
    """
    Recurrent Neural Network for classifying human activity.
    RNN encapsulates all necessary logic for training the network.

    """
    def __init__(self,input_dim = 3,hidden_dim_1 = 128, hidden_dim_2 = 64, seq
_len = 150, learning_rate = 1e-
1, mom_coeff = 0.85, batch_size = 32, output_class = 6):

        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim_1 = hidden_dim_1
        self.hidden_dim_2 = hidden_dim_2
        # Unfold case T = 150 :
```

```
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim_inp2hid = Xavier(self.input_dim,self.hidden_dim_1)
        self.W1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(self.input_dim,self.hidden_dim_1))
        self.B1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.hidden_dim_1))

        lim_hid2hid = Xavier(self.hidden_dim_1,self.hidden_dim_1)
        self.W1_rec= np.random.uniform(-
lim_hid2hid,lim_hid2hid,(self.hidden_dim_1,self.hidden_dim_1))


        lim_hid2hid2 = Xavier(self.hidden_dim_1,self.hidden_dim_2)
        self.W2 = np.random.uniform(-
lim_hid2hid2,lim_hid2hid2,(self.hidden_dim_1,self.hidden_dim_2))
        self.B2 = np.random.uniform(-
lim_hid2hid2,lim_hid2hid2,(1,self.hidden_dim_2))

        lim_hid2out = Xavier(self.hidden_dim_2,self.output_class)
        self.W3 = np.random.uniform(-
lim_hid2out,lim_hid2out,(self.hidden_dim_2,self.output_class))
        self.B3 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # Storing previous momentum updates :
        self.prev_updates = {'W1'       : 0,
                             'B1'       : 0,
                             'W1_rec'   : 0,
                             'W2'       : 0,
                             'B2'       : 0,
                             'W3'       : 0,
                             'B3'       : 0}


    def forward(self,X) -> tuple:
        """
        Forward propagation of the RNN through time.
```

```
        ____________________________________________________________

        Inputs:
        --- X is the bacth.
        --- h_prev_state is the previous state of the hidden layer.
        ____________________________________________________________

        Returns:
        --- (X_state,hidden_state,probs) as a tuple.
        ------ 1) X_state is the input across all time steps
        ------ 2) hidden_state is the hidden stages across time
        -----
- 3) probs is the probabilities of each outputs, i.e. outputs of softmax
        ____________________________________________________________
        """
        X_state = dict()
        hidden_state_1 = dict()
        hidden_state_mlp = dict()
        output_state = dict()
        probs = dict()
        mlp_linear = dict()

        self.h_prev_state = np.zeros((1,self.hidden_dim_1))
        hidden_state_1[-1] = np.copy(self.h_prev_state)

        # Loop over time T = 150 :
        for t in range(self.seq_len):

            # Selecting first record with 3 inputs, dimension = (batch_size,in
put_size)
            X_state[t] = X[:,t]

            # Recurrent hidden layer :
            hidden_state_1[t] = np.tanh(np.dot(X_state[t],self.W1) + np.dot(hi
dden_state_1[t-1],self.W1_rec) + self.B1)
            mlp_linear[t] = np.dot(hidden_state_1[t],self.W2) + self.B2
            hidden_state_mlp[t] = activations.ReLU(mlp_linear[t])
            output_state[t] = np.dot(hidden_state_mlp[t],self.W3) + self.B3

            # Per class probabilites :
            probs[t] = activations.softmax(output_state[t])

        return (X_state,hidden_state_1,mlp_linear,hidden_state_mlp,probs)


    def BPTT(self,cache,Y):
        """
```

```
        Back propagation through time algorihm.
        Inputs:
        -- Cache = (X_state,hidden_state,probs)
        -- Y = desired output

        Returns:
        -- Gradients w.r.t. all configurable elements
        """

        X_state,hidden_state_1,mlp_linear,hidden_state_mlp,probs = cache

        # backward pass: compute gradients going backwards
        dW1, dW1_rec, dW2, dW3 = np.zeros_like(self.W1), np.zeros_like(self.W1
_rec), np.zeros_like(self.W2),np.zeros_like(self.W3)

        dB1, dB2,dB3 = np.zeros_like(self.B1), np.zeros_like(self.B2),np.zeros
_like(self.B3)

        dhnext = np.zeros_like(hidden_state_1[0])

        dy = np.copy(probs[149])
        dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        #dy = probs[0] - Y[0]

        dW3 += np.dot(hidden_state_mlp[149].T,dy)
        dB3 += np.sum(dy,axis = 0, keepdims = True)

        dy1 = np.dot(dy,self.W3.T) * activations.ReLU_grad(mlp_linear[149])

        dB2 += np.sum(dy1,axis = 0, keepdims = True)
        dW2 += np.dot(hidden_state_1[149].T,dy1)


        for t in reversed(range(1,self.seq_len)):




            dh = np.dot(dy1,self.W2.T) + dhnext
            dhrec = (1 - (hidden_state_1[t] * hidden_state_1[t])) * dh

            dB1 += np.sum(dhrec,axis = 0, keepdims = True)
            dW1 += np.dot(X_state[t].T,dhrec)

            dW1_rec += np.dot(hidden_state_1[t-1].T,dhrec)

            dhnext = np.dot(dhrec,self.W1_rec.T)
```

```
        for grad in [dW1,dB1,dW1_rec,dW2,dB2,dW3,dB3]:
            np.clip(grad, -10, 10, out = grad)


        return [dW1,dB1,dW1_rec,dW2,dB2,dW3,dB3]



    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def step(self,grads,momentum = True):


        #for config_param,grad in zip([self.W1,self.B1,self.W1_rec,self.W2,sel
f.B2,self.W3,self.B3],grads):
            #config_param -= self.learning_rate * grad

        if momentum:

            delta_W1 = -
self.learning_rate * grads[0] -  self.mom_coeff * self.prev_updates['W1']
            delta_B1 = -
self.learning_rate * grads[1] -  self.mom_coeff * self.prev_updates['B1']
            delta_W1_rec = -
self.learning_rate * grads[2] -  self.mom_coeff * self.prev_updates['W1_rec']
            delta_W2 = -
self.learning_rate * grads[3] - self.mom_coeff * self.prev_updates['W2']

            delta_B2 = -
self.learning_rate * grads[4] -  self.mom_coeff * self.prev_updates['B2']
            delta_W3 = -
self.learning_rate * grads[5] -  self.mom_coeff * self.prev_updates['W3']
            delta_B3 = -
self.learning_rate * grads[6] -  self.mom_coeff * self.prev_updates['B3']


            self.W1 += delta_W1
            self.W1_rec += delta_W1_rec
            self.W2 += delta_W2
            self.B1 += delta_B1
            self.B2 += delta_B2
```

```
            self.W3 += delta_W3
            self.B3 += delta_B3


            self.prev_updates['W1'] = delta_W1
            self.prev_updates['W1_rec'] = delta_W1_rec
            self.prev_updates['W2'] = delta_W2
            self.prev_updates['B1'] = delta_B1
            self.prev_updates['B2'] = delta_B2
            self.prev_updates['W3'] = delta_W3
            self.prev_updates['B3'] = delta_B3

            self.learning_rate *= 0.9999

    def fit(self,X,Y,X_val,y_val,epochs = 50 ,verbose = True, crossVal = False
):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)

            for i in range(round(X.shape[0]/self.batch_size)):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]

                cache_train = self.forward(X_feed)

                grads = self.BPTT(cache_train,y_feed)
                self.step(grads)

                if crossVal:
                    stop = self.cross_validation(X,val_X,Y,val_Y,threshold = 5
)
                    if stop:
                        break
```

```
            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[4][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            _,__,___,____, probs_test = self.forward(X_val)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)

    def predict(self,X):
        _,__,___,____,probs = self.forward(X)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}



# %%
multilayer_rnn = Multi_Layer_RNN(learning_rate=1e-
4,mom_coeff=0.0,hidden_dim_1 = 128, hidden_dim_2 = 64)
```

```
# %%
multilayer_rnn.fit(X_train,y_train,X_test,y_test,epochs = 35)


# %%
multilayer_rnn_history = multilayer_rnn.history()


# %%
plt.figure()
plt.plot(multilayer_rnn_history['TestLoss'],'-o')
plt.plot(multilayer_rnn_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Categorical Cross Entropy over epochs')
plt.legend(['Test Loss','Train Loss'])
plt.show()


# %%
plt.figure()
plt.plot(multilayer_rnn_history['TestAcc'],'-o')
plt.plot(multilayer_rnn_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Accuracy over epochs')
plt.legend(['Test Acc','Train Acc'])
plt.show()


# %%
plt.figure()
plt.plot(multilayer_rnn_history['TrainAcc'],'-o')
plt.plot(history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['Multi Layer RNN','Vanilla RNN'])
plt.show()


# %%
plt.plot(multilayer_rnn_history['TestAcc'],'-o')
plt.plot(history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['Multi Layer RNN','Vanilla RNN'])
plt.show()
```

```
# %%
train_preds_multilayer_rnn = multilayer_rnn.predict(X_train)
test_preds_multilayer_rnn = multilayer_rnn.predict(X_test)
confusion_mat_train_multilayer_rnn = metrics.confusion_matrix(np.argmax(y_trai
n,1),train_preds_multilayer_rnn)
confusion_mat_test_multilayer_rnn = metrics.confusion_matrix(np.argmax(y_test,
1),test_preds_multilayer_rnn)

body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_multilayer_rnn.columns = body_movements
confusion_mat_train_multilayer_rnn.index = body_movements
confusion_mat_test_multilayer_rnn.columns = body_movements
confusion_mat_test_multilayer_rnn.index = body_movements
confusion_mat_train_multilayer_rnn


# %%
confusion_mat_test_multilayer_rnn


# %%
sns.heatmap(confusion_mat_test_multilayer_rnn/np.sum(confusion_mat_test_multil
ayer_rnn), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
sns.heatmap(confusion_mat_train_multilayer_rnn/np.sum(confusion_mat_train_mult
ilayer_rnn), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
class Three_Hidden_Layer_RNN(object):
    """
    Recurrent Neural Network for classifying human activity.
    RNN encapsulates all necessary logic for training the network.

    """
    def __init__(self,input_dim = 3,hidden_dim_1 = 128, hidden_dim_2 = 64,hidd
en_dim_3 = 32, seq_len = 150, learning_rate = 1e-
1, mom_coeff = 0.85, batch_size = 32, output_class = 6):

        """
```

```
        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim_1 = hidden_dim_1
        self.hidden_dim_2 = hidden_dim_2
        self.hidden_dim_3 = hidden_dim_3

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim_inp2hid = Xavier(self.input_dim,self.hidden_dim_1)
        self.W1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(self.input_dim,self.hidden_dim_1))
        self.B1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.hidden_dim_1))

        lim_hid2hid = Xavier(self.hidden_dim_1,self.hidden_dim_1)
        self.W1_rec= np.random.uniform(-
lim_hid2hid,lim_hid2hid,(self.hidden_dim_1,self.hidden_dim_1))


        lim_hid2hid2 = Xavier(self.hidden_dim_1,self.hidden_dim_2)
        self.W2 = np.random.uniform(-
lim_hid2hid2,lim_hid2hid2,(self.hidden_dim_1,self.hidden_dim_2))
        self.B2 = np.random.uniform(-
lim_hid2hid2,lim_hid2hid2,(1,self.hidden_dim_2))

        lim_hid2hid3 = Xavier(self.hidden_dim_2,self.hidden_dim_3)
        self.W3 = np.random.uniform(-
lim_hid2hid3,lim_hid2hid3,(self.hidden_dim_2,self.hidden_dim_3))
        self.B3 = np.random.uniform(-
lim_hid2hid3,lim_hid2hid3,(1,self.hidden_dim_3))

        lim_hid2out = Xavier(self.hidden_dim_3,self.output_class)
        self.W4 = np.random.uniform(-
lim_hid2out,lim_hid2out,(self.hidden_dim_3,self.output_class))
        self.B4 = np.random.uniform(-
lim_hid2out,lim_hid2out,(1,self.output_class))
```

```
        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # Storing previous momentum updates :
        self.prev_updates = {'W1'       : 0,
                             'B1'       : 0,
                             'W1_rec'   : 0,
                             'W2'       : 0,
                             'B2'       : 0,
                             'W3'       : 0,
                             'W4'       : 0,
                             'B3'       : 0,
                             'B4'       : 0}


    def forward(self,X) -> tuple:
        """
        Forward propagation of the RNN through time.
        ____________________________________________________________

        Inputs:
        --- X is the bacth.
        --- h_prev_state is the previous state of the hidden layer.
        ____________________________________________________________

        Returns:
        --- (X_state,hidden_state,probs) as a tuple.
        ------ 1) X_state is the input across all time steps
        ------ 2) hidden_state is the hidden stages across time
        -----
- 3) probs is the probabilities of each outputs, i.e. outputs of softmax
        ____________________________________________________________
        """
        X_state = dict()
        hidden_state_1 = dict()
        hidden_state_mlp = dict()
        hidden_state_mlp_2 = dict()
        output_state = dict()
        probs = dict()
        mlp_linear = dict()
        mlp_linear_2 = dict()

        self.h_prev_state = np.zeros((1,self.hidden_dim_1))
        hidden_state_1[-1] = np.copy(self.h_prev_state)

        # Loop over time T = 150 :
```

```
        for t in range(self.seq_len):

            # Selecting first record with 3 inputs, dimension = (batch_size,in
put_size)
            X_state[t] = X[:,t]

            # Recurrent hidden layer :
            hidden_state_1[t] = np.tanh(np.dot(X_state[t],self.W1) + np.dot(hi
dden_state_1[t-1],self.W1_rec) + self.B1)
            mlp_linear[t] = np.dot(hidden_state_1[t],self.W2) + self.B2
            hidden_state_mlp[t] = activations.ReLU(mlp_linear[t])
            mlp_linear_2[t] = np.dot(hidden_state_mlp[t],self.W3) + self.B3
            hidden_state_mlp_2[t] = activations.ReLU(mlp_linear_2[t])
            output_state[t] = np.dot(hidden_state_mlp_2[t],self.W4) + self.B4

            # Per class probabilites :
            probs[t] = activations.softmax(output_state[t])

        return (X_state,hidden_state_1,mlp_linear,hidden_state_mlp,mlp_linear_
2,hidden_state_mlp_2,probs)


    def BPTT(self,cache,Y):
        """

        Back propagation through time algorihm.
        Inputs:
        -- Cache = (X_state,hidden_state,probs)
        -- Y = desired output

        Returns:
        -- Gradients w.r.t. all configurable elements
        """

        X_state,hidden_state_1,mlp_linear,hidden_state_mlp,mlp_linear_2,hidden
_state_mlp_2,probs = cache

        # backward pass: compute gradients going backwards
        dW1, dW1_rec, dW2, dW3, dW4 = np.zeros_like(self.W1), np.zeros_like(se
lf.W1_rec), np.zeros_like(self.W2),np.zeros_like(self.W3),np.zeros_like(self.W
4)

        dB1, dB2,dB3,dB4 = np.zeros_like(self.B1), np.zeros_like(self.B2),np.z
eros_like(self.B3),np.zeros_like(self.B4)

        dhnext = np.zeros_like(hidden_state_1[0])

        for t in reversed(range(1,self.seq_len)):
```

```
            dy = np.copy(probs[t])
            dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1
            #dy = probs[0] - Y[0]

            dW4 += np.dot(hidden_state_mlp_2[t].T,dy)
            dB4 += np.sum(dy,axis = 0, keepdims = True)

            dy1 = np.dot(dy,self.W4.T) * activations.ReLU_grad(mlp_linear_2[t]
)


            dW3 += np.dot(hidden_state_mlp[t].T,dy1)
            dB3 += np.sum(dy1,axis = 0, keepdims = True)

            dy2 = np.dot(dy1,self.W3.T) * activations.ReLU_grad(mlp_linear[t])

            dB2 += np.sum(dy2,axis = 0, keepdims = True)
            dW2 += np.dot(hidden_state_1[t].T,dy2)

            dh = np.dot(dy2,self.W2.T) + dhnext
            dhrec = (1 - (hidden_state_1[t] * hidden_state_1[t])) * dh

            dB1 += np.sum(dhrec,axis = 0, keepdims = True)
            dW1 += np.dot(X_state[t].T,dhrec)

            dW1_rec += np.dot(hidden_state_1[t-1].T,dhrec)

            dhnext = np.dot(dhrec,self.W1_rec.T)


        for grad in [dW1,dB1,dW1_rec,dW2,dB2,dW3,dB3,dW4,dB4]:
            np.clip(grad, -10, 10, out = grad)


        return [dW1,dB1,dW1_rec,dW2,dB2,dW3,dB3,dW4,dB4]



    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def step(self,grads,momentum = True):
```

```
        #for config_param,grad in zip([self.W1,self.B1,self.W1_rec,self.W2,sel
f.B2,self.W3,self.B3],grads):
            #config_param -= self.learning_rate * grad

        if momentum:

            delta_W1 = -
self.learning_rate * grads[0] +  self.mom_coeff * self.prev_updates['W1']
            delta_B1 = -
self.learning_rate * grads[1] +  self.mom_coeff * self.prev_updates['B1']
            delta_W1_rec = -
self.learning_rate * grads[2] +  self.mom_coeff * self.prev_updates['W1_rec']
            delta_W2 = -
self.learning_rate * grads[3] +  self.mom_coeff * self.prev_updates['W2']

            delta_B2 = -
self.learning_rate * grads[4] +  self.mom_coeff * self.prev_updates['B2']
            delta_W3 = -
self.learning_rate * grads[5] +  self.mom_coeff * self.prev_updates['W3']

            delta_B3 = -
self.learning_rate * grads[6] +  self.mom_coeff * self.prev_updates['B3']
            delta_W4 = -
self.learning_rate * grads[7] +  self.mom_coeff * self.prev_updates['W4']

            delta_B4 = -
self.learning_rate * grads[8] +  self.mom_coeff * self.prev_updates['B4']


            self.W1 += delta_W1
            self.W1_rec += delta_W1_rec
            self.W2 += delta_W2
            self.B1 += delta_B1
            self.B2 += delta_B2
            self.W3 += delta_W3
            self.B3 += delta_B3
            self.W4 += delta_W4
            self.B4 += delta_B4


            self.prev_updates['W1'] = delta_W1
            self.prev_updates['W1_rec'] = delta_W1_rec
            self.prev_updates['W2'] = delta_W2
            self.prev_updates['B1'] = delta_B1
            self.prev_updates['B2'] = delta_B2
            self.prev_updates['W3'] = delta_W3
            self.prev_updates['B3'] = delta_B3
```

```
            self.prev_updates['W4'] = delta_W4
            self.prev_updates['B4'] = delta_B4

            self.learning_rate *= 0.9999

    def fit(self,X,Y,X_val,y_val,epochs = 50 ,verbose = True, crossVal = False
):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)

            for i in range(round(X.shape[0]/self.batch_size)):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]

                cache_train = self.forward(X_feed)

                grads = self.BPTT(cache_train,y_feed)
                self.step(grads)

                if crossVal:
                    stop = self.cross_validation(X,val_X,Y,val_Y,threshold = 5
)
                    if stop:
                        break

            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[6][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            _,__,___,____,_____,______, probs_test = self.forward(X_val)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
```

```
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)

    def predict(self,X):
        _,__,___,____,_____,______,probs = self.forward(X)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}



# %%
three_layer_rnn = Three_Hidden_Layer_RNN(hidden_dim_1 = 128, hidden_dim_2 = 64
,hidden_dim_3 = 32, learning_rate = 1e-
4, mom_coeff = 0.0, batch_size = 32, output_class = 6)


# %%
three_layer_rnn.fit(X_train,y_train,X_test,y_test,epochs=15)


# %%
three_layer_rnn_v1 = Three_Hidden_Layer_RNN(hidden_dim_1 = 128, hidden_dim_2 =
 64,hidden_dim_3 = 32, learning_rate = 5e-
5, mom_coeff = 0.0, batch_size = 32, output_class = 6)
```

```
three_layer_rnn_v1.fit(X_train,y_train,X_test,y_test,epochs=15)


# %%
three_layer_rnn_v2 = Three_Hidden_Layer_RNN(hidden_dim_1 = 128, hidden_dim_2 =
 64,hidden_dim_3 = 32, learning_rate = 1e-
4, mom_coeff = 0.0, batch_size = 32, output_class = 6)
three_layer_rnn_v2.fit(X_train,y_train,X_test,y_test,epochs=15)


# %%
three_layer_rnn_history = three_layer_rnn.history()
plt.figure()
plt.plot(three_layer_rnn_history['TestLoss'],'-o')
plt.plot(three_layer_rnn_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Categorical Cross Entropy over epochs')
plt.legend(['Test Loss','Train Loss'])
plt.show()


# %%
plt.figure()
plt.plot(three_layer_rnn_history['TestAcc'],'-o')
plt.plot(three_layer_rnn_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Accuracy over epochs')
plt.legend(['Test Acc','Train Acc'])
plt.show()


# %%
plt.figure()
plt.plot(three_layer_rnn_history['TrainAcc'],'-o')
plt.plot(multilayer_rnn_history['TrainAcc'],'-o')
plt.plot(history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['3 hidden layer Rnn','Multi Layer RNN','Vanilla RNN'])
plt.show()


# %%
plt.figure()
plt.plot(three_layer_rnn_history['TestAcc'],'-o')
plt.plot(multilayer_rnn_history['TestAcc'],'-o')
```

```
plt.plot(history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['3 hidden layer Rnn','Multi Layer RNN','Vanilla RNN'])
plt.show()


# %%
train_preds_three_layer_rnn_history = three_layer_rnn.predict(X_train)
test_preds_three_layer_rnn_history = three_layer_rnn.predict(X_test)
confusion_mat_train_three_layer_rnn_history = metrics.confusion_matrix(np.argm
ax(y_train,1),train_preds_three_layer_rnn_history)
confusion_mat_test_three_layer_rnn_history = metrics.confusion_matrix(np.argma
x(y_test,1),test_preds_three_layer_rnn_history)

body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_three_layer_rnn_history.columns = body_movements
confusion_mat_train_three_layer_rnn_history.index = body_movements
confusion_mat_test_three_layer_rnn_history.columns = body_movements
confusion_mat_test_three_layer_rnn_history.index = body_movements
confusion_mat_train_three_layer_rnn_history


# %%
sns.heatmap(confusion_mat_test_three_layer_rnn_history/np.sum(confusion_mat_te
st_three_layer_rnn_history), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
sns.heatmap(confusion_mat_train_three_layer_rnn_history/np.sum(confusion_mat_t
rain_three_layer_rnn_history), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%

# %%
class Five_Hidden_Layer_RNN(object):
    """
    Recurrent Neural Network for classifying human activity.
    RNN encapsulates all necessary logic for training the network.

    """
```

```
    def __init__(self,input_dim = 3,hidden_dim_1 = 128, hidden_dim_2 = 64,hidd
en_dim_3 = 32,hidden_dim_4 = 16 ,hidden_dim_5 = 8, seq_len = 150, learning_rat
e = 1e-1, mom_coeff = 0.85, batch_size = 32, output_class = 6):

        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim_1 = hidden_dim_1
        self.hidden_dim_2 = hidden_dim_2
        self.hidden_dim_3 = hidden_dim_3
        self.hidden_dim_4 = hidden_dim_4
        self.hidden_dim_5 = hidden_dim_5

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim_inp2hid = Xavier(self.input_dim,self.hidden_dim_1)
        self.W1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(self.input_dim,self.hidden_dim_1))
        self.B1 = np.random.uniform(-
lim_inp2hid,lim_inp2hid,(1,self.hidden_dim_1))

        lim_hid2hid = Xavier(self.hidden_dim_1,self.hidden_dim_1)
        self.W1_rec= np.random.uniform(-
lim_hid2hid,lim_hid2hid,(self.hidden_dim_1,self.hidden_dim_1))


        lim_hid2hid2 = Xavier(self.hidden_dim_1,self.hidden_dim_2)
        self.W2 = np.random.uniform(-
lim_hid2hid2,lim_hid2hid2,(self.hidden_dim_1,self.hidden_dim_2))
        self.B2 = np.random.uniform(-
lim_hid2hid2,lim_hid2hid2,(1,self.hidden_dim_2))

        lim_hid2hid3 = Xavier(self.hidden_dim_2,self.hidden_dim_3)
        self.W3 = np.random.uniform(-
lim_hid2hid3,lim_hid2hid3,(self.hidden_dim_2,self.hidden_dim_3))
```

```
        self.B3 = np.random.uniform(-
lim_hid2hid3,lim_hid2hid3,(1,self.hidden_dim_3))

        lim_hid2hid4 = Xavier(self.hidden_dim_3,self.hidden_dim_4)
        self.W4 = np.random.uniform(-
lim_hid2hid4,lim_hid2hid4,(self.hidden_dim_3,self.hidden_dim_4))
        self.B4 = np.random.uniform(-
lim_hid2hid4,lim_hid2hid4,(1,self.hidden_dim_4))

        lim_hid2hid5 = Xavier(self.hidden_dim_4,self.hidden_dim_5)
        self.W5 = np.random.uniform(-
lim_hid2hid5,lim_hid2hid5,(self.hidden_dim_4,self.hidden_dim_5))
        self.B5 = np.random.uniform(-
lim_hid2hid5,lim_hid2hid5,(1,self.hidden_dim_5))


        lim_hid2out = Xavier(self.hidden_dim_5,self.output_class)
        self.W6 = np.random.uniform(-
lim_hid2out,lim_hid2out,(self.hidden_dim_5,self.output_class))
        self.B6 = np.random.uniform(-
lim_hid2out,lim_hid2out,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # Storing previous momentum updates :
        self.prev_updates = {'W1'       : 0,
                             'B1'       : 0,
                             'W1_rec'   : 0,
                             'W2'       : 0,
                             'B2'       : 0,
                             'W3'       : 0,
                             'W4'       : 0,
                             'B3'       : 0,
                             'B4'       : 0,
                             'W5'       : 0,
                             'W6'       : 0,
                             'B5'       : 0,
                             'B6'       : 0}


    def forward(self,X) -> tuple:
        """
        Forward propagation of the RNN through time.
        ---------------------------------------------------------------

        Inputs:
        --- X is the bacth.
```

```
        --- h_prev_state is the previous state of the hidden layer.
        ____________________________________________________________

        Returns:
        --- (X_state,hidden_state,probs) as a tuple.
        ------ 1) X_state is the input across all time steps
        ------ 2) hidden_state is the hidden stages across time
        -----
- 3) probs is the probabilities of each outputs, i.e. outputs of softmax
        ____________________________________________________________
        """
        X_state = dict()
        hidden_state_1 = dict()
        hidden_state_mlp = dict()
        hidden_state_mlp_2 = dict()
        hidden_state_mlp_3 = dict()
        hidden_state_mlp_4 = dict()
        output_state = dict()
        probs = dict()
        mlp_linear = dict()
        mlp_linear_2 = dict()
        mlp_linear_3 = dict()
        mlp_linear_4 = dict()

        self.h_prev_state = np.zeros((1,self.hidden_dim_1))
        hidden_state_1[-1] = np.copy(self.h_prev_state)

        # Loop over time T = 150 :
        for t in range(self.seq_len):

            # Selecting first record with 3 inputs, dimension = (batch_size,in
put_size)
            X_state[t] = X[:,t]

            # Recurrent hidden layer :
            hidden_state_1[t] = np.tanh(np.dot(X_state[t],self.W1) + np.dot(hi
dden_state_1[t-1],self.W1_rec) + self.B1)
            mlp_linear[t] = np.dot(hidden_state_1[t],self.W2) + self.B2

            hidden_state_mlp[t] = activations.ReLU(mlp_linear[t])

            mlp_linear_2[t] = np.dot(hidden_state_mlp[t],self.W3) + self.B3
            hidden_state_mlp_2[t] = activations.ReLU(mlp_linear_2[t])

            mlp_linear_3[t] = np.dot(hidden_state_mlp_2[t],self.W4) + self.B4
            hidden_state_mlp_3[t] = activations.ReLU(mlp_linear_3[t])

            mlp_linear_4[t] = np.dot(hidden_state_mlp_3[t],self.W5) + self.B5
```

```
            hidden_state_mlp_4[t] = activations.ReLU(mlp_linear_4[t])

            output_state[t] = np.dot(hidden_state_mlp_4[t],self.W6) + self.B6

            # Per class probabilites :
            probs[t] = activations.softmax(output_state[t])

        return (X_state,hidden_state_1,mlp_linear,hidden_state_mlp,mlp_linear_
2,hidden_state_mlp_2,mlp_linear_3,hidden_state_mlp_3,mlp_linear_4,hidden_state
_mlp_4,probs)


    def BPTT(self,cache,Y):
        """

        Back propagation through time algorihm.
        Inputs:
        -
- Cache = (X_state,hidden_state_1,mlp_linear,hidden_state_mlp,mlp_linear_2,hid
den_state_mlp_2,mlp_linear_3,hidden_state_mlp_3,mlp_linear_4,hidden_state_mlp_
4,probs)
        -- Y = desired output

        Returns:
        -- Gradients w.r.t. all configurable elements
        """

        X_state,hidden_state_1,mlp_linear,hidden_state_mlp,mlp_linear_2,hidden
_state_mlp_2,mlp_linear_3,hidden_state_mlp_3,mlp_linear_4,hidden_state_mlp_4,p
robs = cache

        # backward pass: compute gradients going backwards
        dW1, dW1_rec, dW2, dW3, dW4, dW5, dW6 = np.zeros_like(self.W1), np.zer
os_like(self.W1_rec), np.zeros_like(self.W2),np.zeros_like(self.W3),np.zeros_l
ike(self.W4),np.zeros_like(self.W5),np.zeros_like(self.W6)

        dB1, dB2,dB3,dB4,dB5,dB6 = np.zeros_like(self.B1), np.zeros_like(self.
B2),np.zeros_like(self.B3),np.zeros_like(self.B4),np.zeros_like(self.B5),np.ze
ros_like(self.B6)

        dhnext = np.zeros_like(hidden_state_1[0])


        for t in reversed(range(1,self.seq_len)):

            dy = np.copy(probs[149])
            dy[np.arange(len(Y)),np.argmax(Y,1)] -= 1
            #dy = probs[0] - Y[0]
```

```
            dW6 += np.dot(hidden_state_mlp_4[t].T,dy)
            dB6 += np.sum(dy,axis = 0, keepdims = True)

            dy1 = np.dot(dy,self.W6.T) * activations.ReLU_grad(mlp_linear_4[t]
)

            dW5 += np.dot(hidden_state_mlp_3[t].T,dy1)
            dB5 += np.sum(dy1,axis = 0, keepdims = True)

            dy2 = np.dot(dy1,self.W5.T) * activations.ReLU_grad(mlp_linear_3[t
])

            dW4 += np.dot(hidden_state_mlp_2[t].T,dy2)
            dB4 += np.sum(dy2,axis = 0, keepdims = True)

            dy3 = np.dot(dy2,self.W4.T) * activations.ReLU_grad(mlp_linear_2[t
])

            dW3 += np.dot(hidden_state_mlp[t].T,dy3)
            dB3 += np.sum(dy3,axis = 0, keepdims = True)

            dy4 = np.dot(dy3,self.W3.T) * activations.ReLU_grad(mlp_linear[t])

            dB2 += np.sum(dy4,axis = 0, keepdims = True)
            dW2 += np.dot(hidden_state_1[t].T,dy4)

            dh = np.dot(dy4,self.W2.T) + dhnext
            dhrec = (1 - (hidden_state_1[t] * hidden_state_1[t])) * dh

            dB1 += np.sum(dhrec,axis = 0, keepdims = True)
            dW1 += np.dot(X_state[t].T,dhrec)

            dW1_rec += np.dot(hidden_state_1[t-1].T,dhrec)

            dhnext = np.dot(dhrec,self.W1_rec.T)


        for grad in [dW1,dB1,dW1_rec,dW2,dB2,dW3,dB3,dW4,dB4,dW5,dB5,dW6,dB6]:
            np.clip(grad, -10, 10, out = grad)


        return [dW1,dB1,dW1_rec,dW2,dB2,dW3,dB3,dW4,dB4,dW5,dB5,dW6,dB6]



    def CategoricalCrossEntropy(self,labels,preds):
        """
```

```
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def step(self,grads,momentum = True):


        #for config_param,grad in zip([self.W1,self.B1,self.W1_rec,self.W2,sel
f.B2,self.W3,self.B3],grads):
            #config_param -= self.learning_rate * grad

        if momentum:

            delta_W1 = -
self.learning_rate * grads[0] +  self.mom_coeff * self.prev_updates['W1']
            delta_B1 = -
self.learning_rate * grads[1] +  self.mom_coeff * self.prev_updates['B1']
            delta_W1_rec = -
self.learning_rate * grads[2] +  self.mom_coeff * self.prev_updates['W1_rec']
            delta_W2 = -
self.learning_rate * grads[3] +  self.mom_coeff * self.prev_updates['W2']

            delta_B2 = -
self.learning_rate * grads[4] +  self.mom_coeff * self.prev_updates['B2']
            delta_W3 = -
self.learning_rate * grads[5] +  self.mom_coeff * self.prev_updates['W3']

            delta_B3 = -
self.learning_rate * grads[6] +  self.mom_coeff * self.prev_updates['B3']
            delta_W4 = -
self.learning_rate * grads[7] +  self.mom_coeff * self.prev_updates['W4']

            delta_B4 = -
self.learning_rate * grads[8] +  self.mom_coeff * self.prev_updates['B4']
            delta_W5 = -
self.learning_rate * grads[9] +  self.mom_coeff * self.prev_updates['W5']

            delta_B5 = -
self.learning_rate * grads[10] +  self.mom_coeff * self.prev_updates['B5']
            delta_W6 = -
self.learning_rate * grads[11] +  self.mom_coeff * self.prev_updates['W6']

            delta_B6 = -
self.learning_rate * grads[12] +  self.mom_coeff * self.prev_updates['B6']
```

```
            self.W1 += delta_W1
            self.W1_rec += delta_W1_rec
            self.W2 += delta_W2
            self.B1 += delta_B1
            self.B2 += delta_B2
            self.W3 += delta_W3
            self.B3 += delta_B3
            self.W4 += delta_W4
            self.B4 += delta_B4
            self.W5 += delta_W5
            self.B5 += delta_B5
            self.W6 += delta_W6
            self.B6 += delta_B6


            self.prev_updates['W1'] = delta_W1
            self.prev_updates['W1_rec'] = delta_W1_rec
            self.prev_updates['W2'] = delta_W2
            self.prev_updates['B1'] = delta_B1
            self.prev_updates['B2'] = delta_B2
            self.prev_updates['W3'] = delta_W3
            self.prev_updates['B3'] = delta_B3
            self.prev_updates['W4'] = delta_W4
            self.prev_updates['B4'] = delta_B4
            self.prev_updates['W5'] = delta_W5
            self.prev_updates['B5'] = delta_B5
            self.prev_updates['W6'] = delta_W6
            self.prev_updates['B6'] = delta_B6

            self.learning_rate *= 0.9999

    def fit(self,X,Y,X_val,y_val,epochs = 50 ,verbose = True, crossVal = False
):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)

            for i in range(round(X.shape[0]/self.batch_size)):
```

```
                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]

                cache_train = self.forward(X_feed)

                grads = self.BPTT(cache_train,y_feed)
                self.step(grads)

                if crossVal:
                    stop = self.cross_validation(X,val_X,Y,val_Y,threshold = 5
)
                    if stop:
                        break

            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[10][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            _,__,___,____,_____,______,_______,________,_________,__________
, probs_test = self.forward(X_val)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
```

```
            self.test_acc.append(acc_val)

    def predict(self,X):
        _,__,___,____,_____,______,_______,________,_________,__________, pr
obs = self.forward(X)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}



# %%
five_hidden_layer_rnn = Five_Hidden_Layer_RNN(hidden_dim_1 = 128, hidden_dim_2
 = 64,hidden_dim_3 = 32,hidden_dim_4 = 16 ,hidden_dim_5 = 8,  learning_rate =
1e-4, mom_coeff = 0.0)


# %%
five_hidden_layer_rnn.fit(X_train,y_train,X_test,y_test,epochs = 35)


# %%
five_hidden_layer_rnn_history = five_hidden_layer_rnn.history()
plt.figure()
plt.plot(five_hidden_layer_rnn_history['TestLoss'],'-o')
plt.plot(five_hidden_layer_rnn_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Categorical Cross Entropy over epochs')
plt.legend(['Test Loss','Train Loss'])
plt.show()


# %%
plt.figure()
plt.plot(five_hidden_layer_rnn_history['TestAcc'],'-o')
plt.plot(five_hidden_layer_rnn_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Accuracy over epochs')
plt.legend(['Test Acc','Train Acc'])
plt.show()


# %%
```

```
plt.figure()
plt.plot(five_hidden_layer_rnn_history['TrainAcc'],'-o')
plt.plot(three_layer_rnn_history['TrainAcc'],'-o')
plt.plot(multilayer_rnn_history['TrainAcc'],'-o')
plt.plot(history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['Five hidden layer RNN','3 hidden layer RNN','Multi Layer RNN','Va
nilla RNN'])
plt.show()


# %%
plt.figure()
plt.plot(five_hidden_layer_rnn_history['TestAcc'],'-o')
plt.plot(three_layer_rnn_history['TestAcc'],'-o')
plt.plot(multilayer_rnn_history['TestAcc'],'-o')
plt.plot(history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['Five hidden layer RNN','3 hidden layer RNN','Multi Layer RNN','Va
nilla RNN'])
plt.show()


# %%
train_preds_five_hidden_layer_rnn = five_hidden_layer_rnn.predict(X_train)
test_preds_five_hidden_layer_rnn = five_hidden_layer_rnn.predict(X_test)
confusion_mat_train_five_hidden_layer_rnn = metrics.confusion_matrix(np.argmax
(y_train,1),train_preds_five_hidden_layer_rnn)
confusion_mat_test_five_hidden_layer_rnn = metrics.confusion_matrix(np.argmax(
y_test,1),test_preds_five_hidden_layer_rnn)

body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_five_hidden_layer_rnn.columns = body_movements
confusion_mat_train_five_hidden_layer_rnn.index = body_movements
confusion_mat_test_five_hidden_layer_rnn.columns = body_movements
confusion_mat_test_five_hidden_layer_rnn.index = body_movements
confusion_mat_test_five_hidden_layer_rnn


# %%
sns.heatmap(confusion_mat_test_five_hidden_layer_rnn/np.sum(confusion_mat_test
_five_hidden_layer_rnn), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()
```

```
# %%
sns.heatmap(confusion_mat_train_five_hidden_layer_rnn/np.sum(confusion_mat_tra
in_five_hidden_layer_rnn), annot=True,
            fmt='.2%',cmap = 'Blues')
plt.show()

# %% [markdown]
# LSTM

# %%
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


def dsigmoid(y):
    return y * (1 - y)


def tanh(x):
    return np.tanh(x)


def dtanh(y):
    return 1 - y * y


# %%
class LSTM(object):
    """

    Long-
Short Term Memory Recurrent neural network, encapsulates all necessary logic f
or training, then built the hyperparameters and architecture of the network.
    """

    def __init__(self,input_dim = 3,hidden_dim = 100,output_class = 6,seq_len
= 150,batch_size = 30,learning_rate = 1e-1,mom_coeff = 0.85):
        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Unfold case T = 150 :
```

```
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        self.input_stack_dim = self.input_dim + self.hidden_dim

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim1 = Xavier(self.input_dim,self.hidden_dim)
        self.W_f = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim))
        self.B_f = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))

        self.W_i = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim))
        self.B_i = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))

        self.W_c = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim))
        self.B_c = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))

        self.W_o = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim))
        self.B_o = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))

        lim2 = Xavier(self.hidden_dim,self.output_class)
        self.W = np.random.uniform(-
lim2,lim2,(self.hidden_dim,self.output_class))
        self.B = np.random.uniform(-lim2,lim2,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # To keep previous updates in momentum :
        self.previous_updates = [0] * 10

        # For AdaGrad:
        self.cache = [0] * 10
        self.cache_rmsprop = [0] * 10
        self.m = [0] * 10
        self.v = [0] * 10
        self.t = 1

    def cell_forward(self,X,h_prev,C_prev):
```

```
        """

        Takes input, previous hidden state and previous cell state, compute:
        --- Forget gate + Input gate + New candidate input + New cell state +
            output gate + hidden state. Then, classify by softmax.
        """
        #print(X.shape,h_prev.shape)
        # Stacking previous hidden state vector with inputs:
        stack = np.column_stack([X,h_prev])

        # Forget gate:
        forget_gate = activations.sigmoid(np.dot(stack,self.W_f) + self.B_f)

        # İnput gate:
        input_gate = activations.sigmoid(np.dot(stack,self.W_i) + self.B_i)

        # New candidate:
        cell_bar = np.tanh(np.dot(stack,self.W_c) + self.B_c)

        # New Cell state:
        cell_state = forget_gate * C_prev + input_gate * cell_bar

        # Output fate:
        output_gate = activations.sigmoid(np.dot(stack,self.W_o) + self.B_o)

        # Hidden state:
        hidden_state = output_gate * np.tanh(cell_state)

        # Classifiers (Softmax) :
        dense = np.dot(hidden_state,self.W) + self.B
        probs = activations.softmax(dense)

        return (stack,forget_gate,input_gate,cell_bar,cell_state,output_gate,h
idden_state,dense,probs)



    def forward(self,X,h_prev,C_prev):
        x_s,z_s,f_s,i_s = {},{},{},{}
        C_bar_s,C_s,o_s,h_s = {},{},{},{}
        v_s,y_s = {},{}


        h_s[-1] = np.copy(h_prev)
        C_s[-1] = np.copy(C_prev)

        for t in range(self.seq_len):
            x_s[t] = X[:,t,:]
```

```
            z_s[t], f_s[t], i_s[t], C_bar_s[t], C_s[t], o_s[t], h_s[t],v_s[t],
 y_s[t] = self.cell_forward(x_s[t],h_s[t-1],C_s[t-1])

        return (z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s,v_s, y_s)

    def BPTT(self,outs,Y):

        z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s,v_s, y_s = outs

        dW_f, dW_i,dW_c, dW_o,dW = np.zeros_like(self.W_f), np.zeros_like(self
.W_i), np.zeros_like(self.W_c),np.zeros_like(self.W_o),np.zeros_like(self.W)

        dB_f, dB_i,dB_c,dB_o,dB = np.zeros_like(self.B_f), np.zeros_like(self.
B_i),np.zeros_like(self.B_c),np.zeros_like(self.B_o),np.zeros_like(self.B)

        dh_next = np.zeros_like(h_s[0])
        dC_next = np.zeros_like(C_s[0])

        # w.r.t. softmax input
        ddense = np.copy(y_s[149])
        ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        #ddense[np.argmax(Y,1)] -=1
        #ddense = y_s[149] - Y
        # Softmax classifier's :
        dW = np.dot(h_s[149].T,ddense)
        dB = np.sum(ddense,axis = 0, keepdims = True)

        # Backprop through time:
        for t in reversed(range(1,self.seq_len)):

            # Just equating more meaningful names
            stack,forget_gate,input_gate,cell_bar,cell_state,output_gate,hidde
n_state,dense,probs = z_s[t], f_s[t], i_s[t], C_bar_s[t], C_s[t], o_s[t], h_s[
t],v_s[t], y_s[t]
            C_prev = C_s[t-1]

            # w.r.t. softmax input
            #ddense = np.copy(probs)
            #ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
            #ddense[np.arange(len(Y)),np.argmax(Y,1)] -=1
            # Softmax classifier's :
            #dW += np.dot(hidden_state.T,ddense)
            #dB += np.sum(ddense,axis = 0, keepdims = True)

            # Output gate :
            dh = np.dot(ddense,self.W.T) + dh_next
            do = dh * np.tanh(cell_state)
            do = do * dsigmoid(output_gate)
```

```
            dW_o += np.dot(stack.T,do)
            dB_o += np.sum(do,axis = 0, keepdims = True)

            # Cell state:
            dC = np.copy(dC_next)
            dC += dh * output_gate * activations.dtanh(cell_state)
            dC_bar = dC * input_gate
            dC_bar = dC_bar * dtanh(cell_bar)
            dW_c += np.dot(stack.T,dC_bar)
            dB_c += np.sum(dC_bar,axis = 0, keepdims = True)

            # Input gate:
            di = dC * cell_bar
            di = dsigmoid(input_gate) * di
            dW_i += np.dot(stack.T,di)
            dB_i += np.sum(di,axis = 0,keepdims = True)

            # Forget gate:
            df = dC * C_prev
            df = df * dsigmoid(forget_gate)
            dW_f += np.dot(stack.T,df)
            dB_f += np.sum(df,axis = 0, keepdims = True)

            dz = np.dot(df,self.W_f.T) + np.dot(di,self.W_i.T) + np.dot(dC_bar
,self.W_c.T) + np.dot(do,self.W_o.T)

            dh_next = dz[:,-self.hidden_dim:]
            dC_next = forget_gate * dC

        # List of gradients :
        grads = [dW,dB,dW_o,dB_o,dW_c,dB_c,dW_i,dB_i,dW_f,dB_f]

        # Clipping gradients anyway
        for grad in grads:
            np.clip(grad, -15, 15, out = grad)

        return h_s[self.seq_len - 1],C_s[self.seq_len -1 ],grads



    def fit(self,X,Y,X_val,y_val,epochs = 50 ,optimizer = 'SGD',verbose = True
, crossVal = False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """
```

```
        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)
            h_prev,C_prev = np.zeros((self.batch_size,self.hidden_dim)),np.zer
os((self.batch_size,self.hidden_dim))
            for i in range(round(X.shape[0]/self.batch_size) - 1):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                # Feeding random indexes:
                X_feed = X[index]
                y_feed = Y[index]

                # Forward + BPTT + SGD:
                cache_train = self.forward(X_feed,h_prev,C_prev)
                h,c,grads = self.BPTT(cache_train,y_feed)

                if optimizer == 'SGD':

                  self.SGD(grads)

                elif optimizer == 'AdaGrad' :
                  self.AdaGrad(grads)

                elif optimizer == 'RMSprop':
                  self.RMSprop(grads)

                elif optimizer == 'VanillaAdam':
                  self.VanillaAdam(grads)
                else:
                  self.Adam(grads)

                # Hidden state --------> Previous hidden state
                # Cell state ----------> Previous cell state
                h_prev,C_prev = h,c

            # Training metrics calculations:
            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[8][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            # Validation metrics calculations:
```

```
            test_prevs = np.zeros((X_val.shape[0],self.hidden_dim))
            _,__,___,____,_____,______,_______,________,probs_test = self.forw
ard(X_val,test_prevs,test_prevs)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)


    def params(self):
        """
        Return all weights/biases in sequential order starting from end in lis
t form.

        """
        return [self.W,self.B,self.W_o,self.B_o,self.W_c,self.B_c,self.W_i,sel
f.B_i,self.W_f,self.B_f]


    def SGD(self,grads):
      """

      Stochastic gradient descent with momentum on mini-batches.
      """
      prevs = []
      for param,grad,prev_update in zip(self.params(),grads,self.previous_upda
tes):
          delta = self.learning_rate * grad - self.mom_coeff * prev_update
```

```
          param -= delta
          prevs.append(delta)

      self.previous_updates = prevs

      self.learning_rate *= 0.99999


    def AdaGrad(self,grads):
      """
      AdaGrad adaptive optimization algorithm.
      """

      i = 0
      for param,grad in zip(self.params(),grads):

        self.cache[i] += grad **2
        param += -self.learning_rate * grad / (np.sqrt(self.cache[i]) + 1e-6)

        i += 1


    def RMSprop(self,grads,decay_rate = 0.9):
      """

      RMSprop adaptive optimization algorithm
      """



      i = 0
      for param,grad in zip(self.params(),grads):
        self.cache_rmsprop[i] = decay_rate * self.cache_rmsprop[i] + (1-
decay_rate) * grad **2
        param += - self.learning_rate * grad / (np.sqrt(self.cache_rmsprop[i])
+ 1e-6)
        i += 1


    def VanillaAdam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """
        Adam optimizer, but bias correction is not implemented
        """
        i = 0

        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
```

```
          param += -self.learning_rate * self.m[i] / (np.sqrt(self.v[i]) + 1e-
8)
          i += 1


    def Adam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """

        Adam optimizer, bias correction is implemented.
        """

        i = 0

        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          m_corrected = self.m[i] / (1-beta1**self.t)
          v_corrected = self.v[i] / (1-beta2**self.t)
          param += -
self.learning_rate * m_corrected / (np.sqrt(v_corrected) + 1e-8)
          i += 1

        self.t +=1


    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def predict(self,X):
        """
        Return predictions, (not one hot encoded format)
        """

        # Give zeros to hidden/cell states:
        pasts = np.zeros((X.shape[0],self.hidden_dim))
        _,__,___,____,_____,______,_______,_______,probs = self.forward(X,past
s,pasts)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
```

```
                'TestLoss'   : self.test_loss,
                'TestAcc'    : self.test_acc}


# %%
lstm = LSTM(learning_rate = 5e-
4,mom_coeff = 0.0,batch_size = 32,hidden_dim=128)


# %%
lstm.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer='SGD')


# %%
lstm_history = lstm.history()


# %%
train_preds_lstm = lstm.predict(X_train)
test_preds_lstm = lstm.predict(X_test)
confusion_mat_train_lstm = metrics.confusion_matrix(np.argmax(y_train,1),train
_preds_lstm)
confusion_mat_test_lstm = metrics.confusion_matrix(np.argmax(y_test,1),test_pr
eds_lstm)

body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_lstm.columns = body_movements
confusion_mat_train_lstm.index = body_movements
confusion_mat_test_lstm.columns = body_movements
confusion_mat_test_lstm.index = body_movements

sns.heatmap(confusion_mat_train_lstm/np.sum(confusion_mat_train_lstm), annot=T
rue,
            fmt='.2%',cmap = 'Blues')
plt.show()
sns.heatmap(confusion_mat_test_lstm/np.sum(confusion_mat_test_lstm), annot=Tru
e,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
lstm2 = LSTM(learning_rate = 2e-
3,mom_coeff = 0.0,batch_size = 32,hidden_dim=128)
lstm2.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer='RMSprop')


# %%
```

```
lstm2_history = lstm2.history()


# %%
lstm3 = LSTM(learning_rate = 3e-
3,mom_coeff = 0.0,batch_size = 32,hidden_dim=128)
lstm3.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer='Adam')


# %%
lstm4 = LSTM(learning_rate = 1e-
3,mom_coeff = 0.0,batch_size = 32,hidden_dim=128)
lstm4.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer='AdaGrad')


# %%
lstm5 = LSTM(learning_rate = 1e-
3,mom_coeff = 0.0,batch_size = 32,hidden_dim=128)
lstm5.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer='VanillaAdam')


# %%
lstm3_history = lstm3.history()
lstm4_history = lstm4.history()
lstm5_history = lstm5.history()
plt.figure()
plt.plot(lstm_history['TrainAcc'],'-o')
plt.plot(lstm2_history['TrainAcc'],'-o')
plt.plot(lstm3_history['TrainAcc'],'-o')
plt.plot(lstm4_history['TrainAcc'],'-o')
plt.plot(lstm5_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['SGD','RMSprop','Adam','AdaGrad','Vanilla Adam'])
plt.show()

plt.figure()
plt.plot(lstm_history['TestAcc'],'-o')
plt.plot(lstm2_history['TestAcc'],'-o')
plt.plot(lstm3_history['TestAcc'],'-o')
plt.plot(lstm4_history['TestAcc'],'-o')
plt.plot(lstm5_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['SGD','RMSprop','Adam','AdaGrad','Vanilla Adam'])
plt.show()
```

```
plt.figure()
plt.plot(lstm_history['TrainLoss'],'-o')
plt.plot(lstm2_history['TrainLoss'],'-o')
plt.plot(lstm3_history['TrainLoss'],'-o')
plt.plot(lstm4_history['TrainLoss'],'-o')
plt.plot(lstm5_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Loss over epochs')
plt.legend(['SGD','RMSprop','Adam','AdaGrad','Vanilla Adam'])
plt.show()

plt.figure()
plt.plot(lstm_history['TestLoss'],'-o')
plt.plot(lstm2_history['TestLoss'],'-o')
plt.plot(lstm3_history['TestLoss'],'-o')
plt.plot(lstm4_history['TestLoss'],'-o')
plt.plot(lstm5_history['TestLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.legend(['SGD','RMSprop','Adam','AdaGrad','Vanilla Adam'])
plt.show()


# %%
three_layer_rnn_v2_history = three_layer_rnn_v2.history()
plt.figure()
plt.plot(three_layer_rnn_v2_history['TrainAcc'],'-o')
plt.plot(lstm_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['Best RNN','Best LSTM'])
plt.show()


plt.figure()
plt.plot(three_layer_rnn_v2_history['TestAcc'],'-o')
plt.plot(lstm_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['Best RNN','Best LSTM'])
plt.show()

plt.figure()
plt.plot(three_layer_rnn_v2_history['TrainLoss'],'-o')
plt.plot(lstm_history['TrainLoss'],'-o')
```

```
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Loss over epochs')
plt.legend(['Best RNN','Best LSTM'])
plt.show()


plt.figure()
plt.plot(three_layer_rnn_v2_history['TestLoss'],'-o')
plt.plot(lstm_history['TestLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.legend(['Best RNN','Best LSTM'])
plt.show()


# %%
train_preds_lstm = lstm3.predict(X_train)
test_preds_lstm = lstm3.predict(X_test)
confusion_mat_train_lstm = metrics.confusion_matrix(np.argmax(y_train,1),train
_preds_lstm)
confusion_mat_test_lstm = metrics.confusion_matrix(np.argmax(y_test,1),test_pr
eds_lstm)

body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_lstm.columns = body_movements
confusion_mat_train_lstm.index = body_movements
confusion_mat_test_lstm.columns = body_movements
confusion_mat_test_lstm.index = body_movements

sns.heatmap(confusion_mat_train_lstm/np.sum(confusion_mat_train_lstm), annot=T
rue,
            fmt='.2%',cmap = 'Blues')
plt.show()
sns.heatmap(confusion_mat_test_lstm/np.sum(confusion_mat_test_lstm), annot=Tru
e,
            fmt='.2%',cmap = 'Blues')
plt.show()


# %%



# %%
class Multi_Layer_LSTM(object):
    """
```

```
    Long-
Short Term Memory Recurrent neural network, encapsulates all necessary logic f
or training, then built the hyperparameters and architecture of the network.
    """

    def __init__(self,input_dim = 3,hidden_dim_1 = 128,hidden_dim_2 =64,output
_class = 6,seq_len = 150,batch_size = 30,learning_rate = 1e-
1,mom_coeff = 0.85):
        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim_1 = hidden_dim_1
        self.hidden_dim_2 = hidden_dim_2

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff

        self.input_stack_dim = self.input_dim + self.hidden_dim_1

        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim1 = Xavier(self.input_dim,self.hidden_dim_1)
        self.W_f = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim_1))
        self.B_f = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))

        self.W_i = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim_1))
        self.B_i = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))

        self.W_c = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim_1))
        self.B_c = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))

        self.W_o = np.random.uniform(-
lim1,lim1,(self.input_stack_dim,self.hidden_dim_1))
        self.B_o = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))
```

```
        lim2 = Xavier(self.hidden_dim_1,self.hidden_dim_2)
        self.W_hid = np.random.uniform(-
lim2,lim2,(self.hidden_dim_1,self.hidden_dim_2))
        self.B_hid = np.random.uniform(-lim2,lim2,(1,self.hidden_dim_2))

        lim3 = Xavier(self.hidden_dim_2,self.output_class)
        self.W = np.random.uniform(-
lim3,lim3,(self.hidden_dim_2,self.output_class))
        self.B = np.random.uniform(-lim3,lim3,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # To keep previous updates in momentum :
        self.previous_updates = [0] * 13

        # For AdaGrad:
        self.cache = [0] * 13
        self.cache_rmsprop = [0] * 13
        self.m = [0] * 13
        self.v = [0] * 13
        self.t = 1

    def cell_forward(self,X,h_prev,C_prev):
        """

        Takes input, previous hidden state and previous cell state, compute:
        --- Forget gate + Input gate + New candidate input + New cell state +
            output gate + hidden state. Then, classify by softmax.
        """
        #print(X.shape,h_prev.shape)
        # Stacking previous hidden state vector with inputs:
        stack = np.column_stack([X,h_prev])

        # Forget gate:
        forget_gate = activations.sigmoid(np.dot(stack,self.W_f) + self.B_f)

        # İnput gate:
        input_gate = activations.sigmoid(np.dot(stack,self.W_i) + self.B_i)

        # New candidate:
        cell_bar = np.tanh(np.dot(stack,self.W_c) + self.B_c)

        # New Cell state:
        cell_state = forget_gate * C_prev + input_gate * cell_bar

        # Output fate:
```

```
        output_gate = activations.sigmoid(np.dot(stack,self.W_o) + self.B_o)

        # Hidden state:
        hidden_state = output_gate * np.tanh(cell_state)

        # Classifiers (Softmax) :
        dense_hid = np.dot(hidden_state,self.W_hid) + self.B_hid
        act = activations.ReLU(dense_hid)

        dense = np.dot(act,self.W) + self.B
        probs = activations.softmax(dense)

        return (stack,forget_gate,input_gate,cell_bar,cell_state,output_gate,h
idden_state,dense,probs,dense_hid,act)



    def forward(self,X,h_prev,C_prev):
        x_s,z_s,f_s,i_s = {},{},{},{}
        C_bar_s,C_s,o_s,h_s = {},{},{},{}
        v_s,y_s,v_1s,y_1s = {},{},{},{}


        h_s[-1] = np.copy(h_prev)
        C_s[-1] = np.copy(C_prev)

        for t in range(self.seq_len):
            x_s[t] = X[:,t,:]
            z_s[t], f_s[t], i_s[t], C_bar_s[t], C_s[t], o_s[t], h_s[t],v_s[t],
 y_s[t],v_1s[t],y_1s[t] = self.cell_forward(x_s[t],h_s[t-1],C_s[t-1])

        return (z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s,v_s, y_s,v_1s,y_1s)

    def BPTT(self,outs,Y):

        z_s, f_s, i_s, C_bar_s, C_s, o_s, h_s,v_s, y_s,v_1s,y_1s = outs

        dW_f, dW_i,dW_c, dW_o,dW,dW_hid = np.zeros_like(self.W_f), np.zeros_li
ke(self.W_i), np.zeros_like(self.W_c),np.zeros_like(self.W_o),np.zeros_like(se
lf.W),np.zeros_like(self.W_hid)

        dB_f, dB_i,dB_c,dB_o,dB,dB_hid  = np.zeros_like(self.B_f), np.zeros_li
ke(self.B_i),np.zeros_like(self.B_c),np.zeros_like(self.B_o),np.zeros_like(sel
f.B),np.zeros_like(self.B_hid)

        dh_next = np.zeros_like(h_s[0])
        dC_next = np.zeros_like(C_s[0])
```

```
        # w.r.t. softmax input
        ddense = np.copy(y_s[149])
        ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        #ddense[np.argmax(Y,1)] -=1
        #ddense = y_s[149] - Y
        # Softmax classifier's :
        dW = np.dot(v_1s[149].T,ddense)
        dB = np.sum(ddense,axis = 0, keepdims = True)

        ddense_hid = np.dot(ddense,self.W.T) * activations.dReLU(v_1s[149])
        dW_hid = np.dot(h_s[149].T,ddense_hid)
        dB_hid = np.sum(ddense_hid,axis = 0, keepdims = True)


        # Backprop through time:
        for t in reversed(range(1,self.seq_len)):

            # Just equating more meaningful names
            stack,forget_gate,input_gate,cell_bar,cell_state,output_gate,hidde
n_state,dense,probs = z_s[t], f_s[t], i_s[t], C_bar_s[t], C_s[t], o_s[t], h_s[
t],v_s[t], y_s[t]
            C_prev = C_s[t-1]

            # w.r.t. softmax input
            #ddense = np.copy(probs)
            #ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
            #ddense[np.arange(len(Y)),np.argmax(Y,1)] -=1
            # Softmax classifier's :
            #dW += np.dot(hidden_state.T,ddense)
            #dB += np.sum(ddense,axis = 0, keepdims = True)

            # Output gate :
            dh = np.dot(ddense_hid,self.W_hid.T) + dh_next
            do = dh * np.tanh(cell_state)
            do = do * dsigmoid(output_gate)
            dW_o += np.dot(stack.T,do)
            dB_o += np.sum(do,axis = 0, keepdims = True)

            # Cell state:
            dC = np.copy(dC_next)
            dC += dh * output_gate * activations.dtanh(cell_state)
            dC_bar = dC * input_gate
            dC_bar = dC_bar * dtanh(cell_bar)
            dW_c += np.dot(stack.T,dC_bar)
            dB_c += np.sum(dC_bar,axis = 0, keepdims = True)

            # Input gate:
            di = dC * cell_bar
```

```
            di = dsigmoid(input_gate) * di
            dW_i += np.dot(stack.T,di)
            dB_i += np.sum(di,axis = 0,keepdims = True)

            # Forget gate:
            df = dC * C_prev
            df = df * dsigmoid(forget_gate)
            dW_f += np.dot(stack.T,df)
            dB_f += np.sum(df,axis = 0, keepdims = True)

            dz = np.dot(df,self.W_f.T) + np.dot(di,self.W_i.T) + np.dot(dC_bar
,self.W_c.T) + np.dot(do,self.W_o.T)

            dh_next = dz[:,-self.hidden_dim_1:]
            dC_next = forget_gate * dC

        # List of gradients :
        grads = [dW,dB,dW_hid,dB_hid,dW_o,dB_o,dW_c,dB_c,dW_i,dB_i,dW_f,dB_f]

        # Clipping gradients anyway
        for grad in grads:
            np.clip(grad, -15, 15, out = grad)

        return h_s[self.seq_len - 1],C_s[self.seq_len -1 ],grads



    def fit(self,X,Y,X_val,y_val,epochs = 50 ,optimizer = 'SGD',verbose = True
, crossVal = False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)
            h_prev,C_prev = np.zeros((self.batch_size,self.hidden_dim_1)),np.z
eros((self.batch_size,self.hidden_dim_1))
            for i in range(round(X.shape[0]/self.batch_size) - 1):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]
```

```
                # Feeding random indexes:
                X_feed = X[index]
                y_feed = Y[index]

                # Forward + BPTT + SGD:
                cache_train = self.forward(X_feed,h_prev,C_prev)
                h,c,grads = self.BPTT(cache_train,y_feed)

                if optimizer == 'SGD':

                  self.SGD(grads)

                elif optimizer == 'AdaGrad' :
                  self.AdaGrad(grads)

                elif optimizer == 'RMSprop':
                  self.RMSprop(grads)

                elif optimizer == 'VanillaAdam':
                  self.VanillaAdam(grads)
                else:
                  self.Adam(grads)

                # Hidden state -------> Previous hidden state
                # Cell state ---------> Previous cell state
                h_prev,C_prev = h,c

            # Training metrics calculations:
            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[8][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            # Validation metrics calculations:
            test_prevs = np.zeros((X_val.shape[0],self.hidden_dim_1))
            _,__,___,____,_____,______,_______,________,probs_test,a,b = self.
forward(X_val,test_prevs,test_prevs)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
```

```
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)


    def params(self):
        """
        Return all weights/biases in sequential order starting from end in lis
t form.

        """
        return [self.W,self.B,self.W_hid,self.B_hid,self.W_o,self.B_o,self.W_c
,self.B_c,self.W_i,self.B_i,self.W_f,self.B_f]


    def SGD(self,grads):
      """

      Stochastic gradient descent with momentum on mini-batches.
      """
      prevs = []

      for param,grad,prev_update in zip(self.params(),grads,self.previous_upda
tes):
          delta = self.learning_rate * grad - self.mom_coeff * prev_update
          param -= delta
          prevs.append(delta)



      self.previous_updates = prevs

      self.learning_rate *= 0.99999


    def AdaGrad(self,grads):
```

```
      """
      AdaGrad adaptive optimization algorithm.
      """
      i = 0
      for param,grad in zip(self.params(),grads):
        self.cache[i] += grad **2
        param += -self.learning_rate * grad / (np.sqrt(self.cache[i]) + 1e-6)
        i += 1


    def RMSprop(self,grads,decay_rate = 0.9):
      """

      RMSprop adaptive optimization algorithm
      """



      i = 0
      for param,grad in zip(self.params(),grads):
        self.cache_rmsprop[i] = decay_rate * self.cache_rmsprop[i] + (1-
decay_rate) * grad **2
        param += - self.learning_rate * grad / (np.sqrt(self.cache_rmsprop[i])
+ 1e-6)
        i += 1


    def VanillaAdam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """
        Adam optimizer, but bias correction is not implemented
        """
        i = 0

        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          param += -self.learning_rate * self.m[i] / (np.sqrt(self.v[i]) + 1e-
8)
          i += 1


    def Adam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """

        Adam optimizer, bias correction is implemented.
        """

        i = 0
```

```
        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          m_corrected = self.m[i] / (1-beta1**self.t)
          v_corrected = self.v[i] / (1-beta2**self.t)
          param += -
self.learning_rate * m_corrected / (np.sqrt(v_corrected) + 1e-8)
          i += 1

        self.t +=1


    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def predict(self,X):
        """
        Return predictions, (not one hot encoded format)
        """

        # Give zeros to hidden/cell states:
        pasts = np.zeros((X.shape[0],self.hidden_dim_1))
        _,__,___,____,_____,______,_______,_______,probs,a,b = self.forward(X,
pasts,pasts)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}


# %%
mutl_layer_lstm = Multi_Layer_LSTM(learning_rate=1e-
3,batch_size=32,hidden_dim_1 = 128,hidden_dim_2=64,mom_coeff=0.0)
mutl_layer_lstm.fit(X_train,y_train,X_test,y_test,epochs=15,optimizer='Adam')


# %%
mutl_layer_lstm_history = mutl_layer_lstm.history()
plt.figure()
plt.plot(mutl_layer_lstm_history['TrainAcc'],'-o')
```

```
plt.plot(lstm_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['Multi Layer LSTM','LSTM'])
plt.show()


plt.figure()
plt.plot(mutl_layer_lstm_history['TestAcc'],'-o')
plt.plot(lstm_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['Multi Layer LSTM','LSTM'])
plt.show()

plt.figure()
plt.plot(mutl_layer_lstm_history['TrainLoss'],'-o')
plt.plot(lstm_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Loss over epochs')
plt.legend(['Multi Layer LSTM','LSTM'])
plt.show()


plt.figure()
plt.plot(mutl_layer_lstm_history['TestLoss'],'-o')
plt.plot(lstm_history['TestLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.legend(['Multi Layer LSTM','LSTM'])
plt.show()


# %%
mutl_layer_lstm.fit(X_train,y_train,X_test,y_test,epochs=15,optimizer = 'Vanil
la')


# %%
mutl_layer_lstm_history = mutl_layer_lstm.history()


# %%
plt.figure()
plt.plot(mutl_layer_lstm_history['TestLoss'],'-o')
```

```
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.show()

plt.figure()
plt.plot(mutl_layer_lstm_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.show()


plt.figure()
plt.plot(mutl_layer_lstm_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.show()


plt.figure()
plt.plot(mutl_layer_lstm_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.show()

# %%
class GRU(object):
    """

    Gater recurrent unit, encapsulates all necessary logic for training, then
built the hyperparameters and architecture of the network.
    """

    def __init__(self,input_dim = 3,hidden_dim = 128,output_class = 6,seq_len
= 150,batch_size = 32,learning_rate = 1e-1,mom_coeff = 0.85):
        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(32)
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Unfold case T = 150 :
        self.seq_len = seq_len
```

```
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff



        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim1 = Xavier(self.input_dim,self.hidden_dim)
        lim1_hid = Xavier(self.hidden_dim,self.hidden_dim)
        self.W_z = np.random.uniform(-
lim1,lim1,(self.input_dim,self.hidden_dim))
        self.U_z = np.random.uniform(-
lim1_hid,lim1_hid,(self.hidden_dim,self.hidden_dim))
        self.B_z = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))

        self.W_r = np.random.uniform(-
lim1,lim1,(self.input_dim,self.hidden_dim))
        self.U_r = np.random.uniform(-
lim1_hid,lim1_hid,(self.hidden_dim,self.hidden_dim))
        self.B_r = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))

        self.W_h = np.random.uniform(-
lim1,lim1,(self.input_dim,self.hidden_dim))
        self.U_h = np.random.uniform(-
lim1_hid,lim1_hid,(self.hidden_dim,self.hidden_dim))
        self.B_h = np.random.uniform(-lim1,lim1,(1,self.hidden_dim))


        lim2 = Xavier(self.hidden_dim,self.output_class)
        self.W = np.random.uniform(-
lim2,lim2,(self.hidden_dim,self.output_class))
        self.B = np.random.uniform(-lim2,lim2,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # To keep previous updates in momentum :
        self.previous_updates = [0] * 10

        # For AdaGrad:
        self.cache = [0] * 11
        self.cache_rmsprop = [0] * 11
        self.m = [0] * 11
        self.v = [0] * 11
```

```
        self.t = 1

    def cell_forward(self,X,h_prev):
        """

        Takes input, previous hidden state and previous cell state, compute:
        --- Forget gate + Input gate + New candidate input + New cell state +
            output gate + hidden state. Then, classify by softmax.
        """



        # Update gate:
        update_gate = activations.sigmoid(np.dot(X,self.W_z) + np.dot(h_prev,s
elf.U_z) + self.B_z)

        # Reset gate:
        reset_gate = activations.sigmoid(np.dot(X,self.W_r) + np.dot(h_prev,se
lf.U_r) + self.B_r)

        # Current memory content:
        h_hat = np.tanh(np.dot(X,self.W_h) + np.dot(np.multiply(reset_gate,h_p
rev),self.U_h) + self.B_h)

        # Hidden state:
        hidden_state = np.multiply(update_gate,h_prev) + np.multiply((1-
update_gate),h_hat)


        # Classifiers (Softmax) :
        dense = np.dot(hidden_state,self.W) + self.B
        probs = activations.softmax(dense)

        return (update_gate,reset_gate,h_hat,hidden_state,dense,probs)



    def forward(self,X,h_prev):
        x_s,z_s,r_s,h_hat = {},{},{},{}
        h_s = {}
        y_s,p_s = {},{}

        h_s[-1] = np.copy(h_prev)


        for t in range(self.seq_len):
            x_s[t] = X[:,t,:]
            z_s[t], r_s[t], h_hat[t], h_s[t], y_s[t], p_s[t] = self.cell_forwa
rd(x_s[t],h_s[t-1])
```

```
        return (x_s,z_s, r_s, h_hat, h_s, y_s, p_s)

    def BPTT(self,outs,Y):

        x_s,z_s, r_s, h_hat, h_s, y_s, p_s = outs

        dW_z, dW_r,dW_h, dW = np.zeros_like(self.W_z), np.zeros_like(self.W_r)
, np.zeros_like(self.W_h),np.zeros_like(self.W)

        dU_z, dU_r,dU_h, = np.zeros_like(self.U_z), np.zeros_like(self.U_r), n
p.zeros_like(self.U_h)


        dB_z, dB_r,dB_h,dB = np.zeros_like(self.B_z), np.zeros_like(self.B_r),
np.zeros_like(self.B_h),np.zeros_like(self.B)

        dh_next = np.zeros_like(h_s[0])


        # w.r.t. softmax input
        ddense = np.copy(p_s[149])
        ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        #ddense[np.argmax(Y,1)] -=1
        #ddense = y_s[149] - Y
        # Softmax classifier's :
        dW = np.dot(h_s[149].T,ddense)
        dB = np.sum(ddense,axis = 0, keepdims = True)

        # Backprop through time:
        for t in reversed(range(1,self.seq_len)):

            # w.r.t. softmax input
            #ddense = np.copy(probs)
            #ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
            #ddense[np.arange(len(Y)),np.argmax(Y,1)] -=1
            # Softmax classifier's :
            #dW += np.dot(hidden_state.T,ddense)
            #dB += np.sum(ddense,axis = 0, keepdims = True)


            # Curernt memort state :
            dh = np.dot(ddense,self.W.T) + dh_next
            dh_hat = dh * (1-z_s[t])
            dh_hat = dh_hat * dtanh(h_hat[t])
            dW_h += np.dot(x_s[t].T,dh_hat)
            dU_h += np.dot((r_s[t] * h_s[t-1]).T,dh_hat)
            dB_h += np.sum(dh_hat,axis = 0, keepdims = True)
```

```
            # Reset gate:
            dr_1 = np.dot(dh_hat,self.U_h.T)
            dr = dr_1  * h_s[t-1]
            dr = dr * dsigmoid(r_s[t])
            dW_r += np.dot(x_s[t].T,dr)
            dU_r += np.dot(h_s[t-1].T,dr)
            dB_r += np.sum(dr,axis = 0, keepdims = True)

            # Forget gate:
            dz = dh * (h_s[t-1] - h_hat[t])
            dz = dz * dsigmoid(z_s[t])
            dW_z += np.dot(x_s[t].T,dz)
            dU_z += np.dot(h_s[t-1].T,dz)
            dB_z += np.sum(dz,axis = 0, keepdims = True)


            # Nexts:
            dh_next = np.dot(dz,self.U_z.T) + (dh * z_s[t]) + (dr_1 * r_s[t])
+ np.dot(dr,self.U_r.T)


        # List of gradients :
        grads = [dW,dB,dW_z,dU_z,dB_z,dW_r,dU_r,dB_r,dW_h,dU_h,dB_h]

        # Clipping gradients anyway
        for grad in grads:
            np.clip(grad, -15, 15, out = grad)

        return h_s[self.seq_len - 1],grads



    def fit(self,X,Y,X_val,y_val,epochs = 50 ,optimizer = 'SGD',verbose = True
, crossVal = False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)
            h_prev = np.zeros((self.batch_size,self.hidden_dim))
            for i in range(round(X.shape[0]/self.batch_size) - 1):
```

```
                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                # Feeding random indexes:
                X_feed = X[index]
                y_feed = Y[index]

                # Forward + BPTT + SGD:
                cache_train = self.forward(X_feed,h_prev)
                h,grads = self.BPTT(cache_train,y_feed)

                if optimizer == 'SGD':

                  self.SGD(grads)

                elif optimizer == 'AdaGrad' :
                  self.AdaGrad(grads)

                elif optimizer == 'RMSprop':
                  self.RMSprop(grads)

                elif optimizer == 'VanillaAdam':
                  self.VanillaAdam(grads)
                else:
                  self.Adam(grads)

                # Hidden state --------> Previous hidden state
                h_prev= h

            # Training metrics calculations:
            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[6][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            # Validation metrics calculations:
            test_prevs = np.zeros((X_val.shape[0],self.hidden_dim))
            _,__,___,____,_____,______,probs_test = self.forward(X_val,test_pr
evs)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:
```

```
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)


    def params(self):
        """
        Return all weights/biases in sequential order starting from end in lis
t form.

        """
        return [self.W,self.B,self.W_z,self.U_z,self.B_z,self.W_r,self.U_r,sel
f.B_r,self.W_h,self.U_h,self.B_h]

    def SGD(self,grads):
      """

      Stochastic gradient descent with momentum on mini-batches.
      """
      prevs = []
      for param,grad,prev_update in zip(self.params(),grads,self.previous_upda
tes):
          delta = self.learning_rate * grad - self.mom_coeff * prev_update
          param -= delta
          prevs.append(delta)

      self.previous_updates = prevs

      self.learning_rate *= 0.99999


    def AdaGrad(self,grads):
      """
      AdaGrad adaptive optimization algorithm.
```

```
      """

      i = 0
      for param,grad in zip(self.params(),grads):

        self.cache[i] += grad **2
        param += -self.learning_rate * grad / (np.sqrt(self.cache[i]) + 1e-6)

        i += 1


    def RMSprop(self,grads,decay_rate = 0.9):
      """

      RMSprop adaptive optimization algorithm
      """



      i = 0
      for param,grad in zip(self.params(),grads):
        self.cache_rmsprop[i] = decay_rate * self.cache_rmsprop[i] + (1-
decay_rate) * grad **2
        param += - self.learning_rate * grad / (np.sqrt(self.cache_rmsprop[i])
+ 1e-6)
        i += 1


    def VanillaAdam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """
        Adam optimizer, but bias correction is not implemented
        """
        i = 0

        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          param += -self.learning_rate * self.m[i] / (np.sqrt(self.v[i]) + 1e-
8)
          i += 1


    def Adam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """

        Adam optimizer, bias correction is implemented.
        """

        i = 0
```

```
        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          m_corrected = self.m[i] / (1-beta1**self.t)
          v_corrected = self.v[i] / (1-beta2**self.t)
          param += -
self.learning_rate * m_corrected / (np.sqrt(v_corrected) + 1e-8)
          i += 1

        self.t +=1


    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def predict(self,X):
        """
        Return predictions, (not one hot encoded format)
        """

        # Give zeros to hidden/cell states:
        pasts = np.zeros((X.shape[0],self.hidden_dim))
        _,__,___,____,_____,______,probs = self.forward(X,pasts)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}


# %%
gru = GRU(hidden_dim=128,learning_rate=1e-3,batch_size=32,mom_coeff=0.0)


# %%
gru.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer = 'RMSprop')


# %%
gru_history = gru.history()
```

```
# %%
# For figure 97:

plt.figure()
plt.plot(gru_history['TrainLoss'],'-o')
plt.plot(gru_history['TestLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Loss over epochs')
plt.legend(['Train Loss','Test Loss'])
plt.show()


plt.figure()
plt.plot(gru_history['TrainAcc'],'-o')
plt.plot(gru_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Accuracy over epochs')
plt.legend(['Train Acc','Test Acc'])
plt.show()


# %%
# For figure 98:
multi_layer_gru_history = multi_layer_gru.history()
plt.figure()
plt.plot(multi_layer_gru_history['TrainAcc'],'-o')
plt.plot(gru_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['Multi Layer GRU','GRU'])
plt.show()


plt.figure()
plt.plot(multi_layer_gru_history['TestAcc'],'-o')
plt.plot(gru_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['Multi Layer GRU','GRU'])
plt.show()

plt.figure()
plt.plot(multi_layer_gru_history['TrainLoss'],'-o')
```

```
plt.plot(gru_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Loss over epochs')
plt.legend(['Multi Layer GRU','GRU'])
plt.show()


plt.figure()
plt.plot(multi_layer_gru_history['TestLoss'],'-o')
plt.plot(gru_history['TestLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.legend(['Multi Layer GRU','GRU'])
plt.show()


# %%
# For figure 99:
three_layer_rnn_history = three_layer_rnn.history()
plt.figure()
plt.plot(gru_history['TrainAcc'],'-o')
plt.plot(lstm_history['TrainAcc'],'-o')
plt.plot(three_layer_rnn_history['TrainAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Accuracy over epochs')
plt.legend(['GRU','LSTM','RNN'])
plt.show()


plt.figure()
plt.plot(gru_history['TestAcc'],'-o')
plt.plot(lstm_history['TestAcc'],'-o')
plt.plot(three_layer_rnn_history['TestAcc'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Accuracy over epochs')
plt.legend(['GRU','LSTM','RNN'])
plt.show()

plt.figure()
plt.plot(gru_history['TrainLoss'],'-o')
plt.plot(lstm_history['TrainLoss'],'-o')
plt.plot(three_layer_rnn_history['TrainLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Training Loss over epochs')
```

```
plt.legend(['GRU','LSTM','RNN'])
plt.show()


plt.figure()
plt.plot(gru_history['TestLoss'],'-o')
plt.plot(lstm_history['TestLoss'],'-o')
plt.plot(three_layer_rnn_history['TestLoss'],'-o')
plt.xlabel('# of epochs')
plt.ylabel('Loss')
plt.title('Testing Loss over epochs')
plt.legend(['GRU','LSTM','RNN'])
plt.show()


# %%
train_preds_gru = gru.predict(X_train)
test_preds_gru = gru.predict(X_test)
confusion_mat_train_gru = metrics.confusion_matrix(np.argmax(y_train,1),train_
preds_gru)
confusion_mat_test_gru = metrics.confusion_matrix(np.argmax(y_test,1),test_pre
ds_gru)

body_movements = ['downstairs','jogging','sitting','standing','upstairs','walk
ing']
confusion_mat_train_gru.columns = body_movements
confusion_mat_train_gru.index = body_movements
confusion_mat_test_gru.columns = body_movements
confusion_mat_test_gru.index = body_movements

sns.heatmap(confusion_mat_train_gru/np.sum(confusion_mat_train_gru), annot=Tru
e,
            fmt='.2%',cmap = 'Blues')
plt.show()
sns.heatmap(confusion_mat_test_gru/np.sum(confusion_mat_test_gru), annot=True,

            fmt='.2%',cmap = 'Blues')
plt.show()


# %%
class Multi_layer_GRU(object):
    """

    Gater recurrent unit, encapsulates all necessary logic for training, then
built the hyperparameters and architecture of the network.
    """
```

```
    def __init__(self,input_dim = 3,hidden_dim_1 = 128,hidden_dim_2 = 64,outpu
t_class = 6,seq_len = 150,batch_size = 32,learning_rate = 1e-
1,mom_coeff = 0.85):
        """

        Initialization of weights/biases and other configurable parameters.

        """
        np.random.seed(150)
        self.input_dim = input_dim
        self.hidden_dim_1 = hidden_dim_1
        self.hidden_dim_2 = hidden_dim_2

        # Unfold case T = 150 :
        self.seq_len = seq_len
        self.output_class = output_class
        self.learning_rate = learning_rate
        self.batch_size = batch_size
        self.mom_coeff = mom_coeff



        # Xavier uniform scaler :
        Xavier = lambda fan_in,fan_out : math.sqrt(6/(fan_in + fan_out))

        lim1 = Xavier(self.input_dim,self.hidden_dim_1)
        lim1_hid = Xavier(self.hidden_dim_1,self.hidden_dim_1)
        self.W_z = np.random.uniform(-
lim1,lim1,(self.input_dim,self.hidden_dim_1))
        self.U_z = np.random.uniform(-
lim1_hid,lim1_hid,(self.hidden_dim_1,self.hidden_dim_1))
        self.B_z = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))

        self.W_r = np.random.uniform(-
lim1,lim1,(self.input_dim,self.hidden_dim_1))
        self.U_r = np.random.uniform(-
lim1_hid,lim1_hid,(self.hidden_dim_1,self.hidden_dim_1))
        self.B_r = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))

        self.W_h = np.random.uniform(-
lim1,lim1,(self.input_dim,self.hidden_dim_1))
        self.U_h = np.random.uniform(-
lim1_hid,lim1_hid,(self.hidden_dim_1,self.hidden_dim_1))
        self.B_h = np.random.uniform(-lim1,lim1,(1,self.hidden_dim_1))

        lim2_hid = Xavier(self.hidden_dim_1,self.hidden_dim_2)
        self.W_hid = np.random.uniform(-
lim2_hid,lim2_hid,(self.hidden_dim_1,self.hidden_dim_2))
```

```
        self.B_hid = np.random.uniform(-
lim2_hid,lim2_hid,(1,self.hidden_dim_2))

        lim2 = Xavier(self.hidden_dim_2,self.output_class)
        self.W = np.random.uniform(-
lim2,lim2,(self.hidden_dim_2,self.output_class))
        self.B = np.random.uniform(-lim2,lim2,(1,self.output_class))

        # To keep track loss and accuracy score :
        self.train_loss,self.test_loss,self.train_acc,self.test_acc = [],[],[]
,[]

        # To keep previous updates in momentum :
        self.previous_updates = [0] * 13

        # For AdaGrad:
        self.cache = [0] * 13
        self.cache_rmsprop = [0] * 13
        self.m = [0] * 13
        self.v = [0] * 13
        self.t = 1

    def cell_forward(self,X,h_prev):


        # Update gate:
        update_gate = activations.sigmoid(np.dot(X,self.W_z) + np.dot(h_prev,s
elf.U_z) + self.B_z)

        # Reset gate:
        reset_gate = activations.sigmoid(np.dot(X,self.W_r) + np.dot(h_prev,se
lf.U_r) + self.B_r)

        # Current memory content:
        h_hat = np.tanh(np.dot(X,self.W_h) + np.dot(np.multiply(reset_gate,h_p
rev),self.U_h) + self.B_h)

        # Hidden state:
        hidden_state = np.multiply(update_gate,h_prev) + np.multiply((1-
update_gate),h_hat)

        # Hidden MLP:
        hid_dense = np.dot(hidden_state,self.W_hid) + self.B_hid
        relu = activations.ReLU(hid_dense)

        # Classifiers (Softmax) :
        dense = np.dot(relu,self.W) + self.B
        probs = activations.softmax(dense)
```

```
        return (update_gate,reset_gate,h_hat,hidden_state,hid_dense,relu,dense
,probs)



    def forward(self,X,h_prev):
        x_s,z_s,r_s,h_hat = {},{},{},{}
        h_s = {}
        hd_s,relu_s = {},{}
        y_s,p_s = {},{}

        h_s[-1] = np.copy(h_prev)


        for t in range(self.seq_len):
            x_s[t] = X[:,t,:]
            z_s[t], r_s[t], h_hat[t], h_s[t],hd_s[t],relu_s[t], y_s[t], p_s[t]
 = self.cell_forward(x_s[t],h_s[t-1])

        return (x_s,z_s, r_s, h_hat, h_s, hd_s,relu_s, y_s, p_s)

    def BPTT(self,outs,Y):

        x_s,z_s, r_s, h_hat, h_s, hd_s,relu_s, y_s, p_s = outs

        dW_z, dW_r,dW_h, dW = np.zeros_like(self.W_z), np.zeros_like(self.W_r)
, np.zeros_like(self.W_h),np.zeros_like(self.W)
        dW_hid = np.zeros_like(self.W_hid)
        dU_z, dU_r,dU_h = np.zeros_like(self.U_z), np.zeros_like(self.U_r), np
.zeros_like(self.U_h)


        dB_z, dB_r,dB_h,dB = np.zeros_like(self.B_z), np.zeros_like(self.B_r),
np.zeros_like(self.B_h),np.zeros_like(self.B)
        dB_hid = np.zeros_like(self.B_hid)
        dh_next = np.zeros_like(h_s[0])


        # w.r.t. softmax input
        ddense = np.copy(p_s[149])
        ddense[np.arange(len(Y)),np.argmax(Y,1)] -= 1
        #ddense[np.argmax(Y,1)] -=1
        #ddense = y_s[149] - Y
        # Softmax classifier's :
        dW = np.dot(relu_s[149].T,ddense)
        dB = np.sum(ddense,axis = 0, keepdims = True)
```

```
        ddense_hid = np.dot(ddense,self.W.T) * activations.dReLU(hd_s[149])
        dW_hid = np.dot(h_s[149].T,ddense_hid)
        dB_hid = np.sum(ddense_hid,axis = 0, keepdims = True)


        # Backprop through time:
        for t in reversed(range(1,self.seq_len)):

            # Curernt memort state :
            dh = np.dot(ddense_hid,self.W_hid.T) + dh_next
            dh_hat = dh * (1-z_s[t])
            dh_hat = dh_hat * dtanh(h_hat[t])
            dW_h += np.dot(x_s[t].T,dh_hat)
            dU_h += np.dot((r_s[t] * h_s[t-1]).T,dh_hat)
            dB_h += np.sum(dh_hat,axis = 0, keepdims = True)

            # Reset gate:
            dr_1 = np.dot(dh_hat,self.U_h.T)
            dr = dr_1  * h_s[t-1]
            dr = dr * dsigmoid(r_s[t])
            dW_r += np.dot(x_s[t].T,dr)
            dU_r += np.dot(h_s[t-1].T,dr)
            dB_r += np.sum(dr,axis = 0, keepdims = True)

            # Forget gate:
            dz = dh * (h_s[t-1] - h_hat[t])
            dz = dz * dsigmoid(z_s[t])
            dW_z += np.dot(x_s[t].T,dz)
            dU_z += np.dot(h_s[t-1].T,dz)
            dB_z += np.sum(dz,axis = 0, keepdims = True)


            # Nexts:
            dh_next = np.dot(dz,self.U_z.T) + (dh * z_s[t]) + (dr_1 * r_s[t])
+ np.dot(dr,self.U_r.T)


        # List of gradients :
        grads = [dW,dB,dW_hid,dB_hid,dW_z,dU_z,dB_z,dW_r,dU_r,dB_r,dW_h,dU_h,d
B_h]

        # Clipping gradients anyway
        for grad in grads:
            np.clip(grad, -15, 15, out = grad)

        return h_s[self.seq_len - 1],grads
```

```
    def fit(self,X,Y,X_val,y_val,epochs = 50 ,optimizer = 'SGD',verbose = True
, crossVal = False):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """


        for epoch in range(epochs):

            print(f'Epoch : {epoch + 1}')

            perm = np.random.permutation(3000)

            # Equate 0 in every epoch:
            h_prev = np.zeros((self.batch_size,self.hidden_dim_1))

            for i in range(round(X.shape[0]/self.batch_size) - 1):

                batch_start  =  i * self.batch_size
                batch_finish = (i+1) * self.batch_size
                index = perm[batch_start:batch_finish]

                # Feeding random indexes:
                X_feed = X[index]
                y_feed = Y[index]

                # Forward + BPTT + Optimization:
                cache_train = self.forward(X_feed,h_prev)
                h,grads = self.BPTT(cache_train,y_feed)

                if optimizer == 'SGD':

                  self.SGD(grads)

                elif optimizer == 'AdaGrad' :
                  self.AdaGrad(grads)

                elif optimizer == 'RMSprop':
                  self.RMSprop(grads)

                elif optimizer == 'VanillaAdam':
                  self.VanillaAdam(grads)
                else:
                  self.Adam(grads)

                # Hidden state --------> Previous hidden state
```

```
                h_prev = h

            # Training metrics calculations:
            cross_loss_train = self.CategoricalCrossEntropy(y_feed,cache_train
[8][149])
            predictions_train = self.predict(X)
            acc_train = metrics.accuracy(np.argmax(Y,1),predictions_train)

            # Validation metrics calculations:
            test_prevs = np.zeros((X_val.shape[0],self.hidden_dim_1))
            _,__,___,____,_____,______,_______,________,probs_test = self.forw
ard(X_val,test_prevs)
            cross_loss_val = self.CategoricalCrossEntropy(y_val,probs_test[149
])
            predictions_val = np.argmax(probs_test[149],1)
            acc_val = metrics.accuracy(np.argmax(y_val,1),predictions_val)

            if verbose:

                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Accuracy : {acc_train}")
                print(f"[{epoch + 1}/{epochs}] ------
> Training :  Loss     : {cross_loss_train}")
                print('_____________________________________________________
_____________________________\n')
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Accuracy : {acc_val}")
                print(f"[{epoch + 1}/{epochs}] ------
> Testing  :  Loss     : {cross_loss_val}")
                print('_____________________________________________________
_____________________________\n')

            self.train_loss.append(cross_loss_train)
            self.test_loss.append(cross_loss_val)
            self.train_acc.append(acc_train)
            self.test_acc.append(acc_val)


    def params(self):
        """
        Return all weights/biases in sequential order starting from end in lis
t form.

        """
        return [self.W,self.B,self.W_hid,self.B_hid,self.W_z,self.U_z,self.B_z
,self.W_r,self.U_r,self.B_r,self.W_h,self.U_h,self.B_h]

    def SGD(self,grads):
```

```
      """

      Stochastic gradient descent with momentum on mini-batches.
      """
      prevs = []

      for param,grad,prev_update in zip(self.params(),grads,self.previous_upda
tes):

          delta = self.learning_rate * grad + self.mom_coeff * prev_update
          param -= delta
          prevs.append(delta)


      self.previous_updates = prevs
      self.learning_rate *= 0.99999


    def AdaGrad(self,grads):
      """
      AdaGrad adaptive optimization algorithm.
      """

      i = 0
      for param,grad in zip(self.params(),grads):

        self.cache[i] += grad **2
        param += -self.learning_rate * grad / (np.sqrt(self.cache[i]) + 1e-6)

        i += 1


    def RMSprop(self,grads,decay_rate = 0.9):
      """
      RMSprop adaptive optimization algorithm
      """
      i = 0
      for param,grad in zip(self.params(),grads):
        self.cache_rmsprop[i] = decay_rate * self.cache_rmsprop[i] + (1-
decay_rate) * grad **2
        param += - self.learning_rate * grad / (np.sqrt(self.cache_rmsprop[i])
+ 1e-6)
        i += 1


    def VanillaAdam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """
        Adam optimizer, but bias correction is not implemented
        """
```

```
        i = 0

        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          param += -self.learning_rate * self.m[i] / (np.sqrt(self.v[i]) + 1e-
8)
          i += 1


    def Adam(self,grads,beta1 = 0.9,beta2 = 0.999):
        """

        Adam optimizer, bias correction is implemented.
        """

        i = 0

        for param,grad  in zip(self.params(),grads):

          self.m[i] = beta1 * self.m[i] + (1-beta1) * grad
          self.v[i] = beta2 * self.v[i] + (1-beta2) * grad **2
          m_corrected = self.m[i] / (1-beta1**self.t)
          v_corrected = self.v[i] / (1-beta2**self.t)
          param += -
self.learning_rate * m_corrected / (np.sqrt(v_corrected) + 1e-8)
          i += 1

        self.t +=1


    def CategoricalCrossEntropy(self,labels,preds):
        """
        Computes cross entropy between labels and model's predictions
        """
        predictions = np.clip(preds, 1e-12, 1. - 1e-12)
        N = predictions.shape[0]
        return -np.sum(labels * np.log(predictions + 1e-9)) / N

    def predict(self,X):
        """
        Return predictions, (not one hot encoded format)
        """

        # Give zeros to hidden states:
        pasts = np.zeros((X.shape[0],self.hidden_dim_1))
```

```
        _,__,___,____,_____,______,_______,________,probs = self.forward(X,pas
ts)
        return np.argmax(probs[149],axis=1)

    def history(self):
        return {'TrainLoss' : self.train_loss,
                'TrainAcc'  : self.train_acc,
                'TestLoss'  : self.test_loss,
                'TestAcc'   : self.test_acc}


# %%
multi_layer_gru = Multi_layer_GRU(hidden_dim_1=128,hidden_dim_2=64,learning_ra
te=1e-3,mom_coeff=0.0,batch_size=32)


# %%
multi_layer_gru.fit(X_train,y_train,X_test,y_test,epochs = 15,optimizer = 'RMS
prop')
```

4.2 APPENDIX II

Part A

Convolutional Networks

So far we have worked with deep fully-connected networks, using them to explore different optimization strategies and network architectures. Fully-connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

```
# As usual, a bit of setup
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array, eval_numerical_gr
adient
from cs231n.layers import *
from cs231n.fast_layers import *
from cs231n.solver import Solver

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

```
# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
  """ returns relative error """
  return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
run the following from the cs231n directory and try again:
python setup.py build_ext --inplace
You may also need to restart your iPython kernel
```

```
# Load the (preprocessed) CIFAR10 data.

data = get_CIFAR10_data()
for k, v in data.items():
  print('%s: ' % k, v.shape)
```

```
X_train:  (49000, 3, 32, 32)
y_train:  (49000,)
X_val:  (1000, 3, 32, 32)
y_val:  (1000,)
X_test:  (1000, 3, 32, 32)
y_test:  (1000,)
```

Convolution: Naive forward pass

The core of a convolutional network is the convolution operation. In the file `cs231n/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find clearest.

You can test your implementation by running the following:

```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

Aside: Image processing via convolutions

As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

```
from scipy.misc import imread, imresize

kitten, puppy = imread('kitten.jpg'), imread('puppy.jpg')
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = imresize(puppy, (img_size, img_size)).transpose((2, 0, 1))
x[1, :, :, :] = imresize(kitten_cropped, (img_size, img_size)).transpose((2, 0, 1)
)

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_noax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
```

```
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```

```
C:\Users\Administrator\Anaconda3\envs\TensorFlow\lib\site-packages\ipykernel_launc
her.py:3: DeprecationWarning:     `imread` is deprecated!
    `imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
    Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports until
C:\Users\Administrator\Anaconda3\envs\TensorFlow\lib\site-packages\ipykernel_launc
her.py:10: DeprecationWarning:     `imresize` is deprecated!
    `imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
    Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
C:\Users\Administrator\Anaconda3\envs\TensorFlow\lib\site-packages\ipykernel_launc
her.py:11: DeprecationWarning:     `imresize` is deprecated!
    `imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
    Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```

png

png

Convolution: Naive backward pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b, conv_
param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_
param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_
param)[0], b, dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around e-8 or less.
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
```

```
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

Max-Pooling: Naive forward

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be on the order of e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

Max-Pooling: Naive backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x, pool_pa
ram)[0], x, dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be on the order of e-12
```

```
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.27562514223145e-12
```

Fast layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py`.

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

NOTE: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

```
# Rel errors should be around e-9 or less
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
```

```
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))

# Relative errors should be close to 0.0
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.133607s
fast: 0.002985s
speedup: 44.755770x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.404671s
fast: 0.009974s
speedup: 40.574536x
dx difference:  0.0
```

Convolutional "sandwich" layers

Previously we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks.

```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
```

```
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w, b, c
onv_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, c
onv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, c
onv_param, pool_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b, conv_p
aram)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_p
aram)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_p
aram)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

Three-layer ConvNet

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

Sanity check loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for C classes. When we add regularization this should go up.

```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)
```

Gradient check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(num_filters=3, filter_size=3,
                          input_dim=input_dim, hidden_dim=7,
                          dtype=np.float64)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verbose=
False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, gra
ds[param_name])))
```

Overfit small data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
np.random.seed(231)

num_train = 100
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(model, small_data,
```

```
                num_epochs=15, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=1)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
```

```
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```

Train the net

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(model, data,
                num_epochs=1, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
```

```
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```

Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully-connected networks. As proposed in the original paper [3], batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect the statistics of each feature channel to be relatively consistent both between different images and different locations within the same image. Therefore spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over both the minibatch dimension `N` and the spatial dimensions `H` and `W`.

[3] [Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.](https://arxiv.org/abs/1502.03167)

Spatial batch normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x.mean(axis=(0, 2, 3)))
print('  Stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  Shape: ', out.shape)
print('  Means: ', out.mean(axis=(0, 2, 3)))
print('  Stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [9.33463814 8.90909116 9.11056338]
  Stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  Shape:  (2, 3, 4, 5)
  Means:  [ 5.85642645e-16  5.93969318e-16 -8.88178420e-17]
  Stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  Shape:  (2, 3, 4, 5)
  Means:  [6. 7. 8.]
  Stds:  [2.99999885 3.99999804 4.99999798]
```

```
np.random.seed(231)
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
```

```
# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

Spatial batch normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  3.083846820796372e-07
dgamma error:  7.09738489671469e-12
dbeta error:  3.275608725278405e-12
```

Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [4] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [5] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups, and a per-group per-datapoint normalization instead.

Comparison of normalization techniques discussed so far

Visual comparison of the normalization techniques discussed so far (image edited from [5])
Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional Computer Vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [6]--after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization. Note that this normalization technique that you are to implement in the following cells was introduced and published to arXiv *less than a month ago* -- this truly is still an ongoing and excitingly active field of research!

[4] [Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.](https://arxiv.org/pdf/1607.06450.pdf)

[5] [Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).](https://arxiv.org/abs/1803.08494)

[6] [N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.](https://ieeexplore.ieee.org/abstract/document/1467360/)

Group normalization: forward

In the file `cs231n/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

```
np.random.seed(231)
# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization

N, C, H, W = 2, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  Shape: ', x.shape)
print('  Means: ', x_g.mean(axis=1))
print('  Stds: ', x_g.std(axis=1))

# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  Shape: ', out.shape)
print('  Means: ', out_g.mean(axis=1))
print('  Stds: ', out_g.std(axis=1))
```

```
Before spatial group normalization:
  Shape:  (2, 6, 4, 5)
  Means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  Stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  Shape:  (1, 1, 1, 2, 6, 4, 5)
  Means:  [-2.14643118e-16  5.25505565e-16  2.58126853e-16 -3.62672855e-16]
  Stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

Spatial group normalization: backward

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

```
np.random.seed(231)
N, C, H, W = 2, 6, 4, 5
G = 2
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)
#You should expect errors of magnitudes between 1e-12~1e-07
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  6.34590431845254e-08
dgamma error:  1.0546047434202244e-11
dbeta error:  3.810857316122484e-12
```

Part B

What's this PyTorch business?

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you switch over to that notebook).

What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

Why?

- Our code will now run on GPUs! Much faster training. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- We want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- We want you to be exposed to the sort of deep learning code you might run into in academia or industry.

PyTorch versions

This notebook assumes that you are using **PyTorch version 0.4**. Prior to this version, Tensors had to be wrapped in Variable objects to be used in autograd; however Variables have now been deprecated. In addition 0.4 also separates a Tensor's datatype from its device, and uses numpy-style factories for constructing Tensors rather than directly invoking Tensor constructors.

How will I learn PyTorch?

Justin Johnson has made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

Table of Contents

This assignment has 5 parts. You will learn PyTorch on different levels of abstractions, which will help you understand it better and prepare you for the final project.

1. Preparation: we will use CIFAR-10 dataset.
2. Barebones PyTorch: we will work directly with the lowest-level PyTorch Tensors.
3. PyTorch Module API: we will use `nn.Module` to define arbitrary neural network architecture.
4. PyTorch Sequential API: we will use `nn.Sequential` to define a linear feed-forward network very conveniently.
5. CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`nn.Module`	High	Medium
`nn.Sequential`	Low	High

Part I. Preparation

First, we load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np

NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000
)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

You have an option to **use GPU by setting the flag to True below**. It is not necessary to use GPU for this assignment. Note that if your computer does not have CUDA enabled, `torch.cuda.is_available()` will return False and this notebook will fallback to CPU mode.

The global variables `dtype` and `device` will control the data types throughout this assignment.

```
USE_GPU = True

dtype = torch.float32 # we will be using float throughout this tutorial
```

```
if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss
print_every = 100

print('using device:', device)
```

```
using device: cuda
```

Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the scalar loss at the end.

PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector -- it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector pe
r image

def test_flatten():
```

```
    x = torch.arange(12).view(2, 1, 3, 2)
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[  0.,   1.],
          [  2.,   3.],
          [  4.,   5.]]],


        [[[  6.,   7.],
          [  8.,   9.],
          [ 10.,  11.]]]])
After flattening:  tensor([[  0.,   1.,   2.,   3.,   4.,   5.],
        [  6.,   7.,   8.,   9.,  10.,  11.]])
```

Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the network;
      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification scores for
      the input data x.
    """
    # first we flatten the image
    x = flatten(x)  # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Since w1 and
    # w2 have requires_grad=True, operations involving these Tensors will cause
    # PyTorch to build a computational graph, allowing automatic computation of
    # gradients. Since we are no longer implementing the backward pass by hand we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
```

```
    x = F.relu(x.mm(w1))
    x = x.mm(w2)
    return x


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature dimension
 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1` x `KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2` x `KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

HINT: For convolutions: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; pay attention to the shapes of convolutional filters!

```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
        for the first convolutional layer
      - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the first
        convolutional layer
      - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2) giving
        weights for the second convolutional layer
      - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the second
        convolutional layer
      - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can you
        figure out what the shape should be?
      - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can you
        figure out what the shape should be?

    Returns:
```

```
    - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ################################################################################
##
    # TODO: Implement the forward pass for the three-layer ConvNet.
 #
    ################################################################################
##

    conv1 = F.conv2d(x, weight=conv_w1, bias=conv_b1, padding=2)
    relu1 = F.relu(conv1)
    conv2 = F.conv2d(relu1, weight=conv_w2, bias=conv_b2, padding=1)
    relu2 = F.relu(conv2)
    relu2_flat = flatten(relu2)
    scores = relu2_flat.mm(fc_w) + fc_b

    ################################################################################
##
    #                                 END OF YOUR CODE
 #
    ################################################################################
##
    return scores
```

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

When you run this function, scores should have shape (64, 10).

```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image size
 [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel, in_channel,
kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel, in_channel,
kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before the
 fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_
b])
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.

- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The `random_weight` function uses the Kaiming normal initialization method, described in:

He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015, https://arxiv.org/abs/1502.01852

```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH, kW
]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

```
tensor([[-0.0183, -0.0282,  0.5041, -0.6743,  0.1662],
        [-0.3404,  0.4510,  0.3055, -0.3067, -0.4398],
        [-0.5025, -0.4530, -0.2349,  2.3406,  1.0483]], device='cuda:0')
```

Barebones PyTorch: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
```

```
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 * ac
c))
```

BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can read about it here.

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

```
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model.
      It should have the signature scores = model_fn(x, params) where x is a
      PyTorch Tensor of image data, params is a list of PyTorch Tensors giving
      model weights, and scores is a PyTorch Tensor of shape (N, C) giving
      scores for the elements in x.
    - params: List of PyTorch Tensors giving weights for the model
    - learning_rate: Python scalar giving the learning rate to use for SGD

    Returns: Nothing
    """
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss
        scores = model_fn(x, params)
        loss = F.cross_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational
        # graph has requires_grad=True and uses backpropagation to compute the
        # gradient of the loss with respect to these Tensors, and stores the
        # gradients in the .grad attribute of each Tensor.
        loss.backward()

        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the backward pass
```

```
            w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 40% after training for one epoch.

```
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 3.3849
Checking accuracy on the val set
Got 161 / 1000 correct (16.10%)

Iteration 100, loss = 2.1713
Checking accuracy on the val set
Got 315 / 1000 correct (31.50%)

Iteration 200, loss = 2.1726
Checking accuracy on the val set
Got 393 / 1000 correct (39.30%)

Iteration 300, loss = 2.3629
Checking accuracy on the val set
Got 391 / 1000 correct (39.10%)

Iteration 400, loss = 1.6259
Checking accuracy on the val set
Got 419 / 1000 correct (41.90%)

Iteration 500, loss = 1.7335
Checking accuracy on the val set
Got 440 / 1000 correct (44.00%)

Iteration 600, loss = 1.7878
Checking accuracy on the val set
Got 435 / 1000 correct (43.50%)

Iteration 700, loss = 1.7003
```

```
Checking accuracy on the val set
Got 448 / 1000 correct (44.80%)
```

BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

```
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

################################################################################
# TODO: Initialize the parameters of a three-layer ConvNet.                    #
################################################################################

conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight((channel_1,))
conv_w2 = random_weight((channel_2, 32, 3, 3))
conv_b2 = zero_weight((channel_2,))
fc_w = random_weight((channel_2*32*32, 10))
fc_b = zero_weight((10,))

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

```
Iteration 0, loss = 3.2011
Checking accuracy on the val set
Got 118 / 1000 correct (11.80%)

Iteration 100, loss = 1.8087
Checking accuracy on the val set
Got 350 / 1000 correct (35.00%)

Iteration 200, loss = 1.7734
```

```
Checking accuracy on the val set
Got 409 / 1000 correct (40.90%)

Iteration 300, loss = 1.4523
Checking accuracy on the val set
Got 445 / 1000 correct (44.50%)

Iteration 400, loss = 1.5071
Checking accuracy on the val set
Got 447 / 1000 correct (44.70%)

Iteration 500, loss = 1.7607
Checking accuracy on the val set
Got 475 / 1000 correct (47.50%)

Iteration 600, loss = 1.5027
Checking accuracy on the val set
Got 463 / 1000 correct (46.30%)

Iteration 700, loss = 1.5302
Checking accuracy on the val set
Got 498 / 1000 correct (49.80%)
```

Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the doc for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
```

```
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64, feature d
imension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

```
torch.Size([64, 10])
```

Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: http://pytorch.org/docs/stable/nn.html#conv2d

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 10)` for the shape of the output scores.

```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
        # architecture defined above.                                          #
        ########################################################################

        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2, bi
as=True)
        nn.init.kaiming_normal_(self.conv1.weight)
        nn.init.constant_(self.conv1.bias, 0)

        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1, bia
s=True)
```

```
        nn.init.kaiming_normal_(self.conv2.weight)
        nn.init.constant_(self.conv2.bias, 0)

        self.fc = nn.Linear(channel_2*32*32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        nn.init.constant_(self.fc.bias, 0)

        ############################################################################
        #                            END OF YOUR CODE                              #

        ############################################################################

    def forward(self, x):
        scores = None
        ############################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you      #
        # should use the layers you defined in __init__ and specify the        #
        # connectivity of those layers in forward()                            #
        ############################################################################

        relu1 = F.relu(self.conv1(x))
        relu2 = F.relu(self.conv2(relu1))
        scores = self.fc(flatten(relu2))

        ############################################################################
        #                                END OF YOUR CODE                          #
        ############################################################################
        return scores


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image size
 [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8, num_classes
=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 10])
```

Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
```

```
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * acc)
)
```

Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.

```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train for

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the optimizer
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

You don't need to tune any hyperparameters, but you should see model accuracies above 40% after training for one epoch.

```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.4818
Checking accuracy on validation set
Got 121 / 1000 correct (12.10)

Iteration 100, loss = 1.8961
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Iteration 200, loss = 2.0630
Checking accuracy on validation set
Got 400 / 1000 correct (40.00)

Iteration 300, loss = 1.9615
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)

Iteration 400, loss = 1.8572
Checking accuracy on validation set
Got 415 / 1000 correct (41.50)

Iteration 500, loss = 2.1470
Checking accuracy on validation set
Got 429 / 1000 correct (42.90)

Iteration 600, loss = 1.7190
Checking accuracy on validation set
Got 423 / 1000 correct (42.30)

Iteration 700, loss = 1.6291
Checking accuracy on validation set
Got 451 / 1000 correct (45.10)
```

Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve above above 45% after training for one epoch.

You should train the model using stochastic gradient descent without momentum.

```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
```

```
################################################################################

model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

################################################################################
#                                END OF YOUR CODE
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.8759
Checking accuracy on validation set
Got 112 / 1000 correct (11.20)

Iteration 100, loss = 1.7760
Checking accuracy on validation set
Got 355 / 1000 correct (35.50)

Iteration 200, loss = 1.7987
Checking accuracy on validation set
Got 395 / 1000 correct (39.50)

Iteration 300, loss = 1.7966
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)

Iteration 400, loss = 1.6835
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)

Iteration 500, loss = 1.5035
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Iteration 600, loss = 1.3668
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Iteration 700, loss = 1.6702
Checking accuracy on validation set
Got 462 / 1000 correct (46.20)
```

Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology than a feed-forward stack, but it's good enough for many use cases.

Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

```
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.4076
Checking accuracy on validation set
Got 178 / 1000 correct (17.80)

Iteration 100, loss = 1.8445
Checking accuracy on validation set
Got 391 / 1000 correct (39.10)

Iteration 200, loss = 2.1736
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Iteration 300, loss = 2.3242
Checking accuracy on validation set
Got 416 / 1000 correct (41.60)

Iteration 400, loss = 1.6490
Checking accuracy on validation set
Got 412 / 1000 correct (41.20)

Iteration 500, loss = 1.5034
Checking accuracy on validation set
Got 448 / 1000 correct (44.80)

Iteration 600, loss = 1.7844
Checking accuracy on validation set
Got 427 / 1000 correct (42.70)

Iteration 700, loss = 2.0401
```

```
Checking accuracy on validation set
Got 432 / 1000 correct (43.20)
```

Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see accuracy above 55% after one epoch of training.

```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

################################################################################
# TODO: Rewrite the 3-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################

model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2*32*32, 10),
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

# Weight initialization
# Ref: http://pytorch.org/docs/stable/nn.html#torch.nn.Module.apply
def init_weights(m):
    # print(m)
    if type(m) == nn.Conv2d or type(m) == nn.Linear:
        random_weight(m.weight.size())
        zero_weight(m.bias.size())

model.apply(init_weights)

################################################################################
#                                END OF YOUR CODE
################################################################################
```

```
train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.2960
Checking accuracy on validation set
Got 116 / 1000 correct (11.60)

Iteration 100, loss = 1.5991
Checking accuracy on validation set
Got 434 / 1000 correct (43.40)

Iteration 200, loss = 1.4879
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Iteration 300, loss = 1.3116
Checking accuracy on validation set
Got 499 / 1000 correct (49.90)

Iteration 400, loss = 1.3159
Checking accuracy on validation set
Got 562 / 1000 correct (56.20)

Iteration 500, loss = 1.2486
Checking accuracy on validation set
Got 542 / 1000 correct (54.20)

Iteration 600, loss = 1.1722
Checking accuracy on validation set
Got 566 / 1000 correct (56.60)

Iteration 700, loss = 1.3335
Checking accuracy on validation set
Got 591 / 1000 correct (59.10)
```

Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?

- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
- ResNets where the input from the previous layer is added to the output.
- DenseNets where inputs into previous layers are concatenated together.
- This blog has an in-depth overview

Have fun and happy training!

```
################################################################################
# TODO:                                                                        #

# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.      #
#                                                                              #
# Note that you can use the check_accuracy function to evaluate on either      #
# the test set or the validation set, by passing either loader_test or         #
# loader_val as the second argument to check_accuracy. You should not touch    #
# the test set until you have finished your architecture and  hyperparameter   #
# tuning, and only run the test set once at the end to report a final value.   #
```

```
################################################################################
model = None
optimizer = None

# A 4-layer convolutional network
# (conv -> batchnorm -> relu -> maxpool) * 3 -> fc
layer1 = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=5, padding=2),
    nn.BatchNorm2d(16),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer2 = nn.Sequential(
    nn.Conv2d(16, 32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

layer3 = nn.Sequential(
    nn.Conv2d(32, 64, kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2)
)

fc = nn.Linear(64*4*4, 10)

model = nn.Sequential(
    layer1,
    layer2,
    layer3,
    Flatten(),
    fc
)

learning_rate = 1e-3

optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Print training status every epoch: set print_every to a large number
print_every = 10000

################################################################################
#                                END OF YOUR CODE
################################################################################

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

```
Iteration 0, loss = 2.3787
Checking accuracy on validation set
Got 149 / 1000 correct (14.90)

Iteration 0, loss = 0.7987
Checking accuracy on validation set
Got 639 / 1000 correct (63.90)
```

```
Iteration 0, loss = 0.7713
Checking accuracy on validation set
Got 677 / 1000 correct (67.70)

Iteration 0, loss = 0.7390
Checking accuracy on validation set
Got 726 / 1000 correct (72.60)

Iteration 0, loss = 0.8759
Checking accuracy on validation set
Got 719 / 1000 correct (71.90)

Iteration 0, loss = 0.8025
Checking accuracy on validation set
Got 750 / 1000 correct (75.00)

Iteration 0, loss = 0.3281
Checking accuracy on validation set
Got 737 / 1000 correct (73.70)

Iteration 0, loss = 0.4678
Checking accuracy on validation set
Got 737 / 1000 correct (73.70)

Iteration 0, loss = 0.7667
Checking accuracy on validation set
Got 742 / 1000 correct (74.20)

Iteration 0, loss = 0.5843
Checking accuracy on validation set
Got 739 / 1000 correct (73.90)
```

Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

TODO: Describe what you did

Test set -- run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy.

```
best_model = model
check_accuracy_part34(loader_test, best_model)

Checking accuracy on test set
Got 7379 / 10000 correct (73.79)
```

5 REFERENCES

[1] "How to Convert an RGB Image to Grayscale," *KDnuggets*. [Online]. Available: https://www.kdnuggets.com/2019/12/convert-rgb-image-grayscale.html. [Accessed: 13-Dec-2020].

[2] "Grayscale to RGB Conversion," *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm. [Accessed: 13-Dec-2020].

[3] *Unsupervised Feature Learning and Deep Learning Tutorial*. [Online]. Available: http://ufldl.stanford.edu/tutorial/unsupervised/Autoencoders/. [Accessed: 13-Dec-2020].

[4] Jeremy Jordan, "Introduction to autoencoders.," *Jeremy Jordan*, 19-Mar-2018. [Online]. Available: https://www.jeremyjordan.me/autoencoders/. [Accessed: 13-Dec-2020].

[5] F. Chollet, "The Keras Blog," *The Keras Blog ATOM*. [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html. [Accessed: 13-Dec-2020].

[6] A. Jana, A. S. says, and A. Sharma, "Understanding and implementing Neural Network with SoftMax in Python from scratch," *A Developer Diary*, 29-Apr-2019. [Online]. Available: https://www.adeveloperdiary.com/data-science/deep-learning/neural-network-with-softmax-in-python/. [Accessed: 13-Dec-2020].

[7] "Backpropagation in a convolutional layer," *NEODELPHIS*, 10-Jul-2019. [Online]. Available: https://neodelphis.github.io/convnet/maths/python/english/2019/07/10/convnet-bp-en.html. [Accessed: 13-Dec-2020].

[8] N. Kumar, "Batch Normalization and Dropout in Neural Networks Explained with Pytorch," *Medium*, 17-Dec-2019. [Online]. Available: https://towardsdatascience.com/batch-normalization-and-dropout-in-neural-networks-explained-with-pytorch-47d7a8459bcd. [Accessed: 13-Dec-2020].

[9] "Building your Recurrent Neural Network - Step by Step¶," *Building a Recurrent Neural Network - Step by Step - v3*. [Online]. Available: https://jmyao17.github.io/Machine_Learning/Sequence/RNN-1.html. [Accessed: 13-Dec-2020].

[10] "Recurrent Neural Networks cheatsheet Star," *CS 230 - Recurrent Neural Networks Cheatsheet*. [Online]. Available: https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks. [Accessed: 13-Dec-2020].

[11] W. Wolf, "Will Wolf," *will wolf*, 18-Oct-2016. [Online]. Available: http://willwolf.io/2016/10/18/recurrent-neural-network-gradients-and-lessons-learned-therein/. [Accessed: 13-Dec-2020].

[12] J. Ramapuram, "RNN Backprop Through Time Equations," *Back Propaganda*, 06-Jun-2016. [Online]. Available: https://jramapuram.github.io/ramblings/rnn-backrpop/. [Accessed: 13-Dec-2020].

[13] "Understanding LSTM Networks," *Understanding LSTM Networks -- colah's blog*. [Online]. Available: http://colah.github.io/posts/2015-08-Understanding-LSTMs/. [Accessed: 13-Dec-2020].

[14] S. Kostadinov, "Understanding GRU Networks," *Medium*, 10-Nov-2019. [Online]. Available: https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be. [Accessed: 13-Dec-2020].

[15] M. Khandekar, “Forward and Backpropagation in GRUs-Derived: Deep Learning,” *Medium*, 07-Apr-2020. [Online]. Available: https://towardsdatascience.com/forward-and-backpropagation-in-grus-derived-deep-learning-5764f374f3f5. [Accessed: 13-Dec-2020].