Assignment 5:

Name: Ameya Vishwanath konkar

UID: 118191058

Please submit to ELMS

- a PDF containing all outputs (by executing **Run all**)
- your ipynb notebook containing all the code

I understand the policy on academic integraty (collaboration and the use of online material). Please sign your name here:

```
# import the necessary packages
import numpy as np
import gzip, os
from urllib.request import urlretrieve
from random import random
from math import exp
from random import seed
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Part 1: Backpropagation in Neural Networks (20 Points)

Overview

Artificial Neural Networks are computational learning systems that uses a network of functions to understand and translate a data input of one form into a desired output, usually in another form. The concept of the artificial neural network was inspired by human biology and the way neurons of the human brain function together to understand inputs from human senses.

A simple neural network consists of Input Layer, Hidden Layer and Output Layer. To train these the network, we will use Backpropagation algorithm. Backpropagation is the cornerstone of modern neural networks. To understand the algorithm in details, here is a mathematical description in the Chapter 2 of *How the backpropagation algorithm works from Neural Networks and Deep Learning* (http://neuralnetworksanddeeplearning.com/chap2.html).

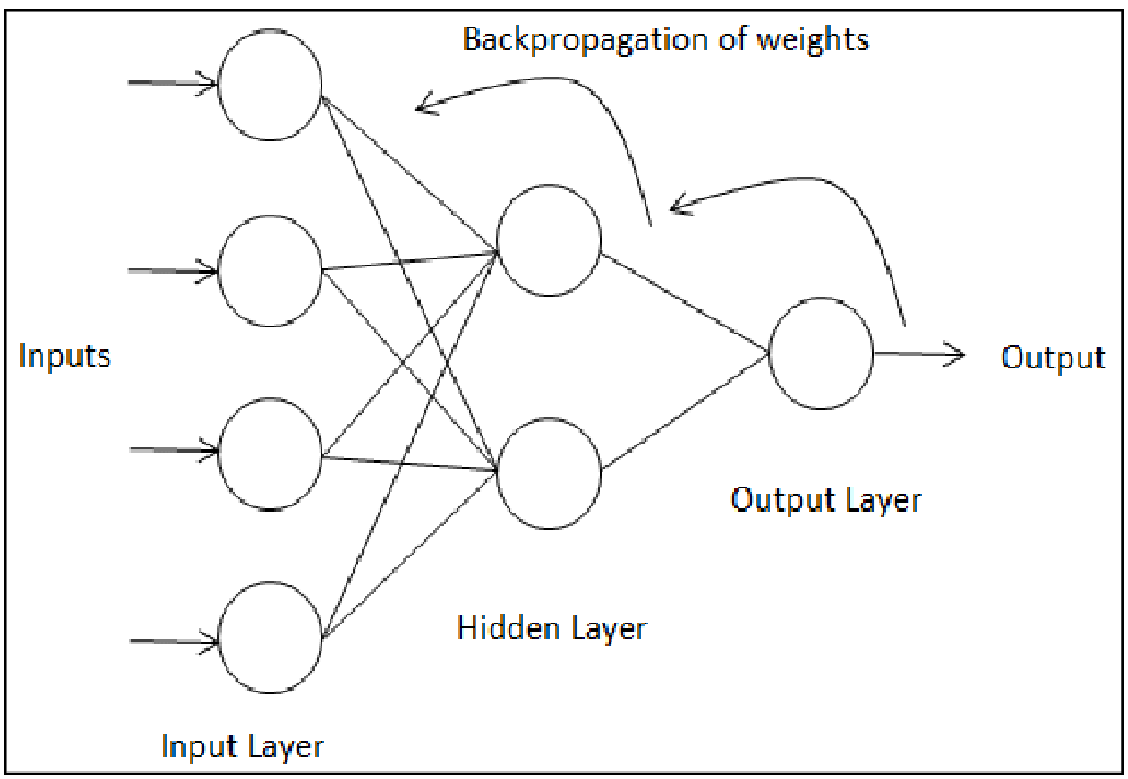
In this part, you are required to implement the following architecture and write training code of a neural network from scratch using the numpy library alone.

Architecture Definition :

- An Input Layer with the following 2-dimensions:
- 0: Batch Size
- 1: 784 = 28*28 pixels
- A hidden layer with 500 units
- A second hidden layer with 50 units
- An output layer with 10 units

There are five major steps to the implementation:

1. Define neural network: initialize_network()
2. Forward Propagation: pre_activation(), sigmoid_activation(), forward_propagation()
3. Backpropagation: backward_propagate_error()
4. Loss function and updation of weights (SGD): update_weights()
5. Training: train()



Data

```
# Download Data -- run this cell only one time per runtime
!gdown 11SpETIc56PReKuaUKEwWDvdkiynyyGFA
!unzip "/content/MNISTArchive.zip" -d "/content/"
!gzip -d "/content/t10k-labels-idx1-ubyte.gz"
!gzip -d "/content/t10k-images-idx3-ubyte.gz"
```

```
!gzip -d "/content/train-labels-idx1-ubyte.gz"
!gzip -d "/content/train-images-idx3-ubyte.gz"
```

```
Downloading...
From: https://drive.google.com/uc?id=11SpETIc56PReKuaUKEwWDvdkiynyyGFA
To: /content/MNISTArchive.zip
   0% 0.00/11.6M [00:00<?, ?B/s] 100% 11.6M/11.6M [00:00<00:00,
137MB/s]
Archive:  /content/MNISTArchive.zip
replace /content/t10k-labels-idx1-ubyte.gz? [y]es, [n]o, [A]ll,
[N]one, [r]ename: N
gzip: /content/t10k-labels-idx1-ubyte already exists; do you wish to
overwrite (y or n)? n
	not overwritten
gzip: /content/t10k-images-idx3-ubyte already exists; do you wish to
overwrite (y or n)? n
	not overwritten
gzip: /content/train-labels-idx1-ubyte already exists; do you wish to
overwrite (y or n)? n
	not overwritten
gzip: /content/train-images-idx3-ubyte already exists; do you wish to
overwrite (y or n)? n
	not overwritten
```

Helper Functions:

Code (10 pts)

```
def read_mnist(path=None):
    r"""Return (train_images, train_labels, test_images, test_labels).

    Args:
        path (str): Directory containing MNIST. Default is
            /home/USER/data/mnist or C:\Users\USER\data\mnist.
            Create if nonexistant. Download any missing files.

    Returns:
        Tuple of (train_images, train_labels, test_images,
test_labels), each
            a matrix. Rows are examples. Columns of images are pixel
values.
            Columns of labels are a onehot encoding of the correct
class.
    """
    url = 'http://yann.lecun.com/exdb/mnist/'
    files = ['train-images-idx3-ubyte.gz',
             'train-labels-idx1-ubyte.gz',
             't10k-images-idx3-ubyte.gz',
             't10k-labels-idx1-ubyte.gz']

    if path is None:
```

```python
        # Set path to /home/USER/data/mnist or C:\Users\USER\data\
mnist
        path = os.path.join(os.path.expanduser('~'), 'data', 'mnist')

    # Create path if it doesn't exist
    os.makedirs(path, exist_ok=True)

    # Download any missing files
    for file in files:
        if file not in os.listdir(path):
            urlretrieve(url + file, os.path.join(path, file))
            print("Downloaded %s to %s" % (file, path))

    def _images(path):
        """Return images loaded locally."""
        with gzip.open(path) as f:
            # First 16 bytes are magic_number, n_imgs, n_rows, n_cols
            pixels = np.frombuffer(f.read(), 'B', offset=16)
        return pixels.reshape(-1, 784).astype('float32') / 255

    def _labels(path):
        """Return labels loaded locally."""
        with gzip.open(path) as f:
            # First 8 bytes are magic_number, n_labels
            integer_labels = np.frombuffer(f.read(), 'B', offset=8)

        def _onehot(integer_labels):
            """Return matrix whose rows are onehot encodings of
integers."""
            n_rows = len(integer_labels)
            n_cols = integer_labels.max() + 1
            onehot = np.zeros((n_rows, n_cols), dtype='uint8')
            onehot[np.arange(n_rows), integer_labels] = 1
            return onehot

        return _onehot(integer_labels)

    train_images = _images(os.path.join(path, files[0]))
    train_labels = _labels(os.path.join(path, files[1]))
    test_images = _images(os.path.join(path, files[2]))
    test_labels = _labels(os.path.join(path, files[3]))

    return train_images, train_labels, test_images, test_labels

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
     network = list()

     # W1 = np.random.rand(n_inputs, n_hidden)
     W1 = np.random.normal(0, 0.01, (n_inputs, n_hidden))
```

```python
    b1 = np.zeros((1, n_hidden))
    hidden_layer = [W1, b1]
    ## Write your code. Initialize hidden layer here.
    network.append(hidden_layer)

    # W2 = np.random.rand(n_hidden, n_outputs)
    W2 = np.random.normal(0, 0.01, (n_hidden, n_outputs))
    b2 = np.zeros((1, n_outputs))
    output_layer = [W2, b2]
    ## Write your code. Initialize output_layer layer here.
    network.append(output_layer)
    return network

# Forward Propagation:
def forward_propagation(network, inputs):
    input_cache = []
    inputs_ = inputs
    for layer in network:
        new_inputs = []
        ## write you code here.
        ## for each hidden neuron this 'layer', compute \
        ## pre_activation, sigmoid_activation and save then output
in 'new_inputs.'
        input_cache.append(inputs_)
        new_inputs = pre_activation(layer, inputs)
        # input_cache.append(new_inputs)
        inputs = sigmoid_activation(new_inputs)
        inputs_ = inputs
    # inputs = new_inputs
    return inputs, input_cache

# softmax loss
def softmax_loss(x, y):

        loss = 0.0
        scores = x
        max_scores = np.amax(scores, axis=1)[:, None] # (N,) #
subtracted by the max value of the score to avoid instability cause by
a very high value.
        scores_exp = np.exp(scores - max_scores)      # extract
correct score from the scores matrix
        loss += -np.sum(np.log(scores_exp[np.arange(0, x.shape[0]),
y])) + np.sum(np.log(np.sum(scores_exp, axis=1)))  # loss = -y +
log(sum(e.^(scores)))

        S = scores_exp / np.sum(scores_exp, axis=1)[:,None] # (N,
C)  # divide a exponential of a particular class score with the sum of
exp of all the class scores
        S_corr = S - np.equal(np.arange(x.shape[1]), y[:,None]) #
(N, C)
```

```
        dx = S_corr/x.shape[0]
        loss /= x.shape[0]
        return loss, dx

def normal_loss(x, y):
    loss = 0.0
    loss = np.sum(np.subtract(x,y))**2/x.shape[0]**2
    dx = sigmoid_derivative(np.subtract(x,y), 1)/x.shape[0]
    return loss, dx

def affine_backward(dout, cache, network):
        x = cache
        [w, b] = network
        dx, dw, db = None, None, None

        input_dimensions = np.prod(x.shape[1:])
        x_new = np.reshape(x, (x.shape[0],input_dimensions))
# reshape each input into a vector of dimension D = d_1 * ... * d_k
        dw = x_new.T.dot(dout)
# calculate dw = x_new.T*dout
        dx = np.reshape(dout.dot(w.T), x.shape)
# calculate dx = dout*w.T
        db = np.sum(dout, axis = 0)
# calculate db = dout
        return dx, dw, db

# Backpropagation:
def backward_propagate_error(network, input_cache, grad_x):

    grads = []
    back_grad = grad_x
    j=len(network)-1
    for i in reversed(range(len(network))):
        layer = network[i]
        dh, dW, db = affine_backward(back_grad, input_cache[i],
network[i])    # backward pass to calculate dL/dW2, dL/db2, dL/dh
based on dL/dscores and input parameters
        grads.append([dW, db])
        back_grad = sigmoid_derivative(input_cache[i], dh)
# sigmoid backward to find dL/da1 from dL/dh
        # j -= 2
    grads.reverse()
    return grads

# Stochastic GD for weight updation:
def update_weights(network, grads, l_rate):
    for i in range(len(network)):
        [dW, db] = grads[i]
        network[i][0] -= l_rate*dW
        network[i][1] -= l_rate*db
```

```
    return network

# Train a network for a fixed number of epochs
def train(network, train_x,train_y, l_rate, n_epoch, n_outputs):
    losses = []
    for epoch in range(n_epoch):
        outputs, input_cache = forward_propagation(network,
train_x)
        # loss, grad_x = softmax_loss(outputs, train_y)
        loss, grad_x = normal_loss(outputs, train_y)
        grads = backward_propagate_error(network, input_cache,
grad_x)
        network = update_weights(network, grads, l_rate)
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate,
loss))
        losses.append(loss)
    return losses

# Calculate neuron activation for an input
def pre_activation(weights, inputs):
    activation = weights[-1]
    [W, b] = weights
    input_dimensions = np.prod(inputs.shape[1:])
    x_new = np.reshape(inputs, (inputs.shape[0],input_dimensions))
# reshape each input into a vector of dimension D = d_1 * ... * d_k

    out = np.add(x_new.dot(W), b)                          # out = WX
+ b
    return out

def sigmoid_activation(activation):
    out_sigmoid= 1/(1 + np.exp(-activation))
    ## write code. implement sigmoid function
    return out_sigmoid

# Calculate the derivative of a neuron output
def sigmoid_derivative(x, dout):
    ## write code. implement sigmoid function
    out_sigmoid_deriv = x*(1-x)*dout
    return out_sigmoid_deriv

# 1. Test your code for backprop algorithm on this sample dataset.
seed(1)
sample_dataset = [[2.7810836,2.550537003,0],
    [1.465489372,2.362125076,0],
    [3.396561688,4.400293529,0],
    [1.38807019,1.850220317,0],
    [3.06407232,3.005305973,0],
    [7.627531214,2.759262235,1],
    [5.332441248,2.088626775,1],
```

```python
    [6.922596716,1.77106367,1],
    [8.675418651,-0.242068655,1],
    [7.673756466,3.508563011,1]]

n_inputs = len(sample_dataset[0]) - 1
n_outputs = len(set([sample[-1] for sample in sample_dataset]))
network = initialize_network(n_inputs, 2, n_outputs)

train_ = np.array(sample_dataset)
train_x = train_[:, :-1]
train_y_temp = train_[:, -1].astype(int)
train_y = np.zeros((train_.shape[0], n_outputs))
train_y[np.arange(0, train_.shape[0]), train_y_temp] = 1

loss_init = train(network, train_x, train_y, l_rate=0.002, n_epoch=50,
n_outputs=n_outputs)
# for layer in network:
#     print(layer)
```

```
>epoch=0, lrate=0.002, error=0.324
>epoch=1, lrate=0.002, error=0.324
>epoch=2, lrate=0.002, error=0.324
>epoch=3, lrate=0.002, error=0.324
>epoch=4, lrate=0.002, error=0.324
>epoch=5, lrate=0.002, error=0.324
>epoch=6, lrate=0.002, error=0.324
>epoch=7, lrate=0.002, error=0.323
>epoch=8, lrate=0.002, error=0.323
>epoch=9, lrate=0.002, error=0.323
>epoch=10, lrate=0.002, error=0.323
>epoch=11, lrate=0.002, error=0.323
>epoch=12, lrate=0.002, error=0.323
>epoch=13, lrate=0.002, error=0.323
>epoch=14, lrate=0.002, error=0.323
>epoch=15, lrate=0.002, error=0.323
>epoch=16, lrate=0.002, error=0.322
>epoch=17, lrate=0.002, error=0.322
>epoch=18, lrate=0.002, error=0.322
>epoch=19, lrate=0.002, error=0.322
>epoch=20, lrate=0.002, error=0.322
>epoch=21, lrate=0.002, error=0.322
>epoch=22, lrate=0.002, error=0.322
>epoch=23, lrate=0.002, error=0.322
>epoch=24, lrate=0.002, error=0.322
>epoch=25, lrate=0.002, error=0.321
>epoch=26, lrate=0.002, error=0.321
>epoch=27, lrate=0.002, error=0.321
>epoch=28, lrate=0.002, error=0.321
>epoch=29, lrate=0.002, error=0.321
>epoch=30, lrate=0.002, error=0.321
```

```
>epoch=31, lrate=0.002, error=0.321
>epoch=32, lrate=0.002, error=0.321
>epoch=33, lrate=0.002, error=0.321
>epoch=34, lrate=0.002, error=0.321
>epoch=35, lrate=0.002, error=0.320
>epoch=36, lrate=0.002, error=0.320
>epoch=37, lrate=0.002, error=0.320
>epoch=38, lrate=0.002, error=0.320
>epoch=39, lrate=0.002, error=0.320
>epoch=40, lrate=0.002, error=0.320
>epoch=41, lrate=0.002, error=0.320
>epoch=42, lrate=0.002, error=0.320
>epoch=43, lrate=0.002, error=0.320
>epoch=44, lrate=0.002, error=0.319
>epoch=45, lrate=0.002, error=0.319
>epoch=46, lrate=0.002, error=0.319
>epoch=47, lrate=0.002, error=0.319
>epoch=48, lrate=0.002, error=0.319
>epoch=49, lrate=0.002, error=0.319
```

```python
# 2. Read MNIST data and test above algorithm on it.

# Read MNIST data
train_images, train_labels, test_images, test_labels =
read_mnist(path='/content/')
print(train_images.shape, train_labels.shape)

# Run Backpropagation.
# Write you code here.
n_inputs = train_images.shape[1]
n_outputs = train_labels.shape[1]
network = initialize_network(n_inputs, 800, n_outputs)

train_x = train_images
train_y = train_labels
train(network, train_x, train_y, l_rate=0.02, n_epoch=25,
n_outputs=n_outputs)
```

```
(60000, 784) (60000, 10)
>epoch=0, lrate=0.020, error=17.940
>epoch=1, lrate=0.020, error=7.446
>epoch=2, lrate=0.020, error=3.159
>epoch=3, lrate=0.020, error=1.811
>epoch=4, lrate=0.020, error=1.344
>epoch=5, lrate=0.020, error=1.156
>epoch=6, lrate=0.020, error=1.073
>epoch=7, lrate=0.020, error=1.034
>epoch=8, lrate=0.020, error=1.016
>epoch=9, lrate=0.020, error=1.007
>epoch=10, lrate=0.020, error=1.002
```

```
>epoch=11, lrate=0.020, error=1.000
>epoch=12, lrate=0.020, error=0.999
>epoch=13, lrate=0.020, error=0.998
>epoch=14, lrate=0.020, error=0.997
>epoch=15, lrate=0.020, error=0.997
>epoch=16, lrate=0.020, error=0.996
>epoch=17, lrate=0.020, error=0.996
>epoch=18, lrate=0.020, error=0.996
>epoch=19, lrate=0.020, error=0.996
>epoch=20, lrate=0.020, error=0.995
>epoch=21, lrate=0.020, error=0.995
>epoch=22, lrate=0.020, error=0.995
>epoch=23, lrate=0.020, error=0.994
>epoch=24, lrate=0.020, error=0.994
```

```
[17.939659994912134,
 7.4458525621673814,
 3.159320119901842,
 1.8114591900923593,
 1.343799861045793,
 1.1555791343800796,
 1.0725015657553594,
 1.033952149298972,
 1.0155476180816203,
 1.0065758706836416,
 1.0020992006547804,
 0.9997839545530154,
 0.9985129503710439,
 0.9977477797273211,
 0.9972280233159323,
 0.9968274518249239,
 0.9964847926722702,
 0.9961702700726818,
 0.9958693984935675,
 0.9955751218307066,
 0.9952839966918168,
 0.9949943376975913,
 0.9947053155399709,
 0.9944165180618673,
 0.9941277365171273]
```

Write-up (10 pts)

1. You are required to report a) train error w.r.t epoch, b) train and test accuracy numbers on MNIST dataset at the end of training.

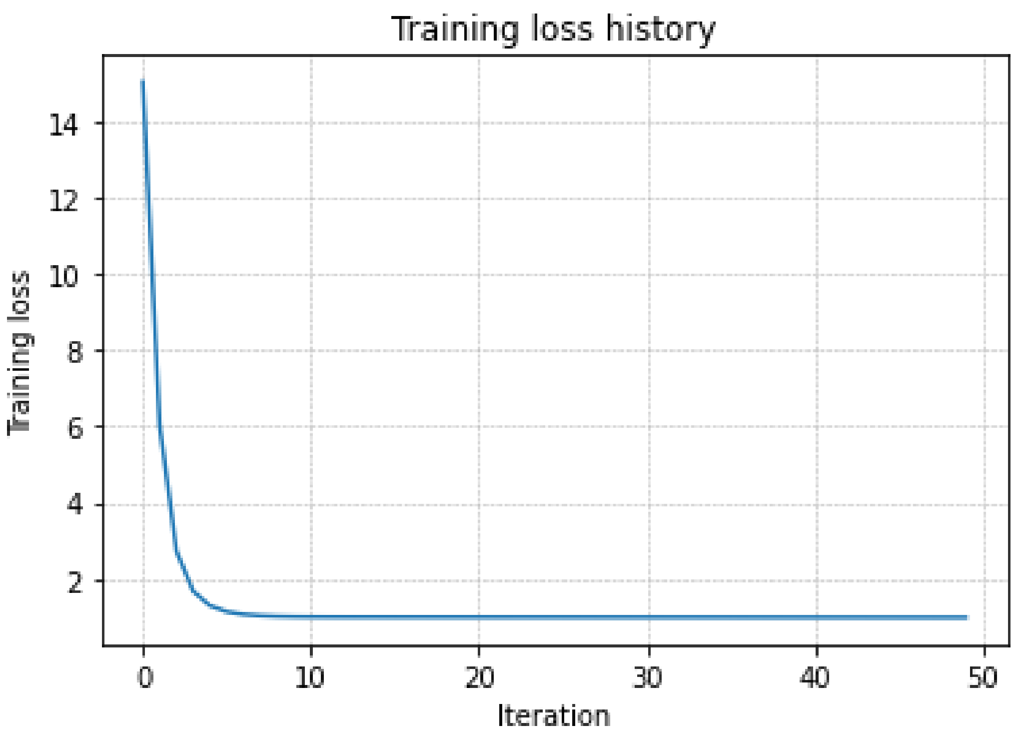
```
# 2. Read MNIST data and test above algorithm on it.

# Read MNIST data
train_images, train_labels, test_images, test_labels =
read_mnist(path='/content/')
print(train_images.shape, train_labels.shape)
```

```python
# Run Backpropagation.
# Write you code here.
n_inputs = train_images.shape[1]
n_outputs = train_labels.shape[1]
network = initialize_network(n_inputs, 800, n_outputs)

train_x = train_images
train_y = train_labels
losses = train(network, train_x, train_y, l_rate=0.02, n_epoch=50,
n_outputs=n_outputs)
```

```
(60000, 784) (60000, 10)
>epoch=0, lrate=0.020, error=15.055
>epoch=1, lrate=0.020, error=6.045
>epoch=2, lrate=0.020, error=2.720
>epoch=3, lrate=0.020, error=1.669
>epoch=4, lrate=0.020, error=1.290
>epoch=5, lrate=0.020, error=1.133
>epoch=6, lrate=0.020, error=1.062
>epoch=7, lrate=0.020, error=1.029
>epoch=8, lrate=0.020, error=1.013
>epoch=9, lrate=0.020, error=1.006
>epoch=10, lrate=0.020, error=1.002
>epoch=11, lrate=0.020, error=1.000
>epoch=12, lrate=0.020, error=0.998
>epoch=13, lrate=0.020, error=0.998
>epoch=14, lrate=0.020, error=0.997
>epoch=15, lrate=0.020, error=0.997
>epoch=16, lrate=0.020, error=0.996
>epoch=17, lrate=0.020, error=0.996
>epoch=18, lrate=0.020, error=0.996
>epoch=19, lrate=0.020, error=0.996
>epoch=20, lrate=0.020, error=0.995
>epoch=21, lrate=0.020, error=0.995
>epoch=22, lrate=0.020, error=0.995
>epoch=23, lrate=0.020, error=0.994
>epoch=24, lrate=0.020, error=0.994
>epoch=25, lrate=0.020, error=0.994
>epoch=26, lrate=0.020, error=0.993
>epoch=27, lrate=0.020, error=0.993
>epoch=28, lrate=0.020, error=0.993
>epoch=29, lrate=0.020, error=0.993
>epoch=30, lrate=0.020, error=0.992
>epoch=31, lrate=0.020, error=0.992
>epoch=32, lrate=0.020, error=0.992
>epoch=33, lrate=0.020, error=0.991
>epoch=34, lrate=0.020, error=0.991
>epoch=35, lrate=0.020, error=0.991
>epoch=36, lrate=0.020, error=0.990
```

```
>epoch=37, lrate=0.020, error=0.990
>epoch=38, lrate=0.020, error=0.990
>epoch=39, lrate=0.020, error=0.990
>epoch=40, lrate=0.020, error=0.989
>epoch=41, lrate=0.020, error=0.989
>epoch=42, lrate=0.020, error=0.989
>epoch=43, lrate=0.020, error=0.988
>epoch=44, lrate=0.020, error=0.988
>epoch=45, lrate=0.020, error=0.988
>epoch=46, lrate=0.020, error=0.987
>epoch=47, lrate=0.020, error=0.987
>epoch=48, lrate=0.020, error=0.987
>epoch=49, lrate=0.020, error=0.987
```

```
final_loss = losses
# for data in losses:
#   if type(data) == float:
#     final_loss.append(data)

plt.plot(final_loss)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
print('Finished Training')
```




```
Finished Training
```

1. Experiment with different number of a) hidden layers b) training epochs and report the ablation study.

a) hidden layers - 2

The hidden layer1 size - 800

The hidden layer2 size - 300

As the network layers increases, the accuracy improves. The network is able to train better as the network is deeper.

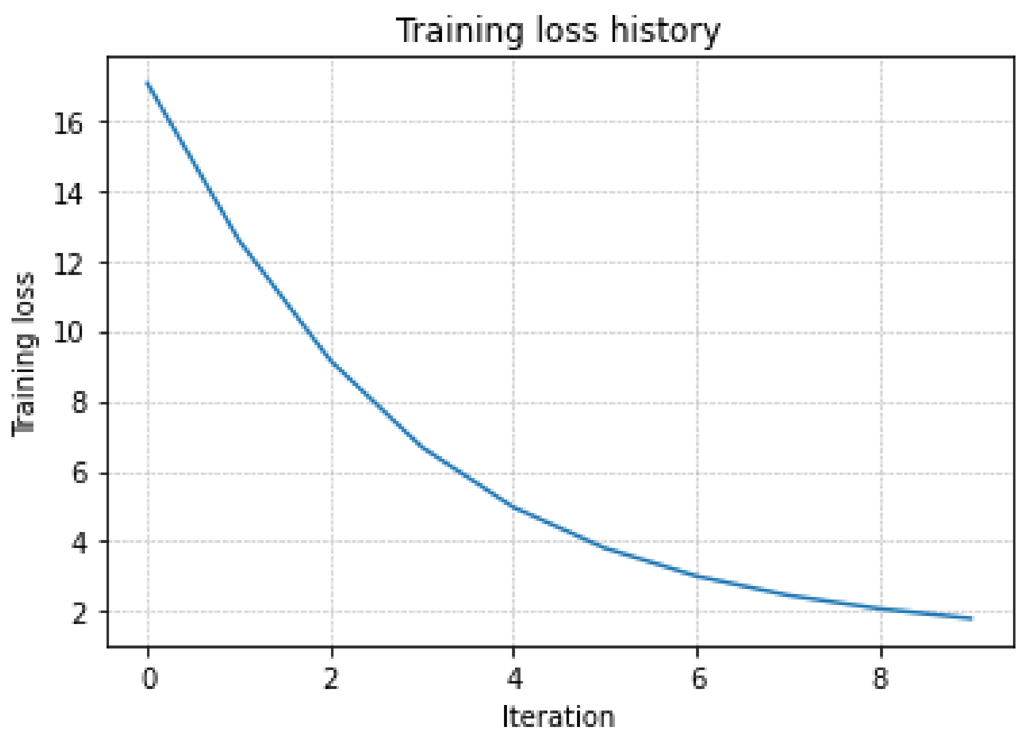
b) 1) training epochs - 10 2) training epochs - 50

When the training epochs are 10 the network loss is not reduced by a considerable amount, which results in poor accuracy. On the contrary, when the training epochs are 50, the network accuracy improves as the network loss is able to better converge to the minimum.

```
def initialize_network_2(n_inputs, n_hidden1, n_hidden2, n_outputs):
     network = list()

     #W1 = np.random.rand(n_inputs, n_hidden)
     W1 = np.random.normal(0, 0.01, (n_inputs, n_hidden1))
     b1 = np.zeros((1, n_hidden1))
     hidden_layer1 = [W1, b1]
     ## Write your code. Initialize hidden layer here.
     network.append(hidden_layer1)

     # W2 = np.random.rand(n_hidden, n_outputs)
     W2 = np.random.normal(0, 0.01, (n_hidden1, n_hidden2))
     b2 = np.zeros((1, n_hidden2))
     hidden_layer2 = [W2, b2]
     ## Write your code. Initialize output_layer layer here.
     network.append(hidden_layer2)

     W3 = np.random.normal(0, 0.01, (n_hidden2, n_outputs))
     b3 = np.zeros((1, n_outputs))
     output_layer = [W3, b3]
     ## Write your code. Initialize output_layer layer here.
     network.append(output_layer)

     return network

network = initialize_network_2(n_inputs, 800, 300, n_outputs)
losses = train(network, train_x, train_y, l_rate=0.02, n_epoch=10,
n_outputs=n_outputs)

final_loss = losses
# for data in losses:
#   if type(data) == float:
#     final_loss.append(data)
```

```
plt.plot(final_loss)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
print('Finished Training')
```

```
>epoch=0, lrate=0.020, error=17.042
>epoch=1, lrate=0.020, error=12.573
>epoch=2, lrate=0.020, error=9.147
>epoch=3, lrate=0.020, error=6.681
>epoch=4, lrate=0.020, error=4.972
>epoch=5, lrate=0.020, error=3.807
>epoch=6, lrate=0.020, error=3.012
>epoch=7, lrate=0.020, error=2.464
>epoch=8, lrate=0.020, error=2.081
>epoch=9, lrate=0.020, error=1.807
```




```
Finished Training
```

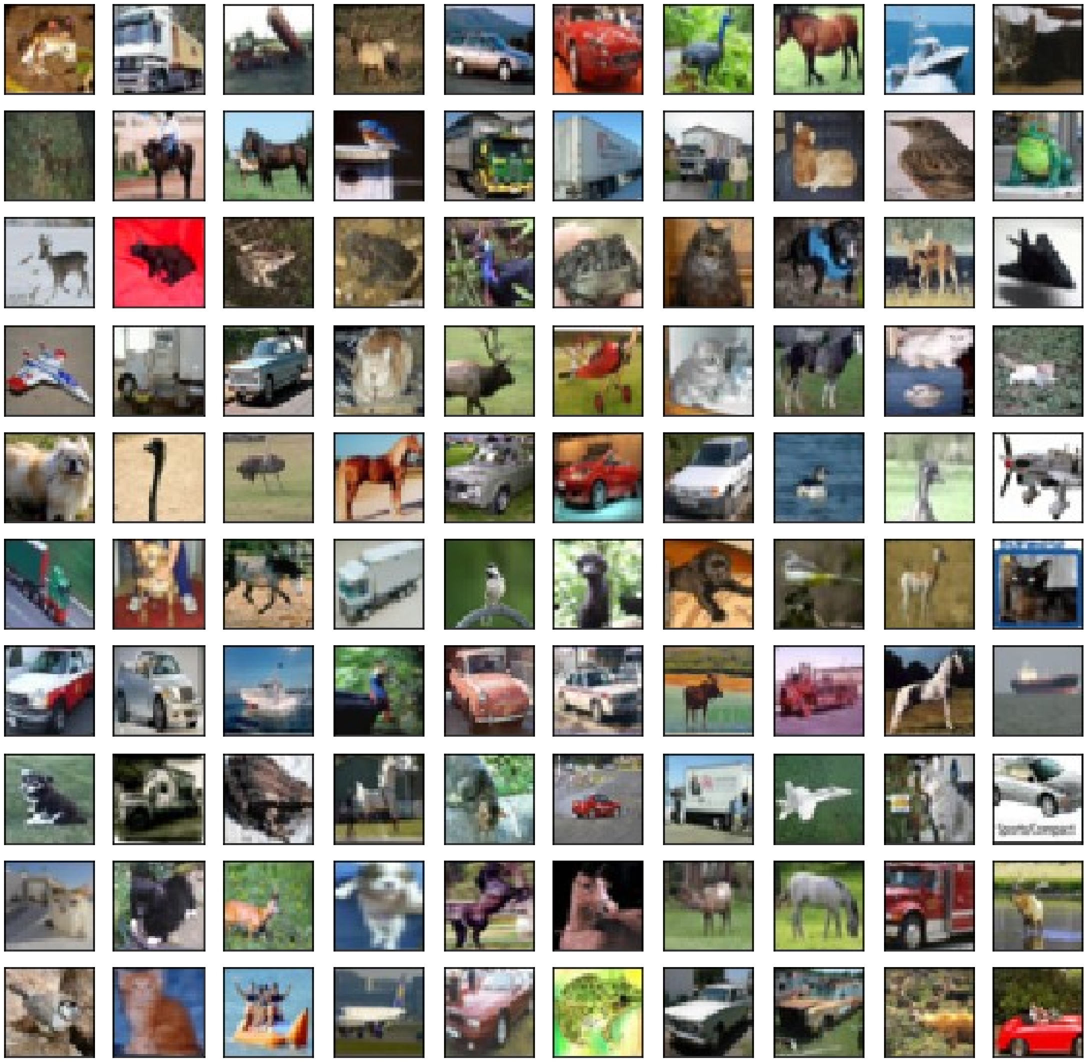
```
def predict(network, test_data, test_labels):

  print(test_data.shape)
  outputs, _ = forward_propagation(network, test_data)
  outputs = (outputs == outputs.max(axis=0,
keepdims=True)).astype(int)
  count = 0
```

```
  add = 1
  for i in range(outputs.shape[0]):
    y_label_correct = np.argmax(test_labels[i])
    y_label_op = np.argmax(outputs[i])

    if y_label_correct == y_label_op:
      count += add

  # correct = (((test_labels - outputs).sum(axis=0))==0).sum()
  test_acc = count / (test_labels.shape[0])

  return test_acc

train_accuracy = predict(network, train_images, train_labels)
test_accuracy = predict(network, test_images, test_labels)

print('Test Accuracy:', (train_accuracy*100))
print('Test Accuracy:', (test_accuracy*100))
```

```
(60000, 784)
(10000, 784)
Test Accuracy: 9.875
Test Accuracy: 9.8
```

Part 2: Training an Image Classifier

##Overview CIFAR10 dataset will be used to train an image classifier.



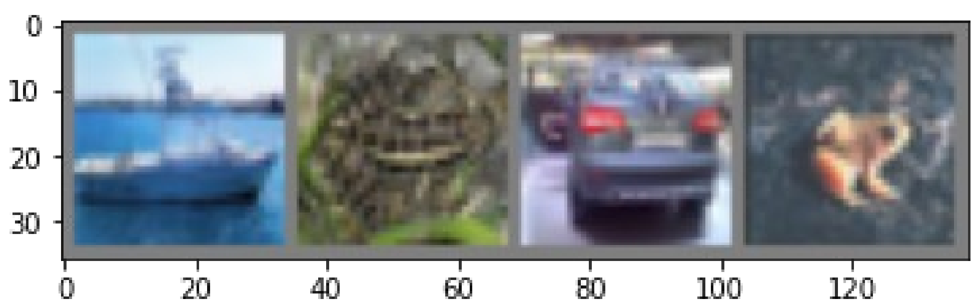
##Data Using `torchvision`, it's extremely easy to load `CIFAR10`.

```
## The output of torchvision datasets are PILImage images of range [0,
1]. We transform them to Tensors of normalized range [-1, 1].
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset,
batch_size=batch_size,
                                          shuffle=True, num_workers=2)
```

```
testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset,
batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

print(len(trainset))
```

```
Files already downloaded and verified
Files already downloaded and verified
50000
```

```
## Let us show some of the training images, for fun.

# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```




```
ship  frog  car   frog
```

##Code (20 pts)

###Define a Convolutional Neural Network (10 pt)

Create a neural network that take 3-channel images. It should go as Conv2d --> ReLU --> MaxPool2d --> Conv2d --> ReLU --> MaxPool2d --> Flatten --> Linear --> ReLU --> Linear --> ReLU --> Linear

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        ## TODO: Add layers to your neural net.
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        ## TODO: run forward pass as mentioned above.
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

###Define a Loss function and optimizer (5 pt)

Let's use a Classification Cross-Entropy loss and SGD with momentum. (Feel free to experiment with other loss functions and optimizers to observe differences)

```
criterion = nn.CrossEntropyLoss() ## TODO: Add loss function
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9) ##
TODO: Add optimizer
```

###Train the network (5 pts)

This is when things start to get interesting. We simply have to loop over our data iterator, and feed the inputs to the network and optimize.

```
epochs = 5 ## TODO: define number of epochs to train
losses = []
for epoch in range(epochs):  # loop over the dataset multiple times

    loss = 0.0
    running_loss = 0.0
    count = 0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
```

```
        inputs, labels = data

        # TODO: add line to zero the parameter gradients below
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        loss += loss.item()
        count = i
        if i % 2000 == 1999:    # print every 2000 mini-batches
            losses.append(running_loss / 2000)
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss /
2000:.3f}')
            running_loss = 0.0

    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    # again no gradients needed
    count_class = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = net(images)
            _, predictions = torch.max(outputs, 1)
            # collect the correct predictions for each class
            for label, prediction in zip(labels, predictions):
                if label == prediction:
                    count_class += 1
                    correct_pred[classes[label]] += 1
                total_pred[classes[label]] += 1

    print(f'[{epoch + 1}] accuracy: {(count_class /
len(trainset)*100):.3f}')
    losses.append(loss/count)
```

```
[1,  2000] loss: 1.313
[1,  4000] loss: 1.306
[1,  6000] loss: 1.279
[1,  8000] loss: 1.266
[1, 10000] loss: 1.240
[1, 12000] loss: 1.251
[1] accuracy: 11.392
[2,  2000] loss: 1.163
```

```
[2,  4000] loss: 1.167
[2,  6000] loss: 1.163
[2,  8000] loss: 1.172
[2, 10000] loss: 1.138
[2, 12000] loss: 1.161
[2] accuracy: 11.286
[3,  2000] loss: 1.076
[3,  4000] loss: 1.091
[3,  6000] loss: 1.100
[3,  8000] loss: 1.084
[3, 10000] loss: 1.090
[3, 12000] loss: 1.082
[3] accuracy: 11.888
[4,  2000] loss: 1.015
[4,  4000] loss: 1.032
[4,  6000] loss: 1.031
[4,  8000] loss: 1.013
[4, 10000] loss: 1.042
[4, 12000] loss: 1.043
[4] accuracy: 12.224
[5,  2000] loss: 0.944
[5,  4000] loss: 0.977
[5,  6000] loss: 0.979
[5,  8000] loss: 1.007
[5, 10000] loss: 0.979
[5, 12000] loss: 0.995
[5] accuracy: 12.368
```

```
final_loss = []
for data in losses:
  if type(data) == float:
    final_loss.append(data)

plt.plot(final_loss)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
print('Finished Training')

## Let’s quickly save our trained model:

PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```


```
Finished Training
```

###Test the network on the test data We have trained the network over the training dataset. But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

```python
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
```

```python
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: plane is 59.8 %
Accuracy for class: car   is 77.7 %
Accuracy for class: bird  is 46.2 %
Accuracy for class: cat   is 35.7 %
Accuracy for class: deer  is 53.9 %
Accuracy for class: dog   is 52.5 %
Accuracy for class: frog  is 82.2 %
Accuracy for class: horse is 69.0 %
Accuracy for class: ship  is 73.7 %
Accuracy for class: truck is 67.7 %
```

Write-up (5 pt)

(1 pt) Show plot for loss over epochs.

```python
final_loss = []
for data in losses:
  if type(data) == float:
    final_loss.append(data)

plt.plot(final_loss)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
print('Finished Training')
```

```python
## Let’s quickly save our trained model:
```


```
Finished Training
```

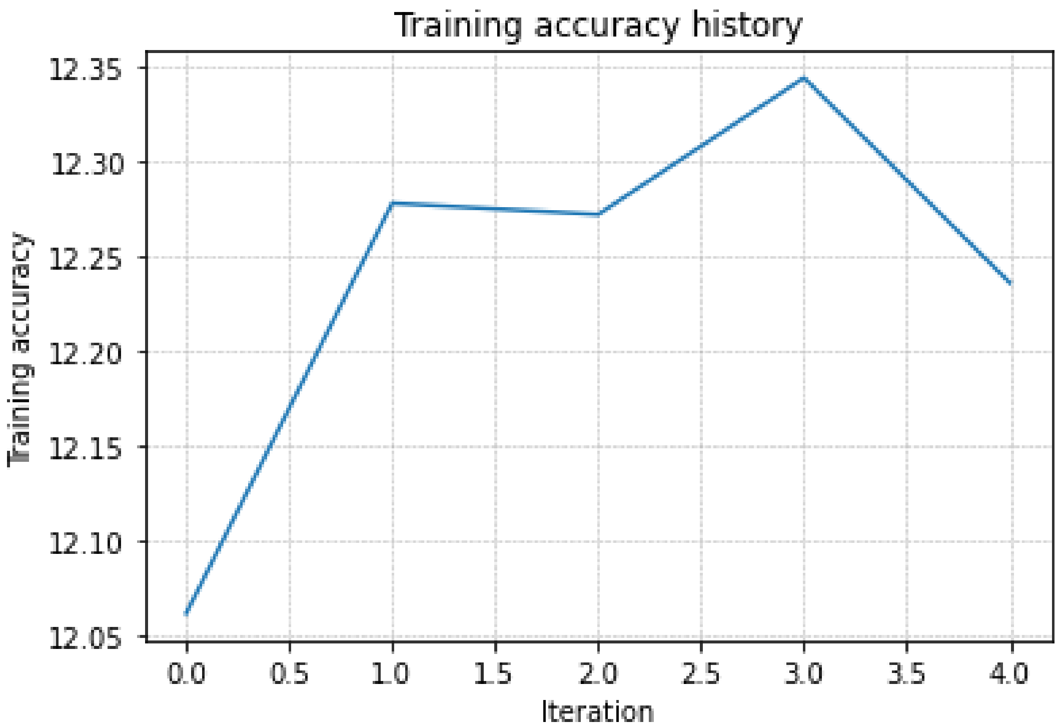
(1 pt) Show plot for accuracy over epochs.

```
epochs = 5 ## TODO: define number of epochs to train
losses = []
accuracies = []
for epoch in range(epochs):  # loop over the dataset multiple times

    loss = 0.0
    running_loss = 0.0
    count = 0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # TODO: add line to zero the parameter gradients below
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
```

```
        loss += loss.item()
        count = i
        if i % 2000 == 1999:    # print every 2000 mini-batches
            losses.append(running_loss / 2000)
            # print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss /
2000:.3f}')
            # running_loss = 0.0

    correct_pred = {classname: 0 for classname in classes}
    total_pred = {classname: 0 for classname in classes}

    # again no gradients needed
    count_class = 0
    with torch.no_grad():
        for data in testloader:
            images, labels = data
            outputs = net(images)
            _, predictions = torch.max(outputs, 1)
            # collect the correct predictions for each class
            for label, prediction in zip(labels, predictions):
                if label == prediction:
                    count_class += 1
                    correct_pred[classes[label]] += 1
                total_pred[classes[label]] += 1

    print(f'epoch: {epoch + 1} accuracy: {(count_class /
len(trainset))*100:.3f}')
    accuracies.append((count_class / len(trainset))*100)
    losses.append(loss/count)
```

```
epoch: 1 accuracy: 12.062
epoch: 2 accuracy: 12.278
epoch: 3 accuracy: 12.272
epoch: 4 accuracy: 12.344
epoch: 5 accuracy: 12.236
```

```
final_acc = []
for data in accuracies:
  if type(data) == float:
    final_acc.append(data)

plt.plot(final_acc)
plt.title("Training accuracy history")
plt.xlabel("Iteration")
plt.ylabel("Training accuracy")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
print('Finished Training')
```




```
Finished Training
```

(3 pt) Show confusion matrix on test data.

```
from sklearn.metrics import confusion_matrix

# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

pred_list = []
label_list = []

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1

            pred_list.append(classes[prediction])
            label_list.append(classes[label])
```

```
conf_mat = confusion_matrix(label_list, pred_list)

print()
print('Confusion Matrix on test data: ')
print()
print(conf_mat)
```

```
Confusion Matrix on test data:

[[533   6 110  77  58  49  44  91  24   8]
 [ 10 803  12   4  12  15   8  26  52  58]
 [ 70  21 498  47 208  58  36  35  15  12]
 [154   6  82 496  53  52  97  38  17   5]
 [111   5 217  28 501  24  77  24   6   7]
 [ 76   8 119  65  46 630  17  12  18   9]
 [ 47  12  66  68  81  12 664  28   4  18]
 [ 47  42  29  27  13  10  16 672 119  25]
 [ 20  60  31  10  14   8   4  98 738  17]
 [ 20 187  42   9  15   8  24  28  84 583]]
```

Extra Credits (5 pt)

Run VGG with pre-trained weights in this colab. Test any two images of your choice to your model and to VGG model and show accuracy (images must include objects from CIFAR10 classes). Discuss which model performs better and why.

Part 3: Semantic Segmentation

Overview

Semantic Segmentation is an image analysis task in which we classify each pixel in the image into a class. So, let's say we have the following image.

And then given the above image its semantically segmentated image would be the following

As you can see, that each pixel in the image is classified to its respective class.

Data

WARNING: Colab deletes all files everytime runtime is disconnected. Make sure to re-download the inputs when it happens.

```python
import os
import tarfile
import shutil
import urllib.request

url='http://host.robots.ox.ac.uk/pascal/VOC/voc2007/VOCtrainval_06-
Nov-2007.tar'
path='VOC'
def get_archive(path,url):
  try:
    os.mkdir(path)
  except:
    path=path

  filename='devkit'
  urllib.request.urlretrieve(url,f"{path}/{filename}.tar")

get_archive(path,url)
def extract(path):
  tar_file=tarfile.open(f"{path}/devkit.tar")
  tar_file.extractall('./')
  tar_file.close()
  shutil.rmtree(path)

extract(path)
```

Helper Functions

```python
from PIL import Image
import matplotlib.pyplot as plt
import torch
from torchvision import models
import torchvision.transforms as T
import numpy as np

"""Various RGB palettes for coloring segmentation labels."""
VOC_CLASSES = [
    "background",
    "aeroplane",
    "bicycle",
    "bird",
    "boat",
    "bottle",
    "bus",
    "car",
    "cat",
    "chair",
    "cow",
    "diningtable",
    "dog",
    "horse",
```

```
    "motorbike",
    "person",
    "potted plant",
    "sheep",
    "sofa",
    "train",
    "tv/monitor",
]


VOC_COLORMAP = [
    [0, 0, 0],
    [128, 0, 0],
    [0, 128, 0],
    [128, 128, 0],
    [0, 0, 128],
    [128, 0, 128],
    [0, 128, 128],
    [128, 128, 128],
    [64, 0, 0],
    [192, 0, 0],
    [64, 128, 0],
    [192, 128, 0],
    [64, 0, 128],
    [192, 0, 128],
    [64, 128, 128],
    [192, 128, 128],
    [0, 64, 0],
    [128, 64, 0],
    [0, 192, 0],
    [128, 192, 0],
    [0, 64, 128],
]
```

Code (25 pt)

1. Implement Data Loader for training and validation (5 pt)

```
import os
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import cv2

# You can modify this class
class VocDataset(Dataset):
    def __init__(self, dir, color_map):

        self.root=os.path.join(dir,'VOCdevkit/VOC2007')
        self.target_dir=os.path.join(self.root,'SegmentationClass')
        self.images_dir=os.path.join(self.root,'JPEGImages')
```

```
file_list=os.path.join(self.root,'ImageSets/Segmentation/trainval.txt'
)
        self.files = [line.rstrip() for line in tuple(open(file_list,
"r"))]
        self.color_map=color_map

    def convert_to_segmentation_mask(self, mask):

        height, width = mask.shape[:2]

        segmentation_mask = np.zeros((height, width,
len(self.color_map)), dtype=np.float32)

        for label_index, label in enumerate(self.color_map):
            segmentation_mask[:, :, label_index] = np.all(mask ==
label, axis=-1).astype(float)

        return segmentation_mask


    def __getitem__(self, index):

        image_id = self.files[index]
        image_path = os.path.join(self.images_dir,f"{image_id}.jpg")
        label_path = os.path.join(self.target_dir,f"{image_id}.png")
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        image = cv2.resize(image,(256, 256))
        image = image.reshape(3, 256, 256)
        image = torch.tensor(image).float()
        label = cv2.imread(label_path)
        label = cv2.cvtColor(label, cv2.COLOR_BGR2RGB)
        label = cv2.resize(label, (256,256))
        label = self.convert_to_segmentation_mask(label)
        label = label.reshape(label.shape[-1], 256, 256)
        label = torch.tensor(label).float()

        return image, label

    def __len__(self):
        return len(self.files)

from google.colab.patches import cv2_imshow
import numpy as np
import torch.optim as optim
from torchvision.models import vgg16

data=VocDataset('/content', VOC_COLORMAP)
```

```
train_set,val_set=torch.utils.data.random_split(data,
[int(len(data)*0.9), round(len(data)*0.1)+1])

train_loader=DataLoader(train_set,batch_size=10,shuffle = True)

val_loader=DataLoader(val_set,batch_size=10,shuffle = False)
# cv2_imshow(np.array(train_dataset[0][0]))
# model = FCN32(21, pretrained_model)
```

###2. Define model and training code (15 pt) Implement FCN-32 model. You can use encoder as pretrained model provided by torchvision.

```
import torch
from PIL import Image

class FCN32(torch.nn.Module):
    def __init__(self, n_classes, pretrained_model):
        # YOUR CODE HERE:
        super().__init__()

        self.n_classes = n_classes
        self.pretrained_model = pretrained_model

features,classifiers=list(self.pretrained_model.features.children()),l
ist(self.pretrained_model.classifier.children())

        features[0].padding=(100,100)
        self.features_map=nn.Sequential(*features)
        self.conv=nn.Sequential(nn.Conv2d(512,4096,7),
                                nn.ReLU(inplace=True),
                                nn.Dropout(),
                                nn.Conv2d(4096,4096,1),
                                nn.ReLU(inplace=True),
                                nn.Dropout()
                                )
        self.score_fr=nn.Conv2d(4096,self.n_classes,1)
        self.upscore=nn.ConvTranspose2d(self.n_classes,
self.n_classes, kernel_size=64, stride=32, bias=False)
        self.upscale
=nn.ConvTranspose2d(self.n_classes,self.n_classes, kernel_size=3,
stride=2, padding =2)

    def forward(self,x):
        # print(x.size)
        # x = Image.fromarray(x)
        x_size=x.size()
        pool=self.conv(self.features_map(x))
        score_fr=self.score_fr(pool)
        upscore=self.upscore(score_fr)
        return upscore[:, :, 16:-16, 16:-16]
```

```
pretrained_model=vgg16(pretrained=True)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = FCN32(21, pretrained_model).to(device)
# model = torchvision.models.vgg16_bn(pretrained=True)
# model = FCN32(model, 1)
# model.to(device)
```

```
/usr/local/lib/python3.8/dist-packages/torchvision/models/
_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated
since 0.13 and may be removed in the future, please use 'weights'
instead.
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:22
3: UserWarning: Arguments other than a weight enum or `None` for
'weights' are deprecated since 0.13 and may be removed in the future.
The current behavior is equivalent to passing
`weights=VGG16_Weights.IMAGENET1K_V1`. You can also use
`weights=VGG16_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```

Training code for the semantic segmentation model. Implment both training and validation parts.

```
import torch.optim as optim
from torch.autograd import Variable
from tqdm import tqdm

criterion = nn.CrossEntropyLoss() ## TODO: Add loss function
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9) ##
TODO: Add optimizer

epochs = 5

train_loss_list = []
train_accuracy_list = []
val_loss_list = []
val_accuracy_list = []

for epoch in tqdm(range(epochs)):  # loop over the dataset multiple
times

    train_loss = 0.0
    train_accuracy = 0.0
    correct = 0
    total = 0

    for i, data in enumerate(train_loader):

        inputs, labels = data
```

```python
        optimizer.zero_grad()

        # forward + backward + optimize
        inputs = Variable(torch.from_numpy(np.array(inputs)))
        labels = Variable(torch.from_numpy(np.array(labels)))

        inputs = inputs.to(device)
        labels = labels.to(device)

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        train_loss += loss.item()

        for j in range(inputs.size(0)):

            output = outputs[j]
            label = labels[j]

            pred_class = torch.argmax(output, dim=0)
            act_class = torch.argmax(label, dim=0)

            correct += torch.sum(pred_class == act_class)
            total += float(act_class.numel())

    train_loss = train_loss / float(len(train_loader))
    train_accuracy = float(correct) / float(total)
    train_loss_list.append(train_loss)
    train_accuracy_list.append(train_accuracy)


    val_loss = 0.0
    val_accuracy = 0.0
    correct = 0

    for i, data in enumerate(val_loader):

        with torch.no_grad():

            inputs, labels = data
            optimizer.zero_grad()

            # forward + backward + optimize
            inputs = Variable(torch.from_numpy(np.array(inputs)))
            labels = Variable(torch.from_numpy(np.array(labels)))
```

```
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, labels)

            # print statistics
            val_loss += loss.item()

            for j in range(inputs.size(0)):

                output = outputs[j]
                label = labels[j]

                pred_class = torch.argmax(output, dim=0)
                act_class = torch.argmax(label, dim=0)

                correct += torch.sum(pred_class == act_class)
                total += float(act_class.numel())


    val_loss = val_loss / float(len(val_loader))
    val_accuracy = float(correct) / float(total)
    val_loss_list.append(val_loss)
    val_accuracy_list.append(val_accuracy)



    print()
    print("Epoch: " + str(epoch))
    print("Training Loss: " + str(train_loss) + "    Validation Loss:
" + str(val_loss))
    print("Training Accuracy: " + str(train_accuracy*100) + "
Validation Accuracy: " + str(val_accuracy*100))
    print()

 20%|██        | 1/5 [01:01<04:06, 61.59s/it]


Epoch: 0
Training Loss: 2.8543059825897217    Validation Loss:
2.889460802078247
Training Accuracy: 4.79059546478191    Validation Accuracy:
0.49051944678428616


  40%|████      | 2/5 [02:01<03:02, 60.71s/it]


Epoch: 1
Training Loss: 2.8544179012900903    Validation Loss:
```

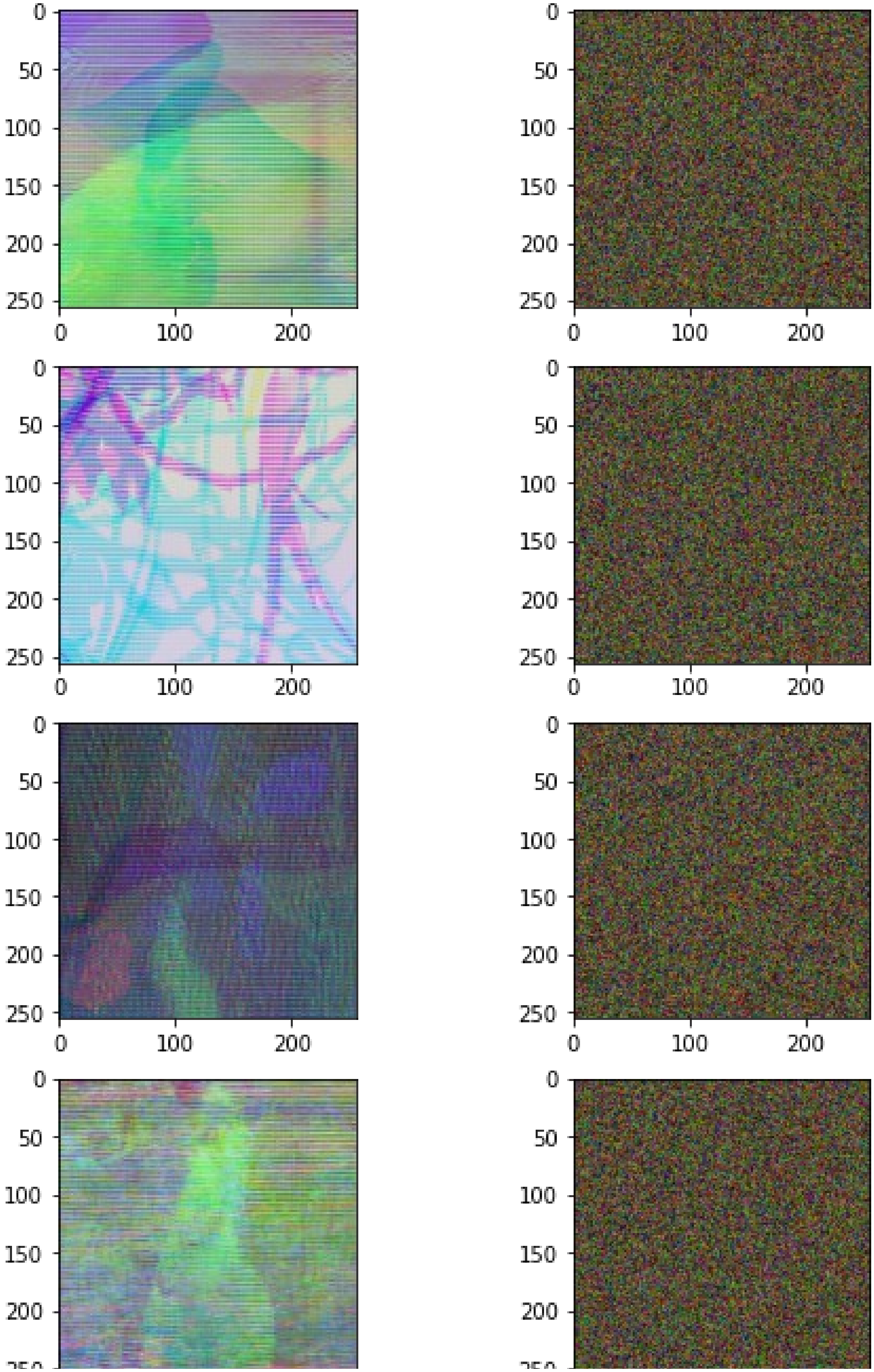
```
2.889178657531738
Training Accuracy: 4.8160522783022754    Validation Accuracy:
0.49197300915469494


  60%|██████    | 3/5 [03:04<02:02, 61.47s/it]


Epoch: 2
Training Loss: 2.8541707616103325    Validation Loss:
2.889130783081055
Training Accuracy: 4.8256101583113455    Validation Accuracy:
0.4888778614206901


  80%|████████  | 4/5 [04:07<01:02, 62.17s/it]


Epoch: 3
Training Loss: 2.8540554234856055    Validation Loss:
2.888978385925293
Training Accuracy: 4.845096316375329    Validation Accuracy:
0.4944136922393365


100%|██████████| 5/5 [05:10<00:00, 62.11s/it]


Epoch: 4
Training Loss: 2.8540659578222978    Validation Loss:
2.888877773284912
Training Accuracy: 4.868113334071982    Validation Accuracy:
0.4971147148530065
```

3. Inference for semantic segmentation (5 pt)

Implement the inference code for semantic segmentation. Display the visualization results of the model. Plot the image and colorized image (similar to the results in overview).

```
# YOUR CODE HERE:

colmap = np.array(VOC_COLORMAP)
def prediction(val_loader, model):
  count = 0
  preds = []
  fig3, ax3 = plt.subplots(5, 2, figsize=(8, 15))
  for k, data in enumerate(val_loader):
        with torch.no_grad():
```

```
            # print(f'image no: {k}')
            # print()
            inputs, labels = data
            # input_img = inputs
            optimizer.zero_grad()

            # forward + backward + optimize
            inputs = Variable(torch.from_numpy(np.array(inputs)))
            labels = Variable(torch.from_numpy(np.array(labels)))

            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)

            outputs_ = outputs[0].permute(1, 2, 0)
            inputs = (inputs[0].permute(1, 2, 0).to('cpu')).numpy()
            outputs_ = (outputs_.to('cpu')).numpy()
            op_img = np.zeros(inputs.shape)
            for i in range(inputs.shape[0]):
              for j in range(inputs.shape[1]):
                idx = np.argmax(outputs_[i, j])
                op_img[i, j, :] = colmap[idx]

            ax3[k, 0].imshow(inputs.astype(np.uint8))
            ax3[k, 1].imshow(op_img.astype(np.uint8))

            if count == 5:
              plt.show()
              return preds


  return preds

preds = prediction(val_loader, model)
```


0
50
100
150
200
250
0
100
200

Write-up (5 pt)

- Describe the properties of segmentation model

One of the two following fundamental features of intensity values is frequently the foundation of segmentation algorithms:

1. Similarity partitioning an image into areas based on a set of predefined criteria that are comparable.
2. Discontinuity detection regional borders based on local intensity discontinuities.

Using the information from the edges, it assists in locating characteristics of related items in the picture. Edge detection helps reduce the size of photos and makes analysis easier by removing extraneous data. Based on differences in contrast, texture, color, and saturation, edge-based segmentation algorithms locate edges.

- Describe the evaluation metric (IoU) for segmentation model

The Intersection-Over-Union (IoU), also known as the Jaccard Index, is one of the most commonly used metrics in semantic segmentation

IoU, as indicated on the left image, is the area of union between the predicted segmentation and the ground truth divided by the area of overlap between the predicted segmentation and the ground truth. This statistic has a range of 0 to 1 (0 to 100%), with 0 denoting complete overlap and 1 denoting no overlap at all.

Intersection-Over-Union is a common evaluation metric for semantic image segmentation.

For an individual class, the IoU metric is defined as follows:

iou = true_positives / (true_positives + false_positives + false_negatives)

Hint

- Refer to original paper FCNet : https://arxiv.org/abs/1411.4038
- Figures for FCNet Structure: https://towardsdatascience.com/review-fcn-semantic-segmentation-eb8c9b50d2d1
- PyTorch Tutorial for Image semgnetation: https://towardsdatascience.com/train-neural-net-for-semantic-segmentation-with-pytorch-in-50-lines-of-code-830c71a6544f

Part 4: Text2Img Generation (10 Points)

We have provided link to 'DALL.E' mini model to generate images from a text prompt in an interactive way.

https://colab.research.google.com/github/borisdayma/dalle-mini/blob/main/tools/inference/inference_pipeline.ipynb#scrollTo=118UKH5bWCGa

Write-up (10 pts)

1. Try different prompts (as per your understanding) to reveal biases encoded by model (for example, birds always exist in the similar surroundings like trees).

   Bias towards old age

   Image for the input “A renaissance-style painting of a modern supermarket aisle. In the aisle is a crowd of shoppers with shopping trolleys trying to get reduced items”. Although there are customers and shopping carts in the picture, the store does not appear to be new.

1. By inputting creative text prompts, you should report the failure cases in your writeup i.e. when model doesn't quite understand the semantics of text prompt (for example, in case of long and complex sentences).

   DALL-E has problems with faces, coherent plans like a site plan or a maze, and with text. The system could not handle negations at all: An input like “A spaceship without an apple” results in a spaceship with an apple.