In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

a) load/merge data and visualize logerror

In [2]:
```
# Load data into DataFrames
df_train = pd.read_csv('train.csv')
df_prop = pd.read_csv('properties.csv')
df_merge = pd.merge(df_train, df_prop,on ='id')
```

In [3]:
```
# eliminate outliers
percent_range = np.percentile(df_merge.logerror, [1, 99]);
print(df_merge)
df_merge[(df_merge.logerror < percent_range[0])] = df_merge[(df_merge.logerror
df_merge[(df_merge.logerror > percent_range[1])] = df_merge[(df_merge.logerror
```

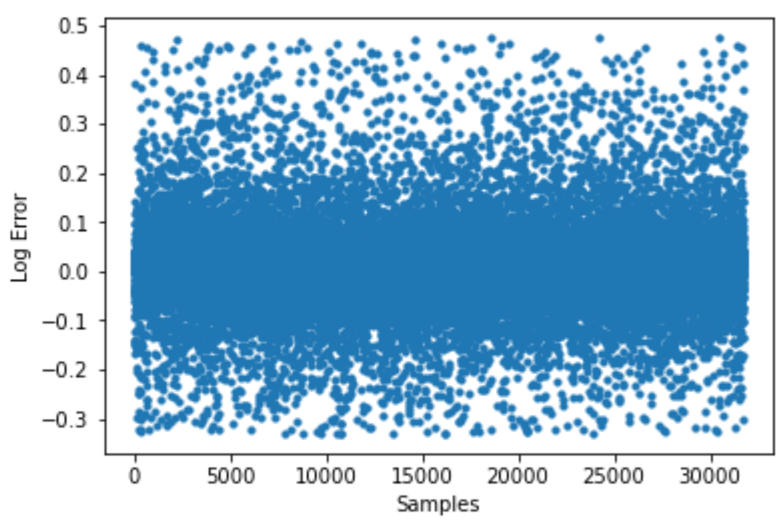
```
             id  logerror transactiondate  airconditioningtypeid  \
0      14366692   -0.1684          1/1/16                    NaN   
1      14739064   -0.0030          1/2/16                    NaN   
2      10854446    0.3825          1/3/16                    NaN   
3      11672170   -0.0161          1/3/16                    1.0   
4      12524288   -0.0419          1/3/16                    NaN   
...         ...       ...             ...                    ...   
31720  12756771    0.0658        12/30/16                    NaN   
31721  11295458   -0.0294        12/30/16                    1.0   
31722  11308315    0.0070        12/30/16                    1.0   
31723  11703478    0.0431        12/30/16                    NaN   
31724  12566293    0.4207        12/30/16                    NaN   

       architecturalstyletypeid  basementsqft  bathroomcnt  bedroomcnt  \
0                           NaN           NaN          3.5         4.0   
1                           NaN           NaN          1.0         2.0   
2                           NaN           NaN          2.0         2.0   
3                           NaN           NaN          4.0         5.0   
4                           NaN           NaN          1.0         1.0   
...                         ...           ...          ...         ...   
31720                       NaN           NaN          1.0         3.0   
31721                       NaN           NaN          2.0         2.0   
31722                       NaN           NaN          3.0         5.0   
31723                       NaN           NaN          1.0         3.0   
31724                       NaN           NaN          1.0         3.0   

       buildingclasstypeid  buildingqualitytypeid  ...  numberofstories  \
0                      NaN                    NaN  ...              NaN   
1                      NaN                    NaN  ...              NaN   
2                      NaN                    7.0  ...              NaN   
3                      NaN                    1.0  ...              NaN   
4                      NaN                    7.0  ...              NaN   
...                    ...                    ...  ...              ...   
31720                  NaN                    7.0  ...              NaN   
31721                  NaN                    7.0  ...              NaN   
31722                  NaN                    4.0  ...              NaN   
31723                  NaN                    7.0  ...              NaN   
31724                  NaN                    7.0  ...              NaN   

       fireplaceflag  structuretaxvaluedollarcnt  taxvaluedollarcnt  \
0                NaN                    346458.0           585529.0   
1                NaN                     66834.0           210064.0   
2                NaN                     55396.0           105954.0   
3                NaN                    559040.0          1090127.0   
4                NaN                     56233.0            70316.0   
...              ...                         ...                ...   
31720            NaN                     65728.0           307167.0   
31721            NaN                     40163.0            50203.0   
31722            NaN                    248378.0           331525.0   
31723            NaN                     17520.0            39934.0   
```

```
31724           NaN                     66258.0          163037.0

       assessmentyear  landtaxvaluedollarcnt  taxamount  taxdelinquencyflag  \
0                2015               239071.0   10153.02                 NaN
1                2015               143230.0    2172.88                 NaN
2                2015                50558.0    1443.69                 NaN
3                2015               531087.0   13428.94                 NaN
4                2015                14083.0     913.17                 NaN
...               ...                    ...        ...                 ...
31720            2015               241439.0    4038.70                 NaN
31721            2015                10040.0    1263.39                   Y
31722            2015                83147.0    6461.79                 NaN
31723            2015                22414.0     627.91                 NaN
31724            2015                96779.0    2560.96                 NaN

       taxdelinquencyyear  censustractandblock
0                     NaN                  NaN
1                     NaN         6.059040e+13
2                     NaN         6.037140e+13
3                     NaN         6.037260e+13
4                     NaN         6.037570e+13
...                   ...                  ...
31720                 NaN         6.037550e+13
31721                15.0         6.037900e+13
31722                 NaN         6.037900e+13
31723                 NaN         6.037230e+13
31724                 NaN         6.037540e+13

[31725 rows x 60 columns]
```

In [4]:
```
# scatter of logerr
print(df_merge.logerror)
x = np.arange(0,31725, 1)
plt.scatter(x, df_merge.logerror,s = 10)
plt.ylabel('Log Error')
plt.xlabel('Samples')
```

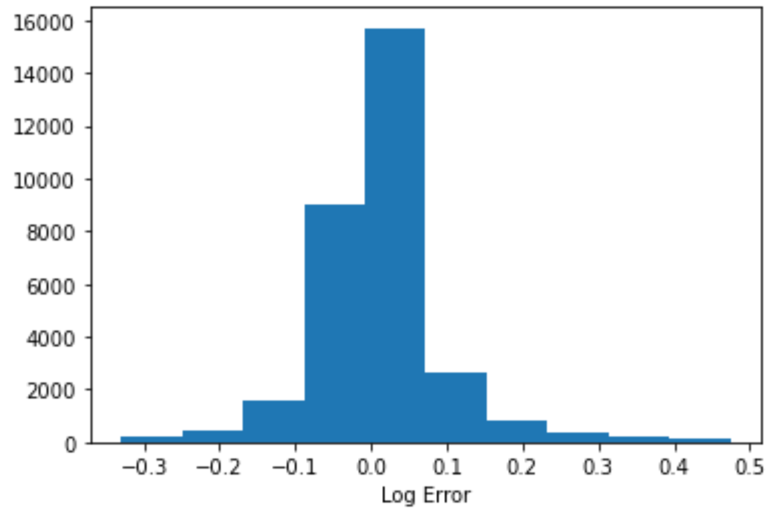
```
0       -0.1684
1       -0.0030
2        0.3825
3       -0.0161
4       -0.0419
          ...
31720    0.0658
31721   -0.0294
31722    0.0070
31723    0.0431
31724    0.4207
Name: logerror, Length: 31725, dtype: float64
```

Out[4]:
```
Text(0.5, 0, 'Samples')
```



In [5]:
```
# histogram of logerr
plt.hist(df_merge.logerror)
plt.xlabel('Log Error')
```

Out[5]: Text(0.5, 0, 'Log Error')



b) data cleaning

In [6]:
```
# build new data frame
missing_vals = df_merge.isna().sum()
col_num = df_merge.columns.transpose()
new_df = pd.DataFrame(list(zip(col_num, missing_vals)), columns = ["column_nam

missing_ratio = df_merge.isna().sum() / len(df_merge)
new_df.insert(2, "missing_ratio", missing_ratio.values)
```

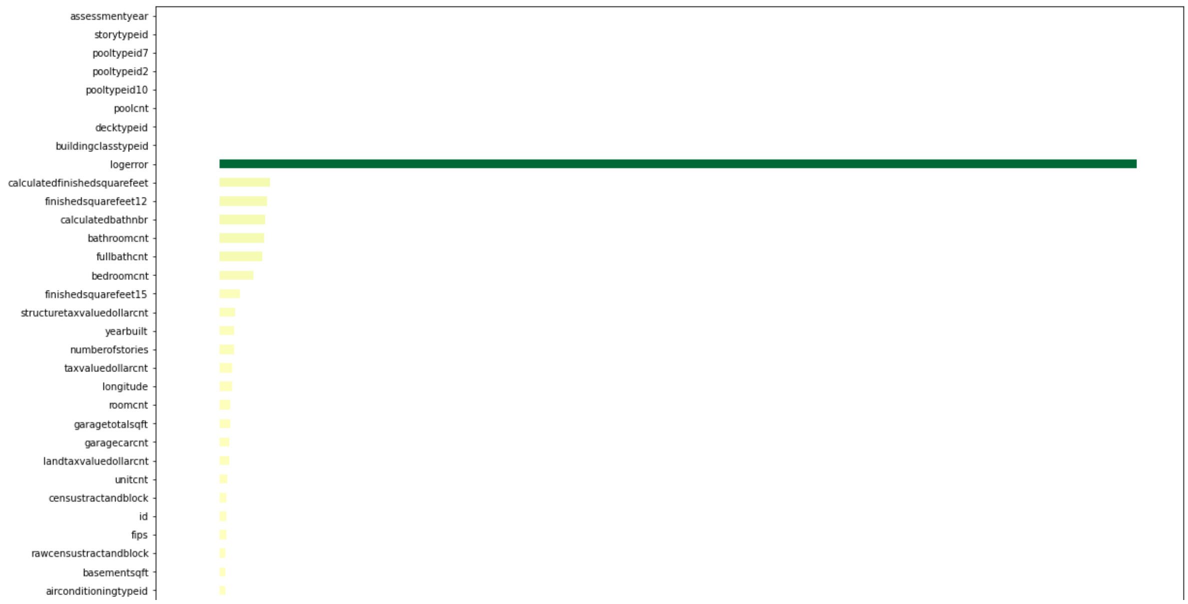
In [7]:
```
# fill missing data
df_merge = df_merge.fillna(df_merge.mean())
```

```
C:\Users\gandi\AppData\Local\Temp\ipykernel_27800\1960727304.py:2: FutureWarn
ing: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only
=None') is deprecated; in a future version this will raise TypeError.  Select
only valid columns before calling the reduction.
  df_merge = df_merge.fillna(df_merge.mean())
```

c) univariate analysis

In [8]:
```
# make bar chart
corr_mat = df_merge.corrwith(df_merge["logerror"])
corr_mat = corr_mat.sort_values()

from matplotlib.colors import TwoSlopeNorm
fig, ax = plt.subplots(figsize =(19, 19))
norm = TwoSlopeNorm(vmin=-1, vcenter = 0, vmax=1)
colors = [plt.cm.RdYlGn(norm(c)) for c in corr_mat.values]
corr_mat.plot.barh(color=colors)
```

Out[8]: <AxesSubplot:>



In [9]:
```
#Explain

#The reason behind why some of the values have no correlation values and becom
#has the standard deviation in the denominator. For these NaN cases, since the
#between the terms is NaN as a result of this division.
```

d) non-linear regression model

In [10]:
```
non_lin_df = df_merge.drop(columns=['id', 'transactiondate',"hashottuborspa",
```

In [11]:
```
# split and train
from sklearn.model_selection import train_test_split
x_train, x_test = train_test_split(non_lin_df, test_size = 0.3, shuffle = True

x_train_stats = []
x_test_stats = []
for i in x_train.columns:
    x_train_stats.append([np.mean(x_train[i]), np.std(x_train[i])])
for i in x_test.columns:
    x_test_stats.append([np.mean(x_test[i]), np.std(x_test[i])])

count = 0;
for i in x_train.columns:
    stats = x_train_stats[count]
    if (stats[1] != 0):
        x_train[i] = (x_train[i] - stats[0]) / stats[1]
    count +=1

count = 0;
for i in x_test.columns:
    stats = x_test_stats[count]
    if (stats[1] != 0):
        x_test[i] = (x_test[i] - stats[0]) / stats[1]
    count +=1
from sklearn.neural_network import MLPRegressor as mlp
from sklearn.datasets import make_regression


x_train_new = x_train.drop(["logerror"], axis = 1)
reg = mlp(random_state = 1, max_iter = 500).fit(x_train_new, x_train.logerror)

x_test_new = x_test.drop(['logerror'], axis = 1)
pred = reg.predict(x_test_new)
```

In [12]:
```
# report importances and mse
from sklearn.metrics import mean_squared_error
x = mean_squared_error(x_test.logerror, pred)
print("Mean Squared Error: ", x)
```

```
Mean Squared Error:  1.1200822431414335
```

Problem 2a.

i. $\frac{\partial L}{\partial b_k}$ $\quad \hat{y}_k = \frac{e^{b_k}}{\sum_{e=1}^{k} e^{b_e}}$ $\quad b_k = \sum_{j=0}^{p} B_{kj} z_j$

$L = \sum_{k=1}^{k} y_k^{(n)} \log(\hat{y} x^{(n)})$

$\frac{\partial L}{\partial b_k} = \sum_{k=1}^{k} y_k \cdot \frac{\partial \log \hat{y}_k}{\partial b_k} = \sum_{k=1}^{k} y_k \frac{1}{\hat{y}_k} \frac{\partial \hat{y}_k}{\partial b_k}$

$\hat{y}_k = \frac{e^{b_k}}{\sum_{\ell=1}^{k} e^{b_\ell}}$ $\quad \frac{\partial \hat{y}_k}{\partial b_k} = \hat{y}_k(1-\hat{y}_k)$

$\frac{\partial \hat{y}_k}{\partial b_\ell} = -\hat{y}_k \hat{y}_\ell$ at $k \neq \ell$

$\frac{\partial L}{\partial b_k} = -y_k^{(n)}(1-\hat{y}_k) - \sum_{\ell \neq k} y_\ell \frac{1}{\hat{y}_\ell}(-\hat{y}_\ell \hat{y}_k)$

$= -y_k^{(n)}(1-\hat{y}_k) + \sum_{k \neq \ell} y_\ell^{(n)} \hat{y}_k$

$\sum_\ell y_\ell = 1$ hot encoded

$$\boxed{\frac{\partial L}{\partial b_k} = \hat{y}_k - y_k}$$

ii. $\frac{\partial L}{\partial \beta_{kj}}$ in terms of $\frac{\partial L}{\partial b_k}$

$\frac{\partial L}{\partial \beta_{kj}} = \frac{\partial L}{\partial b_k} \times \frac{\partial b_k}{\partial \beta_{kj}}$ $\quad \frac{\partial b_k}{\partial \beta_{kj}} = \frac{\partial}{\partial B_{kj}} \sum_{j=0}^{p} B_{kj} z_j = z$

$$\boxed{\frac{\partial L}{\partial B_{kj}} = \frac{\partial L}{\partial b} \cdot z^T}$$

iii. $\frac{\partial L}{\partial z} = \frac{\partial L}{\partial b} + B'$

$= \frac{\partial L}{\partial b} \cdot \frac{\partial b}{\partial z}$

$$\boxed{\frac{\partial L}{\partial z} = \frac{\partial L}{\partial b} \cdot B'^T}$$

iv. $\frac{\partial L}{\partial a_{ji}} \rightarrow \frac{\partial z_i}{\partial a_j} = \frac{e^{-a_j}}{(1+e^{-a})^2} \cdot \sum_{i=0}^{p} x_i^n$

$$\boxed{\frac{\partial L}{\partial a} = \left(\frac{\partial L}{\partial z}\right)^T x^n \left(\frac{e^{-a}}{(1-e^{-a})^2}\right)^T}$$

Problem 2: Implementing a Multi-layer Perceptron

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

In [2]:
```
#!pip install seaborn
# Install seaborn (needed to plot confusion matrix) by uncommenting the above
```

In [16]:
```
def sigmoid_forward(a):
    # calculates the sigmoid activation function
    # a: pre-activation values
    # returns: activated values

    return 1 / (1 + np.exp(-a));

def sigmoid_backward(grad_accum, a):
    # grad_accum: the gradient of the loss function w.r.t to z
    # a: the pre-activation values
    # returns: the gradient of the loss w.r.t to the preactivation values, a

    return (grad_accum * (np.exp(-a) / ((1 + np.exp(-a)) ** 2)))

def linear_forward(x, weight, bias):
    # Computes the forward pass of the linear layer
    # x: input of layer
    # weight, bias: weights and bias of neural network layer
    # returns: output of linear layer

    x = np.column_stack((np.ones(np.shape(x)[0]), x))
    weight = np.column_stack((bias, weight))
    return x @ weight.T

def linear_backward(grad_accum, x, weight, bias):
    #  Derivative of the linear layer w.r.t
    # grad_accum: gradient of loss w.r.t function after linear layer
    # returns dl_dw: gradient of loss w.r.t to weights
    # returns dl_dx: gradient of loss w.r.t to input, x
    # return dl_dw, dl_dx

    x = np.column_stack((np.ones(np.shape(x)[0]), x))
    return grad_accum.T @ x, grad_accum @ weight

def softmax_xeloss_forward(b, labels):
    # Input parameters:
    ## b: pre-activation
    # calculates the softmax of the vector b
    # calculates the cross entropy loss between the softmax of b and the label
    # returns: l

    den = np.sum(np.exp(b), axis = 1)
    pred = np.exp(b) / np.tile(den[:, np.newaxis], (1, 10))
    l = -np.sum(labels * (np.log(pred)), axis = 1)
    return l

def softmax_xeloss_backward(yhat, labels):
    # Input parameters:
    # yhat: predictions of the neural network
    # labels: target of the network
    # returns: dl_db gradient of loss w.r.t to b

    dl_db = (-labels * (1 - yhat))
    return dl_db
```

```python
def data_load():
    # load in the data provided in "data/"
    # Unzip fashion_mnist.zip

    train = np.loadtxt("fashion_mnist/train.csv", delimiter = ",");
    test = np.loadtxt("fashion_mnist/test.csv", delimiter = ",");

    x_train = train[:, :-1]
    y_train = train[:, -1]

    x_test = test[:, :-1]
    y_test = test[:, -1]

    return x_train, y_train, x_test, y_test

def load_params():
    alpha_weights = np.loadtxt('params/alpha1.txt', delimiter=',')
    beta_weights = np.loadtxt('params/alpha2.txt', delimiter=',')
    alpha_bias= np.loadtxt('params/beta1.txt', delimiter=',')
    beta_bias = np.loadtxt('params/beta2.txt', delimiter=',')
    return alpha_weights, beta_weights, alpha_bias, beta_bias

def one_hot_encode(y):
    # convert categorical target features to one hot encoded data

    encode_data = np.zeros((np.shape(y)[0], 10))
    y = np.array(y, dtype = "int")
    for column in range(np.shape(y)[0]):
        encode_data[column, y[column]] = 1
    return encode_data

def train(batchsize=1 , eta = 0.01, num_epochs=100, h = 256, init='default'):
    x_train, y_train, x_test, y_test = data_load()

    y_train = one_hot_encode(y_train)
    y_test = one_hot_encode(y_test)

    if init == 'default':
       alpha_weights, beta_weights, alpha_bias, beta_bias = load_params()
    elif init=='zeros':
        # initialize weights and biases to 0
        alpha_weights, beta_weights, alpha_bias, beta_bias = load_params() * 0
    elif init=='ones':
        # initialize weights and biases to 1
        alpha_weights, beta_weights, alpha_bias, beta_bias = load_params() * 0
    elif init=='random':
        # initialize weights and biases to random values between -1 and 1
        pass

    train_loss_list = []
    test_loss_list = []
    acc_list = []

    for epoch in (range(num_epochs)):
        print("Epoch :", epoch)
```

```
        for batch in range(int(len(x_train) / batchsize) + (len(x_train) % bat
            batch_x = x_train[batch * batchsize:(batch + 1) * batchsize, :]
            batch_y = y_train[batch * batchsize:(batch + 1) * batchsize, :]

            ######## FORWARD
            # Linear -> Sigmoid -> Linear -> Softmax
            A = linear_forward(batch_x, alpha_weights, alpha_bias)
            Z = sigmoid_forward(A)
            B = linear_forward(Z, beta_weights, beta_bias)
            L = softmax_xeloss_forward(B, batch_y)
            ######## BACKWARD
            den = np.sum(np.exp(B), axis = 1)
            yhat = np.exp(B) / den
            dl_db = softmax_xeloss_backward(yhat, batch_y)
            dl_dbeta, dl_dz = linear_backward(dl_db, Z, beta_weights, beta_bia
            dl_da = sigmoid_backward(dl_dz, A)
            dl_dalpha, dl_dx = linear_backward(dl_da, batch_x, alpha_weights,
            ######## UPDATE
            alpha_weights = alpha_weights - dl_dalpha[:, 1:] * eta
            beta_weights = beta_weights - dl_dbeta[:, 1:] * eta
            alpha_bias = alpha_bias - dl_dalpha[:, 0] * eta
            beta_bias = beta_bias - dl_dbeta[:, 0] * eta

        # store average training loss for the epoch
        # calculate test predictions and loss
        num_test = np.shape(x_train)[0]
        A = linear_forward(x_train, alpha_weights, alpha_bias)
        Z = sigmoid_forward(A)
        B = linear_forward(Z, beta_weights, beta_bias)
        L = softmax_xeloss_forward(B, y_train)

        train_loss_list.append(np.sum(L)/num_test)
        y_hat_train = np.exp(B)/np.sum(np.exp(B), axis = 0)
        train_pred = np.argmax(y_hat_train, axis = 1)[:]
        vector_corr = train_pred == np.argmax(y_train, axis = 1)[:]

        #Test
        num_test = np.shape(x_test)[0]
        A = linear_forward(x_test, alpha_weights, alpha_bias)
        Z = sigmoid_forward(A)
        B = linear_forward(Z, beta_weights, beta_bias)
        L = softmax_xeloss_forward(B, y_test)

        test_loss_list.append(np.sum(L)/num_test)
        den = np.sum(np.exp(B), axis = 1)
        y_hat_test = np.exp(B)/np.sum(np.exp(B), axis = 0)
        test_pred = np.exp(B)/np.sum(np.exp(B), axis = 0)
        vector_correct = np.argmax(test_pred, axis = 1)[:] == np.argmax(y_test

        # calculate test accuracy
        acc_list.append(np.sum(vector_correct) / num_test)

    # return train_loss_list, test_loss_list, as well as test and train predic
    return train_loss_list, test_loss_list, acc_list, y_hat_train, y_hat_test
```

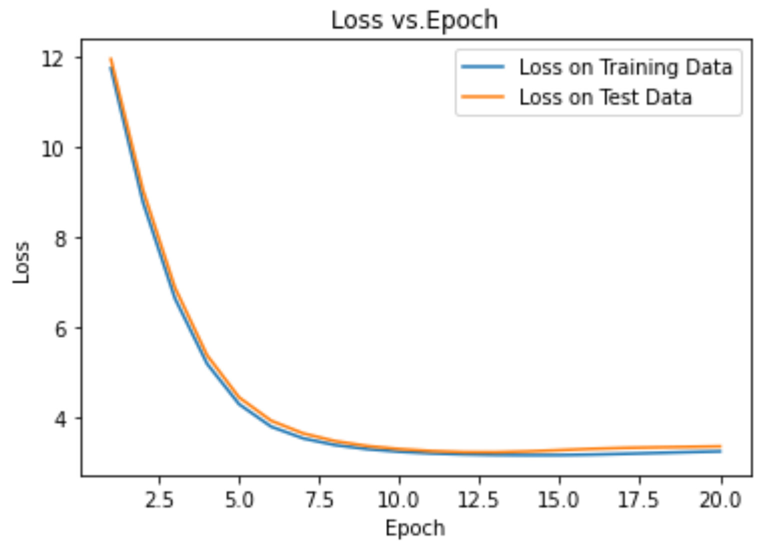
In [17]:
```
train_loss_list, test_loss_list, acc_list, yhatTrain, yhatTest = train(num_epo
```

```
Epoch : 0
Epoch : 1
Epoch : 2
Epoch : 3
Epoch : 4
Epoch : 5
Epoch : 6
Epoch : 7
Epoch : 8
Epoch : 9
Epoch : 10
Epoch : 11
Epoch : 12
Epoch : 13
Epoch : 14
Epoch : 15
Epoch : 16
Epoch : 17
Epoch : 18
Epoch : 19
```

Plot Loss

In [18]:
```
# Plot training loss, testing loss as a function of epochs
```

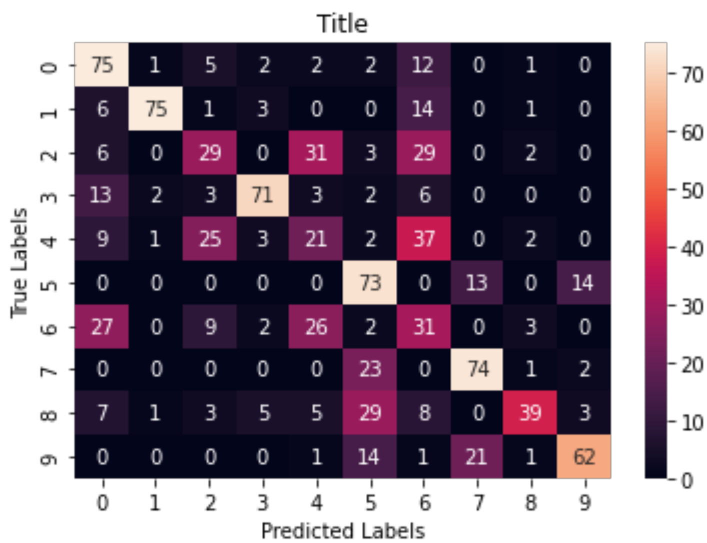
In [19]:
```
epochs = np.arange(1, 21);
plt.figure()
plt.plot(epochs, train_loss_list, label = "Loss on Training Data")
plt.plot(epochs, test_loss_list, label = "Loss on Test Data")
plt.title("Loss vs.Epoch")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.legend()
plt.show()
```



Confusion Matrix

In [20]:
```
def plot_confusion(yhat, y, title = '[Training or Test] Set'):

    pred_train = np.argmax(yhat, axis=1)
    true_train = np.argmax(y, axis=1)
    print(true_train.shape)
    conf_train = np.zeros((10,10))
    for i in range(len(y)):
        conf_train[ true_train[i], pred_train[i] ] += int(1)

    sns.heatmap(conf_train, annot=True, fmt='.3g')
    plt.xlabel('Predicted Labels')
    plt.ylabel('True Labels')
    plt.title('Title')
    plt.show()
# plot_confusion(yhat_train, y_train, title = "Training Set")
# plot_confusion(yhat_test, y_test, title = "Test Set")
#yhat: predictions
#y: one-hot-encoded labels

X_train, y_train, X_test, y_test = data_load()
y_test = one_hot_encode(y_test)
plot_confusion(yhatTest, y_test, title = "Test Data")
```

```
(1000,)
```



Correct and Incorrect Classification Samples

In [8]:
```
def plot_image(vector, out_f_name, label=None):
    """
    Takes a vector as input of size (784) and saves as an image
    """
    image = np.asarray(vector).reshape(28, 28)
    plt.imshow(image, cmap='gray')
    if label:
        plt.title(label)
    plt.axis('off')
    plt.savefig(f'{out_f_name}.png', bbox_inches='tight')
    plt.show()
```

In [9]:
```
# Use plot_image function to display samples that are correctly and incorrectl
```

Effect Of Learning Rate

In [10]:
```
# Plot test loss as a function of epochs
```

Effect of Initialization

In [11]:
```
# Plot test loss as a function of epochs
```

Question 3: CIFAR-10 Classification using CNN

- Please **do not** change the default variable names in this problem, as we will use them in different parts.
- The default variables are initially set to "None".

In [1]:
```
import numpy as np # linear algebra
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torchvision
from torchvision import datasets, transforms, models
from torch.utils.data import *
import random
from tqdm import tqdm
import warnings
```

In [2]:
```
def imshow(img):
  img = img / 2 + 0.5   # unnormalize
  npimg = img.numpy()   # convert from tensor
  plt.imshow(np.transpose(npimg, (1, 2, 0)))
  plt.show()
```

In [3]:

```
# The below two lines are optional and are just there to avoid any SSL
# related errors while downloading the CIFAR-10 dataset
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

#Initializing normalizing transform for the dataset
normalize_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(mean = (0.5, 0.5, 0.5),
                                     std = (0.5, 0.5, 0.5))])

#Downloading the CIFAR10 dataset into train and test sets
train_dataset = torchvision.datasets.CIFAR10(
    root="./CIFAR10/train", train=True,
    transform=normalize_transform,
    download=True)

test_dataset = torchvision.datasets.CIFAR10(
    root="./CIFAR10/test", train=False,
    transform=normalize_transform,
    download=True)


#Generating data loaders from the corresponding datasets
batch_size = 128
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_siz
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size)




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

# get first 100 training images
dataiter = iter(train_loader)
imgs, lbls = dataiter.next()

for i in range(20):
    plt.title(classes[lbls[i]])
    imshow(imgs[i])
```

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

In [4]:
```
# check pytorch cuda and use cuda if possible
device = torch.cuda.is_available()
print('*' * 50)
if torch.cuda.is_available():
  print('CUDA is found! Tranining on %s.......'%torch.cuda.get_device_name(0))
else:
  warnings.warn('CUDA not found! Training may be slow......')
```

```
**************************************************
CUDA is found! Tranining on NVIDIA GeForce RTX 2070 with Max-Q Design.......
```

P1. Build you own CNN model

TODO

- Design your model class in **CNNModel(nn.Module)** and write forward pass in **forward(self, x)**
- Create loss function in **error**, optimizer in **optimizer**
- Define hyparparameters: **learning_rate**, **num_epochs**
- Plot your **loss vs num_epochs** and **accuracy vs num_epochs**

Hints

- Start with low number of epochs for debugging. (eg. num_epochs=1)
- Be careful with the input dimension of fully connected layer.
- The dimension calculation of the output tensor from the input tensor is \ $D_{out} = \frac{D_{in} - K + 2P}{S} + 1$ \ D_{out} : Dimension of output tensor \ D_{in} : Dimension of input tensor \ K : width/height of the kernel \ S : stride \ P : padding

Convolutional and Pooling Layers

A convolutional layer using pyTorch:

```
torch.nn.Conv2d(num_in_channels, num_out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)
```

For example:

```
torch.nn.Conv2d(3, 32, 3)
```

It applies a 2D convolution over an input signal composed of several input planes. If we have input size with (N, C_{in}, H, W) and output size with $(N, C_{out}, H_{out}, W_{out})$, the 2D convolution can described as

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

num_in_channels: is the number of channels of the input tensor. If the previous layer is the input layer, num_in_channels is the number of channels of the image (3 channels for RGB images), otherwise num_in_channels is equal to the number of feature maps of the previous layer.

num_out_channels: is the number of filters (feature extractor) that this layer will apply over the image or feature maps generated by the previous layer.

kernel_size: is the size of the convolving kernel

stride: is the stride of the convolution. Default: 1

padding: is the padding added to all four sides of the input. Default: 0

dilation: is the spacing between kernel elements. Default: 1

group: is the number of blocked connections from input channels to output channels. Default: 1

bias: If True, adds a learnable bias to the output. Default: True

A Simple Convolutional Neural Network

In our convnet we'll initally use this structure shown below:

input -> convolution -> fully connected -> output \

At the end of the last convolutional layer, we get a tensor of dimension (num_channels, height, width). Since now we are going to feed it to a fully connected layer, we need to convert it into a 1-D vector, and for that we use the reshape method:

```
x = x.view(x.size(0), -1)
```

The way of calculating size of the output size from previous convolution layer can be formulized as below:

$$H_{output} = \frac{H_{in} + 2 \times padding - kernel\_Size}{stride} + 1$$

For more details, you can refer to this link: \ https://pytorch.org/docs/stable/generated

In [5]:
```
class CNNModel(nn.Module):
  def __init__(self):
    super(CNNModel, self).__init__()
    # TODO: Create CNNModel using 2D convolution. You should vary the number o
    # In this function, you should define each of the individual components of
    # Example:
    # self.cnn1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, str
    # self.relu1 = nn.ReLU()
    # self.maxpool1 = nn.MaxPool2d(kernel_size=2)
    self.cnn1 = torch.nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size
    self.relu = torch.nn.ReLU()
    self.cnn2 = torch.nn.Conv2d(in_channels = 6, out_channels = 16, kernel_siz
    self.cnn3 = torch.nn.Conv2d(in_channels = 16, out_channels = 24, kernel_si

    # TODO: Create Fully connected layers. You should calculate the dimension
    # Example:
    # self.fc1 = nn.Linear(16 *110 * 110, 5)
    # Fully connected 1
    self.fc1 = torch.nn.Linear(24*24*24, 120)
    self.fc2 = torch.nn.Linear(120, 84)
    self.fc3 = torch.nn.Linear(84, 10)

  def forward(self,x):

    # TODO: Perform forward pass in below section
    # In this function, you will apply the components defined earlier to the i
    # Example:
    out = self.cnn1(x)
    out = self.relu(out)
    out = self.cnn2(out)
    out = self.relu(out)
    #plt.imshow(out[0][0].cpu().detach().numpy())
    #plt.show()
    #plt.close('all')
    out = self.cnn3(out)
    out = self.relu(out)
    #out = self.relu1(out)
    # out = self.maxpool1(out)
    # to visualize feature map in part a, part b.i), use the following three l

    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.relu(out)
    out = self.fc2(out)
    out = self.relu(out)
    out = self.fc3(out)
    return out
```

Starting Up Our Model

We'll send the model to our GPU if you have one so we need to create a CUDA device and instantiate our model. Then we will define the loss function and hyperparameters that we need to train the model: \

###TODO

- Define Cross Entropy Loss
- Create Adam Optimizer
- Define hyperparameters

In [6]:
```
# Create CNN
device = "cuda" if torch.cuda.is_available() else "cpu"
model = CNNModel()
model.to(device)

# TODO: define Cross Entropy Loss
error = torch.nn.CrossEntropyLoss()

# TODO: create Adam Optimizer and define your hyperparameters
learning_rate = 0.001
optimizer = torch.optim.Adam(model.parameters(), learning_rate)
num_epochs = 20

from torchsummary import summary
batch_size = 16
summary(model, input_size=(3, 32, 32))
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─Conv2d: 1-4                            3,480
├─Linear: 1-5                            1,659,000
├─Linear: 1-6                            10,164
├─Linear: 1-7                            850
=================================================================
Total params: 1,674,830
Trainable params: 1,674,830
Non-trainable params: 0
=================================================================
```

Out[6]:

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─Conv2d: 1-4                            3,480
├─Linear: 1-5                            1,659,000
├─Linear: 1-6                            10,164
├─Linear: 1-7                            850
=================================================================
Total params: 1,674,830
Trainable params: 1,674,830
Non-trainable params: 0
=================================================================
```

Training the Model

TODO

- Make predictions from your model
- Calculate Cross Entropy Loss from predictions and labels

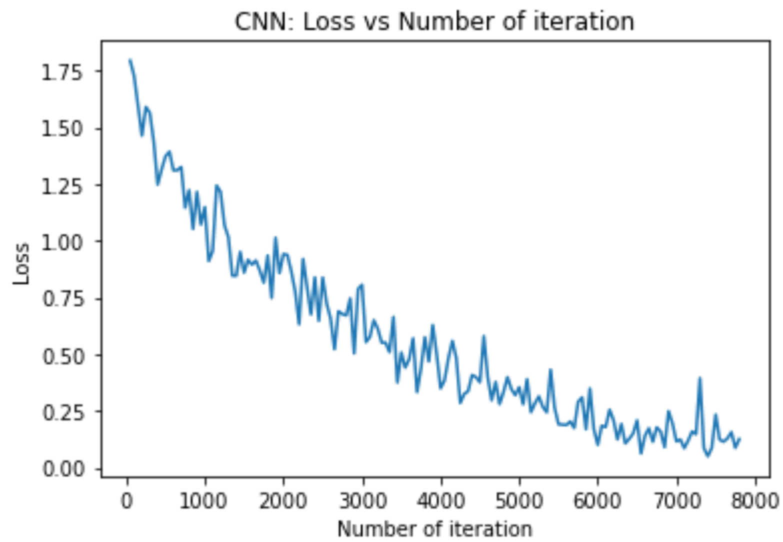
In [7]:
```
count = 0
loss_list = []
iteration_list = []
accuracy_list = []
for epoch in tqdm(range(num_epochs)):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Clear gradients
        optimizer.zero_grad()

        # TODO: Forward propagation
        outputs = model(images)

        # TODO: Calculate softmax and cross entropy loss
        loss = error(outputs, labels)

        # Backprop agate your Loss
        torch.sum(loss).backward()

        # Update CNN model
        optimizer.step()

        count += 1

        if count % 50 == 0:
            model.eval()
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)

                # Forward propagation
                outputs = model(images)

                # Get predictions from the maximum value
                predicted = torch.argmax(outputs,1)

                # Total number of labels
                total += len(labels)

                correct += (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # store loss and iteration
            loss_list.append(loss.item())
            iteration_list.append(count)
            accuracy_list.append(accuracy.item())
        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss
```

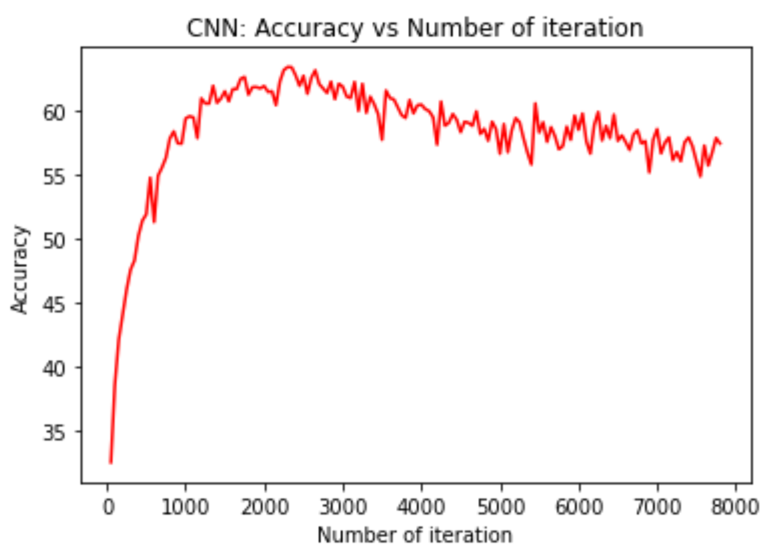
```
  5%|▌         | 1/20 [00:39<12:22, 39.06s/it]
Iteration: 500  Loss: 1.3727850914001465  Accuracy: 51.8599967956543 %

 10%|█         | 2/20 [01:14<11:01, 36.77s/it]

Iteration: 1000  Loss: 1.1488912105560303  Accuracy: 59.34000015258789 %

 15%|█▌        | 3/20 [01:51<10:32, 37.19s/it]

Iteration: 1500  Loss: 0.8596102595329285  Accuracy: 61.46999740600586 %

 25%|██▌       | 5/20 [03:04<09:10, 36.67s/it]

Iteration: 2000  Loss: 0.941204845905304  Accuracy: 61.87999725341797 %

 30%|███       | 6/20 [03:37<08:17, 35.53s/it]

Iteration: 2500  Loss: 0.83780837059021  Accuracy: 62.68000030517578 %

 35%|███▌      | 7/20 [04:12<07:37, 35.20s/it]

Iteration: 3000  Loss: 0.8059139847755432  Accuracy: 61.82999801635742 %

 40%|████      | 8/20 [04:49<07:08, 35.74s/it]

Iteration: 3500  Loss: 0.5071962475776672  Accuracy: 57.68000030517578 %

 50%|█████     | 10/20 [06:02<06:01, 36.18s/it]

Iteration: 4000  Loss: 0.3492661714553833  Accuracy: 60.43000030517578 %

 55%|█████▌    | 11/20 [06:40<05:31, 36.88s/it]

Iteration: 4500  Loss: 0.37602102756500244  Accuracy: 58.279998779296875 %

 60%|██████    | 12/20 [07:14<04:47, 35.97s/it]

Iteration: 5000  Loss: 0.35433849692344666  Accuracy: 56.59000015258789 %

 70%|███████   | 14/20 [08:26<03:35, 35.91s/it]

Iteration: 5500  Loss: 0.19080926477909088  Accuracy: 58.25 %

 75%|███████▌  | 15/20 [09:02<02:59, 36.00s/it]

Iteration: 6000  Loss: 0.1000833660364151  Accuracy: 58.459999084472656 %

 80%|████████  | 16/20 [09:38<02:23, 35.87s/it]

Iteration: 6500  Loss: 0.2079406976699829  Accuracy: 57.599998474121094 %

 85%|████████▌ | 17/20 [10:09<01:43, 34.49s/it]

Iteration: 7000  Loss: 0.11518188565969467  Accuracy: 58.519996643066406 %

 95%|█████████▌| 19/20 [11:28<00:36, 36.95s/it]

Iteration: 7500  Loss: 0.23340679705142975  Accuracy: 55.98999786376953 %

100%|██████████| 20/20 [12:04<00:00, 36.23s/it]
```

In [8]:
```python
# !pip install torchsummary
# uncomment above line if you don't have torchsummary installed yet
# Print torchsummary of model
from torchsummary import summary
print(summary(model, input_size = (3, 32, 32)))
print("Kernel Size: 3x3")
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─Conv2d: 1-4                            3,480
├─Linear: 1-5                            1,659,000
├─Linear: 1-6                            10,164
├─Linear: 1-7                            850
=================================================================
Total params: 1,674,830
Trainable params: 1,674,830
Non-trainable params: 0
=================================================================
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─Conv2d: 1-4                            3,480
├─Linear: 1-5                            1,659,000
├─Linear: 1-6                            10,164
├─Linear: 1-7                            850
=================================================================
Total params: 1,674,830
Trainable params: 1,674,830
Non-trainable params: 0
=================================================================
Kernel Size: 3x3
```

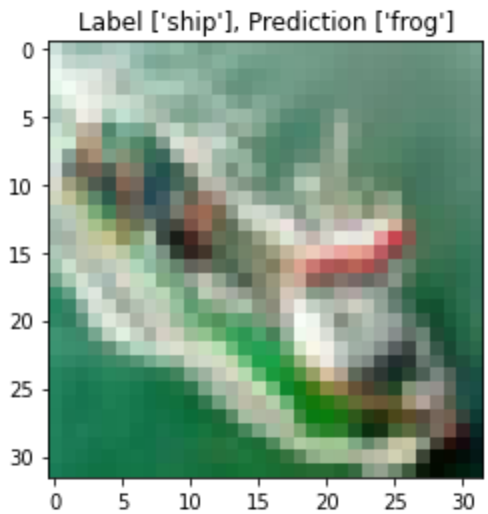
In [9]:
```
# visualization loss
plt.plot(iteration_list,loss_list)
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "red")
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.show()
```





Evaluating the Model

In [10]:
```
import random
#To-do: evaluate on test set, instead of training set
random_image = random.randint(0,len(test_dataset))
image = test_dataset.__getitem__(random_image)
model.eval()
images, labels = next(iter(test_loader))
images, labels = images.to(device), labels.to(device)
predictions = torch.argmax(model(images),1)
num_cols=1
num_rows = 25# len(labels)
label_map = [['airplane'],['automobile'],['bird'],['cat'], ['deer'], ['dog'],

r = list(range(num_rows))
random.shuffle(r)
rand_range = r[0:5]
for idx in rand_range:
  img = images.cpu()[idx]

  plt.title(f"Label {label_map[labels[idx]]}, Prediction {label_map[prediction
  imshow(img)

  plt.axis('off')

plt.show()
```

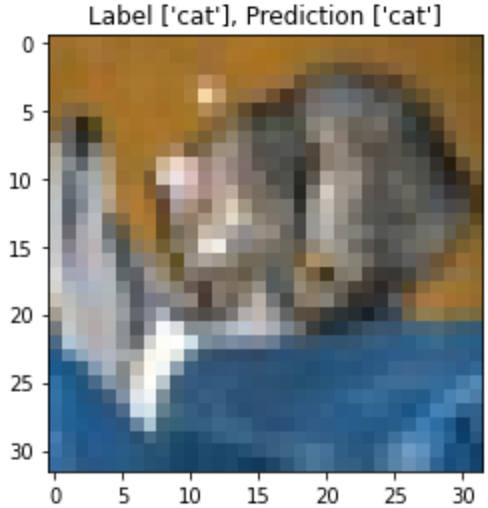



Label ['airplane'], Prediction ['airplane']
Label ['dog'], Prediction ['deer']

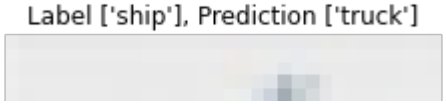

Label ['horse'], Prediction ['horse']

In [11]:
```
#To-do: evaluate on test set, instead of training set
random_image = random.randint(0,len(test_dataset))
image = test_dataset.__getitem__(random_image)
model.eval()
images, labels = next(iter(test_loader))
images, labels = images.to(device), labels.to(device)
predictions = torch.argmax(model(images),1)
num_cols=1
num_rows = 25# len(labels)
label_map = [['airplane'],['automobile'],['bird'],['cat'], ['deer'], ['dog'],

for idx in range(num_rows):
  img = images.cpu()[idx]

  plt.title(f"Label {label_map[labels[idx]]}, Prediction {label_map[prediction
  imshow(img)

  plt.axis('off')
plt.show()
```

Label ['cat'], Prediction ['cat']



Label ['ship'], Prediction ['truck']



Part 2: Additional Components

i.)

In [12]:
```
class CNNModel(nn.Module):
  def __init__(self):
    super(CNNModel, self).__init__()
    # TODO: Create CNNModel using 2D convolution. You should vary the number o
    # In this function, you should define each of the individual components of
    # Example:
    # self.cnn1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, str
    # self.relu1 = nn.ReLU()
    # self.maxpool1 = nn.MaxPool2d(kernel_size=2)
    self.cnn1 = torch.nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size
    self.relu = torch.nn.ReLU()
    self.cnn2 = torch.nn.Conv2d(in_channels = 6, out_channels = 16, kernel_siz

    self.maxpool1 = nn.MaxPool2d(2)

    self.cnn3 = torch.nn.Conv2d(in_channels = 16, out_channels = 24, kernel_si
    self.cnn4 = torch.nn.Conv2d(in_channels = 24, out_channels = 24, kernel_si
    self.cnn5 = torch.nn.Conv2d(in_channels = 24, out_channels = 24, kernel_si

    # TODO: Create Fully connected layers. You should calculate the dimension
    # Example:
    # self.fc1 = nn.Linear(16 *110 * 110, 5)
    # Fully connected 1
    self.fc1 = torch.nn.Linear(13824, 120)
    self.fc2 = torch.nn.Linear(120, 84)
    self.fc3 = torch.nn.Linear(84, 10)

  def forward(self,x):

    # TODO: Perform forward pass in below section
    # In this function, you will apply the components defined earlier to the i
    # Example:
    out = self.cnn1(x)
    out = self.relu(out)
    out = self.cnn2(out)
    out = self.relu(out)
    #plt.imshow(out[0][0].cpu().detach().numpy())
    #plt.show()
    #plt.close('all')
    out = self.cnn3(out)
    out = self.relu(out)
    #out = self.relu1(out)
    # out = self.maxpool1(out)
    # to visualize feature map in part a, part b.i), use the following three l

    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.relu(out)
    out = self.fc2(out)
    out = self.relu(out)
    out = self.fc3(out)
    return out

  # Create CNN
device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
model = CNNModel()
model.to(device)

# TODO: define Cross Entropy Loss
error = torch.nn.CrossEntropyLoss()

# TODO: create Adam Optimizer and define your hyperparameters
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
num_epochs = 20

from torchsummary import summary
print(summary(model,input_size=(3, 32, 32)))

count = 0
loss_list_bi = []
iteration_list_bi = []
accuracy_list_bi = []
for epoch in tqdm(range(num_epochs)):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Clear gradients
        optimizer.zero_grad()

        # TODO: Forward propagation
        outputs = model(images)

        # TODO: Calculate softmax and cross entropy loss
        loss = error(outputs,labels)

        # Backprop agate your Loss
        T = torch.sum(loss)
        T.backward()

        # Update CNN model
        optimizer.step()

        count += 1

        if count % 50 == 0:
            model.eval()
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)

                # Forward propagation
                outputs = model(images)

                # Get predictions from the maximum value
                predicted = torch.argmax(outputs,1)
```

```
                # Total number of labels
                total = total + len(labels)
                correct = correct + (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # store loss and iteration
            loss_list_bi.append(loss.item())
            iteration_list_bi.append(count)
            accuracy_list_bi.append(accuracy.item())
        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─MaxPool2d: 1-4                         --
├─Conv2d: 1-5                            3,480
├─Conv2d: 1-6                            5,208
├─Conv2d: 1-7                            5,208
├─Linear: 1-8                            1,659,000
├─Linear: 1-9                            10,164
├─Linear: 1-10                           850
=================================================================
Total params: 1,685,246
Trainable params: 1,685,246
Non-trainable params: 0
=================================================================
```



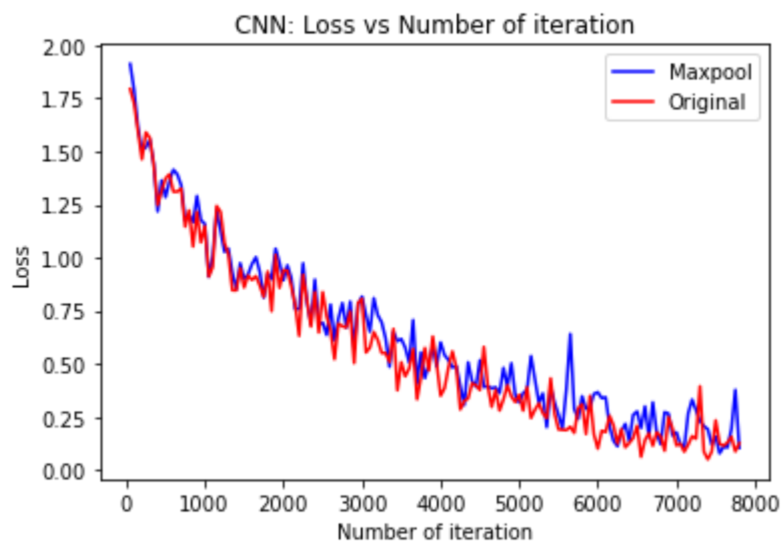
```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─MaxPool2d: 1-4                         --
├─Conv2d: 1-5                            3,480
├─Conv2d: 1-6                            5,208
├─Conv2d: 1-7                            5,208
├─Linear: 1-8                            1,659,000
├─Linear: 1-9                            10,164
├─Linear: 1-10                           850
=================================================================
Total params: 1,685,246
Trainable params: 1,685,246
Non-trainable params: 0
=================================================================
```

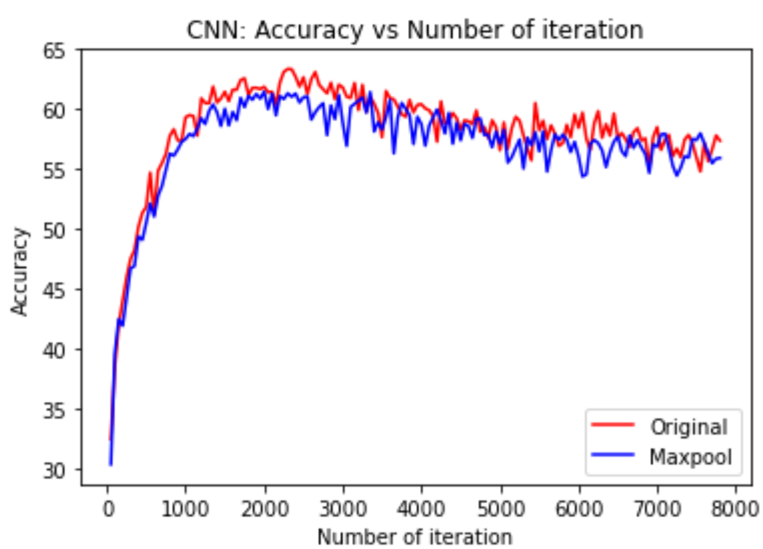


```
  5%|▌         | 1/20 [00:31<10:03, 31.79s/it]

Iteration: 500  Loss: 1.284759521484375  Accuracy: 50.55999755859375 %

 10%|█         | 2/20 [01:08<10:23, 34.62s/it]
```

```
Iteration: 1000  Loss: 1.1590650081634521  Accuracy: 57.529998779296875 %
 15%|█▌        | 3/20 [01:43<09:51, 34.81s/it]

Iteration: 1500  Loss: 0.8989694714546204  Accuracy: 60.04999923706055 %

 25%|██▌       | 5/20 [02:55<08:55, 35.72s/it]

Iteration: 2000  Loss: 0.8936105370521545  Accuracy: 61.47999954223633 %

 30%|███       | 6/20 [03:27<08:03, 34.54s/it]

Iteration: 2500  Loss: 0.6931913495063782  Accuracy: 61.0099983215332 %

 35%|███▌      | 7/20 [04:03<07:31, 34.74s/it]

Iteration: 3000  Loss: 0.816838264465332  Accuracy: 58.89999771118164 %

 40%|████      | 8/20 [04:37<06:54, 34.55s/it]

Iteration: 3500  Loss: 0.6166757941246033  Accuracy: 58.31999969482422 %

 50%|█████     | 10/20 [05:43<05:38, 33.87s/it]

Iteration: 4000  Loss: 0.6007229089736938  Accuracy: 58.769996643066406 %

 55%|█████▌    | 11/20 [06:16<05:00, 33.39s/it]

Iteration: 4500  Loss: 0.5166906118392944  Accuracy: 57.41999816894531 %

 60%|██████    | 12/20 [06:46<04:19, 32.43s/it]

Iteration: 5000  Loss: 0.31766220927238464  Accuracy: 57.119998931884766 %

 70%|███████   | 14/20 [07:54<03:20, 33.37s/it]

Iteration: 5500  Loss: 0.25976741313934326  Accuracy: 56.5 %

 75%|███████▌  | 15/20 [08:28<02:46, 33.32s/it]

Iteration: 6000  Loss: 0.3657153844833374  Accuracy: 56.05999755859375 %

 80%|████████  | 16/20 [09:01<02:13, 33.48s/it]

Iteration: 6500  Loss: 0.27545157074928284  Accuracy: 57.73999786376953 %

 85%|████████▌ | 17/20 [09:33<01:38, 32.99s/it]

Iteration: 7000  Loss: 0.1770372986793518  Accuracy: 56.87999725341797 %

 95%|█████████▌| 19/20 [10:41<00:33, 33.41s/it]

Iteration: 7500  Loss: 0.1567377895116806  Accuracy: 57.439998626708984 %

100%|██████████| 20/20 [11:14<00:00, 33.73s/it]
```

In [13]:
```
# visualization loss
plt.plot(iteration_list_bi,loss_list_bi, color = "blue", label = 'Maxpool')
plt.plot(iteration_list,loss_list, color = "red", label = 'Original')
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.legend()
plt.show()

# visualization accuracy
plt.plot(iteration_list,accuracy_list,color = "red", label = 'Original')
plt.plot(iteration_list_bi,accuracy_list_bi,color = "blue", label = 'Maxpool')
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.legend()
plt.show()

print("\nMemory Required for Model: 6.85 MB")
print("\nIn terms of observations, it seems that the Maxpool and Original meth
```

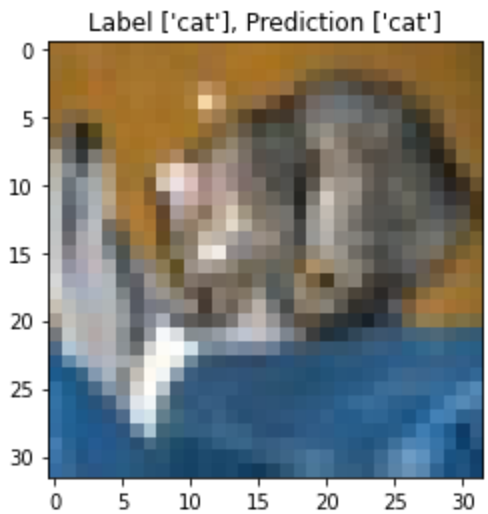


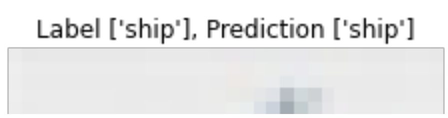


```
Memory Required for Model: 6.85 MB

In terms of observations, it seems that the Maxpool and Original method seem
to have similar trends for the
accuracy and loss rate. However, it appears that Maxpool method is slightly l
ess accurate than the original method.
```

In [14]:

```python
#To-do: evaluate on test set, instead of training set
random_image = random.randint(0,len(test_dataset))
image = test_dataset.__getitem__(random_image)
model.eval()
images, labels = next(iter(test_loader))
images, labels = images.to(device), labels.to(device)
predictions = torch.argmax(model(images),1)
num_cols = 1
num_rows = 25# len(labels)
label_map = [['airplane'],['automobile'],['bird'],['cat'], ['deer'], ['dog'],

for idx in range(num_rows):
  img = images.cpu()[idx]

  plt.title(f"Label {label_map[labels[idx]]}, Prediction {label_map[prediction
  imshow(img)

  plt.axis('off')
plt.show()
```





ii.)

In [15]:
```
class CNNModel(nn.Module):
  def __init__(self):
    super(CNNModel, self).__init__()
    # TODO: Create CNNModel using 2D convolution. You should vary the number o
    # In this function, you should define each of the individual components of
    # Example:
    self.cnn1 = torch.nn.Conv2d(in_channels = 3, out_channels = 6, kernel_size
    self.relu = torch.nn.ReLU()
    #Input = 6 x 28 x 28, Output = 16 x 26 x 26
    self.cnn2 =torch.nn.Conv2d(in_channels = 6, out_channels = 16, kernel_size


    self.maxpool1 = nn.MaxPool2d(2)

    self.cnn3 = torch.nn.Conv2d(in_channels = 16, out_channels = 24, kernel_si
    self.cnn4 = torch.nn.Conv2d(in_channels = 24, out_channels = 24, kernel_si
    self.cnn5 = torch.nn.Conv2d(in_channels = 24, out_channels = 24, kernel_si


    # TODO: Create Fully connected layers. You should calculate the dimension
    # Example:
    # self.fc1 = nn.Linear(16 *110 * 110, 5)
    # Fully connected 1
    self.fc1 = torch.nn.Linear(24*7*7, 120)
    self.fc2 = torch.nn.Linear(120, 84)
    self.fc3 = torch.nn.Linear(84, 10)


  def forward(self,x):

    # TODO: Perform forward pass in below section
    # In this function, you will apply the components defined earlier to the i
    # Example:
    out = self.cnn1(x)
    out = self.relu(out)
    out = self.cnn2(out)
    out = self.relu(out)
    #plt.imshow(out[0][0].cpu().detach().numpy())
    #plt.show()
    #plt.close('all')
    out = self.maxpool1(out)
    out = self.cnn3(out)
    out = self.relu(out)
    out = self.cnn4(out)
    out = self.relu(out)
    out = self.cnn5(out)
    out = self.relu(out)
    #out = self.relu1(out)

    # to visualize feature map in part a, part b.i), use the following three l

    out = out.view(out.size(0), -1)
    out = self.fc1(out)
    out = self.relu(out)
    out = self.fc2(out)
```

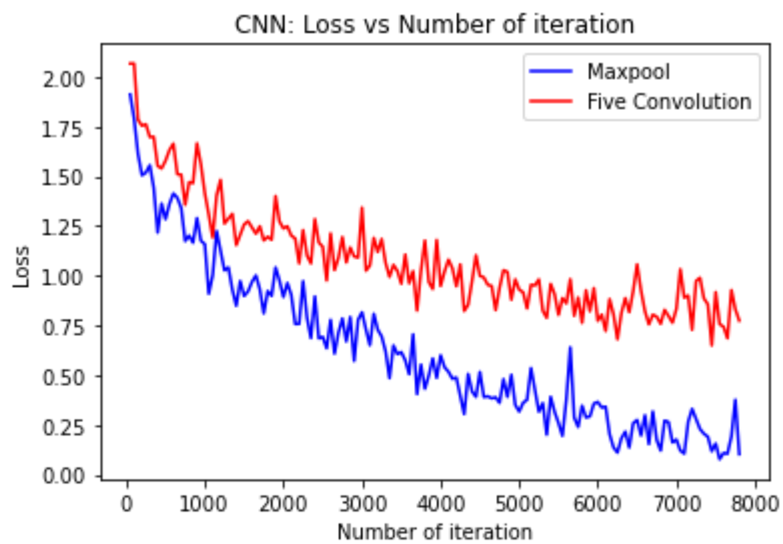
```
    out = self.relu(out)
    out = self.fc3(out)

    return out

# Create CNN
device = "cuda" if torch.cuda.is_available() else "cpu"
model = CNNModel()
model.to(device)

# TODO: define Cross Entropy Loss
error = torch.nn.CrossEntropyLoss()

# TODO: create Adam Optimizer and define your hyperparameters
learning_rate = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
num_epochs = 20

from torchsummary import summary
print(summary(model,input_size=(3, 32, 32)))

count = 0
loss_list_bii = []
iteration_list_bii = []
accuracy_list_bii = []
for epoch in tqdm(range(num_epochs)):
    model.train()
    for i, (images, labels) in enumerate(train_loader):
        images, labels = images.to(device), labels.to(device)

        # Clear gradients
        optimizer.zero_grad()

        # TODO: Forward propagation
        outputs = model(images)

        # TODO: Calculate softmax and cross entropy loss
        loss = error(outputs,labels)

        # Backprop agate your Loss
        T = torch.sum(loss)
        T.backward()

        # Update CNN model
        optimizer.step()
        count = count + 1

        if count % 50 == 0:
            model.eval()
            # Calculate Accuracy
            correct = 0
            total = 0
            # Iterate through test dataset
            for images, labels in test_loader:
                images, labels = images.to(device), labels.to(device)
```

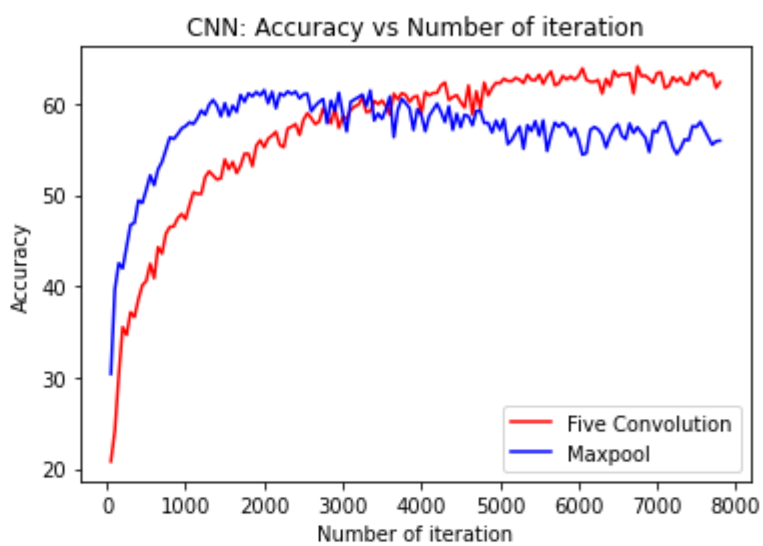
```
                # Forward propagation
                outputs = model(images)

                # Get predictions from the maximum value
                predicted = torch.argmax(outputs,1)

                # Total number of labels
                total = total + len(labels)

                correct = correct + (predicted == labels).sum()

            accuracy = 100 * correct / float(total)

            # store loss and iteration
            loss_list_bii.append(loss.item())
            iteration_list_bii.append(count)
            accuracy_list_bii.append(accuracy.item())
        if count % 500 == 0:
            # Print Loss
            print('Iteration: {}  Loss: {}  Accuracy: {} %'.format(count, loss
```

```
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─MaxPool2d: 1-4                         --
├─Conv2d: 1-5                            3,480
├─Conv2d: 1-6                            5,208
├─Conv2d: 1-7                            5,208
├─Linear: 1-8                            141,240
├─Linear: 1-9                            10,164
├─Linear: 1-10                           850
=================================================================
Total params: 167,486
Trainable params: 167,486
Non-trainable params: 0
=================================================================
=================================================================
Layer (type:depth-idx)                   Param #
=================================================================
├─Conv2d: 1-1                            456
├─ReLU: 1-2                              --
├─Conv2d: 1-3                            880
├─MaxPool2d: 1-4                         --
├─Conv2d: 1-5                            3,480
├─Conv2d: 1-6                            5,208
├─Conv2d: 1-7                            5,208
├─Linear: 1-8                            141,240
├─Linear: 1-9                            10,164
├─Linear: 1-10                           850
=================================================================
Total params: 167,486
Trainable params: 167,486
Non-trainable params: 0
```

```
  5%|▌         | 1/20 [00:31<10:01, 31.66s/it]
Iteration: 500  Loss: 1.5793367624282837  Accuracy: 40.55999755859375 %
 10%|█         | 2/20 [01:07<10:13, 34.07s/it]
Iteration: 1000  Loss: 1.415574073791504  Accuracy: 47.34000015258789 %
 15%|█▌        | 3/20 [01:42<09:44, 34.39s/it]
Iteration: 1500  Loss: 1.2583825588226318  Accuracy: 53.87999725341797 %
 25%|██▌       | 5/20 [02:51<08:39, 34.63s/it]
Iteration: 2000  Loss: 1.2385494709014893  Accuracy: 55.21999740600586 %
 30%|███       | 6/20 [03:20<07:36, 32.60s/it]
Iteration: 2500  Loss: 1.1462451219558716  Accuracy: 58.21999740600586 %
 35%|███▌      | 7/20 [03:50<06:51, 31.62s/it]
Iteration: 3000  Loss: 1.3434181213378906  Accuracy: 58.55999755859375 %
 40%|████      | 8/20 [04:23<06:27, 32.28s/it]
Iteration: 3500  Loss: 0.960294783115387  Accuracy: 60.349998474121094 %
 50%|█████     | 10/20 [05:30<05:26, 32.68s/it]
Iteration: 4000  Loss: 0.9490000009536743  Accuracy: 58.87999725341797 %
 55%|█████▌    | 11/20 [06:04<04:57, 33.04s/it]
Iteration: 4500  Loss: 0.9997236132621765  Accuracy: 60.18000030517578 %
 60%|██████    | 12/20 [06:36<04:22, 32.79s/it]
Iteration: 5000  Loss: 0.9303755164146423  Accuracy: 62.22999954223633 %
 70%|███████   | 14/20 [07:45<03:21, 33.65s/it]
Iteration: 5500  Loss: 0.8039083480834961  Accuracy: 63.18000030517578 %
 75%|███████▌  | 15/20 [08:20<02:50, 34.13s/it]
Iteration: 6000  Loss: 0.7739051580429077  Accuracy: 63.029998779296875 %
 80%|████████  | 16/20 [08:56<02:18, 34.59s/it]
Iteration: 6500  Loss: 1.0581154823303223  Accuracy: 62.97999954223633 %
 85%|████████▌ | 17/20 [09:29<01:42, 34.22s/it]
Iteration: 7000  Loss: 0.8343865871429443  Accuracy: 63.349998474121094 %
 95%|█████████▌| 19/20 [10:39<00:34, 34.81s/it]
Iteration: 7500  Loss: 0.9185515642166138  Accuracy: 62.66999816894531 %
100%|██████████| 20/20 [11:15<00:00, 33.77s/it]
```

In [16]:
```
# visualization loss
plt.plot(iteration_list_bi,loss_list_bi, color = "blue", label = 'Maxpool')
plt.plot(iteration_list_bii,loss_list_bii, color = "red", label = 'Five Convol
plt.xlabel("Number of iteration")
plt.ylabel("Loss")
plt.title("CNN: Loss vs Number of iteration")
plt.legend()
plt.show()

# visualization accuracy
plt.plot(iteration_list_bii,accuracy_list_bii,color = "red", label = 'Five Con
plt.plot(iteration_list_bi,accuracy_list_bi,color = "blue", label = 'Maxpool')
plt.xlabel("Number of iteration")
plt.ylabel("Accuracy")
plt.title("CNN: Accuracy vs Number of iteration")
plt.legend()
plt.show()

print("\nMemory Required for Model: 1.00 MB")
```





```
Memory Required for Model: 1.00 MB
```

In [17]:
```python
print("As the method in part 3 uses more training hyper parameters than 2, we
```

```
As the method in part 3 uses more training hyper parameters than 2, we can se
e that it tends to be
more accurate and takes up much less memory as well
relative to the older method. However, the drawback to this is that we have a
decrease in loss compared to older methods
due to the more channels used to counter hyperparameters.
```