Lab 4: Data Imputation using an Autoencoder

Deadline: Mon, March 01, 5:00pm

Late Penalty: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted between 1 hour and 24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted. Quercus submission time will be used, not your local computer time. You can submit your labs as many times as you want before the deadline, so please submit often and early.

TA: Chris Lucasius christopher.lucasius@mail.utoronto.ca (mailto:christopher.lucasius@mail.utoronto.ca)

In this lab, you will build and train an autoencoder to impute (or "fill in") missing data.

We will be using the Adult Data Set provided by the UCI Machine Learning Repository [1], available at https://archive.ics.uci.edu/ml/datasets/adult (https://archive.ics.uci.edu/ml/datasets/adult). The data set contains census record files of adults, including their age, martial status, the type of work they do, and other features.

Normally, people use this data set to build a supervised classification model to classify whether a person is a high income earner. We will not use the dataset for this original intended purpose.

Instead, we will perform the task of imputing (or "filling in") missing values in the dataset. For example, we may be missing one person's martial status, and another person's age, and a third person's level of education. Our model will predict the missing features based on the information that we do have about each person.

We will use a variation of a denoising autoencoder to solve this data imputation problem. Our autoencoder will be trained using inputs that have one categorical feature artificially removed, and the goal of the autoencoder is to correctly reconstruct all features, including the one removed from the input.

In the process, you are expected to learn to:

1. Clean and process continuous and categorical data for machine learning.
2. Implement an autoencoder that takes continuous and categorical (one-hot) inputs.
3. Tune the hyperparameters of an autoencoder.
4. Use baseline models to help interpret model performance.

[1] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml (http://archive.ics.uci.edu/ml)]. Irvine, CA: University of California, School of Information and Computer Science.

What to submit

Submit a PDF file containing all your code, outputs, and write-up. You can produce a PDF of your Google Colab file by going to File > Print and then save as PDF. The Colab instructions have more information (.html files are also acceptable).

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Colab Link

Include a link to your Colab file here. If you would like the TA to look at your Colab file in case your solutions are cut off, **please make sure that your Colab file is publicly accessible at the time of submission**.

Colab Link: https://drive.google.com/file/d/1BO681Cl_zB5_cyyyJSPvS_rZSO8fPkvo/view?usp=sharing (https://drive.google.com/file/d/1BO681Cl_zB5_cyyyJSPvS_rZSO8fPkvo/view?usp=sharing)

In [29]:
```
import csv
import numpy as np
import random
import torch
import torch.utils.data
import tensorflow as tf
import matplotlib.pyplot as plt
```

Part 0

We will be using a package called `pandas` for this assignment.

If you are using Colab, `pandas` should already be available. If you are using your own computer, installation instructions for `pandas` are available here: https://pandas.pydata.org/pandas-docs/stable/install.html (https://pandas.pydata.org/pandas-docs/stable/install.html)

In [30]:
```
import pandas as pd
```

Part 1. Data Cleaning [15 pt]

The adult.data file is available at `https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data`

The function `pd.read_csv` loads the adult.data file into a pandas dataframe. You can read about the pandas documentation for `pd.read_csv` at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html)

In [31]:
```
header = ['age', 'work', 'fnlwgt', 'edu', 'yredu', 'marriage', 'occupation',
 'relationship', 'race', 'sex', 'capgain', 'caploss', 'workhr', 'country']
df = pd.read_csv(
    "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.dat
a",
    names=header,
    index_col=False)
```

In [32]:
```
df.shape # there are 32561 rows (records) in the data frame, and 14 columns (features)
```

Out[32]: (32561, 14)

Part (a) Continuous Features [3 pt]

For each of the columns `["age", "yredu", "capgain", "caploss", "workhr"]`, report the minimum, maximum, and average value across the dataset.

Then, normalize each of the features `["age", "yredu", "capgain", "caploss", "workhr"]` so that their values are always between 0 and 1. Make sure that you are actually modifying the dataframe `df`.

Like numpy arrays and torch tensors, pandas data frames can be sliced. For example, we can display the first 3 rows of the data frame (3 records) below.

In [33]:
```
df[:3] # show the first 3 records
```

Out[33]:

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationship	race	sex	capg
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	21
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	

Alternatively, we can slice based on column names, for example `df["race"]`, `df["hr"]`, or even index multiple columns like below.

In [34]:
```
subdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
subdf[:3] # show the first 3 records
```

Out[34]:

	age	yredu	capgain	caploss	workhr
0	39	13	2174	0	40
1	50	13	0	0	13
2	38	9	0	0	40

Numpy works nicely with pandas, like below:

In [35]:
```
np.sum(subdf["caploss"])
```

Out[35]: 2842700

Just like numpy arrays, you can modify entire columns of data rather than one scalar element at a time. For example, the code

```
df["age"] = df["age"] + 1
```

would increment everyone's age by 1.

In [36]:
```
#For each of the columns ["age", "yredu", "capgain", "caploss", "workhr"], rep
ort the minimum, maximum, and average value across the dataset.
def getMinMaxAvgInformation(colName):
  minCalculation = min(subdf[colName])
  maxCalculation = max(subdf[colName])
  avgCalculation = sum(subdf[colName])/len(subdf[colName])

  print("The minimum", colName, "is", minCalculation)
  print("The maximum", colName, "is", maxCalculation)
  print("The average", colName, "is", avgCalculation)
  print('\n')

getMinMaxAvgInformation('age')
getMinMaxAvgInformation('yredu')
getMinMaxAvgInformation('capgain')
getMinMaxAvgInformation('caploss')
getMinMaxAvgInformation('workhr')
```

```
The minimum age is 17
The maximum age is 90
The average age is 38.58164675532078


The minimum yredu is 1
The maximum yredu is 16
The average yredu is 10.0806793403151


The minimum capgain is 0
The maximum capgain is 99999
The average capgain is 1077.6488437087312


The minimum caploss is 0
The maximum caploss is 4356
The average caploss is 87.303829734959


The minimum workhr is 1
The maximum workhr is 99
The average workhr is 40.437455852092995
```

In [37]:
```
#Then, normalize each of the features ["age", "yredu", "capgain", "caploss",
 "workhr"] so that their values are always between 0 and 1.
#Make sure that you are actually modifying the dataframe df.

df["age"] = (df["age"] - (df["age"]).min())/((df["age"]).max() - (df["age"]).m
in())
df["yredu"] = (df["yredu"] - (df["yredu"]).min())/((df["yredu"]).max() - (df[
"yredu"]).min())
df["capgain"] = (df["capgain"] - (df["capgain"]).min())/((df["capgain"]).max()
- (df["capgain"]).min())
df["caploss"] = (df["caploss"] - (df["caploss"]).min())/((df["caploss"]).max()
- (df["caploss"]).min())
df["workhr"] = (df["workhr"] - (df["workhr"]).min())/((df["workhr"]).max() - (
df["workhr"]).min())
```

Part (b) Categorical Features [1 pt]

What percentage of people in our data set are male? Note that the data labels all have an unfortunate space in the beginning, e.g. " Male" instead of "Male".

What percentage of people in our data set are female?

In [38]:
```
# hint: you can do something like this in pandas
numMales = sum(df["sex"] == " Male")
ans1 = (numMales*100)/len(df["sex"])

numFemales = sum(df["sex"] == " Female")
ans2 = (numFemales*100)/len(df["sex"])

print("The percentage of people that are males in our dataset is:", ans1, "%")
print("The percentage of people that are females in our dataset is:", ans2,
"%")
```

```
The percentage of people that are males in our dataset is: 66.92054912318417
%
The percentage of people that are females in our dataset is: 33.0794508768158
2 %
```

Part (c) [2 pt]

Before proceeding, we will modify our data frame in a couple more ways:

1. We will restrict ourselves to using a subset of the features (to simplify our autoencoder)
2. We will remove any records (rows) already containing missing values, and store them in a second dataframe. We will only use records without missing values to train our autoencoder.

Both of these steps are done for you, below.

How many records contained missing features? What percentage of records were removed?

In [39]:
```
contcols = ["age", "yredu", "capgain", "caploss", "workhr"]
catcols = ["work", "marriage", "occupation", "edu", "relationship", "sex"]
features = contcols + catcols
df = df[features]
```

In [40]:
```
missing = pd.concat([df[c] == " ?" for c in catcols], axis=1).any(axis=1)
df_with_missing = df[missing]
df_not_missing = df[~missing]
```

In [41]:
```
print(len(df_with_missing), "records contained missing information")
print((len(df_with_missing)*100/len(df)), "% of the records were removed")
```

```
1843 records contained missing information
5.660145572924664 % of the records were removed
```

Part (d) One-Hot Encoding [1 pt]

What are all the possible values of the feature "work" in `df_not_missing` ? You may find the Python function `set` useful.

In [42]:
```
workTypes = set()

for x in df_not_missing['work']:
  workTypes.add(x)

print("All the possible values of the feature 'work' in df_not_missing are: ")
for c in workTypes:
  print(c)
```

```
All the possible values of the feature 'work' in df_not_missing are:
 Federal-gov
 Self-emp-inc
 Local-gov
 Without-pay
 State-gov
 Self-emp-not-inc
 Private
```

We will be using a one-hot encoding to represent each of the categorical variables. Our autoencoder will be trained using these one-hot encodings.

We will use the pandas function `get_dummies` to produce one-hot encodings for all of the categorical variables in `df_not_missing` .

In [43]:
```
data = pd.get_dummies(df_not_missing)
```

In [44]:
```
data[:3]
```

Out[44]:

	age	yredu	capgain	caploss	workhr	work_ Federal-gov	work_ Local-gov	work_ Private	work_ Self-emp-inc	work_ Self-emp-not-inc	wor Stat g
0	0.301370	0.800000	0.02174	0.0	0.397959	0	0	0	0	0	
1	0.452055	0.800000	0.00000	0.0	0.122449	0	0	0	0	1	
2	0.287671	0.533333	0.00000	0.0	0.397959	0	0	1	0	0	

Part (e) One-Hot Encoding [2 pt]

The dataframe `data` contains the cleaned and normalized data that we will use to train our denoising autoencoder.

How many **columns** (features) are in the dataframe `data` ?

Briefly explain where that number come from.

In [45]:
```
data.shape
```

Out[45]:
```
(30718, 57)
```

There are 57 columns or features in the dataframe 'data'. The function, 'get_dummies', transferred strings in data set to values of 0 and 1. Now we have one column representing each possible outcome for things like work. Now we only need 0s and 1s to represent all the information.

Part (f) One-Hot Conversion [3 pt]

We will convert the pandas data frame `data` into numpy, so that it can be further converted into a PyTorch tensor. However, in doing so, we lose the column label information that a panda data frame automatically stores.

Complete the function `get_categorical_value` that will return the named value of a feature given a one-hot embedding. You may find the global variables `cat_index` and `cat_values` useful. (Display them and figure out what they are first.)

We will need this function in the next part of the lab to interpret our autoencoder outputs. So, the input to our function `get_categorical_values` might not actually be "one-hot" -- the input may instead contain real-valued predictions from our neural network.

In [46]:
```
datanp = data.values.astype(np.float32)
```

In [47]:
```
cat_index = {}  # Mapping of feature -> start index of feature in a record
cat_values = {} # Mapping of feature -> list of categorical values the feature
can take

# build up the cat_index and cat_values dictionary
for i, header in enumerate(data.keys()):
    if "_" in header: # categorical header
        feature, value = header.split()
        feature = feature[:-1] # remove the last char; it is always an undersc
ore
        if feature not in cat_index:
            cat_index[feature] = i
            cat_values[feature] = [value]
        else:
            cat_values[feature].append(value)

def get_onehot(record, feature):
    """
    Return the portion of `record` that is the one-hot encoding
    of `feature`. For example, since the feature "work" is stored
    in the indices [5:12] in each record, calling `get_range(record, "work")`
    is equivalent to accessing `record[5:12]`.

    Args:
        - record: a numpy array representing one record, formatted
                  the same way as a row in `data.np`
        - feature: a string, should be an element of `catcols`
    """
    start_index = cat_index[feature]
    stop_index = cat_index[feature] + len(cat_values[feature])
    return record[start_index:stop_index]

def get_categorical_value(onehot, feature):
    """
    Return the categorical value name of a feature given
    a one-hot vector representing the feature.

    Args:
        - onehot: a numpy array one-hot representation of the feature
        - feature: a string, should be an element of `catcols`

    Examples:

    >>> get_categorical_value(np.array([0., 0., 0., 0., 0., 1., 0.]), "work")
    'State-gov'
    >>> get_categorical_value(np.array([0.1, 0., 1.1, 0.2, 0., 1., 0.]), "wor
k")
    'Private'
    """
    # <----- TODO: WRITE YOUR CODE HERE ----->
    # You may find the variables `cat_index` and `cat_values`
    # (created above) useful.
    index = np.where(onehot == onehot.max())[0][0]
    return cat_values[feature][index]
```

In [48]:
```
print("Cat_index information:", cat_index, "\n")
print("Cat_values information:", cat_values)
```

```
Cat_index information: {'work': 5, 'marriage': 12, 'occupation': 19, 'edu': 3
3, 'relationship': 49, 'sex': 55}

Cat_values information: {'work': ['Federal-gov', 'Local-gov', 'Private', 'Sel
f-emp-inc', 'Self-emp-not-inc', 'State-gov', 'Without-pay'], 'marriage': ['Di
vorced', 'Married-AF-spouse', 'Married-civ-spouse', 'Married-spouse-absent',
'Never-married', 'Separated', 'Widowed'], 'occupation': ['Adm-clerical', 'Arm
ed-Forces', 'Craft-repair', 'Exec-managerial', 'Farming-fishing', 'Handlers-c
leaners', 'Machine-op-inspct', 'Other-service', 'Priv-house-serv', 'Prof-spec
ialty', 'Protective-serv', 'Sales', 'Tech-support', 'Transport-moving'], 'ed
u': ['10th', '11th', '12th', '1st-4th', '5th-6th', '7th-8th', '9th', 'Assoc-a
cdm', 'Assoc-voc', 'Bachelors', 'Doctorate', 'HS-grad', 'Masters', 'Preschoo
l', 'Prof-school', 'Some-college'], 'relationship': ['Husband', 'Not-in-famil
y', 'Other-relative', 'Own-child', 'Unmarried', 'Wife'], 'sex': ['Female', 'M
ale']}
```

In [49]:
```
# more useful code, used during training, that depends on the function
# you write above

def get_feature(record, feature):
    """
    Return the categorical feature value of a record
    """
    onehot = get_onehot(record, feature)
    return get_categorical_value(onehot, feature)

def get_features(record):
    """
    Return a dictionary of all categorical feature values of a record
    """
    return { f: get_feature(record, f) for f in catcols }
```

Part (g) Train/Test Split [3 pt]

Randomly split the data into approximately 70% training, 15% validation and 15% test.

Report the number of items in your training, validation, and test set.

In [50]:
```
# set the numpy seed for reproducibility
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html
np.random.seed(50)

# todo
indices = np.arange(len(data))
np.random.shuffle(indices)

dataSplit1 = int(len(indices) * 0.70)
dataSplit2 = int(len(indices) * 0.85)

trainingIndices = indices[:dataSplit1]
validationIndices = indices[dataSplit1:dataSplit2]
testingIndices = indices[dataSplit2:]

trainingData = datanp[trainingIndices]
validationData = datanp[validationIndices]
testingData = datanp[testingIndices]

print("The length of training data is:", len(trainingData))
print("The length of validation data is:", len(validationIndices))
print("The length of testing data is:", len(testingData))
```

```
The length of training data is: 21502
The length of validation data is: 4608
The length of testing data is: 4608
```

Part 2. Model Setup [5 pt]

Part (a) [4 pt]

Design a fully-connected autoencoder by modifying the `encoder` and `decoder` below.

The input to this autoencoder will be the features of the `data`, with one categorical feature recorded as "missing". The output of the autoencoder should be the reconstruction of the same features, but with the missing value filled in.

Note: Do not reduce the dimensionality of the input too much! The output of your embedding is expected to contain information about ~11 features.

In [51]:
```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(57, 57), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(57, 30),
            nn.ReLU(),
            nn.Linear(30, 15)
        )
        self.decoder = nn.Sequential(
            nn.Linear(15, 30), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(30, 57),
            nn.ReLU(),
            nn.Linear(57, 57),
            nn.Sigmoid() # get to the range (0, 1)
        )

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

Part (b) [1 pt]

Explain why there is a sigmoid activation in the last step of the decoder.

(**Note**: the values inside the data frame `data` and the training code in Part 3 might be helpful.)

In [ ]:
```
'''
The sigmoid activation in the last step of the decoder forces the output retur
ned to be values
between 0 to 1. This is neccesarry because the data in the columns is normaliz
ed as well.
Therefore, we want the output of the encoder to match this.
'''
```

Part 3. Training [18]

Part (a) [6 pt]

We will train our autoencoder in the following way:

- In each iteration, we will hide one of the categorical features using the `zero_out_random_features` function
- We will pass the data with one missing feature through the autoencoder, and obtain a reconstruction
- We will check how close the reconstruction is compared to the original data -- including the value of the missing feature

Complete the code to train the autoencoder, and plot the training and validation loss every few iterations. You may also want to plot training and validation "accuracy" every few iterations, as we will define in part (b). You may also want to checkpoint your model every few iterations or epochs.

Use `nn.MSELoss()` as your loss function. (Side note: you might recognize that this loss function is not ideal for this problem, but we will use it anyway.)

In [ ]:
```
def zero_out_feature(records, feature):
    """ Set the feature missing in records, by setting the appropriate
    columns of records to 0
    """
    start_index = cat_index[feature]
    stop_index = cat_index[feature] + len(cat_values[feature])
    records[:, start_index:stop_index] = 0
    return records

def zero_out_random_feature(records):
    """ Set one random feature missing in records, by setting the
    appropriate columns of records to 0
    """
    return zero_out_feature(records, random.choice(catcols))

def train(model, train_loader, valid_loader, batch_size = 128, num_epochs=5, l
earning_rate=1e-4):
    """ Training loop. You should update this."""
    torch.manual_seed(42)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    #Store the accuracy in the array
    trainingAccuracy = []
    validationAccuracy = []

    #Store the loss in an array
    trainingLosses = []

    for epoch in range(num_epochs):
        i = 1
        for data in train_loader:
            if torch.cuda.is_available():
                data = data.cuda()
                model.cuda()

            datam = zero_out_random_feature(data.clone()) # zero out one categ
orical feature
            recon = model(datam)
            loss = criterion(recon, data)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            if i%40 == 0:
                trainingAccuracy.append(get_accuracy(model, train_loader))
                validationAccuracy.append(get_accuracy(model, valid_loader))
                trainingLosses.append(float(loss)/batch_size)

                #print("Epoch:", epoch + 1, ", Training Accuracy:", trainingAc
curacy[-1], ", Validation Accuracy:", validationAccuracy[-1])
                print("Epoch:", epoch + 1, ", Iteration:", i, ", Training Accu
racy:", trainingAccuracy[-1], ", Validation Accuracy:", validationAccuracy[-1
])
            i += 1
```

```
    trainingAccuracy.append(get_accuracy(model, train_loader))
    validationAccuracy.append(get_accuracy(model, valid_loader))
    trainingLosses.append(float(loss)/batch_size)
    print("Epoch:", epoch + 1, ", Training Accuracy:", trainingAccuracy[-1],
", Validation Accuracy:", validationAccuracy[-1])

'''
    #Accuracy Plot
    plt.title("Training Curve")
    plt.plot(trainingAccuracy, label='Training')
    plt.plot(validationAccuracy, label='Validation')
    plt.xlabel('Iterations')
    plt.ylabel('Accuracy')
    plt.show()

    #Loss Plot
    plt.title('Training Curve')
    plt.plot(trainingLosses, label='Training')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.show()
'''
```

Part (b) [3 pt]

While plotting training and validation loss is valuable, loss values are harder to compare than accuracy percentages. It would be nice to have a measure of "accuracy" in this problem.

Since we will only be imputing missing categorical values, we will define an accuracy measure. For each record and for each categorical feature, we determine whether the model can predict the categorical feature given all the other features of the record.

A function `get_accuracy` is written for you. It is up to you to figure out how to use the function. **You don't need to submit anything in this part.** To earn the marks, correctly plot the training and validation accuracy every few iterations as part of your training curve.

In [54]:
```
def get_accuracy(model, data_loader):
    """Return the "accuracy" of the autoencoder model across a data set.
    That is, for each record and for each categorical feature,
    we determine whether the model can successfully predict the value
    of the categorical feature given all the other features of the
    record. The returned "accuracy" measure is the percentage of times
    that our model is successful.

    Args:
       - model: the autoencoder model, an instance of nn.Module
       - data_loader: an instance of torch.utils.data.DataLoader

    Example (to illustrate how get_accuracy is intended to be called.
             Depending on your variable naming this code might require
             modification.)

        >>> model = AutoEncoder()
        >>> vdl = torch.utils.data.DataLoader(data_valid, batch_size=256, shuf
fle=True)
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
          #Allow GPUs to be used
          if torch.cuda.is_available():
            item = item.cuda()
            model.cuda()

            inp = item.detach().cuda().cpu().numpy()
            out = model(zero_out_feature(item.clone(), col).cuda()).detach().c
pu().numpy()
            for i in range(out.shape[0]): # record in minibatch
                acc += int(get_feature(out[i], col) == get_feature(inp[i], col
))
                total += 1
    return acc / total
```

Part (c) [4 pt]

Run your updated training code, using reasonable initial hyperparameters.

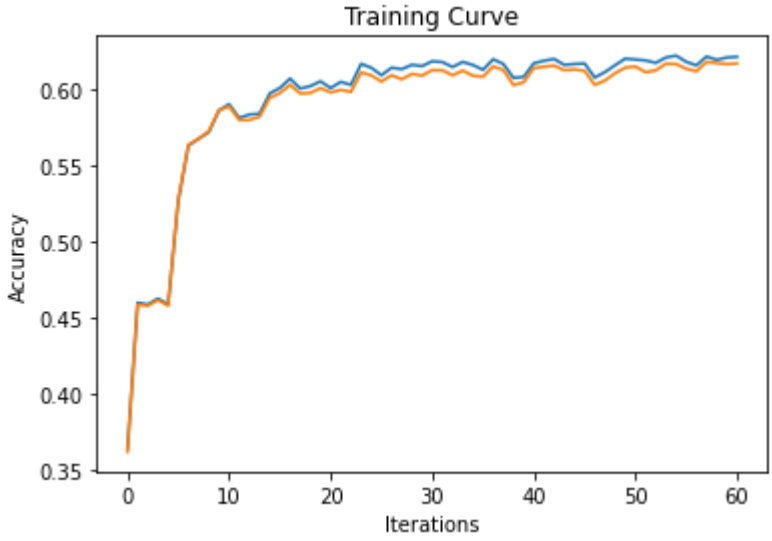
Include your training curve in your submission.

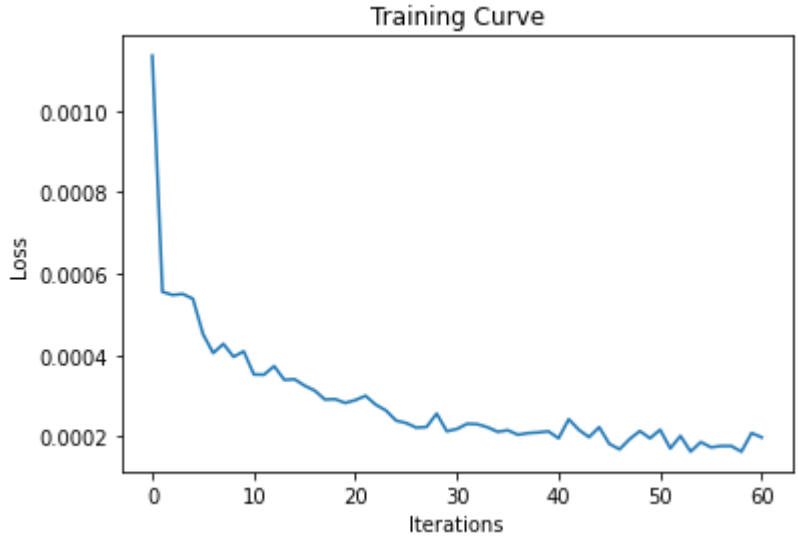
In [57]:
```
autoEncoderModel = AutoEncoder()

use_cuda = True

if use_cuda and torch.cuda.is_available():
  autoEncoderModel.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')

batch_size=128
train_loader = torch.utils.data.DataLoader(trainingData, batch_size=batch_size
, shuffle=True)
valid_loader = torch.utils.data.DataLoader(validationData, batch_size=batch_si
ze, shuffle=True)
train(autoEncoderModel, train_loader, valid_loader, batch_size=batch_size, num
_epochs=15, learning_rate=0.001)
```

```
Using CUDA!
Epoch: 1 , Iteration: 40 , Training Accuracy: 0.3638111183455803 , Validation
Accuracy: 0.3621961805555556
Epoch: 1 , Iteration: 80 , Training Accuracy: 0.4598486962453105 , Validation
Accuracy: 0.4585503472222222
Epoch: 1 , Iteration: 120 , Training Accuracy: 0.458608501534741 , Validation
Accuracy: 0.4577907986111111
Epoch: 1 , Iteration: 160 , Training Accuracy: 0.46256162217468144 , Validati
on Accuracy: 0.4615162037037037
Epoch: 2 , Iteration: 40 , Training Accuracy: 0.458608501534741 , Validation
Accuracy: 0.4578269675925926
Epoch: 2 , Iteration: 80 , Training Accuracy: 0.526594425324776 , Validation
Accuracy: 0.5275607638888888
Epoch: 2 , Iteration: 120 , Training Accuracy: 0.5627461011378786 , Validatio
n Accuracy: 0.5632957175925926
Epoch: 2 , Iteration: 160 , Training Accuracy: 0.5672263045298112 , Validatio
n Accuracy: 0.5671296296296297
Epoch: 3 , Iteration: 40 , Training Accuracy: 0.5714352153288066 , Validation
Accuracy: 0.5721571180555556
Epoch: 3 , Iteration: 80 , Training Accuracy: 0.5858137227544724 , Validation
Accuracy: 0.5857928240740741
Epoch: 3 , Iteration: 120 , Training Accuracy: 0.5898753604315877 , Validatio
n Accuracy: 0.5884693287037037
Epoch: 3 , Iteration: 160 , Training Accuracy: 0.5808374414783121 , Validatio
n Accuracy: 0.5794632523148148
Epoch: 4 , Iteration: 40 , Training Accuracy: 0.5831007968251015 , Validation
Accuracy: 0.5795355902777778
Epoch: 4 , Iteration: 80 , Training Accuracy: 0.5836123771432115 , Validation
Accuracy: 0.5815972222222222
Epoch: 4 , Iteration: 120 , Training Accuracy: 0.596921216631011 , Validation
Accuracy: 0.5940031828703703
Epoch: 4 , Iteration: 160 , Training Accuracy: 0.6004790252069575 , Validatio
n Accuracy: 0.5970775462962963
Epoch: 5 , Iteration: 40 , Training Accuracy: 0.6066257402412178 , Validation
Accuracy: 0.6025390625
Epoch: 5 , Iteration: 80 , Training Accuracy: 0.600153474095433 , Validation
Accuracy: 0.5968967013888888
Epoch: 5 , Iteration: 120 , Training Accuracy: 0.6016959662667039 , Validatio
n Accuracy: 0.5972222222222222
Epoch: 5 , Iteration: 160 , Training Accuracy: 0.6048507115617152 , Validatio
n Accuracy: 0.6003689236111112
Epoch: 6 , Iteration: 40 , Training Accuracy: 0.6004635227730754 , Validation
Accuracy: 0.59765625
Epoch: 6 , Iteration: 80 , Training Accuracy: 0.6044786531485443 , Validation
Accuracy: 0.5992115162037037
Epoch: 6 , Iteration: 120 , Training Accuracy: 0.6025408489132794 , Validatio
n Accuracy: 0.5979817708333334
Epoch: 6 , Iteration: 160 , Training Accuracy: 0.6162915077667194 , Validatio
n Accuracy: 0.6106770833333334
Epoch: 7 , Iteration: 40 , Training Accuracy: 0.6137413573931108 , Validation
Accuracy: 0.6087239583333334
Epoch: 7 , Iteration: 80 , Training Accuracy: 0.6088735931541251 , Validation
Accuracy: 0.6047453703703703
Epoch: 7 , Iteration: 120 , Training Accuracy: 0.613772362260875 , Validation
Accuracy: 0.6088324652777778
Epoch: 7 , Iteration: 160 , Training Accuracy: 0.6128189625771246 , Validatio
n Accuracy: 0.6062282986111112
```

```
Epoch: 8 , Iteration: 40 , Training Accuracy: 0.6156714104114346 , Validation
Accuracy: 0.6096643518518519
Epoch: 8 , Iteration: 80 , Training Accuracy: 0.6148420301987412 , Validation
Accuracy: 0.6086154513888888
Epoch: 8 , Iteration: 120 , Training Accuracy: 0.6180045267106936 , Validatio
n Accuracy: 0.6122323495370371
Epoch: 8 , Iteration: 160 , Training Accuracy: 0.6174541903078783 , Validatio
n Accuracy: 0.6120876736111112
Epoch: 9 , Iteration: 40 , Training Accuracy: 0.6142296840603975 , Validation
Accuracy: 0.6088686342592593
Epoch: 9 , Iteration: 80 , Training Accuracy: 0.6175627073450531 , Validation
Accuracy: 0.6118706597222222
Epoch: 9 , Iteration: 120 , Training Accuracy: 0.6155706445912008 , Validatio
n Accuracy: 0.6085069444444444
Epoch: 9 , Iteration: 160 , Training Accuracy: 0.6124391529470127 , Validatio
n Accuracy: 0.6079282407407407
Epoch: 10 , Iteration: 40 , Training Accuracy: 0.6193067311567916 , Validatio
n Accuracy: 0.6145833333333334
Epoch: 10 , Iteration: 80 , Training Accuracy: 0.6162217468142498 , Validatio
n Accuracy: 0.6123046875
Epoch: 10 , Iteration: 120 , Training Accuracy: 0.6070908132576814 , Validati
on Accuracy: 0.6023943865740741
Epoch: 10 , Iteration: 160 , Training Accuracy: 0.6077419154807304 , Validati
on Accuracy: 0.6042390046296297
Epoch: 11 , Iteration: 40 , Training Accuracy: 0.6166790686137723 , Validatio
n Accuracy: 0.6134982638888888
Epoch: 11 , Iteration: 80 , Training Accuracy: 0.6183223266052771 , Validatio
n Accuracy: 0.6144024884259259
Epoch: 11 , Iteration: 120 , Training Accuracy: 0.6194772579294949 , Validati
on Accuracy: 0.6151620370370371
Epoch: 11 , Iteration: 160 , Training Accuracy: 0.6156249031097882 , Validati
on Accuracy: 0.6121238425925926
Epoch: 12 , Iteration: 40 , Training Accuracy: 0.6161984931634267 , Validatio
n Accuracy: 0.6126663773148148
Epoch: 12 , Iteration: 80 , Training Accuracy: 0.6165240442749512 , Validatio
n Accuracy: 0.6117259837962963
Epoch: 12 , Iteration: 120 , Training Accuracy: 0.6073543546336775 , Validati
on Accuracy: 0.6025390625
Epoch: 12 , Iteration: 160 , Training Accuracy: 0.6108889095588007 , Validati
on Accuracy: 0.6053964120370371
Epoch: 13 , Iteration: 40 , Training Accuracy: 0.6154156202523796 , Validatio
n Accuracy: 0.6101707175925926
Epoch: 13 , Iteration: 80 , Training Accuracy: 0.6197252968716088 , Validatio
n Accuracy: 0.6138237847222222
Epoch: 13 , Iteration: 120 , Training Accuracy: 0.6191284531671473 , Validati
on Accuracy: 0.6145471643518519
Epoch: 13 , Iteration: 160 , Training Accuracy: 0.6185316094626856 , Validati
on Accuracy: 0.6108217592592593
Epoch: 14 , Iteration: 40 , Training Accuracy: 0.6169658636405916 , Validatio
n Accuracy: 0.6120515046296297
Epoch: 14 , Iteration: 80 , Training Accuracy: 0.6204229063963043 , Validatio
n Accuracy: 0.6163917824074074
Epoch: 14 , Iteration: 120 , Training Accuracy: 0.6217018571915791 , Validati
on Accuracy: 0.6161747685185185
Epoch: 14 , Iteration: 160 , Training Accuracy: 0.6176402195144638 , Validati
on Accuracy: 0.6131365740740741
Epoch: 15 , Iteration: 40 , Training Accuracy: 0.615330356866028 , Validation
```

```
Accuracy: 0.6114366319444444
Epoch: 15 , Iteration: 80 , Training Accuracy: 0.6211127647040585 , Validatio
n Accuracy: 0.6175491898148148
Epoch: 15 , Iteration: 120 , Training Accuracy: 0.6189269215266797 , Validati
on Accuracy: 0.6168258101851852
Epoch: 15 , Iteration: 160 , Training Accuracy: 0.6204461600471274 , Validati
on Accuracy: 0.6161747685185185
Epoch: 15 , Training Accuracy: 0.6208957306297088 , Validation Accuracy: 0.61
65364583333334
```





Part (d) [5 pt]

Tune your hyperparameters, training at least 4 different models (4 sets of hyperparameters).

Do not include all your training curves. Instead, explain what hyperparameters you tried, what their effect was, and what your thought process was as you chose the next set of hyperparameters to try.

In [59]:
```
#In my first iteration of the model, I will be lowering the batch_size to 64,
 and decreasing
#the number of epochs
#My thinking behind this is that smaller batches can go through quicker with l
ess variablility
autoEncoderModel1 = AutoEncoder()

use_cuda = True

if use_cuda and torch.cuda.is_available():
  autoEncoderModel1.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')

batch_size=64
train_loader = torch.utils.data.DataLoader(trainingData, batch_size=batch_size
, shuffle=True)
valid_loader = torch.utils.data.DataLoader(validationData, batch_size=batch_si
ze, shuffle=True)
train(autoEncoderModel1, train_loader, valid_loader, batch_size=batch_size, nu
m_epochs=10, learning_rate=0.001)
```

```
Using CUDA!
Epoch: 1 , Iteration: 40 , Training Accuracy: 0.3778098161411342 , Validation
Accuracy: 0.37666377314814814
Epoch: 1 , Iteration: 80 , Training Accuracy: 0.45512045391126404 , Validatio
n Accuracy: 0.4541015625
Epoch: 1 , Iteration: 120 , Training Accuracy: 0.4587712770905032 , Validatio
n Accuracy: 0.4568504050925926
Epoch: 1 , Iteration: 160 , Training Accuracy: 0.4598486962453105 , Validatio
n Accuracy: 0.4585503472222222
Epoch: 1 , Iteration: 200 , Training Accuracy: 0.45907357455120457 , Validati
on Accuracy: 0.4578993055555556
Epoch: 1 , Iteration: 240 , Training Accuracy: 0.49852726878119863 , Validati
on Accuracy: 0.49692563657407407
Epoch: 1 , Iteration: 280 , Training Accuracy: 0.5491737202740831 , Validatio
n Accuracy: 0.5506727430555556
Epoch: 1 , Iteration: 320 , Training Accuracy: 0.5559327814466871 , Validatio
n Accuracy: 0.5550491898148148
Epoch: 2 , Iteration: 40 , Training Accuracy: 0.5636529935199827 , Validation
Accuracy: 0.5644169560185185
Epoch: 2 , Iteration: 80 , Training Accuracy: 0.5676758751123926 , Validation
Accuracy: 0.5679615162037037
Epoch: 2 , Iteration: 120 , Training Accuracy: 0.5689160698229622 , Validatio
n Accuracy: 0.5675998263888888
Epoch: 2 , Iteration: 160 , Training Accuracy: 0.5669550119368741 , Validatio
n Accuracy: 0.5668402777777778
Epoch: 2 , Iteration: 200 , Training Accuracy: 0.5727916782934921 , Validatio
n Accuracy: 0.5726996527777778
Epoch: 2 , Iteration: 240 , Training Accuracy: 0.5798220320590333 , Validatio
n Accuracy: 0.5805844907407407
Epoch: 2 , Iteration: 280 , Training Accuracy: 0.5833953430688618 , Validatio
n Accuracy: 0.5808015046296297
Epoch: 2 , Iteration: 320 , Training Accuracy: 0.5785430812637584 , Validatio
n Accuracy: 0.5758101851851852
Epoch: 3 , Iteration: 40 , Training Accuracy: 0.5809304560816048 , Validation
Accuracy: 0.5783781828703703
Epoch: 3 , Iteration: 80 , Training Accuracy: 0.5838604160853254 , Validation
Accuracy: 0.5807291666666666
Epoch: 3 , Iteration: 120 , Training Accuracy: 0.5933711592720057 , Validatio
n Accuracy: 0.5902054398148148
Epoch: 3 , Iteration: 160 , Training Accuracy: 0.5936114469971786 , Validatio
n Accuracy: 0.5882523148148148
Epoch: 3 , Iteration: 200 , Training Accuracy: 0.6054553064831178 , Validatio
n Accuracy: 0.6018518518518519
Epoch: 3 , Iteration: 240 , Training Accuracy: 0.6029594146280967 , Validatio
n Accuracy: 0.5988498263888888
Epoch: 3 , Iteration: 280 , Training Accuracy: 0.6055173162186464 , Validatio
n Accuracy: 0.6005497685185185
Epoch: 3 , Iteration: 320 , Training Accuracy: 0.5995876352587356 , Validatio
n Accuracy: 0.5963541666666666
Epoch: 4 , Iteration: 40 , Training Accuracy: 0.6084162713546026 , Validation
Accuracy: 0.6031901041666666
Epoch: 4 , Iteration: 80 , Training Accuracy: 0.5983629429820482 , Validation
Accuracy: 0.5932436342592593
Epoch: 4 , Iteration: 120 , Training Accuracy: 0.6049204725141847 , Validatio
n Accuracy: 0.6012369791666666
Epoch: 4 , Iteration: 160 , Training Accuracy: 0.5954407341952687 , Validatio
n Accuracy: 0.5949797453703703
```

```
Epoch: 4 , Iteration: 200 , Training Accuracy: 0.5886196632871361 , Validatio
n Accuracy: 0.5859375
Epoch: 4 , Iteration: 240 , Training Accuracy: 0.6052537748426503 , Validatio
n Accuracy: 0.6017433449074074
Epoch: 4 , Iteration: 280 , Training Accuracy: 0.6000527082751992 , Validatio
n Accuracy: 0.5960648148148148
Epoch: 4 , Iteration: 320 , Training Accuracy: 0.6043468824605464 , Validatio
n Accuracy: 0.6007667824074074
Epoch: 5 , Iteration: 40 , Training Accuracy: 0.6060831550553437 , Validation
Accuracy: 0.6036964699074074
Epoch: 5 , Iteration: 80 , Training Accuracy: 0.6055948283880569 , Validation
Accuracy: 0.6018156828703703
Epoch: 5 , Iteration: 120 , Training Accuracy: 0.5986264843580442 , Validatio
n Accuracy: 0.5951244212962963
Epoch: 5 , Iteration: 160 , Training Accuracy: 0.6020447710290515 , Validatio
n Accuracy: 0.5987051504629629
Epoch: 5 , Iteration: 200 , Training Accuracy: 0.6022540538864601 , Validatio
n Accuracy: 0.6000072337962963
Epoch: 5 , Iteration: 240 , Training Accuracy: 0.600812327535423 , Validation
Accuracy: 0.5971498842592593
Epoch: 5 , Iteration: 280 , Training Accuracy: 0.6059668868012278 , Validatio
n Accuracy: 0.6022858796296297
Epoch: 5 , Iteration: 320 , Training Accuracy: 0.6072613400303848 , Validatio
n Accuracy: 0.6036241319444444
Epoch: 6 , Iteration: 40 , Training Accuracy: 0.6036027656342046 , Validation
Accuracy: 0.5987774884259259
Epoch: 6 , Iteration: 80 , Training Accuracy: 0.6003007472173131 , Validation
Accuracy: 0.5960286458333334
Epoch: 6 , Iteration: 120 , Training Accuracy: 0.6071683254270921 , Validatio
n Accuracy: 0.6019965277777778
Epoch: 6 , Iteration: 160 , Training Accuracy: 0.6009596006573033 , Validatio
n Accuracy: 0.5964265046296297
Epoch: 6 , Iteration: 200 , Training Accuracy: 0.6057110966421728 , Validatio
n Accuracy: 0.6031901041666666
Epoch: 6 , Iteration: 240 , Training Accuracy: 0.6088270858524788 , Validatio
n Accuracy: 0.6057219328703703
Epoch: 6 , Iteration: 280 , Training Accuracy: 0.6068582767494497 , Validatio
n Accuracy: 0.6019965277777778
Epoch: 6 , Iteration: 320 , Training Accuracy: 0.6010991225622423 , Validatio
n Accuracy: 0.5936414930555556
Epoch: 7 , Iteration: 40 , Training Accuracy: 0.6050134871174775 , Validation
Accuracy: 0.5995370370370371
Epoch: 7 , Iteration: 80 , Training Accuracy: 0.609485939292469 , Validation
Accuracy: 0.6057581018518519
Epoch: 7 , Iteration: 120 , Training Accuracy: 0.6034632437292655 , Validatio
n Accuracy: 0.5987774884259259
Epoch: 7 , Iteration: 160 , Training Accuracy: 0.6107571388708027 , Validatio
n Accuracy: 0.6069155092592593
Epoch: 7 , Iteration: 200 , Training Accuracy: 0.6049204725141847 , Validatio
n Accuracy: 0.5994285300925926
Epoch: 7 , Iteration: 240 , Training Accuracy: 0.6070675596068583 , Validatio
n Accuracy: 0.6031539351851852
Epoch: 7 , Iteration: 280 , Training Accuracy: 0.6046414287043066 , Validatio
n Accuracy: 0.6002965856481481
Epoch: 7 , Iteration: 320 , Training Accuracy: 0.6017579760022324 , Validatio
n Accuracy: 0.5978370949074074
Epoch: 8 , Iteration: 40 , Training Accuracy: 0.6054165503984126 , Validation
```

```
Accuracy: 0.6017795138888888
Epoch: 8 , Iteration: 80 , Training Accuracy: 0.6055328186525284 , Validation
Accuracy: 0.6006582754629629
Epoch: 8 , Iteration: 120 , Training Accuracy: 0.607160574210151 , Validation
Accuracy: 0.6006221064814815
Epoch: 8 , Iteration: 160 , Training Accuracy: 0.608710817598363 , Validation
Accuracy: 0.6044560185185185
Epoch: 8 , Iteration: 200 , Training Accuracy: 0.6128344650110067 , Validatio
n Accuracy: 0.6077835648148148
Epoch: 8 , Iteration: 240 , Training Accuracy: 0.6057188478591139 , Validatio
n Accuracy: 0.5993200231481481
Epoch: 8 , Iteration: 280 , Training Accuracy: 0.6057033454252317 , Validatio
n Accuracy: 0.5990306712962963
Epoch: 8 , Iteration: 320 , Training Accuracy: 0.6072690912473259 , Validatio
n Accuracy: 0.6022858796296297
Epoch: 9 , Iteration: 40 , Training Accuracy: 0.6066102378073358 , Validation
Accuracy: 0.6026114004629629
Epoch: 9 , Iteration: 80 , Training Accuracy: 0.6037422875391436 , Validation
Accuracy: 0.5992476851851852
Epoch: 9 , Iteration: 120 , Training Accuracy: 0.6110129290298577 , Validatio
n Accuracy: 0.6056495949074074
Epoch: 9 , Iteration: 160 , Training Accuracy: 0.6014634297584721 , Validatio
n Accuracy: 0.5977647569444444
Epoch: 9 , Iteration: 200 , Training Accuracy: 0.6041608532539609 , Validatio
n Accuracy: 0.5984157986111112
Epoch: 9 , Iteration: 240 , Training Accuracy: 0.6155706445912008 , Validatio
n Accuracy: 0.6102068865740741
Epoch: 9 , Iteration: 280 , Training Accuracy: 0.609485939292469 , Validation
Accuracy: 0.6046730324074074
Epoch: 9 , Iteration: 320 , Training Accuracy: 0.6096022075465848 , Validatio
n Accuracy: 0.6048900462962963
Epoch: 10 , Iteration: 40 , Training Accuracy: 0.6148962887173286 , Validatio
n Accuracy: 0.6095920138888888
Epoch: 10 , Iteration: 80 , Training Accuracy: 0.6159116981366074 , Validatio
n Accuracy: 0.6115451388888888
Epoch: 10 , Iteration: 120 , Training Accuracy: 0.6131057576039438 , Validati
on Accuracy: 0.6087962962962963
Epoch: 10 , Iteration: 160 , Training Accuracy: 0.6160279663907233 , Validati
on Accuracy: 0.6123046875
Epoch: 10 , Iteration: 200 , Training Accuracy: 0.6080364617244908 , Validati
on Accuracy: 0.6021773726851852
Epoch: 10 , Iteration: 240 , Training Accuracy: 0.6104238365423371 , Validati
on Accuracy: 0.6046006944444444
Epoch: 10 , Iteration: 280 , Training Accuracy: 0.6193997457600844 , Validati
on Accuracy: 0.6144386574074074
Epoch: 10 , Iteration: 320 , Training Accuracy: 0.6161597370787214 , Validati
on Accuracy: 0.6107494212962963
Epoch: 10 , Training Accuracy: 0.6114004898769106 , Validation Accuracy: 0.60
30454282407407
```

As we can see above, the training accuracy and the validation accuracy slightly decreased. I will increase the number of epochs a little to allow longer training. I also want to increase the learning rate and see whether this has any benefits.

In [60]:
```
#In my second iteration of the model, I will be increasing the learning rate t
o 0.005, and increasing the
#number of epochs to 12
autoEncoderModel2 = AutoEncoder()

use_cuda = True

if use_cuda and torch.cuda.is_available():
  autoEncoderModel2.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')

batch_size=64
train_loader = torch.utils.data.DataLoader(trainingData, batch_size=batch_size
, shuffle=True)
valid_loader = torch.utils.data.DataLoader(validationData, batch_size=batch_si
ze, shuffle=True)
train(autoEncoderModel2, train_loader, valid_loader, batch_size=batch_size, nu
m_epochs=12, learning_rate=0.005)
```

```
Using CUDA!
Epoch: 1 , Iteration: 40 , Training Accuracy: 0.4417651071218181 , Validation
Accuracy: 0.4409722222222222
Epoch: 1 , Iteration: 80 , Training Accuracy: 0.493775772796329 , Validation
Accuracy: 0.49171730324074076
Epoch: 1 , Iteration: 120 , Training Accuracy: 0.5546693330852944 , Validatio
n Accuracy: 0.5542173032407407
Epoch: 1 , Iteration: 160 , Training Accuracy: 0.5569171859982017 , Validatio
n Accuracy: 0.5591724537037037
Epoch: 1 , Iteration: 200 , Training Accuracy: 0.5847595572504883 , Validatio
n Accuracy: 0.5831886574074074
Epoch: 1 , Iteration: 240 , Training Accuracy: 0.5979908845688773 , Validatio
n Accuracy: 0.5920500578703703
Epoch: 1 , Iteration: 280 , Training Accuracy: 0.5959833193811428 , Validatio
n Accuracy: 0.5927372685185185
Epoch: 1 , Iteration: 320 , Training Accuracy: 0.6052692772765325 , Validatio
n Accuracy: 0.6034794560185185
Epoch: 2 , Iteration: 40 , Training Accuracy: 0.6005952934610734 , Validation
Accuracy: 0.5974754050925926
Epoch: 2 , Iteration: 80 , Training Accuracy: 0.6056723405574675 , Validation
Accuracy: 0.5984519675925926
Epoch: 2 , Iteration: 120 , Training Accuracy: 0.5999674448888476 , Validatio
n Accuracy: 0.5962094907407407
Epoch: 2 , Iteration: 160 , Training Accuracy: 0.6068815304002728 , Validatio
n Accuracy: 0.6011284722222222
Epoch: 2 , Iteration: 200 , Training Accuracy: 0.6075016277555576 , Validatio
n Accuracy: 0.6034794560185185
Epoch: 2 , Iteration: 240 , Training Accuracy: 0.6008200787523641 , Validatio
n Accuracy: 0.5947265625
Epoch: 2 , Iteration: 280 , Training Accuracy: 0.6014324248907078 , Validatio
n Accuracy: 0.5972222222222222
Epoch: 2 , Iteration: 320 , Training Accuracy: 0.6117647970731405 , Validatio
n Accuracy: 0.6072048611111112
Epoch: 3 , Iteration: 40 , Training Accuracy: 0.6141986791926333 , Validation
Accuracy: 0.6103153935185185
Epoch: 3 , Iteration: 80 , Training Accuracy: 0.6077651691315537 , Validation
Accuracy: 0.6036603009259259
Epoch: 3 , Iteration: 120 , Training Accuracy: 0.6118035531578457 , Validatio
n Accuracy: 0.6069155092592593
Epoch: 3 , Iteration: 160 , Training Accuracy: 0.6129274796142995 , Validatio
n Accuracy: 0.6077112268518519
Epoch: 3 , Iteration: 200 , Training Accuracy: 0.6031764487024462 , Validatio
n Accuracy: 0.5955222800925926
Epoch: 3 , Iteration: 240 , Training Accuracy: 0.6055948283880569 , Validatio
n Accuracy: 0.5997178819444444
Epoch: 3 , Iteration: 280 , Training Accuracy: 0.6176557219483458 , Validatio
n Accuracy: 0.6134620949074074
Epoch: 3 , Iteration: 320 , Training Accuracy: 0.6114547483954981 , Validatio
n Accuracy: 0.6050708912037037
Epoch: 4 , Iteration: 40 , Training Accuracy: 0.6117337922053763 , Validation
Accuracy: 0.6069878472222222
Epoch: 4 , Iteration: 80 , Training Accuracy: 0.6141444206740458 , Validation
Accuracy: 0.6096281828703703
Epoch: 4 , Iteration: 120 , Training Accuracy: 0.6043778873283105 , Validatio
n Accuracy: 0.5985604745370371
Epoch: 4 , Iteration: 160 , Training Accuracy: 0.6006262983288376 , Validatio
n Accuracy: 0.5955584490740741
```

```
Epoch: 4 , Iteration: 200 , Training Accuracy: 0.60327721452268 , Validation
Accuracy: 0.5947265625
Epoch: 4 , Iteration: 240 , Training Accuracy: 0.617810746287167 , Validation
Accuracy: 0.6129918981481481
Epoch: 4 , Iteration: 280 , Training Accuracy: 0.6121213530524292 , Validatio
n Accuracy: 0.6074580439814815
Epoch: 4 , Iteration: 320 , Training Accuracy: 0.6204461600471274 , Validatio
n Accuracy: 0.6144024884259259
Epoch: 5 , Iteration: 40 , Training Accuracy: 0.6254224413232877 , Validation
Accuracy: 0.6196469907407407
Epoch: 5 , Iteration: 80 , Training Accuracy: 0.6080519641583728 , Validation
Accuracy: 0.6055049189814815
Epoch: 5 , Iteration: 120 , Training Accuracy: 0.6116485288190245 , Validatio
n Accuracy: 0.6067708333333334
Epoch: 5 , Iteration: 160 , Training Accuracy: 0.6160512200415466 , Validatio
n Accuracy: 0.6084346064814815
Epoch: 5 , Iteration: 200 , Training Accuracy: 0.609369671038353 , Validation
Accuracy: 0.6019965277777778
Epoch: 5 , Iteration: 240 , Training Accuracy: 0.627274982172201 , Validation
Accuracy: 0.6222873263888888
Epoch: 5 , Iteration: 280 , Training Accuracy: 0.6223684618485102 , Validatio
n Accuracy: 0.6159215856481481
Epoch: 5 , Iteration: 320 , Training Accuracy: 0.6096409636312902 , Validatio
n Accuracy: 0.6046730324074074
Epoch: 6 , Iteration: 40 , Training Accuracy: 0.6246085635444765 , Validation
Accuracy: 0.6199725115740741
Epoch: 6 , Iteration: 80 , Training Accuracy: 0.6093774222552941 , Validation
Accuracy: 0.6010561342592593
Epoch: 6 , Iteration: 120 , Training Accuracy: 0.6178805072396366 , Validatio
n Accuracy: 0.6123046875
Epoch: 6 , Iteration: 160 , Training Accuracy: 0.624151241744954 , Validation
Accuracy: 0.6197554976851852
Epoch: 6 , Iteration: 200 , Training Accuracy: 0.6156869128453167 , Validatio
n Accuracy: 0.6114004629629629
Epoch: 6 , Iteration: 240 , Training Accuracy: 0.623763680897901 , Validation
Accuracy: 0.6217809606481481
Epoch: 6 , Iteration: 280 , Training Accuracy: 0.6191362043840883 , Validatio
n Accuracy: 0.6159577546296297
Epoch: 6 , Iteration: 320 , Training Accuracy: 0.6165240442749512 , Validatio
n Accuracy: 0.6113642939814815
Epoch: 7 , Iteration: 40 , Training Accuracy: 0.6173301708368214 , Validation
Accuracy: 0.6120876736111112
Epoch: 7 , Iteration: 80 , Training Accuracy: 0.6147722692462717 , Validation
Accuracy: 0.6119791666666666
Epoch: 7 , Iteration: 120 , Training Accuracy: 0.6287322109571203 , Validatio
n Accuracy: 0.6243851273148148
Epoch: 7 , Iteration: 160 , Training Accuracy: 0.6253526803708183 , Validatio
n Accuracy: 0.6204788773148148
Epoch: 7 , Iteration: 200 , Training Accuracy: 0.6264145970917434 , Validatio
n Accuracy: 0.6223234953703703
Epoch: 7 , Iteration: 240 , Training Accuracy: 0.6130670015192385 , Validatio
n Accuracy: 0.6073857060185185
Epoch: 7 , Iteration: 280 , Training Accuracy: 0.6174464390909373 , Validatio
n Accuracy: 0.6125578703703703
Epoch: 7 , Iteration: 320 , Training Accuracy: 0.626104548414101 , Validation
Accuracy: 0.6207320601851852
Epoch: 8 , Iteration: 40 , Training Accuracy: 0.6217173596254612 , Validation
```

```
Accuracy: 0.6159577546296297
Epoch: 8 , Iteration: 80 , Training Accuracy: 0.6207639599417109 , Validation
Accuracy: 0.6157769097222222
Epoch: 8 , Iteration: 120 , Training Accuracy: 0.6301351812234521 , Validatio
n Accuracy: 0.6238787615740741
Epoch: 8 , Iteration: 160 , Training Accuracy: 0.636088115834186 , Validation
Accuracy: 0.6302445023148148
Epoch: 8 , Iteration: 200 , Training Accuracy: 0.6255387095774037 , Validatio
n Accuracy: 0.6195023148148148
Epoch: 8 , Iteration: 240 , Training Accuracy: 0.6261278020649242 , Validatio
n Accuracy: 0.6186342592592593
Epoch: 8 , Iteration: 280 , Training Accuracy: 0.6260735435463368 , Validatio
n Accuracy: 0.6230107060185185
Epoch: 8 , Iteration: 320 , Training Accuracy: 0.6164000248038942 , Validatio
n Accuracy: 0.6095558449074074
Epoch: 9 , Iteration: 40 , Training Accuracy: 0.631669922177782 , Validation
Accuracy: 0.6284360532407407
Epoch: 9 , Iteration: 80 , Training Accuracy: 0.6308405419650885 , Validation
Accuracy: 0.6268446180555556
Epoch: 9 , Iteration: 120 , Training Accuracy: 0.6200818528508976 , Validatio
n Accuracy: 0.619140625
Epoch: 9 , Iteration: 160 , Training Accuracy: 0.6287399621740614 , Validatio
n Accuracy: 0.6217086226851852
Epoch: 9 , Iteration: 200 , Training Accuracy: 0.6232753542306142 , Validatio
n Accuracy: 0.6178023726851852
Epoch: 9 , Iteration: 240 , Training Accuracy: 0.635584286733017 , Validation
Accuracy: 0.6297381365740741
Epoch: 9 , Iteration: 280 , Training Accuracy: 0.6285539329674759 , Validatio
n Accuracy: 0.6207682291666666
Epoch: 9 , Iteration: 320 , Training Accuracy: 0.6308095370973242 , Validatio
n Accuracy: 0.6240957754629629
Epoch: 10 , Iteration: 40 , Training Accuracy: 0.6289182401637057 , Validatio
n Accuracy: 0.6252170138888888
Epoch: 10 , Iteration: 80 , Training Accuracy: 0.6169038539050631 , Validatio
n Accuracy: 0.6125217013888888
Epoch: 10 , Iteration: 120 , Training Accuracy: 0.63363098006387 , Validation
Accuracy: 0.6256872106481481
Epoch: 10 , Iteration: 160 , Training Accuracy: 0.6317861904318978 , Validati
on Accuracy: 0.6239149305555556
Epoch: 10 , Iteration: 200 , Training Accuracy: 0.6309568102192045 , Validati
on Accuracy: 0.6270254629629629
Epoch: 10 , Iteration: 240 , Training Accuracy: 0.6348789259913806 , Validati
on Accuracy: 0.6287615740740741
Epoch: 10 , Iteration: 280 , Training Accuracy: 0.6270967041825567 , Validati
on Accuracy: 0.6215277777777778
Epoch: 10 , Iteration: 320 , Training Accuracy: 0.6329178681052925 , Validati
on Accuracy: 0.6277126736111112
Epoch: 11 , Iteration: 40 , Training Accuracy: 0.6306932688432084 , Validatio
n Accuracy: 0.6259765625
Epoch: 11 , Iteration: 80 , Training Accuracy: 0.6291197718041732 , Validatio
n Accuracy: 0.6222873263888888
Epoch: 11 , Iteration: 120 , Training Accuracy: 0.6242132514804825 , Validati
on Accuracy: 0.6165002893518519
Epoch: 11 , Iteration: 160 , Training Accuracy: 0.636739218057235 , Validatio
n Accuracy: 0.6300274884259259
Epoch: 11 , Iteration: 200 , Training Accuracy: 0.631685424611664 , Validatio
n Accuracy: 0.6259765625
```

```
Epoch: 11 , Iteration: 240 , Training Accuracy: 0.6368554863113509 , Validati
on Accuracy: 0.6289785879629629
Epoch: 11 , Iteration: 280 , Training Accuracy: 0.6460949369050941 , Validati
on Accuracy: 0.6395760995370371
Epoch: 11 , Iteration: 320 , Training Accuracy: 0.6385064955197967 , Validati
on Accuracy: 0.6305338541666666
Epoch: 12 , Iteration: 40 , Training Accuracy: 0.6359563451461879 , Validatio
n Accuracy: 0.6281828703703703
Epoch: 12 , Iteration: 80 , Training Accuracy: 0.6336542337146932 , Validatio
n Accuracy: 0.6265190972222222
Epoch: 12 , Iteration: 120 , Training Accuracy: 0.6294608253495799 , Validati
on Accuracy: 0.6268807870370371
Epoch: 12 , Iteration: 160 , Training Accuracy: 0.6427696648373795 , Validati
on Accuracy: 0.6379123263888888
Epoch: 12 , Iteration: 200 , Training Accuracy: 0.6307087712770905 , Validati
on Accuracy: 0.6253978587962963
Epoch: 12 , Iteration: 240 , Training Accuracy: 0.6357315598548973 , Validati
on Accuracy: 0.6314019097222222
Epoch: 12 , Iteration: 280 , Training Accuracy: 0.6293755619632282 , Validati
on Accuracy: 0.6218171296296297
Epoch: 12 , Iteration: 320 , Training Accuracy: 0.6195237652311413 , Validati
on Accuracy: 0.6123408564814815
Epoch: 12 , Training Accuracy: 0.6353517502247853 , Validation Accuracy: 0.62
99913194444444
```

As you can see above, both the training accuracy and the validation increased due to the increase in the learning rate. Therefore, I will be keeping all the other hyperparameters constant, and I will be increasing the learning eate even more. I expect the accuracy to increase even more, or fall again.

In [61]:
```
#In my third iteration of the model, I will be increasing the learning rate to
0.01, and will be keeping
#other hyperparameters the same
autoEncoderModel3 = AutoEncoder()

use_cuda = True

if use_cuda and torch.cuda.is_available():
  autoEncoderModel3.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')

batch_size=64
train_loader = torch.utils.data.DataLoader(trainingData, batch_size=batch_size
, shuffle=True)
valid_loader = torch.utils.data.DataLoader(validationData, batch_size=batch_si
ze, shuffle=True)
train(autoEncoderModel3, train_loader, valid_loader, batch_size=batch_size, nu
m_epochs=12, learning_rate=0.01)
```

```
Using CUDA!
Epoch: 1 , Iteration: 40 , Training Accuracy: 0.4598486962453105 , Validation
Accuracy: 0.4585503472222222
Epoch: 1 , Iteration: 80 , Training Accuracy: 0.5207345053173348 , Validation
Accuracy: 0.5223524305555556
Epoch: 1 , Iteration: 120 , Training Accuracy: 0.5619089697082442 , Validatio
n Accuracy: 0.5625723379629629
Epoch: 1 , Iteration: 160 , Training Accuracy: 0.560459492140266 , Validation
Accuracy: 0.5614872685185185
Epoch: 1 , Iteration: 200 , Training Accuracy: 0.5799925588317366 , Validatio
n Accuracy: 0.5777994791666666
Epoch: 1 , Iteration: 240 , Training Accuracy: 0.6008278299693052 , Validatio
n Accuracy: 0.6000072337962963
Epoch: 1 , Iteration: 280 , Training Accuracy: 0.5884646389483149 , Validatio
n Accuracy: 0.5831886574074074
Epoch: 1 , Iteration: 320 , Training Accuracy: 0.6013704151551794 , Validatio
n Accuracy: 0.5988859953703703
Epoch: 2 , Iteration: 40 , Training Accuracy: 0.5887669364090162 , Validation
Accuracy: 0.5828993055555556
Epoch: 2 , Iteration: 80 , Training Accuracy: 0.6026726196012774 , Validation
Accuracy: 0.59765625
Epoch: 2 , Iteration: 120 , Training Accuracy: 0.6064862183362788 , Validatio
n Accuracy: 0.6023220486111112
Epoch: 2 , Iteration: 160 , Training Accuracy: 0.6030834340991535 , Validatio
n Accuracy: 0.5976200810185185
Epoch: 2 , Iteration: 200 , Training Accuracy: 0.6118035531578457 , Validatio
n Accuracy: 0.609375
Epoch: 2 , Iteration: 240 , Training Accuracy: 0.6104625926270425 , Validatio
n Accuracy: 0.6051793981481481
Epoch: 2 , Iteration: 280 , Training Accuracy: 0.6039050630949059 , Validatio
n Accuracy: 0.6007306134259259
Epoch: 2 , Iteration: 320 , Training Accuracy: 0.6064087061668681 , Validatio
n Accuracy: 0.6004774305555556
Epoch: 3 , Iteration: 40 , Training Accuracy: 0.5980528943044058 , Validation
Accuracy: 0.5938223379629629
Epoch: 3 , Iteration: 80 , Training Accuracy: 0.6016107028803522 , Validation
Accuracy: 0.5972583912037037
Epoch: 3 , Iteration: 120 , Training Accuracy: 0.606773013363098 , Validation
Accuracy: 0.6009114583333334
Epoch: 3 , Iteration: 160 , Training Accuracy: 0.6008743372709515 , Validatio
n Accuracy: 0.5974030671296297
Epoch: 3 , Iteration: 200 , Training Accuracy: 0.6063699500821629 , Validatio
n Accuracy: 0.6019241898148148
Epoch: 3 , Iteration: 240 , Training Accuracy: 0.6035097510309119 , Validatio
n Accuracy: 0.6017795138888888
Epoch: 3 , Iteration: 280 , Training Accuracy: 0.6044011409791338 , Validatio
n Accuracy: 0.6006582754629629
Epoch: 3 , Iteration: 320 , Training Accuracy: 0.616376771153071 , Validation
Accuracy: 0.6154513888888888
Epoch: 4 , Iteration: 40 , Training Accuracy: 0.6008588348370694 , Validation
Accuracy: 0.5982349537037037
Epoch: 4 , Iteration: 80 , Training Accuracy: 0.6037500387560847 , Validation
Accuracy: 0.6003327546296297
Epoch: 4 , Iteration: 120 , Training Accuracy: 0.6050367407683006 , Validatio
n Accuracy: 0.6017071759259259
Epoch: 4 , Iteration: 160 , Training Accuracy: 0.5991923231947416 , Validatio
n Accuracy: 0.5949435763888888
```

```
Epoch: 4 , Iteration: 200 , Training Accuracy: 0.607021052305212 , Validation
Accuracy: 0.6053240740740741
Epoch: 4 , Iteration: 240 , Training Accuracy: 0.602486590394692 , Validation
Accuracy: 0.5993923611111112
Epoch: 4 , Iteration: 280 , Training Accuracy: 0.6143071962298081 , Validatio
n Accuracy: 0.6084707754629629
Epoch: 4 , Iteration: 320 , Training Accuracy: 0.6034089852106781 , Validatio
n Accuracy: 0.6009114583333334
Epoch: 5 , Iteration: 40 , Training Accuracy: 0.6094084271230583 , Validation
Accuracy: 0.6047815393518519
Epoch: 5 , Iteration: 80 , Training Accuracy: 0.6079977056397855 , Validation
Accuracy: 0.6027199074074074
Epoch: 5 , Iteration: 120 , Training Accuracy: 0.6027113756859827 , Validatio
n Accuracy: 0.6004774305555556
Epoch: 5 , Iteration: 160 , Training Accuracy: 0.6140126499860478 , Validatio
n Accuracy: 0.6116536458333334
Epoch: 5 , Iteration: 200 , Training Accuracy: 0.6047964530431278 , Validatio
n Accuracy: 0.6000795717592593
Epoch: 5 , Iteration: 240 , Training Accuracy: 0.5970762409698323 , Validatio
n Accuracy: 0.5962094907407407
Epoch: 5 , Iteration: 280 , Training Accuracy: 0.6098579977056398 , Validatio
n Accuracy: 0.6065899884259259
Epoch: 5 , Iteration: 320 , Training Accuracy: 0.6116717824698478 , Validatio
n Accuracy: 0.6079282407407407
Epoch: 6 , Iteration: 40 , Training Accuracy: 0.6190586922146777 , Validation
Accuracy: 0.6143301504629629
Epoch: 6 , Iteration: 80 , Training Accuracy: 0.620655442904536 , Validation
Accuracy: 0.6149450231481481
Epoch: 6 , Iteration: 120 , Training Accuracy: 0.6168650978203578 , Validatio
n Accuracy: 0.6124131944444444
Epoch: 6 , Iteration: 160 , Training Accuracy: 0.615462127554026 , Validation
Accuracy: 0.6110749421296297
Epoch: 6 , Iteration: 200 , Training Accuracy: 0.6090596223607107 , Validatio
n Accuracy: 0.6058304398148148
Epoch: 6 , Iteration: 240 , Training Accuracy: 0.6166790686137723 , Validatio
n Accuracy: 0.6129918981481481
Epoch: 6 , Iteration: 280 , Training Accuracy: 0.6104935974948067 , Validatio
n Accuracy: 0.6066261574074074
Epoch: 6 , Iteration: 320 , Training Accuracy: 0.6123616407776021 , Validatio
n Accuracy: 0.603515625
Epoch: 7 , Iteration: 40 , Training Accuracy: 0.6225002325365082 , Validation
Accuracy: 0.6170789930555556
Epoch: 7 , Iteration: 80 , Training Accuracy: 0.6121523579201935 , Validation
Accuracy: 0.6087601273148148
Epoch: 7 , Iteration: 120 , Training Accuracy: 0.6192447214212632 , Validatio
n Accuracy: 0.6135706018518519
Epoch: 7 , Iteration: 160 , Training Accuracy: 0.6143149474467492 , Validatio
n Accuracy: 0.6098813657407407
Epoch: 7 , Iteration: 200 , Training Accuracy: 0.6131135088208849 , Validatio
n Accuracy: 0.6110387731481481
Epoch: 7 , Iteration: 240 , Training Accuracy: 0.6092766564350603 , Validatio
n Accuracy: 0.6029007523148148
Epoch: 7 , Iteration: 280 , Training Accuracy: 0.6159427030043717 , Validatio
n Accuracy: 0.6147280092592593
Epoch: 7 , Iteration: 320 , Training Accuracy: 0.6139196353827551 , Validatio
n Accuracy: 0.6094835069444444
Epoch: 8 , Iteration: 40 , Training Accuracy: 0.6036415217189098 , Validation
```

```
Accuracy: 0.5984881365740741
Epoch: 8 , Iteration: 80 , Training Accuracy: 0.6108036461724491 , Validation
Accuracy: 0.6084346064814815
Epoch: 8 , Iteration: 120 , Training Accuracy: 0.6138111183455802 , Validatio
n Accuracy: 0.6108579282407407
Epoch: 8 , Iteration: 160 , Training Accuracy: 0.6239574613214275 , Validatio
n Accuracy: 0.6210575810185185
Epoch: 8 , Iteration: 200 , Training Accuracy: 0.6063389452143987 , Validatio
n Accuracy: 0.6029730902777778
Epoch: 8 , Iteration: 240 , Training Accuracy: 0.609369671038353 , Validation
Accuracy: 0.6092664930555556
Epoch: 8 , Iteration: 280 , Training Accuracy: 0.609245651567296 , Validation
Accuracy: 0.6048538773148148
Epoch: 8 , Iteration: 320 , Training Accuracy: 0.6115787678665551 , Validatio
n Accuracy: 0.6057942708333334
Epoch: 9 , Iteration: 40 , Training Accuracy: 0.6101602951663411 , Validation
Accuracy: 0.6061197916666666
Epoch: 9 , Iteration: 80 , Training Accuracy: 0.6207174526400645 , Validation
Accuracy: 0.6175491898148148
Epoch: 9 , Iteration: 120 , Training Accuracy: 0.6198260626918426 , Validatio
n Accuracy: 0.6176215277777778
Epoch: 9 , Iteration: 160 , Training Accuracy: 0.6127492016246551 , Validatio
n Accuracy: 0.6066984953703703
Epoch: 9 , Iteration: 200 , Training Accuracy: 0.615330356866028 , Validation
Accuracy: 0.6089048032407407
Epoch: 9 , Iteration: 240 , Training Accuracy: 0.6104703438439835 , Validatio
n Accuracy: 0.6067346643518519
Epoch: 9 , Iteration: 280 , Training Accuracy: 0.6164852881902458 , Validatio
n Accuracy: 0.6106409143518519
Epoch: 9 , Iteration: 320 , Training Accuracy: 0.5841239574613214 , Validatio
n Accuracy: 0.5814525462962963
Epoch: 10 , Iteration: 40 , Training Accuracy: 0.613656094006759 , Validation
Accuracy: 0.6112919560185185
Epoch: 10 , Iteration: 80 , Training Accuracy: 0.6107106315691564 , Validatio
n Accuracy: 0.6068070023148148
Epoch: 10 , Iteration: 120 , Training Accuracy: 0.6081372275447245 , Validati
on Accuracy: 0.6059751157407407
Epoch: 10 , Iteration: 160 , Training Accuracy: 0.6178339999379903 , Validati
on Accuracy: 0.6147641782407407
Epoch: 10 , Iteration: 200 , Training Accuracy: 0.6263448361392738 , Validati
on Accuracy: 0.6196469907407407
Epoch: 10 , Iteration: 240 , Training Accuracy: 0.6012231420332992 , Validati
on Accuracy: 0.5998625578703703
Epoch: 10 , Iteration: 280 , Training Accuracy: 0.6200585992000744 , Validati
on Accuracy: 0.6148003472222222
Epoch: 10 , Iteration: 320 , Training Accuracy: 0.6112764704058538 , Validati
on Accuracy: 0.6061921296296297
Epoch: 11 , Iteration: 40 , Training Accuracy: 0.6169426099897684 , Validatio
n Accuracy: 0.6087239583333334
Epoch: 11 , Iteration: 80 , Training Accuracy: 0.6259185192075156 , Validatio
n Accuracy: 0.6210214120370371
Epoch: 11 , Iteration: 120 , Training Accuracy: 0.62039190152854 , Validation
Accuracy: 0.6163194444444444
Epoch: 11 , Iteration: 160 , Training Accuracy: 0.6191594580349115 , Validati
on Accuracy: 0.61328125
Epoch: 11 , Iteration: 200 , Training Accuracy: 0.6213220475614671 , Validati
on Accuracy: 0.6153790509259259
```

```
Epoch: 11 , Iteration: 240 , Training Accuracy: 0.6177874926363439 , Validati
on Accuracy: 0.6134620949074074
Epoch: 11 , Iteration: 280 , Training Accuracy: 0.6230738225901467 , Validati
on Accuracy: 0.6192129629629629
Epoch: 11 , Iteration: 320 , Training Accuracy: 0.6195082627972591 , Validati
on Accuracy: 0.6145471643518519
Epoch: 12 , Iteration: 40 , Training Accuracy: 0.6224304715840387 , Validatio
n Accuracy: 0.6190321180555556
Epoch: 12 , Iteration: 80 , Training Accuracy: 0.6063001891296934 , Validatio
n Accuracy: 0.6002965856481481
Epoch: 12 , Iteration: 120 , Training Accuracy: 0.6246163147614175 , Validati
on Accuracy: 0.6218171296296297
Epoch: 12 , Iteration: 160 , Training Accuracy: 0.6152605959135584 , Validati
on Accuracy: 0.6109302662037037
Epoch: 12 , Iteration: 200 , Training Accuracy: 0.6071063156915636 , Validati
on Accuracy: 0.6021412037037037
Epoch: 12 , Iteration: 240 , Training Accuracy: 0.625972777726103 , Validatio
n Accuracy: 0.6235894097222222
Epoch: 12 , Iteration: 280 , Training Accuracy: 0.6184773509440982 , Validati
on Accuracy: 0.6156322337962963
Epoch: 12 , Iteration: 320 , Training Accuracy: 0.6214615694664062 , Validati
on Accuracy: 0.6167896412037037
Epoch: 12 , Training Accuracy: 0.615849688401079 , Validation Accuracy: 0.612
1600115740741
```

After increasing the learning rate to 0.01, I see high variance and the accuracies decreased from the second model. For my fourth and final attempt, I will lower the learning rate slightly to 0.0075. I will also try to increase the batch size to 75. Lastly, I want to increase the number of epochs to 20 to allow for an even longer training period.

In [64]:
```
#In my fourth iteration of the model, I will be decreasing the learning rate t
o 0.0075, I will also
#increase the batch size to 75 and will be also increasing the number of epoch
s to 20 to allow for a
#longer training time
autoEncoderModel4 = AutoEncoder()

use_cuda = True

if use_cuda and torch.cuda.is_available():
  autoEncoderModel4.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')

batch_size=75
train_loader = torch.utils.data.DataLoader(trainingData, batch_size=batch_size
, shuffle=True)
valid_loader = torch.utils.data.DataLoader(validationData, batch_size=batch_si
ze, shuffle=True)
train(autoEncoderModel4, train_loader, valid_loader, batch_size=batch_size, nu
m_epochs=20, learning_rate=0.0075)
```

```
Using CUDA!
Epoch: 1 , Iteration: 40 , Training Accuracy: 0.4578256286236939 , Validation
Accuracy: 0.4564525462962963
Epoch: 1 , Iteration: 80 , Training Accuracy: 0.5470808916999969 , Validation
Accuracy: 0.5498046875
Epoch: 1 , Iteration: 120 , Training Accuracy: 0.5630871546832853 , Validatio
n Accuracy: 0.5641276041666666
Epoch: 1 , Iteration: 160 , Training Accuracy: 0.5842092208476731 , Validatio
n Accuracy: 0.5808015046296297
Epoch: 1 , Iteration: 200 , Training Accuracy: 0.5845270207422565 , Validatio
n Accuracy: 0.5844184027777778
Epoch: 1 , Iteration: 240 , Training Accuracy: 0.5970142312343037 , Validatio
n Accuracy: 0.5956669560185185
Epoch: 1 , Iteration: 280 , Training Accuracy: 0.5959445632964375 , Validatio
n Accuracy: 0.5960286458333334
Epoch: 2 , Iteration: 40 , Training Accuracy: 0.5940920224475242 , Validation
Accuracy: 0.5915436921296297
Epoch: 2 , Iteration: 80 , Training Accuracy: 0.5997271571636748 , Validation
Accuracy: 0.5961733217592593
Epoch: 2 , Iteration: 120 , Training Accuracy: 0.6082302421480172 , Validatio
n Accuracy: 0.6036241319444444
Epoch: 2 , Iteration: 160 , Training Accuracy: 0.6112687191889127 , Validatio
n Accuracy: 0.6072771990740741
Epoch: 2 , Iteration: 200 , Training Accuracy: 0.6178960096735188 , Validatio
n Accuracy: 0.6137152777777778
Epoch: 2 , Iteration: 240 , Training Accuracy: 0.6109896753790345 , Validatio
n Accuracy: 0.6059027777777778
Epoch: 2 , Iteration: 280 , Training Accuracy: 0.6080829690261371 , Validatio
n Accuracy: 0.6043113425925926
Epoch: 3 , Iteration: 40 , Training Accuracy: 0.6161829907295445 , Validation
Accuracy: 0.6137152777777778
Epoch: 3 , Iteration: 80 , Training Accuracy: 0.6171053855455306 , Validation
Accuracy: 0.6117621527777778
Epoch: 3 , Iteration: 120 , Training Accuracy: 0.6106253681828047 , Validatio
n Accuracy: 0.6081090856481481
Epoch: 3 , Iteration: 160 , Training Accuracy: 0.6243450221684804 , Validatio
n Accuracy: 0.6202618634259259
Epoch: 3 , Iteration: 200 , Training Accuracy: 0.6038973118779648 , Validatio
n Accuracy: 0.6030454282407407
Epoch: 3 , Iteration: 240 , Training Accuracy: 0.6019440052088177 , Validatio
n Accuracy: 0.5984157986111112
Epoch: 3 , Iteration: 280 , Training Accuracy: 0.6073776082845007 , Validatio
n Accuracy: 0.6033347800925926
Epoch: 4 , Iteration: 40 , Training Accuracy: 0.6168185905187115 , Validation
Accuracy: 0.6151620370370371
Epoch: 4 , Iteration: 80 , Training Accuracy: 0.6178184975041081 , Validation
Accuracy: 0.6147641782407407
Epoch: 4 , Iteration: 120 , Training Accuracy: 0.6221359253402784 , Validatio
n Accuracy: 0.6182725694444444
Epoch: 4 , Iteration: 160 , Training Accuracy: 0.6107028803522153 , Validatio
n Accuracy: 0.6046730324074074
Epoch: 4 , Iteration: 200 , Training Accuracy: 0.6272439773044368 , Validatio
n Accuracy: 0.6240234375
Epoch: 4 , Iteration: 240 , Training Accuracy: 0.6135320745357021 , Validatio
n Accuracy: 0.6081452546296297
Epoch: 4 , Iteration: 280 , Training Accuracy: 0.6135863330542896 , Validatio
n Accuracy: 0.6101707175925926
```

```
Epoch: 5 , Iteration: 40 , Training Accuracy: 0.6151675813102657 , Validation
Accuracy: 0.6137152777777778
Epoch: 5 , Iteration: 80 , Training Accuracy: 0.6156404055436704 , Validation
Accuracy: 0.6142216435185185
Epoch: 5 , Iteration: 120 , Training Accuracy: 0.6219886522183983 , Validatio
n Accuracy: 0.6209852430555556
Epoch: 5 , Iteration: 160 , Training Accuracy: 0.6243217685176573 , Validatio
n Accuracy: 0.6195023148148148
Epoch: 5 , Iteration: 200 , Training Accuracy: 0.6154466251201438 , Validatio
n Accuracy: 0.6115089699074074
Epoch: 5 , Iteration: 240 , Training Accuracy: 0.6206631941214771 , Validatio
n Accuracy: 0.6178747106481481
Epoch: 5 , Iteration: 280 , Training Accuracy: 0.6182758193036306 , Validatio
n Accuracy: 0.6181640625
Epoch: 6 , Iteration: 40 , Training Accuracy: 0.6187021362353889 , Validation
Accuracy: 0.6152705439814815
Epoch: 6 , Iteration: 80 , Training Accuracy: 0.6198880724273711 , Validation
Accuracy: 0.6158130787037037
Epoch: 6 , Iteration: 120 , Training Accuracy: 0.6224769788856851 , Validatio
n Accuracy: 0.6206235532407407
Epoch: 6 , Iteration: 160 , Training Accuracy: 0.620004340681487 , Validation
Accuracy: 0.6184895833333334
Epoch: 6 , Iteration: 200 , Training Accuracy: 0.6223917154993334 , Validatio
n Accuracy: 0.6168619791666666
Epoch: 6 , Iteration: 240 , Training Accuracy: 0.6093231637367067 , Validatio
n Accuracy: 0.6066623263888888
Epoch: 6 , Iteration: 280 , Training Accuracy: 0.6167333271323597 , Validatio
n Accuracy: 0.6150535300925926
Epoch: 7 , Iteration: 40 , Training Accuracy: 0.6240039686230738 , Validation
Accuracy: 0.6211299189814815
Epoch: 7 , Iteration: 80 , Training Accuracy: 0.6243837782531857 , Validation
Accuracy: 0.6195746527777778
Epoch: 7 , Iteration: 120 , Training Accuracy: 0.6256549778315196 , Validatio
n Accuracy: 0.6213107638888888
Epoch: 7 , Iteration: 160 , Training Accuracy: 0.6248333488357672 , Validatio
n Accuracy: 0.6227936921296297
Epoch: 7 , Iteration: 200 , Training Accuracy: 0.6102455585526928 , Validatio
n Accuracy: 0.6043475115740741
Epoch: 7 , Iteration: 240 , Training Accuracy: 0.6277245527547824 , Validatio
n Accuracy: 0.6225405092592593
Epoch: 7 , Iteration: 280 , Training Accuracy: 0.6286546987877096 , Validatio
n Accuracy: 0.6235532407407407
Epoch: 8 , Iteration: 40 , Training Accuracy: 0.6211825256565281 , Validation
Accuracy: 0.6146918402777778
Epoch: 8 , Iteration: 80 , Training Accuracy: 0.6246240659783586 , Validation
Accuracy: 0.6192129629629629
Epoch: 8 , Iteration: 120 , Training Accuracy: 0.6263525873562149 , Validatio
n Accuracy: 0.6223234953703703
Epoch: 8 , Iteration: 160 , Training Accuracy: 0.6236629150776672 , Validatio
n Accuracy: 0.6155598958333334
Epoch: 8 , Iteration: 200 , Training Accuracy: 0.6184773509440982 , Validatio
n Accuracy: 0.6162109375
Epoch: 8 , Iteration: 240 , Training Accuracy: 0.6305072396366229 , Validatio
n Accuracy: 0.6269892939814815
Epoch: 8 , Iteration: 280 , Training Accuracy: 0.6269959383623229 , Validatio
n Accuracy: 0.6236979166666666
Epoch: 9 , Iteration: 40 , Training Accuracy: 0.6205624283012433 , Validation
```

```
Accuracy: 0.6173321759259259
Epoch: 9 , Iteration: 80 , Training Accuracy: 0.6240892320094255 , Validation
Accuracy: 0.6217086226851852
Epoch: 9 , Iteration: 120 , Training Accuracy: 0.6253914364555235 , Validatio
n Accuracy: 0.6198278356481481
Epoch: 9 , Iteration: 160 , Training Accuracy: 0.626104548414101 , Validation
Accuracy: 0.6214916087962963
Epoch: 9 , Iteration: 200 , Training Accuracy: 0.6144777230025114 , Validatio
n Accuracy: 0.6121238425925926
Epoch: 9 , Iteration: 240 , Training Accuracy: 0.6197795553901962 , Validatio
n Accuracy: 0.6187427662037037
Epoch: 9 , Iteration: 280 , Training Accuracy: 0.6254146901063466 , Validatio
n Accuracy: 0.6207320601851852
Epoch: 10 , Iteration: 40 , Training Accuracy: 0.6236861687284904 , Validatio
n Accuracy: 0.6173321759259259
Epoch: 10 , Iteration: 80 , Training Accuracy: 0.6225467398381546 , Validatio
n Accuracy: 0.6192129629629629
Epoch: 10 , Iteration: 120 , Training Accuracy: 0.6232908566644964 , Validati
on Accuracy: 0.6171513310185185
Epoch: 10 , Iteration: 160 , Training Accuracy: 0.6293368058785229 , Validati
on Accuracy: 0.6259403935185185
Epoch: 10 , Iteration: 200 , Training Accuracy: 0.6196942920038446 , Validati
on Accuracy: 0.6153067129629629
Epoch: 10 , Iteration: 240 , Training Accuracy: 0.6264533531764487 , Validati
on Accuracy: 0.6223596643518519
Epoch: 10 , Iteration: 280 , Training Accuracy: 0.636615198586178 , Validatio
n Accuracy: 0.6312572337962963
Epoch: 11 , Iteration: 40 , Training Accuracy: 0.6246085635444765 , Validatio
n Accuracy: 0.6217086226851852
Epoch: 11 , Iteration: 80 , Training Accuracy: 0.6263138312715096 , Validatio
n Accuracy: 0.6231192129629629
Epoch: 11 , Iteration: 120 , Training Accuracy: 0.6209422379313552 , Validati
on Accuracy: 0.6197916666666666
Epoch: 11 , Iteration: 160 , Training Accuracy: 0.620275633274424 , Validatio
n Accuracy: 0.6164279513888888
Epoch: 11 , Iteration: 200 , Training Accuracy: 0.6226320032245063 , Validati
on Accuracy: 0.6173321759259259
Epoch: 11 , Iteration: 240 , Training Accuracy: 0.6311195857749666 , Validati
on Accuracy: 0.6257957175925926
Epoch: 11 , Iteration: 280 , Training Accuracy: 0.6269804359284408 , Validati
on Accuracy: 0.6227936921296297
Epoch: 12 , Iteration: 40 , Training Accuracy: 0.6296933618578117 , Validatio
n Accuracy: 0.6285083912037037
Epoch: 12 , Iteration: 80 , Training Accuracy: 0.6331736582643475 , Validatio
n Accuracy: 0.6288700810185185
Epoch: 12 , Iteration: 120 , Training Accuracy: 0.6247325830155335 , Validati
on Accuracy: 0.6189236111111112
Epoch: 12 , Iteration: 160 , Training Accuracy: 0.6311040833410846 , Validati
on Accuracy: 0.6250723379629629
Epoch: 12 , Iteration: 200 , Training Accuracy: 0.6256239729637553 , Validati
on Accuracy: 0.6216724537037037
Epoch: 12 , Iteration: 240 , Training Accuracy: 0.6210275013177069 , Validati
on Accuracy: 0.615234375
Epoch: 12 , Iteration: 280 , Training Accuracy: 0.6298251325458097 , Validati
on Accuracy: 0.6235170717592593
Epoch: 13 , Iteration: 40 , Training Accuracy: 0.6282206306390103 , Validatio
n Accuracy: 0.6220341435185185
```

```
Epoch: 13 , Iteration: 80 , Training Accuracy: 0.6284531671472421 , Validatio
n Accuracy: 0.6239872685185185
Epoch: 13 , Iteration: 120 , Training Accuracy: 0.6282671379406567 , Validati
on Accuracy: 0.6241681134259259
Epoch: 13 , Iteration: 160 , Training Accuracy: 0.6296778594239295 , Validati
on Accuracy: 0.6292679398148148
Epoch: 13 , Iteration: 200 , Training Accuracy: 0.6296701082069885 , Validati
on Accuracy: 0.6240596064814815
Epoch: 13 , Iteration: 240 , Training Accuracy: 0.6327008340309429 , Validati
on Accuracy: 0.6300636574074074
Epoch: 13 , Iteration: 280 , Training Accuracy: 0.6273834992093759 , Validati
on Accuracy: 0.6215277777777778
Epoch: 14 , Iteration: 40 , Training Accuracy: 0.6277012991039593 , Validatio
n Accuracy: 0.6234447337962963
Epoch: 14 , Iteration: 80 , Training Accuracy: 0.6274532601618454 , Validatio
n Accuracy: 0.6217447916666666
Epoch: 14 , Iteration: 120 , Training Accuracy: 0.6314063808017859 , Validati
on Accuracy: 0.6253616898148148
Epoch: 14 , Iteration: 160 , Training Accuracy: 0.6253991876724646 , Validati
on Accuracy: 0.6231192129629629
Epoch: 14 , Iteration: 200 , Training Accuracy: 0.633879019005984 , Validatio
n Accuracy: 0.6296296296296297
Epoch: 14 , Iteration: 240 , Training Accuracy: 0.6404520509720026 , Validati
on Accuracy: 0.6358506944444444
Epoch: 14 , Iteration: 280 , Training Accuracy: 0.6269339286267944 , Validati
on Accuracy: 0.6236255787037037
Epoch: 15 , Iteration: 40 , Training Accuracy: 0.6290965181533501 , Validatio
n Accuracy: 0.6268807870370371
Epoch: 15 , Iteration: 80 , Training Accuracy: 0.6420565528788019 , Validatio
n Accuracy: 0.6366102430555556
Epoch: 15 , Iteration: 120 , Training Accuracy: 0.6348866772083217 , Validati
on Accuracy: 0.6276765046296297
Epoch: 15 , Iteration: 160 , Training Accuracy: 0.6278718258766627 , Validati
on Accuracy: 0.6238787615740741
Epoch: 15 , Iteration: 200 , Training Accuracy: 0.6317164294794283 , Validati
on Accuracy: 0.6273871527777778
Epoch: 15 , Iteration: 240 , Training Accuracy: 0.6371810374228754 , Validati
on Accuracy: 0.6332826967592593
Epoch: 15 , Iteration: 280 , Training Accuracy: 0.6375453446191052 , Validati
on Accuracy: 0.6334635416666666
Epoch: 16 , Iteration: 40 , Training Accuracy: 0.6271974700027905 , Validatio
n Accuracy: 0.6232638888888888
Epoch: 16 , Iteration: 80 , Training Accuracy: 0.6344448578426813 , Validatio
n Accuracy: 0.6295934606481481
Epoch: 16 , Iteration: 120 , Training Accuracy: 0.6315769075744891 , Validati
on Accuracy: 0.6264467592592593
Epoch: 16 , Iteration: 160 , Training Accuracy: 0.6320187269401296 , Validati
on Accuracy: 0.6314019097222222
Epoch: 16 , Iteration: 200 , Training Accuracy: 0.6412039190152854 , Validati
on Accuracy: 0.6380570023148148
Epoch: 16 , Iteration: 240 , Training Accuracy: 0.6371500325551112 , Validati
on Accuracy: 0.6323061342592593
Epoch: 16 , Iteration: 280 , Training Accuracy: 0.6367934765758224 , Validati
on Accuracy: 0.6327401620370371
Epoch: 17 , Iteration: 40 , Training Accuracy: 0.6393978854680185 , Validatio
n Accuracy: 0.6352358217592593
Epoch: 17 , Iteration: 80 , Training Accuracy: 0.6435602889653675 , Validatio
```

```
n Accuracy: 0.6383101851851852
Epoch: 17 , Iteration: 120 , Training Accuracy: 0.62843766471336 , Validation
Accuracy: 0.6227936921296297
Epoch: 17 , Iteration: 160 , Training Accuracy: 0.6418705236722165 , Validati
on Accuracy: 0.6366825810185185
Epoch: 17 , Iteration: 200 , Training Accuracy: 0.6378941493814528 , Validati
on Accuracy: 0.6348741319444444
Epoch: 17 , Iteration: 240 , Training Accuracy: 0.635188974669023 , Validatio
n Accuracy: 0.6322337962962963
Epoch: 17 , Iteration: 280 , Training Accuracy: 0.6458779028307444 , Validati
on Accuracy: 0.64453125
Epoch: 18 , Iteration: 40 , Training Accuracy: 0.644257898490063 , Validation
Accuracy: 0.6407335069444444
Epoch: 18 , Iteration: 80 , Training Accuracy: 0.6356850525532508 , Validatio
n Accuracy: 0.6333550347222222
Epoch: 18 , Iteration: 120 , Training Accuracy: 0.6426146404985583 , Validati
on Accuracy: 0.6415292245370371
Epoch: 18 , Iteration: 160 , Training Accuracy: 0.6399559730877747 , Validati
on Accuracy: 0.6373697916666666
Epoch: 18 , Iteration: 200 , Training Accuracy: 0.6402582705484761 , Validati
on Accuracy: 0.6340422453703703
Epoch: 18 , Iteration: 240 , Training Accuracy: 0.6347161504356184 , Validati
on Accuracy: 0.6315465856481481
Epoch: 18 , Iteration: 280 , Training Accuracy: 0.6411419092797569 , Validati
on Accuracy: 0.6356698495370371
Epoch: 19 , Iteration: 40 , Training Accuracy: 0.6419092797569218 , Validatio
n Accuracy: 0.6381293402777778
Epoch: 19 , Iteration: 80 , Training Accuracy: 0.6461724490745047 , Validatio
n Accuracy: 0.6439163773148148
Epoch: 19 , Iteration: 120 , Training Accuracy: 0.6442423960561808 , Validati
on Accuracy: 0.6414207175925926
Epoch: 19 , Iteration: 160 , Training Accuracy: 0.6381886956252132 , Validati
on Accuracy: 0.6342592592592593
Epoch: 19 , Iteration: 200 , Training Accuracy: 0.6431262208166683 , Validati
on Accuracy: 0.6382740162037037
Epoch: 19 , Iteration: 240 , Training Accuracy: 0.6375298421852231 , Validati
on Accuracy: 0.6334273726851852
Epoch: 19 , Iteration: 280 , Training Accuracy: 0.6372663008092271 , Validati
on Accuracy: 0.6334997106481481
Epoch: 20 , Iteration: 40 , Training Accuracy: 0.6404442997550616 , Validatio
n Accuracy: 0.6365740740740741
Epoch: 20 , Iteration: 80 , Training Accuracy: 0.6360803646172449 , Validatio
n Accuracy: 0.6342592592592593
Epoch: 20 , Iteration: 120 , Training Accuracy: 0.644273400923945 , Validatio
n Accuracy: 0.6410590277777778
Epoch: 20 , Iteration: 160 , Training Accuracy: 0.6492574334170464 , Validati
on Accuracy: 0.6436631944444444
Epoch: 20 , Iteration: 200 , Training Accuracy: 0.6478854680184789 , Validati
on Accuracy: 0.6443142361111112
Epoch: 20 , Iteration: 240 , Training Accuracy: 0.6416069822962205 , Validati
on Accuracy: 0.6369719328703703
Epoch: 20 , Iteration: 280 , Training Accuracy: 0.6389560661023781 , Validati
on Accuracy: 0.6358868634259259
Epoch: 20 , Training Accuracy: 0.6370105106501721 , Validation Accuracy: 0.63
32826967592593
```

As you can see above this combination of the hyperparameters had the highest final training accuracy of 63.70% and validation accuracy of 63.33%. This is the model I will be using for the testing stage in Part 4.

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [67]:
```
#Using the best model: autoEncoder4
#Hyperparameters: batch_size = 75, num_epochs = 20, learning_rate = 0.0075

use_cuda = True

if use_cuda and torch.cuda.is_available():
  autoEncoderModel4.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')

batch_size=75
test_loader = torch.utils.data.DataLoader(testingData, batch_size=batch_size,
shuffle=True)

print("The test accuracy for the final model is:", get_accuracy(autoEncoderMod
el4, test_loader)*100, "%")
```

```
Using CUDA!
The test accuracy for the final model is: 63.35358796296296 %
```

Part (b) [4 pt]

Based on the test accuracy alone, it is difficult to assess whether our model is actually performing well. We don't know whether a high accuracy is due to the simplicity of the problem, or if a poor accuracy is a result of the inherent difficulty of the problem.

It is therefore very important to be able to compare our model to at least one alternative. In particular, we consider a simple **baseline** model that is not very computationally expensive. Our neural network should at least outperform this baseline model. If our network is not much better than the baseline, then it is not doing well.

For our data imputation problem, consider the following baseline model: to predict a missing feature, the baseline model will look at the **most common value** of the feature in the training set.

For example, if the feature "marriage" is missing, then this model's prediction will be the most common value for "marriage" in the training set, which happens to be "Married-civ-spouse".

What would be the test accuracy of this baseline model?

In [70]:
```
mostCommonModel = {}

for column in df_not_missing.columns:
  #Retrieve the most common model
  mostCommonModel[column] = df_not_missing[column].value_counts().idxmax()

baselineAccuracy = sum(df_not_missing['marriage'] == mostCommonModel['marriag
e'])/len(df_not_missing)

print("The accuracy of the baseline model, the mostCommonModel, is:", baseline
Accuracy*100, "%")
```

```
The accuracy of the baseline model, the mostCommonModel, is: 46.6794713197473
8 %
```

Part (c) [1 pt]

How does your test accuracy from part (a) compared to your basline test accuracy in part (b)?

In [ ]:
```
'''
Evidently, my autoEncoderModel 4 has a much higher test accuracy of 63.35% com
pared to the baseline
model which has an accuracy of 46.68%. This is expected since the most common
 value is not making any
predictions based on the other data available, and is only choosing the most f
requently used number.
My model is actually making predictions based on the other data available. The
refore, it should
perform better.
'''
```

Part (d) [1 pt]

Look at the first item in your test data. Do you think it is reasonable for a human to be able to guess this person's education level based on their other features? Explain.

In [72]:
```
get_features(testingData[0])
```

Out[72]:
```
{'edu': 'Bachelors',
 'marriage': 'Divorced',
 'occupation': 'Prof-specialty',
 'relationship': 'Not-in-family',
 'sex': 'Male',
 'work': 'Private'}
```

No. It is difficult for a human to guess this person's education level based on their other features. A lot of the times, humans assume a person's education level based on their occupation and marital status. In a case like this, it is hard to predict this education level because their occupation is listed as "Prof-specialty". I would assume they are a person who has a Masters education in the minimum. Evidently, humans can be very biased, and therefore, they do not always make the best predictions.

Part (e) [2 pt]

What is your model's prediction of this person's education level, given their other features?

In [78]:
```
firstTestData = testingData[0]

educationIndexStart = cat_index['edu']
educationIndexStop = cat_index['edu'] + len(cat_values['edu'])

firstTestData[educationIndexStart:educationIndexStop] = 0
firstTestData = torch.from_numpy(firstTestData)
firstTestData = firstTestData.cuda()

modelPrediction = autoEncoderModel4(firstTestData)
modelPrediction = modelPrediction.detach().cpu().numpy()

print("AutoEncoderModel's Predicted Education Level:", get_feature(modelPredic
tion, "edu"))
```

```
AutoEncoderModel's Predicted Education Level: Prof-school
```

As we can see above, my model's prediction for this individual is also at Professional School Level.

Part (f) [2 pt]

What is the baseline model's prediction of this person's education level?

In [73]:
```
baselinePrediction = df_not_missing['edu'].value_counts().idxmax()
print("The baseline model's prediction for this person's education is:", basel
inePrediction)
```

```
The baseline model's prediction for this person's education is:  HS-grad
```

In [79]:
```
%%shell
jupyter nbconvert --to html /content/Lab_4_Data_Imputation.ipynb
```

```
[NbConvertApp] Converting notebook /content/Lab_4_Data_Imputation.ipynb to ht
ml
[NbConvertApp] Writing 481907 bytes to /content/Lab_4_Data_Imputation.html
```

Out[79]: