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.

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

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://colab.research.google.com/drive/1TkZR1tzYVPMAuaPyS5mOt71K1n-WuliL?usp=sharing (https://colab.research.google.com/drive/1TkZR1tzYVPMAuaPyS5mOt71K1n-WuliL?usp=sharing)

In [ ]:
```
import csv
import numpy as np
import random
import torch
import torch.utils.data
```

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 [ ]:
```
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 [39]:
```
header = ['age', 'work', 'fnlwgt', 'edu', 'yredu', 'marriage', 'occupati
on',
 'relationship', 'race', 'sex', 'capgain', 'caploss', 'workhr', 'countr
y']
df = pd.read_csv(
    "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adu
lt.data",
    names=header,
    index_col=False)
```

In [40]:
```
df.shape # there are 32561 rows (records) in the data frame, and 14 colu
mns (features)
```

Out[40]:
```
(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 [41]:
```
df[:3] # show the first 3 records
```

Out[41]:

	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 [42]:
```
subdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
subdf[:3] # show the first 3 records
```

Out[42]:

	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 [43]:
```
np.sum(subdf["caploss"])
```

Out[43]:
```
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 [44]:
```
column = ["age", "yredu", "capgain", "caploss", "workhr"]

for i in range(len(column)):
  print(column[i] + ' minimum: ' + str(df[column[i]].min()))
  print(column[i] + ' maximum: ' + str(df[column[i]].max()))
  print(column[i] + ' average: ' + str(df[column[i]].mean()))
```

```
age minimum: 17
age maximum: 90
age average: 38.58164675532078
yredu minimum: 1
yredu maximum: 16
yredu average: 10.0806793403151
capgain minimum: 0
capgain maximum: 99999
capgain average: 1077.6488437087312
caploss minimum: 0
caploss maximum: 4356
caploss average: 87.303829734959
workhr minimum: 1
workhr maximum: 99
workhr average: 40.437455852092995
```

In [45]:
```
def normalize(data, col):
  return (data[col] - data[col].min())/(data[col].max() - data[col].min
())

for i in range(len(column)):
  df[column[i]] = normalize(df, column[i])

df[:5]
```

Out[45]:

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationship	race	s
0	0.301370	State-gov	77516	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family	White	M
1	0.452055	Self-emp-not-inc	83311	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband	White	M
2	0.287671	Private	215646	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family	White	M
3	0.493151	Private	234721	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husband	Black	M
4	0.150685	Private	338409	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wife	Black	Fem

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 [46]:
```
# hint: you can do something like this in pandas
sum(df["sex"] == " Male")
```

Out[46]: 21790

In [47]:
```
Male_Total = sum(df["sex"] == " Male")
Female_Total = sum(df["sex"] == " Female")

Male_Percent = (Male_Total/(Male_Total + Female_Total))*100
Female_Percent = (Female_Total/(Male_Total + Female_Total))*100

print("Male %: " + str(Male_Percent) + " %")
print("Female %: " + str(Female_Percent) + " %")
```

```
Male %: 66.92054912318419 %
Female %: 33.07945087681583 %
```

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 [48]:
```
contcols = ["age", "yredu", "capgain", "caploss", "workhr"]
catcols = ["work", "marriage", "occupation", "edu", "relationship", "se
x"]
features = contcols + catcols
df = df[features]
```

In [49]:
```
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 [50]:
```
print("# of Records with missing features: " + str(df_with_missing.shape
[0]))
print("# of Records with no missing features: " + str(df_not_missing.sha
pe[0]))
print("% removed: " + str(df_with_missing.shape[0]/(df_with_missing.shap
e[0]+df_not_missing.shape[0])*100))
```

```
# of Records with missing features: 1843
# of Records with no missing features: 30718
% removed: 5.660145572924664
```

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 [51]:
```
set(df_not_missing['work'].unique())
```

Out[51]:
```
{' Federal-gov',
 ' Local-gov',
 ' Private',
 ' Self-emp-inc',
 ' Self-emp-not-inc',
 ' State-gov',
 ' Without-pay'}
```

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 [52]:
```
data = pd.get_dummies(df_not_missing)
```

In [53]:
```
data[:3]
```

Out[53]:

	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 [54]:
```
print(data.shape[1])
```

```
57
```

There are 57 Columns (features) inside the dataframe data. This number comes from the total number of features, including all the categorical values of each original 14 columns.

For example, work --> work_Federal_gov + ... + work_Without-pay.

For example, marriage --> marriage_Divorced + ... + marriage_Widowed.

The 57 contains ALL subcolumns.

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 [55]:
```
datanp = data.values.astype(np.float32)
```

In [59]:
```
cat_index = {}  # Mapping of feature -> start index of feature in a reco
rd
cat_values = {} # Mapping of feature -> list of categorical values the f
eature 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 u
nderscore
        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, "wo
rk")`
    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.]), "w
ork")
    'State-gov'
    >>> get_categorical_value(np.array([0.1, 0., 1.1, 0.2, 0., 1., 0.]),
"work")
    'Private'
    """
    # <----- TODO: WRITE YOUR CODE HERE ----->
    # You may find the variables `cat_index` and `cat_values`
    # (created above) useful.
```

```
    return cat_values[feature][np.argmax(onehot)]
```

In [61]:
```
print("For Cat_Values: " + str(cat_values))
print("For Cat_index: " + str(cat_index))
```

```
For Cat_Values: {'work': ['Federal-gov', 'Local-gov', 'Private', 'Self-emp-inc', 'Self-emp-not-inc', 'State-gov', 'Without-pay'], 'marriage': ['Divorced', 'Married-AF-spouse', 'Married-civ-spouse', 'Married-spouse-absent', 'Never-married', 'Separated', 'Widowed'], 'occupation': ['Adm-clerical', 'Armed-Forces', 'Craft-repair', 'Exec-managerial', 'Farming-fishing', 'Handlers-cleaners', 'Machine-op-inspct', 'Other-service', 'Priv-house-serv', 'Prof-specialty', 'Protective-serv', 'Sales', 'Tech-support', 'Transport-moving'], 'edu': ['10th', '11th', '12th', '1st-4th', '5th-6th', '7th-8th', '9th', 'Assoc-acdm', 'Assoc-voc', 'Bachelors', 'Doctorate', 'HS-grad', 'Masters', 'Preschool', 'Prof-school', 'Some-college'], 'relationship': ['Husband', 'Not-in-family', 'Other-relative', 'Own-child', 'Unmarried', 'Wife'], 'sex': ['Female', 'Male']}
For Cat_index: {'work': 5, 'marriage': 12, 'occupation': 19, 'edu': 33, 'relationship': 49, 'sex': 55}
```

In [62]:
```
# 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 [68]:
```
# set the numpy seed for reproducibility
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.see
d.html
np.random.seed(50)

# todo
np.random.shuffle(datanp)
training = datanp[:int(0.7*datanp.shape[0]), :]
# 0:70
validation = datanp[int(0.7*datanp.shape[0]):int(0.85*datanp.shape[0]),
:]  # 70:85
testing = datanp[int(0.85*datanp.shape[0]):int(datanp.shape[0]), :]
# 85:100
print("Training Data: ", len(training))
print("Validation Data: ", len(validation))
print("Testing Data: ", len(testing))
```

```
Training Data:  21502
Validation Data:  4608
Testing Data:  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 [86]:
```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(57, 38), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(38, 19),
            nn.ReLU(),
            nn.Linear(19, 10)
        )
        self.decoder = nn.Sequential(
            nn.Linear(10, 19), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(19, 38),
            nn.ReLU(),
            nn.Linear(38, 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 [ ]:
```
'''A Sigmoid activation is required since the input values are all betwe
en 0-1,
and the output of decoder needs to be 0 to 1. The Sigmoid forces the out
put to be
in range 0 to 1 '''
```

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 [80]:
```
import matplotlib.pyplot as plt
```

In [88]:
```
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, num_epochs=5, learning_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)

    train_acc = np.zeros(num_epochs)
    val_acc = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    iter = []

    for epoch in range(num_epochs):
        total_train_loss = 0.0
        total_val_loss = 0.0
        for data in train_loader:
            datam = zero_out_random_feature(data.clone()) # zero out one
categorical feature
            recon = model(datam)
            loss = criterion(recon, data)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            total_train_loss = loss

        for data in valid_loader:
            datam = zero_out_random_feature(data.clone()) # zero out one
categorical feature
            recon = model(datam)
            loss = criterion(recon, data)

            total_val_loss = loss

        train_acc[epoch] = get_accuracy(model, train_loader)
        val_acc[epoch] = get_accuracy(model, valid_loader)

        train_loss[epoch] = total_train_loss
        val_loss[epoch] = total_val_loss
```

```
        iter.append(epoch)

        print(("Epoch {}: Train acc: {} |"+ "Validation acc: {}").format
(
                  epoch + 1,
                  train_acc[epoch],
                  val_acc[epoch]))

        print(("Epoch {}: Train Loss: {} |"+ "Validation Loss: {}").form
at(
                  epoch + 1,
                  train_loss[epoch],
                  val_loss[epoch]))

    # plotting
    plt.title("Training Curve")
    plt.plot(iter, train_loss, label="Train")
    plt.plot(iter, val_loss, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training Curve")
    plt.plot(iter, train_acc, label="Train")
    plt.plot(iter, val_acc, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    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 [75]:
```
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=25
6, shuffle=True)
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            inp = item.detach().numpy()
            out = model(zero_out_feature(item.clone(), col)).detach().nu
mpy()
            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]

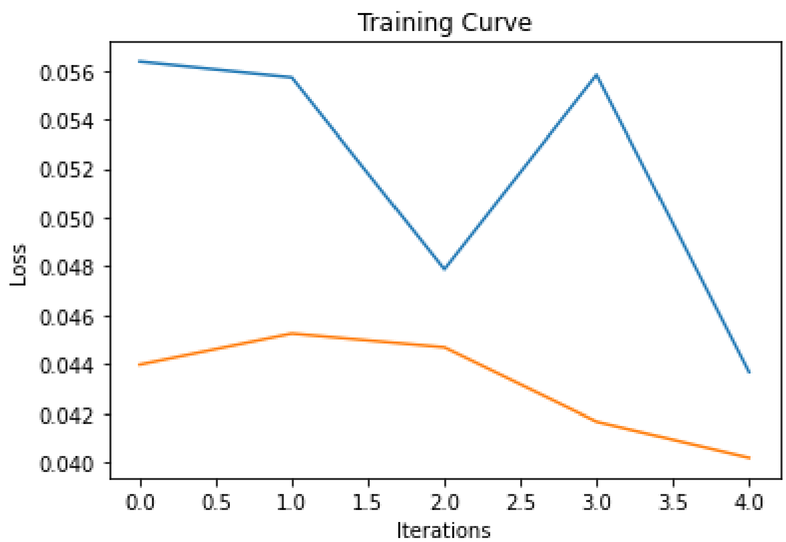
Run your updated training code, using reasonable initial hyperparameters.

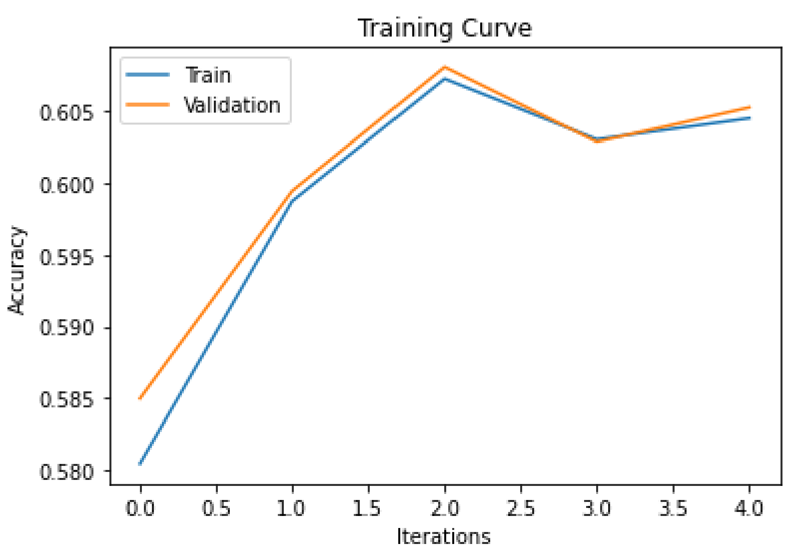
Include your training curve in your submission.

In [ ]:
```
# The initial hyperparameters are
# # of Epoch = 5
# Learning Rate = 1e-4
# 4 Layers
# These initial hyperparamters were chosen the way it was given
```

In [89]:
```
AutoEncoder1 = AutoEncoder()

train_loader = torch.utils.data.DataLoader(dataset=training)
valid_loader = torch.utils.data.DataLoader(dataset=valid)

train(AutoEncoder1, train_loader, valid_loader)
```

```
Epoch 1: Train acc: 0.5804188757634948 |Validation acc: 0.5849609375
Epoch 1: Train Loss: 0.056369930505752563 |Validation Loss: 0.043976679
44431305
Epoch 2: Train acc: 0.5987505038291012 |Validation acc: 0.5994646990740
741
Epoch 2: Train Loss: 0.05571722984313965 |Validation Loss: 0.0452486313
8794899
Epoch 3: Train acc: 0.6072845936812079 |Validation acc: 0.6081090856481
481
Epoch 3: Train Loss: 0.04787912592291832 |Validation Loss: 0.0446843169
6295738
Epoch 4: Train acc: 0.6030756828822125 |Validation acc: 0.6029007523148
148
Epoch 4: Train Loss: 0.05582636222243309 |Validation Loss: 0.0416526421
9045639
Epoch 5: Train acc: 0.6045406628840728 |Validation acc: 0.6052879050925
926
Epoch 5: Train Loss: 0.04367002099752426 |Validation Loss: 0.0401770509
7794533
```





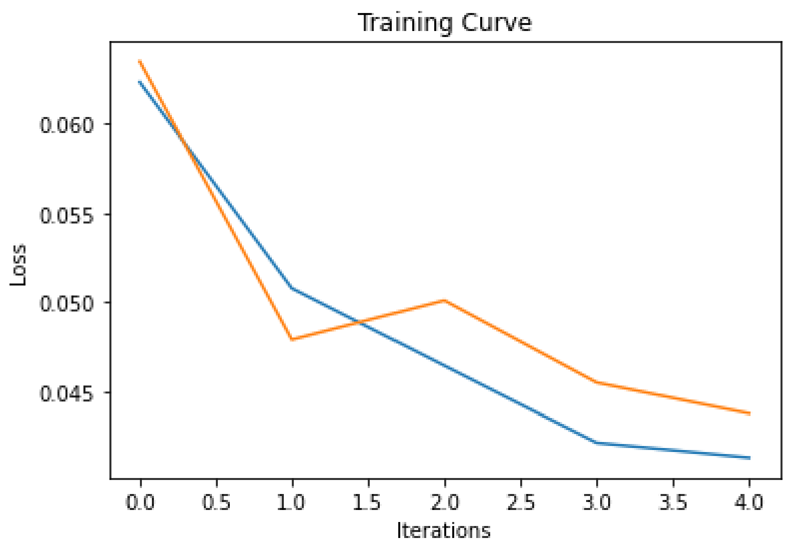
Part (d) [5 pt]

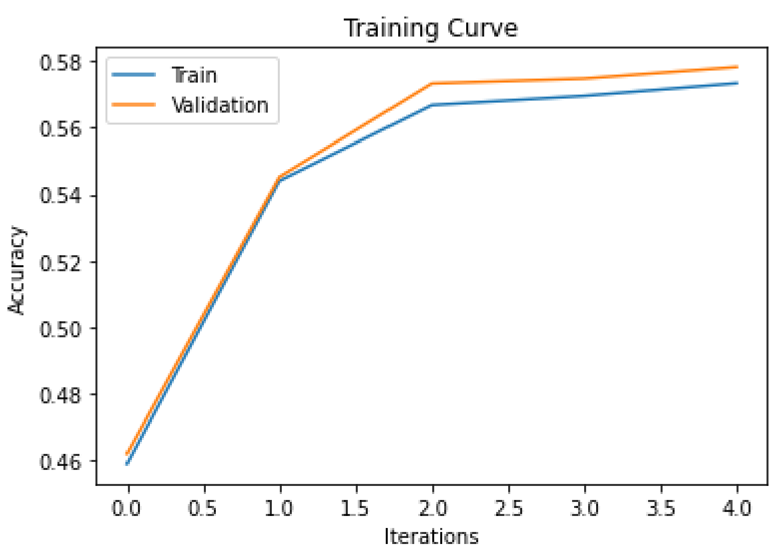
Tune your hyperparameters, training at least 4 different models (4 sets of hyperparameters).

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 [91]:
```
# Changed Batch Size to 10 from 5
# num_epoch = 5
# learning_rate = 0.0001
# batch_size = 10

AutoEncoder2 = AutoEncoder()

train_loader = torch.utils.data.DataLoader(dataset=training, batch_size=
10)
valid_loader = torch.utils.data.DataLoader(dataset=valid, batch_size=10)

train(AutoEncoder2, train_loader, valid_loader, num_epochs=5, learning_r
ate=0.0001)
```

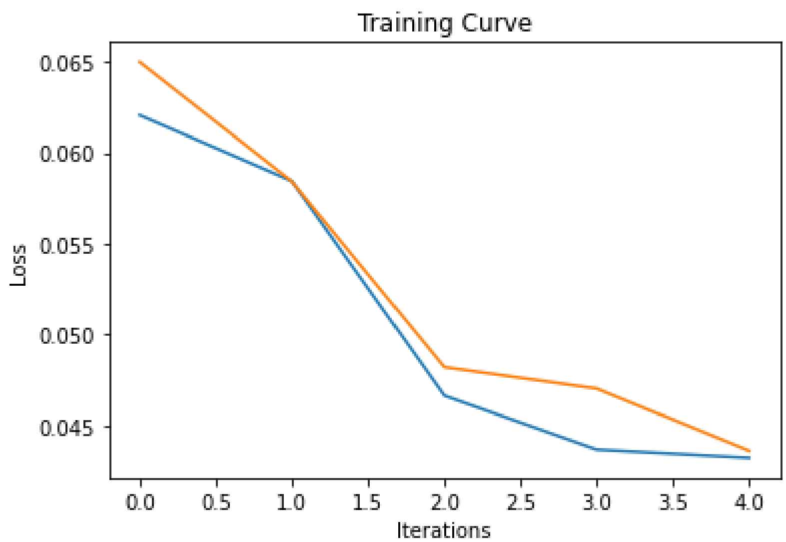
```
Epoch 1: Train acc: 0.4589263014293244 |Validation acc: 0.4619864004629
6297
Epoch 1: Train Loss: 0.06231649965047836 |Validation Loss: 0.0634771659
9702835
Epoch 2: Train acc: 0.5439881561405141 |Validation acc: 0.5451750578703
703
Epoch 2: Train Loss: 0.05075991526246071 |Validation Loss: 0.0478810891
5090561
Epoch 3: Train acc: 0.5667379778625244 |Validation acc: 0.5732421875
Epoch 3: Train Loss: 0.046421945095062256 |Validation Loss: 0.050077926
367521286
Epoch 4: Train acc: 0.5695129135274238 |Validation acc: 0.5747251157407
407
Epoch 4: Train Loss: 0.04210023954510689 |Validation Loss: 0.0454786419
86846924
Epoch 5: Train acc: 0.5733110098285431 |Validation acc: 0.5781973379629
629
Epoch 5: Train Loss: 0.041264910250902176 |Validation Loss: 0.043748687
95275688
```

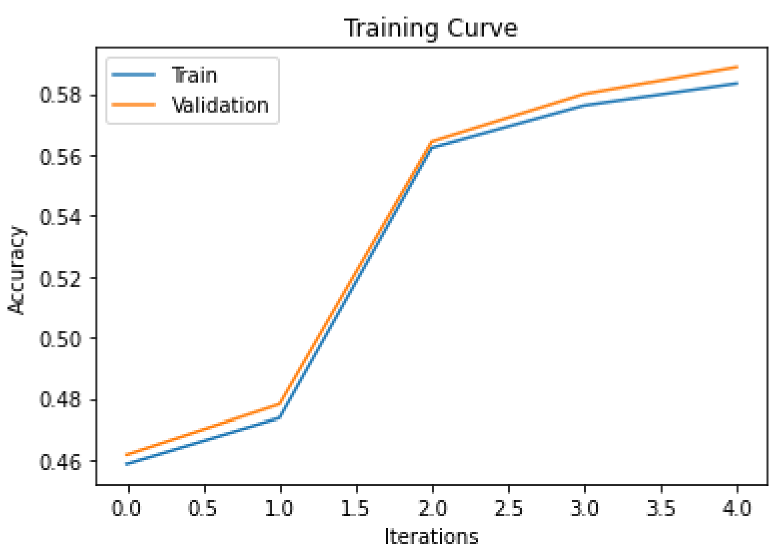




In [93]:
```
# Changed Batch Size to 20 from 10
# num_epoch = 5
# learning_rate = 0.0001
# batch_size = 20

AutoEncoder3 = AutoEncoder()

train_loader = torch.utils.data.DataLoader(dataset=training, batch_size=
20)
valid_loader = torch.utils.data.DataLoader(dataset=valid, batch_size=20)

train(AutoEncoder3, train_loader, valid_loader, num_epochs=5, learning_r
ate=0.0001)
```

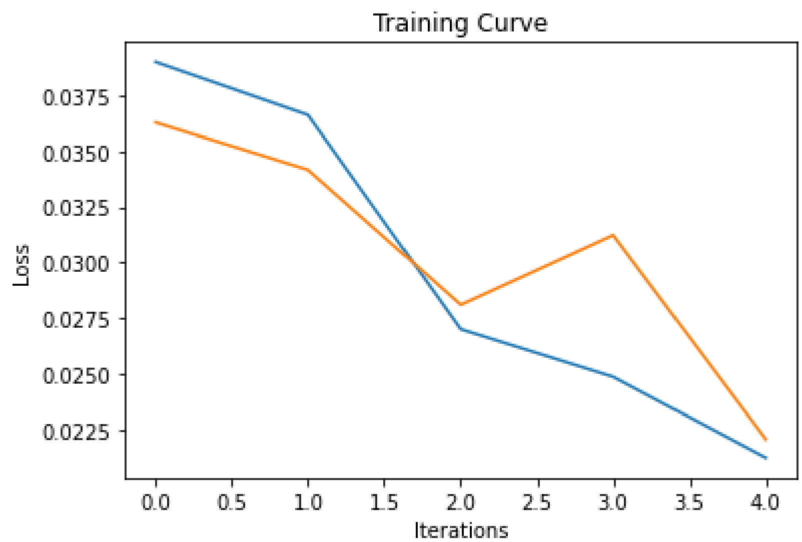
```
Epoch 1: Train acc: 0.4587557746566211 |Validation acc: 0.4618055555555
556
Epoch 1: Train Loss: 0.06206278130412102 |Validation Loss: 0.0649812519
5503235
Epoch 2: Train acc: 0.4737543794375717 |Validation acc: 0.4782262731481
4814
Epoch 2: Train Loss: 0.05844651535153389 |Validation Loss: 0.0584241449
83291626
Epoch 3: Train acc: 0.5623042817722382 |Validation acc: 0.5643446180555
556
Epoch 3: Train Loss: 0.046652838587760925 |Validation Loss: 0.048213265
83623886
Epoch 4: Train acc: 0.5761091991442656 |Validation acc: 0.5798611111111
112
Epoch 4: Train Loss: 0.04369697347283363 |Validation Loss: 0.0470437780
02262115
Epoch 5: Train acc: 0.5833720894180386 |Validation acc: 0.5887586805555
556
Epoch 5: Train Loss: 0.04324262961745262 |Validation Loss: 0.0436300858
8552475
```

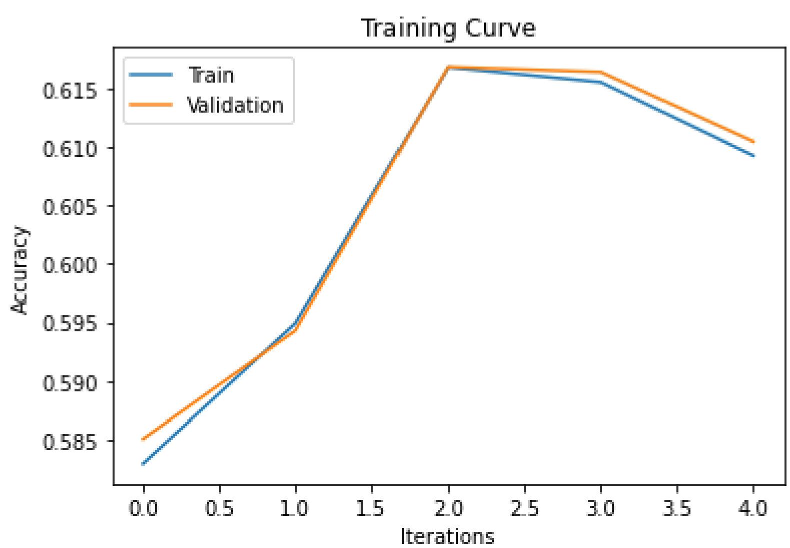




In [94]:
```
# Changed learning_rate to 0.001 from 0.0001
# num_epoch = 5
# learning_rate = 0.001
# batch_size = 20

AutoEncoder4 = AutoEncoder()

train_loader = torch.utils.data.DataLoader(dataset=training, batch_size=
20)
valid_loader = torch.utils.data.DataLoader(dataset=valid, batch_size=20)

train(AutoEncoder4, train_loader, valid_loader, num_epochs=5, learning_r
ate=0.001)
```

```
Epoch 1: Train acc: 0.5829302700523983 |Validation acc: 0.5850332754629
629
Epoch 1: Train Loss: 0.03900599107146263 |Validation Loss: 0.0362900979
8169136
Epoch 2: Train acc: 0.5948981490093945 |Validation acc: 0.5942925347222
222
Epoch 2: Train Loss: 0.036631349474191666 |Validation Loss: 0.034162476
658821106
Epoch 3: Train acc: 0.6167875856509472 |Validation acc: 0.6168258101851
852
Epoch 3: Train Loss: 0.02699468471109867 |Validation Loss: 0.0280952937
90102005
Epoch 4: Train acc: 0.6155241372895545 |Validation acc: 0.6163917824074
074
Epoch 4: Train Loss: 0.024856073781847954 |Validation Loss: 0.031222980
469465256
Epoch 5: Train acc: 0.6092611540011782 |Validation acc: 0.6104600694444
444
Epoch 5: Train Loss: 0.021219123154878616 |Validation Loss: 0.022046253
085136414
```



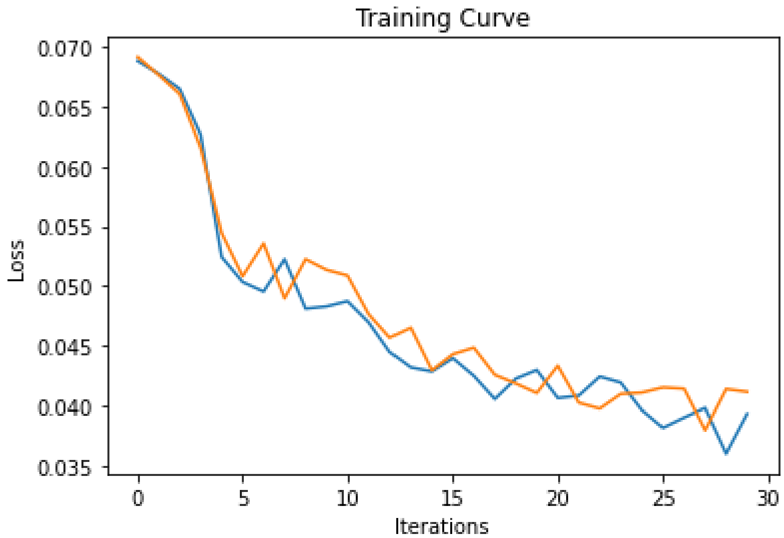


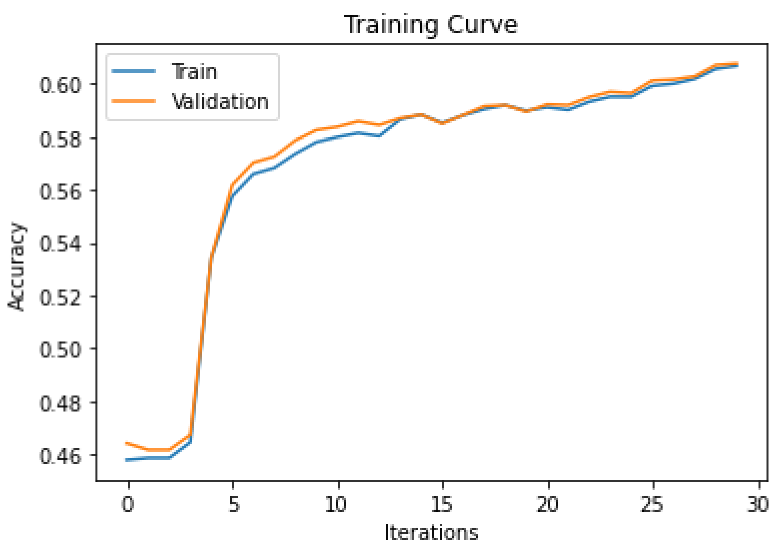
In [97]:
```
# Try Mixing Results
# num_epoch = 30
# learning_rate = 0.0001
# batch_size = 30

AutoEncoder5 = AutoEncoder()

train_loader = torch.utils.data.DataLoader(dataset=training, batch_size=
30)
valid_loader = torch.utils.data.DataLoader(dataset=valid, batch_size=30)

train(AutoEncoder5, train_loader, valid_loader, num_epochs=30, learning_
rate=0.0001)
```

```
Epoch 1: Train acc: 0.45803491148110254 |Validation acc: 0.464265046296
2963
Epoch 1: Train Loss: 0.06880198419094086 |Validation Loss: 0.0691323503
8518906
Epoch 2: Train acc: 0.4587557746566211 |Validation acc: 0.4618055555555
556
Epoch 2: Train Loss: 0.06772919744253159 |Validation Loss: 0.0676305815
577507
Epoch 3: Train acc: 0.4587557746566211 |Validation acc: 0.4618055555555
556
Epoch 3: Train Loss: 0.06647177785634995 |Validation Loss: 0.0660376027
226448
Epoch 4: Train acc: 0.4646699531826497 |Validation acc: 0.4674840856481
4814
Epoch 4: Train Loss: 0.0626591295003891 |Validation Loss: 0.06155933439
731598
Epoch 5: Train acc: 0.5342913837472483 |Validation acc: 0.5344690393518
519
Epoch 5: Train Loss: 0.052446551620960236 |Validation Loss: 0.054449282
586574554
Epoch 6: Train acc: 0.5576768052584256 |Validation acc: 0.5619212962962
963
Epoch 6: Train Loss: 0.050351742655038834 |Validation Loss: 0.050831940
02509117
Epoch 7: Train acc: 0.5660171146870059 |Validation acc: 0.5702763310185
185
Epoch 7: Train Loss: 0.04955750331282616 |Validation Loss: 0.0535800196
2304115
Epoch 8: Train acc: 0.5682959724676774 |Validation acc: 0.5725549768518
519
Epoch 8: Train Loss: 0.052259478718042374 |Validation Loss: 0.048985440
28401375
Epoch 9: Train acc: 0.5736133072892444 |Validation acc: 0.5786313657407
407
Epoch 9: Train Loss: 0.04812362790107727 |Validation Loss: 0.0522595420
4797745
Epoch 10: Train acc: 0.5779927448609432 |Validation acc: 0.582573784722
2222
Epoch 10: Train Loss: 0.04831041023135185 |Validation Loss: 0.051369424
909353256
Epoch 11: Train acc: 0.579922797879267 |Validation acc: 0.5837673611111
112
Epoch 11: Train Loss: 0.04874804988503456 |Validation Loss: 0.050896909
08789635
Epoch 12: Train acc: 0.5814497876166558 |Validation acc: 0.585865162037
0371
Epoch 12: Train Loss: 0.046986211091279984 |Validation Loss: 0.04767648
130655289
Epoch 13: Train acc: 0.5802871050754969 |Validation acc: 0.584454571759
2593
Epoch 13: Train Loss: 0.04449542611837387 |Validation Loss: 0.045698657
631874084
Epoch 14: Train acc: 0.5865733420146964 |Validation acc: 0.587058738425
9259
Epoch 14: Train Loss: 0.04320777580142021 |Validation Loss: 0.046519987
285137177
Epoch 15: Train acc: 0.5883716243450222 |Validation acc: 0.588360821759
```

```
2593
Epoch 15: Train Loss: 0.042870644479990005 |Validation Loss: 0.04296748
340129852
Epoch 16: Train acc: 0.5852633863516572 |Validation acc: 0.5849609375
Epoch 16: Train Loss: 0.04397854954004288 |Validation Loss: 0.044315569
10276413
Epoch 17: Train acc: 0.5881313366198493 |Validation acc: 0.588216145833
3334
Epoch 17: Train Loss: 0.04250174015760422 |Validation Loss: 0.044853899
627923965
Epoch 18: Train acc: 0.590433448051344 |Validation acc: 0.5915436921296
297
Epoch 18: Train Loss: 0.04056732356548309 |Validation Loss: 0.042588010
430336
Epoch 19: Train acc: 0.5919216817040275 |Validation acc: 0.591941550925
9259
Epoch 19: Train Loss: 0.042271506041288376 |Validation Loss: 0.04186980
053782463
Epoch 20: Train acc: 0.5898366043468825 |Validation acc: 0.589482060185
1852
Epoch 20: Train Loss: 0.04298669844865799 |Validation Loss: 0.041072346
27008438
Epoch 21: Train acc: 0.5911543112268626 |Validation acc: 0.592230902777
7778
Epoch 21: Train Loss: 0.040657203644514084 |Validation Loss: 0.04334883
764386177
Epoch 22: Train acc: 0.5901311505906427 |Validation acc: 0.591941550925
9259
Epoch 22: Train Loss: 0.04085388407111168 |Validation Loss: 0.040269117
802381516
Epoch 23: Train acc: 0.5932161349331845 |Validation acc: 0.594979745370
3703
Epoch 23: Train Loss: 0.042447954416275024 |Validation Loss: 0.03978231
1767339706
Epoch 24: Train acc: 0.5950996806498621 |Validation acc: 0.596969039351
8519
Epoch 24: Train Loss: 0.04194619506597519 |Validation Loss: 0.040999148
0410099
Epoch 25: Train acc: 0.5950686757820978 |Validation acc: 0.596462673611
1112
Epoch 25: Train Loss: 0.03966417908668518 |Validation Loss: 0.041106864
80998993
Epoch 26: Train acc: 0.5992620841472112 |Validation acc: 0.601236979166
6666
Epoch 26: Train Loss: 0.038183439522981644 |Validation Loss: 0.04153640
195727348
Epoch 27: Train acc: 0.6000759619260224 |Validation acc: 0.601634837962
9629
Epoch 27: Train Loss: 0.03903285413980484 |Validation Loss: 0.041434373
70657921
Epoch 28: Train acc: 0.6018587418224661 |Validation acc: 0.602828414351
8519
Epoch 28: Train Loss: 0.03983934223651886 |Validation Loss: 0.037975721
061229706
Epoch 29: Train acc: 0.6056568381235854 |Validation acc: 0.607096354166
6666
Epoch 29: Train Loss: 0.03602205589413643 |Validation Loss: 0.041405439
```

```
376831055
Epoch 30: Train acc: 0.6067885157969801 |Validation acc: 0.607675057870
3703
Epoch 30: Train Loss: 0.03934241458773613 |Validation Loss: 0.041175175
458192825
```





Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [100]:

```
test_loader = torch.utils.data.DataLoader(dataset=test)
test_acc = (get_accuracy(AutoEncoder5, test_loader) * 100)

print('Test Accuracy: ' + str(test_acc) + '%')
```

```
Test Accuracy: 60.525173611111114%
```

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 [103]:
```
most_common = {}

for col in df_not_missing.columns:
  most_common[col] = df_not_missing[col].value_counts().idxmax()

baseline_accuracy = sum(df_not_missing['marriage'] == most_common['marri
age'])/len(df_not_missing)

print("The Baseline model accuracy for missing 'marriage': ", baseline_a
ccuracy*100)
```

```
The Baseline model accuracy for missing 'marriage':  46.67947131974738
```

Part (c) [1 pt]

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

In [ ]:
```
# My Test accuracy from part(a) was 60.52%, which is better than the
# basline model accuracy in part(b) being 46.68%.
# Thus, my NN at least outperforms the baseline model.
```

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 [104]:
```
get_features(test_data[0])

# Yes, I think it is reasonable for a human to somewhat guess a person's education level
# based on other features such as their "work" or maybe their "country".
# Work can relate to one's education level because a certain education level
# may have to be met for a certain job.
# The country can also relate because in different countries, different education levels
# are provided and different education levels can lead to different jobs from another
# country.
```

Out[104]:
```
{'edu': 'HS-grad',
 'marriage': 'Married-civ-spouse',
 'occupation': 'Machine-op-inspct',
 'relationship': 'Husband',
 'sex': 'Male',
 'work': 'Private'}
```

In [109]:
```
test_data[0].size
```

Out[109]:
```
57
```

Part (e) [2 pt]

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

In [116]:
```
excl_edu = test_data[0]

# This is from zero_out_feature
start_index = cat_index["edu"]
stop_index = cat_index["edu"] + len(cat_values["edu"])
excl_edu[start_index:stop_index] = 0

excl_edu = torch.from_numpy(excl_edu)

edu_pred = AutoEncoder5(excl_edu) #AutoEncoder5 was chosen because it was my best model

result = edu_pred.detach().numpy()

result = get_feature(result, "edu")

print(result)
```

```
HS-grad
```

Part (f) [2 pt]

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

The baseline model predicts HS-grad

In [118]:
```
print(most_common["edu"])
```

```
HS-grad
```