Lab 4: Data Imputation using an Autoencoder

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. 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]. 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.

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://colab.research.google.com/drive/1jNVyeRUSFoaanCb-vevH82l_ADS4tVBI?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

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

In [ ]:
```
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.data",
    names=header,
    index_col=False)
```

```
/usr/local/lib/python3.8/dist-packages/pandas/util/_decorators.py:311: ParserWarning:
Length of header or names does not match length of data. This leads to a loss of data
with index_col=False.
  return func(*args, **kwargs)
```

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

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

Out[ ]:

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationship	race	sex	capgain
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0

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

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

Out[ ]:

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

Out[ ]:
```
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 [ ]:
```
print("Max values:")
print(df[['age', "yredu", "capgain", "caploss", "workhr"]].max(axis=0))

print("\nMin values:")
print(df[['age', "yredu", "capgain", "caploss", "workhr"]].min(axis=0))

print("\nAverage values")
print(df[['age', "yredu", "capgain", "caploss", "workhr"]].mean(axis=0))
```

```
Max values:
age           90
yredu         16
capgain    99999
caploss     4356
workhr        99
dtype: int64

Min values:
age        17
yredu       1
capgain     0
caploss     0
workhr      1
dtype: int64

Average values
age          38.581647
yredu        10.080679
capgain    1077.648844
caploss      87.303830
workhr       40.437456
dtype: float64
```

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

Out[ ]:
```
21790
```

In [ ]:
```
percent_male = (sum(df["sex"] == " Male") / df["sex"].shape[0] * 100)
percent_female = (sum(df["sex"] == " Female") / df["sex"].shape[0] * 100)

print(f"There are {percent_male}% male entries")
print(f"There are {percent_female}% female entries")
```

```
There are 66.92054912318419% male entries
There are 33.07945087681583% female entries
```

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

In [ ]:
```
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 [ ]:
```
print(f"There are {df_with_missing.shape[0]} records with missing features")

print(f"{df_with_missing.shape[0] / df.shape[0] * 100}% of records were removed")
```

```
There are 1843 records with missing features
5.660145572924664% of 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 [ ]:
```
print(set(df_not_missing["work"]))
```

```
{' Federal-gov', ' Private', ' Self-emp-inc', ' Local-gov', ' Without-pay', ' State-g
ov', ' Self-emp-not-inc'}
```

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

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

Out[ ]:

	age	yredu	capgain	caploss	workhr	work_ Federal-gov	work_ Local-gov	work_ Private	work_ Self-emp-inc	work_ Self-emp-not-inc	...	edu_ Prof-school	edu_ Some-college
0	39	13	2174	0	40	0	0	0	0	0	...	0	0
1	50	13	0	0	13	0	0	0	0	1	...	0	0
2	38	9	0	0	40	0	0	1	0	0	...	0	0

3 rows × 57 columns

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

57

get_dummies turned all unique categorical data values into new columns using one-hot encoding, where recordings that fit the label will have a 1 in columns where they meet the value and 0 otherwise. Since there were many columns with categorical data values, such as the "work" column, this caused a large number of new columns to be added.

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

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

# 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 underscore
        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.]), "work")
    'Private'
    """
    # <----- TODO: WRITE YOUR CODE HERE ----->
    # You may find the variables `cat_index` and `cat_values`
    # (created above) useful.
    max_idx = np.argmax(onehot)
    return cat_values[feature][max_idx]
```

In [ ]:
```
get_categorical_value(np.array([0.1, 0., 1.1, 0.2, 0., 1., 0.]), "work")
```

Out[ ]:
```
'Private'
```

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

# Split data into sets
datanp_shuffled = datanp.copy()
np.random.shuffle(datanp_shuffled)
training_set = datanp_shuffled[0: int(0.7*datanp.shape[0])]
validation_set = datanp_shuffled[int(0.7*datanp.shape[0]): int(0.7*datanp.shape[0]) +
testing_set = datanp_shuffled[int(0.7*datanp.shape[0]) + int(0.15*datanp.shape[0]) + 

# Return number of items in each set
print(f"There are {training_set.shape[0]} training items")
print(f"There are {validation_set.shape[0]} validation items")
print(f"There are {testing_set.shape[0]} testing items")
```

```
There are 21502 training items
There are 4608 validation items
There are 4608 testing items
```

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 [ ]:
```
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, 35),
            nn.ReLU(),
            nn.Linear(35, 13),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(13, 35),
            nn.ReLU(),
            nn.Linear(35, 57),
            nn.ReLU(),
            nn.Linear(57, 57), # TODO -- FILL OUT THE CODE HERE!
            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.)

We are using one-hot encoding to represent categorical data in the input, so we want the model to output a numerical value between 0 and 1 for each category so we can translate it back into the original categorical data for the output.

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

def plot_curve(iters, data, dtype, stype):
  plt.title(f"{dtype} {stype}")
  plt.plot(iters, data)
  plt.xlabel("Iterations")
  plt.ylabel(stype)
  plt.show()
```

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, 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_loss, val_loss = [], []
    train_acc, val_acc = [], []
    iterations = []

    for epoch in range(num_epochs):
        tot_train_loss = 0
        for data in train_loader:
            data = data.to(device)

            datam = zero_out_random_feature(data.clone()) # zero out one categorical f
            recon = model(datam)
            loss = criterion(recon, data)
            loss.backward()
```

```
            optimizer.step()
            optimizer.zero_grad()
            tot_train_loss += loss.item()

        # Calculate validation loss
        tot_val_loss = 0
        for temp in valid_loader:
            datam = zero_out_random_feature(data.clone())
            recon = model(datam)
            tot_val_loss += criterion(recon, data).item()

        # Loss/Accuracy Processing
        iterations.append(epoch)

        train_loss.append(float(tot_train_loss)/(len(train_loader)*train_loader.batch_
        val_loss.append(float(tot_val_loss)/(len(valid_loader)*valid_loader.batch_size

        train_acc.append(get_accuracy(model, train_loader))
        val_acc.append(get_accuracy(model, valid_loader))

        # Plot stats once every five epochs
        if epoch % 5 == 0:
            plot_curve(iterations, train_loss, "Training", "Loss")
            plot_curve(iterations, train_acc, "Training", "Accuracy")
            plot_curve(iterations, val_loss, "Validation", "Loss")
            plot_curve(iterations, val_acc, "Validation", "Acc")

        # Save model once very 2 epochs
        if epoch % 2 == 0:
            torch.save(model.state_dict(), f"autoencoder_epoch_{epoch}")

        # Training feedback
        print(f"Epoch {epoch}")
        print(f"Training: loss {train_loss[-1]}, training acc: {train_acc[-1]}")
        print(f"Validation: loss {val_loss[-1]}, validation acc: {val_acc[-1]}")

    print("Done")
```

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 [ ]:
```
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, shuffle=True
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            item = item.to(device)
            inp = item.cpu().detach().numpy()
            out = model(zero_out_feature(item.clone(), col)).detach().cpu().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
```

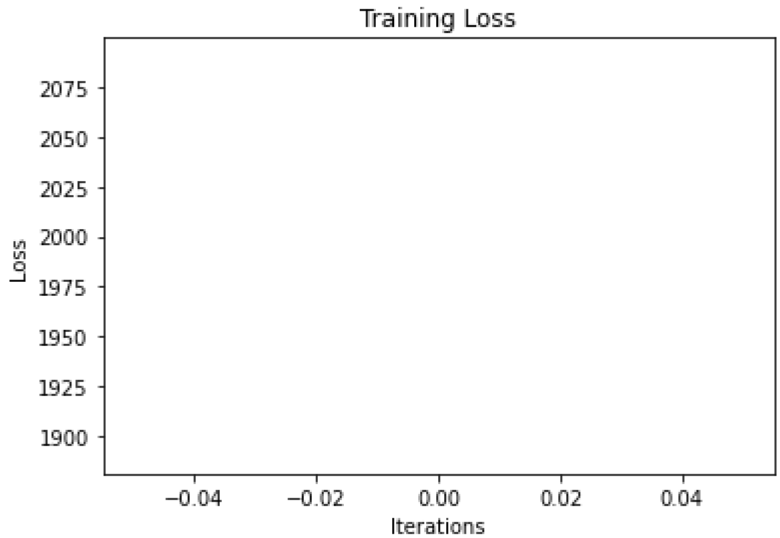
Part (c) [4 pt]

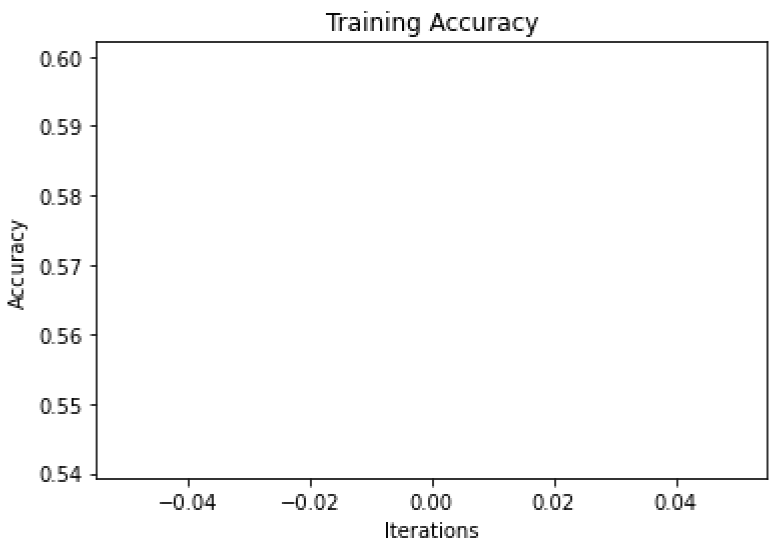
Run your updated training code, using reasonable initial hyperparameters.

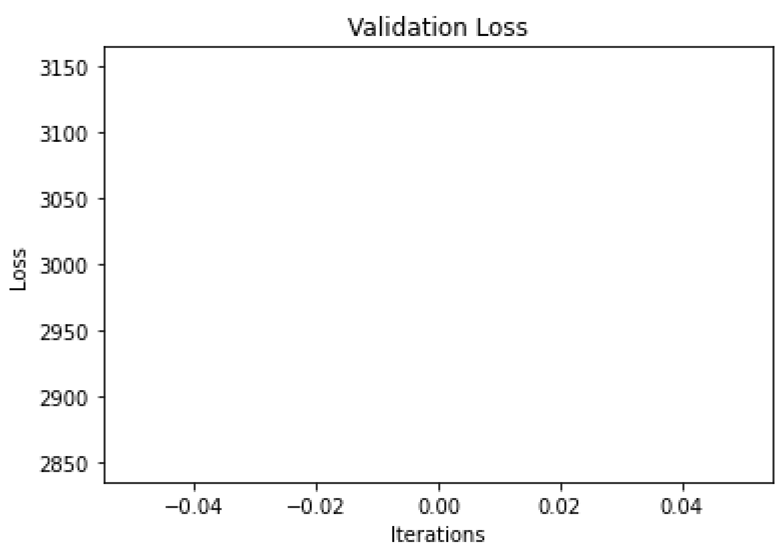
Include your training curve in your submission.

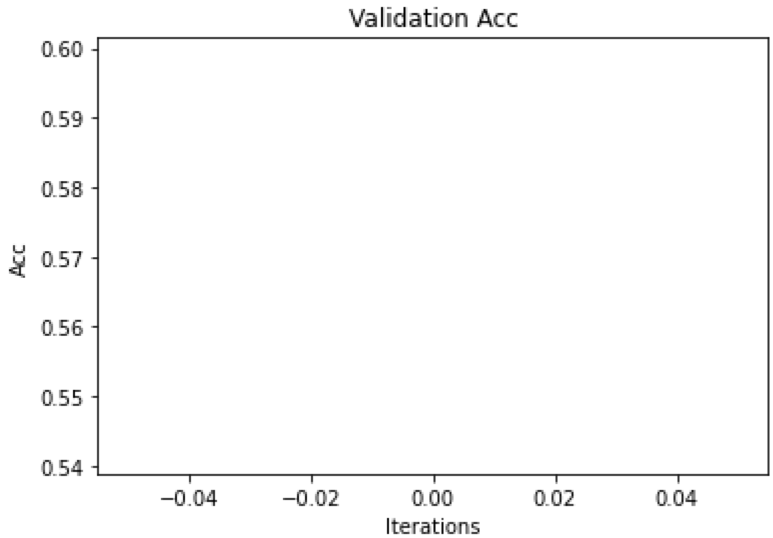
In [ ]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

autoencoder = AutoEncoder()
autoencoder.to(device)

batch_size = 512
train_loader = torch.utils.data.DataLoader(training_set, batch_size=batch_size)
valid_loader = torch.utils.data.DataLoader(validation_set, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(testing_set, batch_size=batch_size)
```

In [ ]:
```
train(autoencoder, train_loader, valid_loader, num_epochs=40, learning_rate=0.01)
```

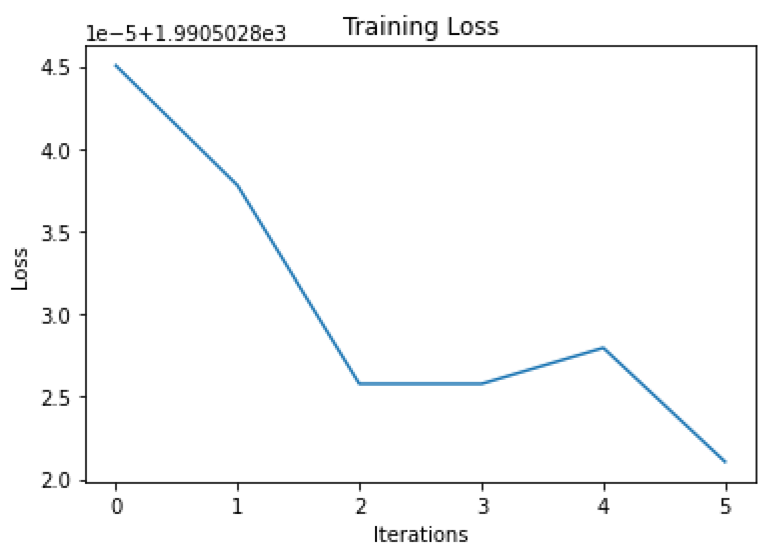

Training Loss
2075
2050
2025
2000
1975
1950
1925
1900
Loss
−0.04
−0.02
0.00
0.02
0.04
Iterations

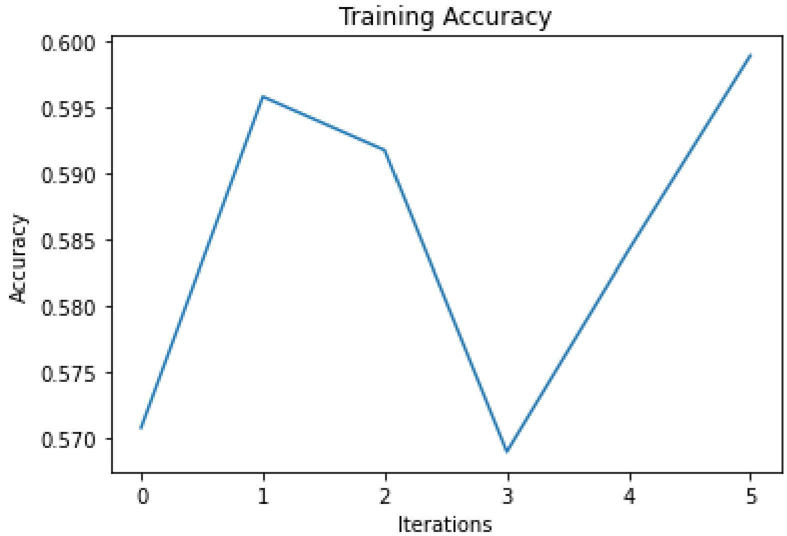

Training Accuracy
0.60
0.59
0.58
0.57
0.56
0.55
0.54
Accuracy
−0.04
−0.02
0.00
0.02
0.04
Iterations

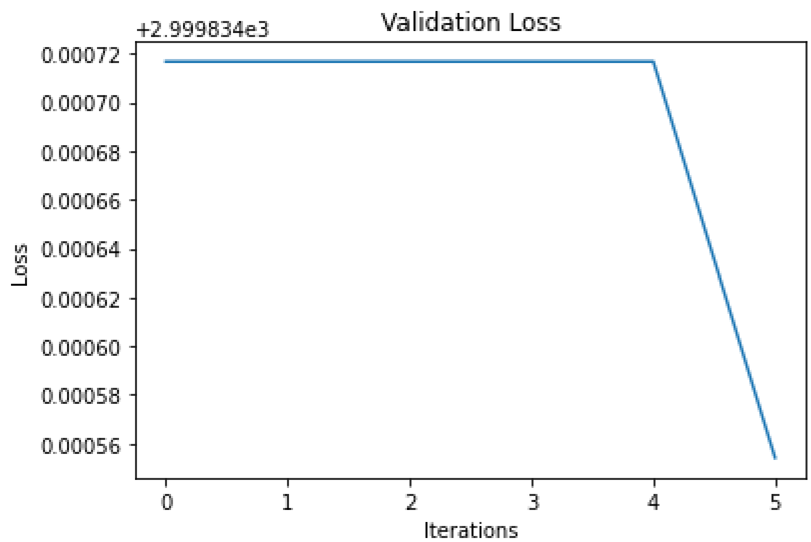

Validation Loss
3150
3100
3050
3000
2950
2900
2850
Loss
−0.04
−0.02
0.00
0.02
0.04
Iterations

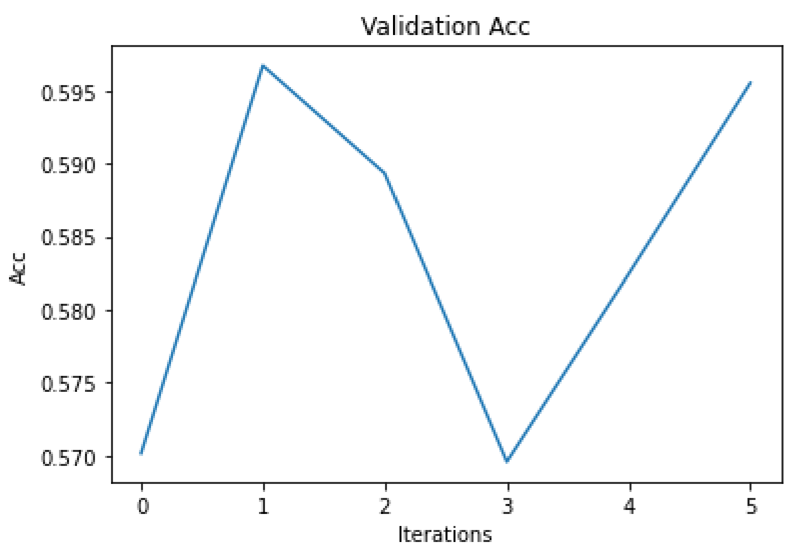



```
Epoch 0
Training: loss 1990.5028450375512, training acc: 0.5707531082379934
Validation: loss 2999.834716796875, validation acc: 0.5701678240740741
Epoch 1
Training: loss 1990.5028377714611, training acc: 0.5957895389576163
Validation: loss 2999.834716796875, validation acc: 0.5967158564814815
Epoch 2
Training: loss 1990.5028257824126, training acc: 0.5917434037143832
Validation: loss 2999.834716796875, validation acc: 0.5893373842592593
Epoch 3
Training: loss 1990.5028257824126, training acc: 0.5689315722568443
Validation: loss 2999.834716796875, validation acc: 0.5695891203703703
Epoch 4
Training: loss 1990.5028279622395, training acc: 0.5841704647629677
Validation: loss 2999.834716796875, validation acc: 0.5824291087962963
```

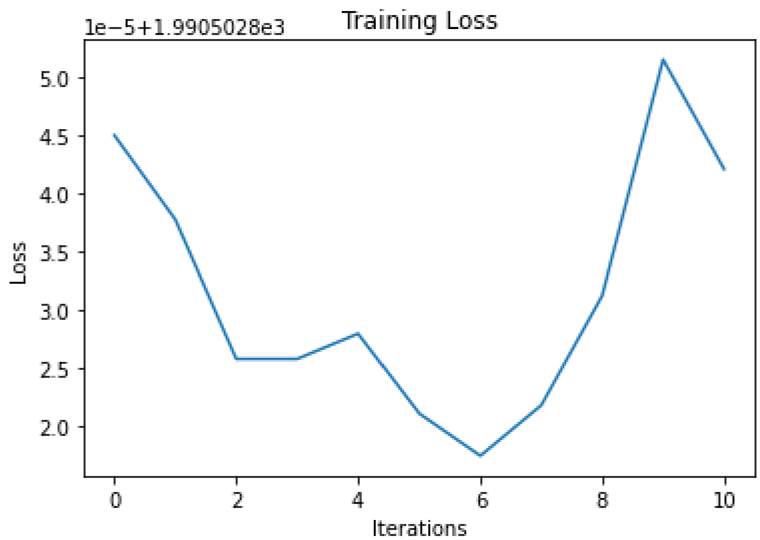


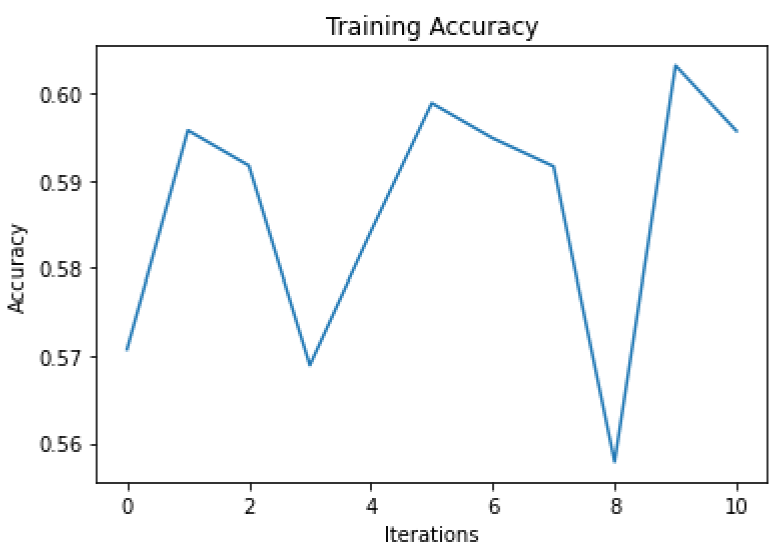

Training Accuracy
0.600
0.595
0.590
0.585
0.580
0.575
0.570
Accuracy
0
1
2
3
4
5
Iterations

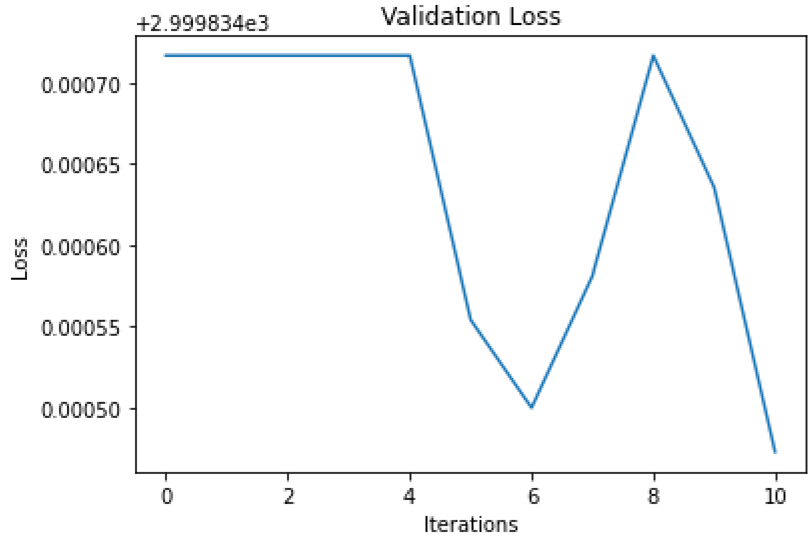

Validation Loss
+2.999834e3
0.00072
0.00070
0.00068
0.00066
0.00064
0.00062
0.00060
0.00058
0.00056
Loss
0
1
2
3
4
5
Iterations

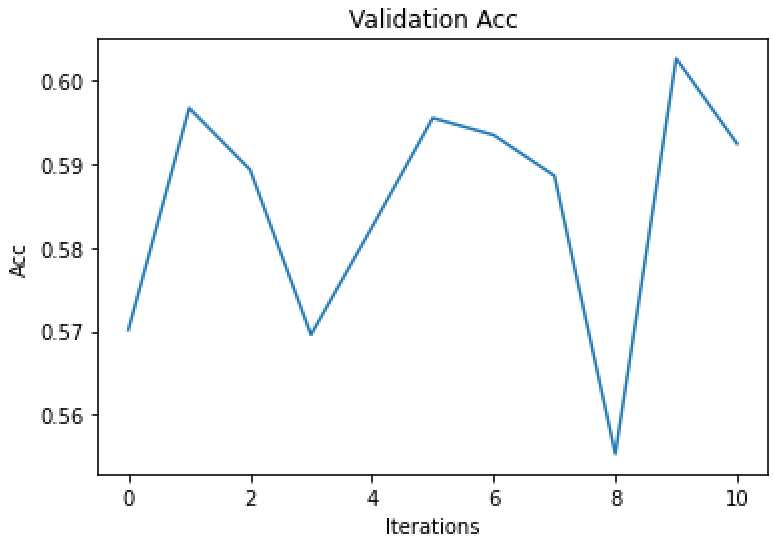

Validation Acc
0.595
0.590
0.585
0.580
0.575
0.570
Acc
0
1
2
3
4
5
Iterations

```
Epoch 5
Training: loss 1990.5028210594542, training acc: 0.5989055281679224
Validation: loss 2999.8345540364585, validation acc: 0.5955222800925926
Epoch 6
Training: loss 1990.5028174264091, training acc: 0.5948903977924535
Validation: loss 2999.8344997829863, validation acc: 0.5934968171296297
Epoch 7
Training: loss 1990.5028217860631, training acc: 0.5916503891110905
Validation: loss 2999.8345811631943, validation acc: 0.5886140046296297
Epoch 8
Training: loss 1990.50283123198, training acc: 0.5578473320311289
Validation: loss 2999.834716796875, validation acc: 0.5553385416666666
Epoch 9
Training: loss 1990.5028515770323, training acc: 0.6032307072210337
Validation: loss 2999.8346354166665, validation acc: 0.6026475694444444
```

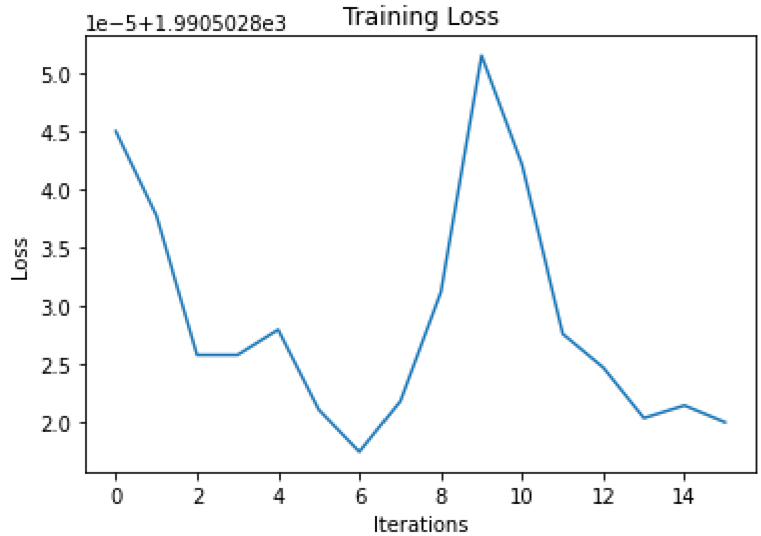




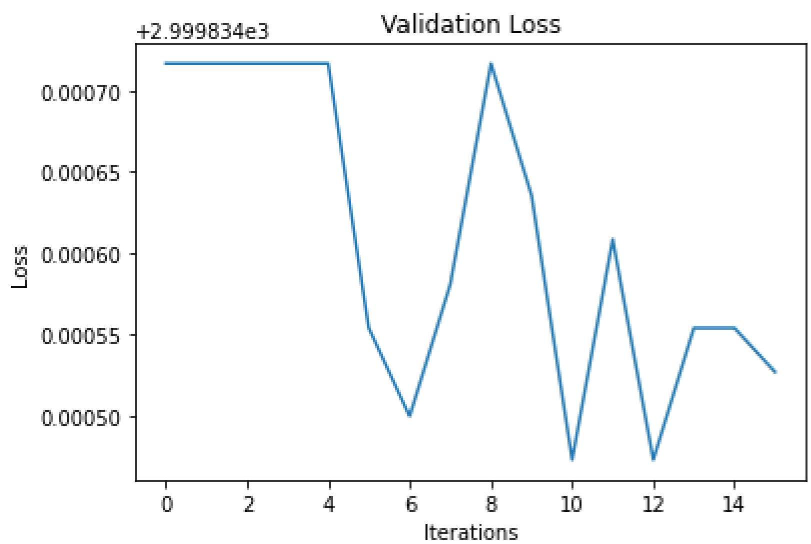


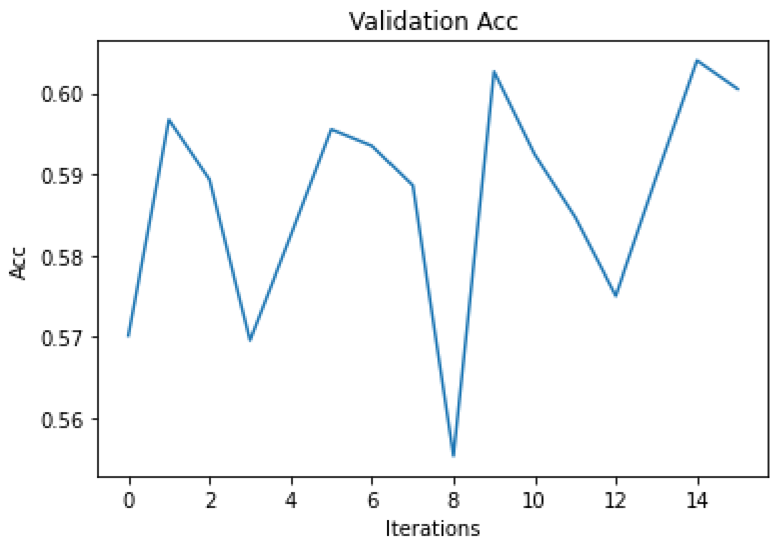



```
Epoch 10
Training: loss 1990.5028421311151, training acc: 0.5957120267882058
Validation: loss 2999.83447265625, validation acc: 0.5924479166666666
Epoch 11
Training: loss 1990.502827598935, training acc: 0.5856354447648281
Validation: loss 2999.8346082899307, validation acc: 0.5847077546296297
Epoch 12
Training: loss 1990.502824692499, training acc: 0.5771246085635445
Validation: loss 2999.83447265625, validation acc: 0.5750506365740741
Epoch 13
Training: loss 1990.502820332845, training acc: 0.5924177595882554
Validation: loss 2999.8345540364585, validation acc: 0.5896990740740741
Epoch 14
Training: loss 1990.5028214227586, training acc: 0.6048739652125383
Validation: loss 2999.8345540364585, validation acc: 0.6040219907407407
```

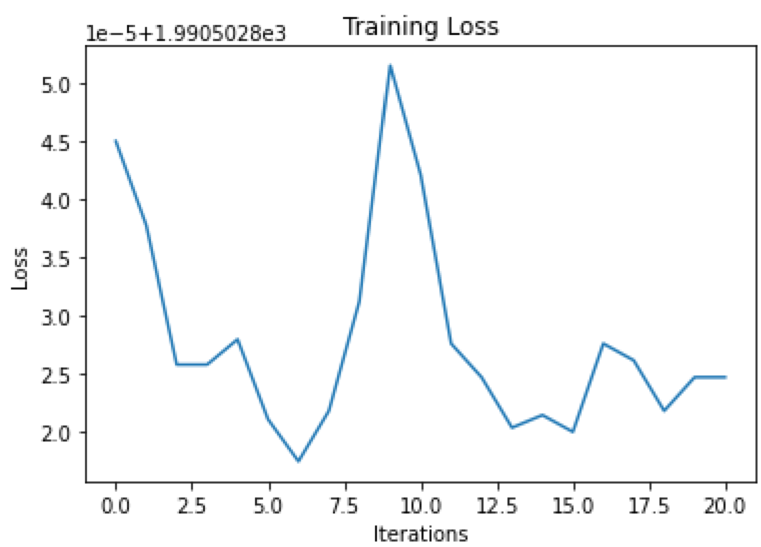

Training Loss
1e−5+1.9905028e3
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Loss
0
2
4
6
8
10
12
14
Iterations

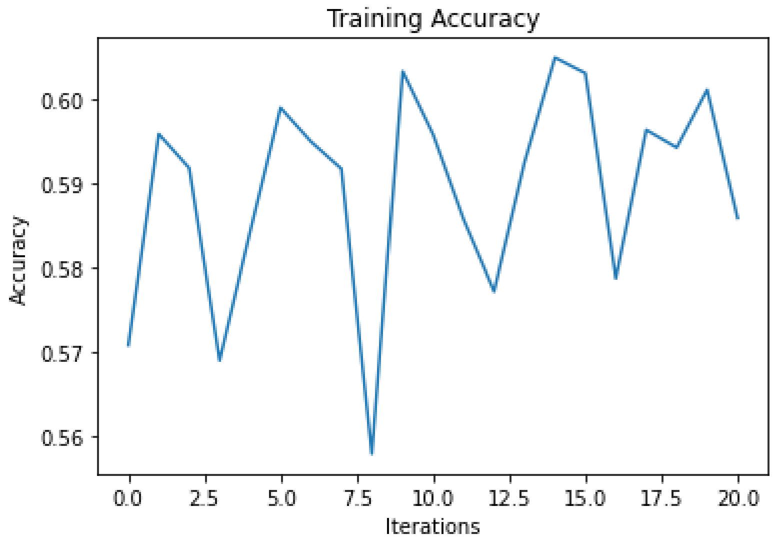

Training Accuracy
0.60
0.59
0.58
0.57
0.56
Accuracy
0
2
4
6
8
10
12
14
Iterations

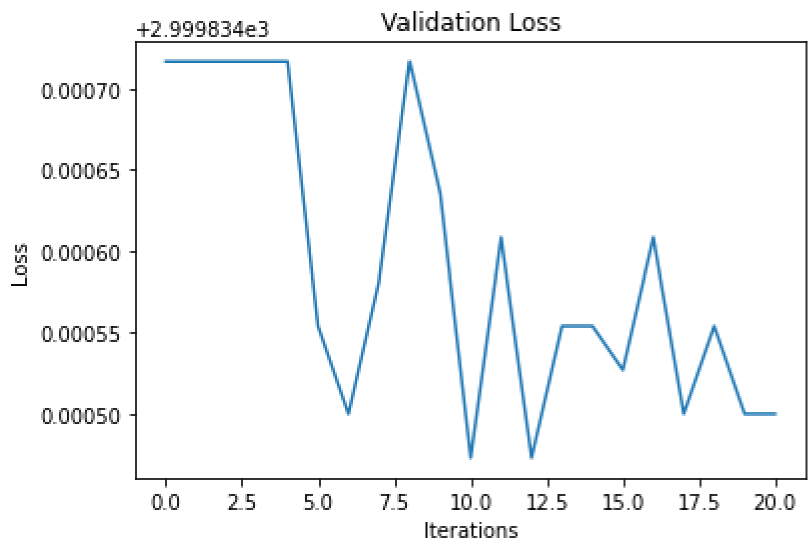

Validation Loss
+2.999834e3
0.00070
0.00065
0.00060
0.00055
0.00050
Loss
0
2
4
6
8
10
12
14
Iterations

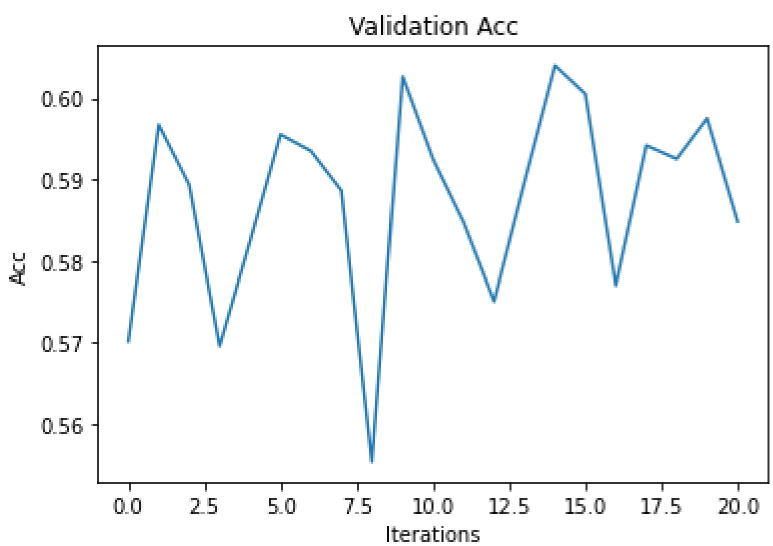



```
Epoch 15
Training: loss 1990.5028199695405, training acc: 0.6030136731466841
Validation: loss 2999.834526909722, validation acc: 0.6005135995370371
Epoch 16
Training: loss 1990.502827598935, training acc: 0.5786748519517564
Validation: loss 2999.8346082899307, validation acc: 0.5770037615740741
Epoch 17
Training: loss 1990.502826145717, training acc: 0.5962778656249031
Validation: loss 2999.8344997829863, validation acc: 0.5941840277777778
Epoch 18
Training: loss 1990.5028217860631, training acc: 0.594177285833876
Validation: loss 2999.8345540364585, validation acc: 0.5925202546296297
Epoch 19
Training: loss 1990.502824692499, training acc: 0.6010448640436549
Validation: loss 2999.8344997829863, validation acc: 0.5975115740740741
```

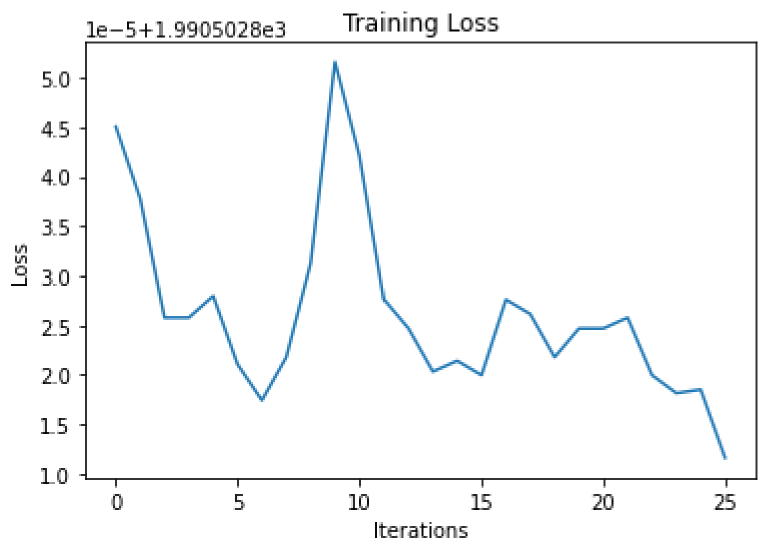


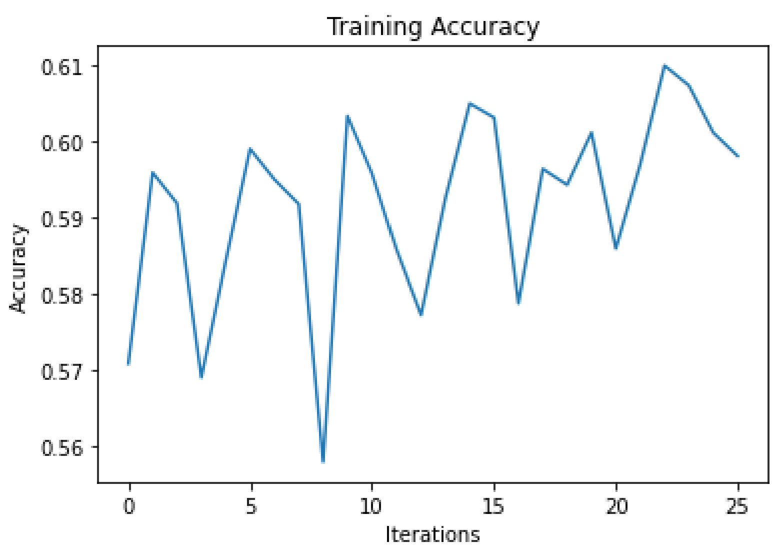

Training Accuracy
Accuracy
Iterations
0.56
0.57
0.58
0.59
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

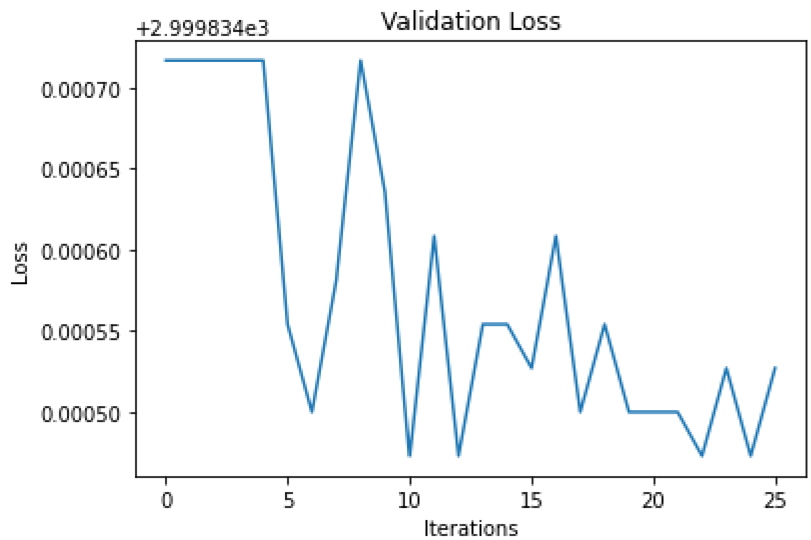

Validation Loss
+2.999834e3
Loss
Iterations
0.00050
0.00055
0.00060
0.00065
0.00070
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

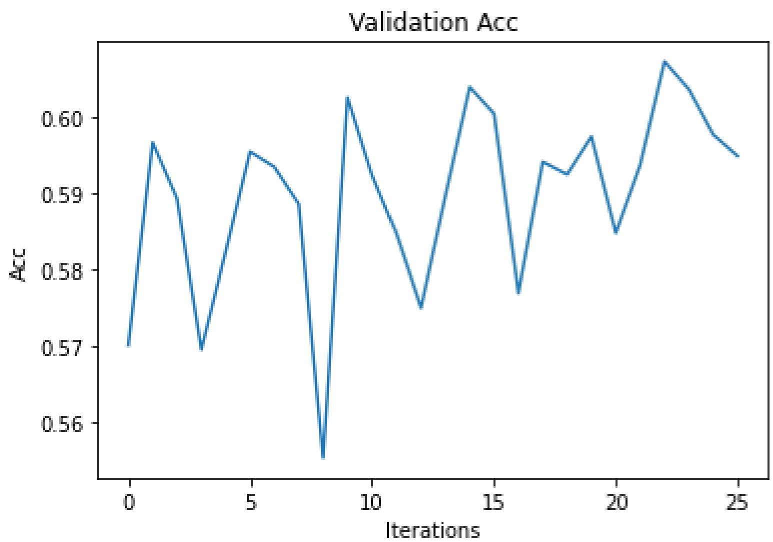

Validation Acc
Acc
Iterations
0.56
0.57
0.58
0.59
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

```
Epoch 20
Training: loss 1990.502824692499, training acc: 0.5857982203205904
Validation: loss 2999.8344997829863, validation acc: 0.5848162615740741
Epoch 21
Training: loss 1990.5028257824126, training acc: 0.5968437044616005
Validation: loss 2999.8344997829863, validation acc: 0.5937861689814815
Epoch 22
Training: loss 1990.5028199695405, training acc: 0.6098579977056398
Validation: loss 2999.83447265625, validation acc: 0.6073857060185185
Epoch 23
Training: loss 1990.502818153018, training acc: 0.6072380863795616
Validation: loss 2999.834526909722, validation acc: 0.6037326388888888
Epoch 24
Training: loss 1990.5028185163226, training acc: 0.6010448640436549
Validation: loss 2999.83447265625, validation acc: 0.5977647569444444
```

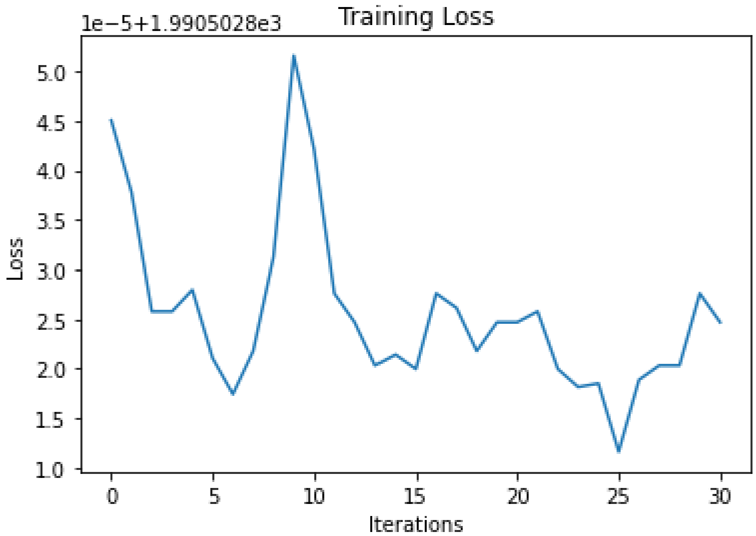


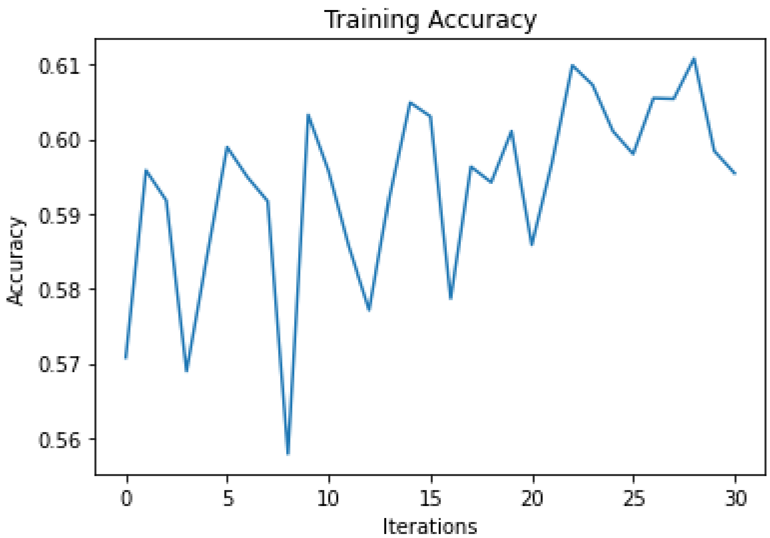


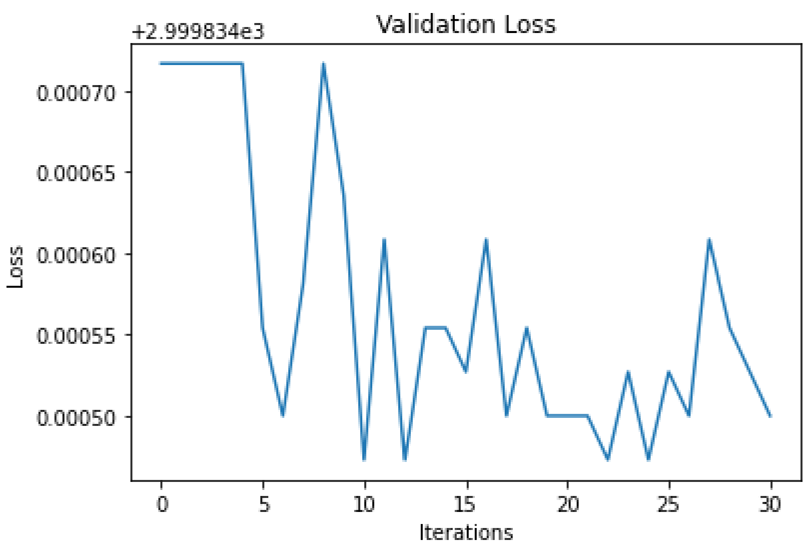


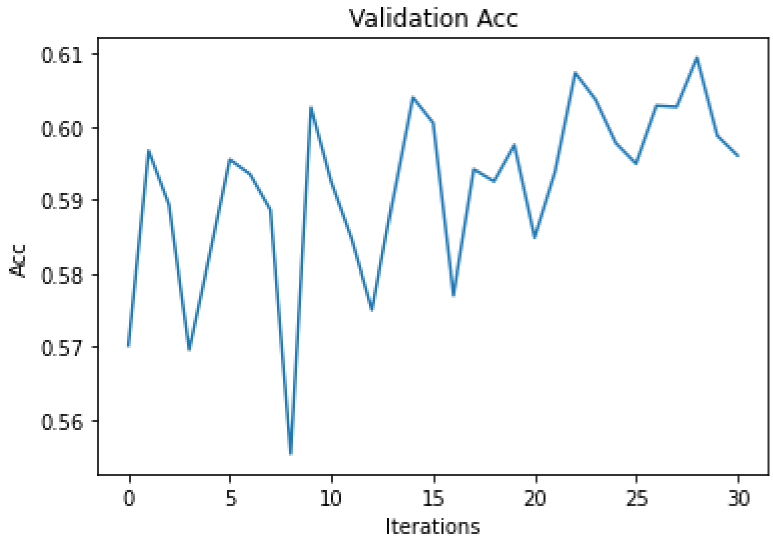



```
Epoch 25
Training: loss 1990.502811613537, training acc: 0.5979753821349952
Validation: loss 2999.834526909722, validation acc: 0.5949435763888888
Epoch 26
Training: loss 1990.502818879627, training acc: 0.6054630577000589
Validation: loss 2999.8344997829863, validation acc: 0.6029007523148148
Epoch 27
Training: loss 1990.502820332845, training acc: 0.6053700430967662
Validation: loss 2999.8346082899307, validation acc: 0.6027199074074074
Epoch 28
Training: loss 1990.502820332845, training acc: 0.6107881437385669
Validation: loss 2999.8345540364585, validation acc: 0.6094835069444444
Epoch 29
Training: loss 1990.502827598935, training acc: 0.5983784454159303
Validation: loss 2999.834526909722, validation acc: 0.5987774884259259
```

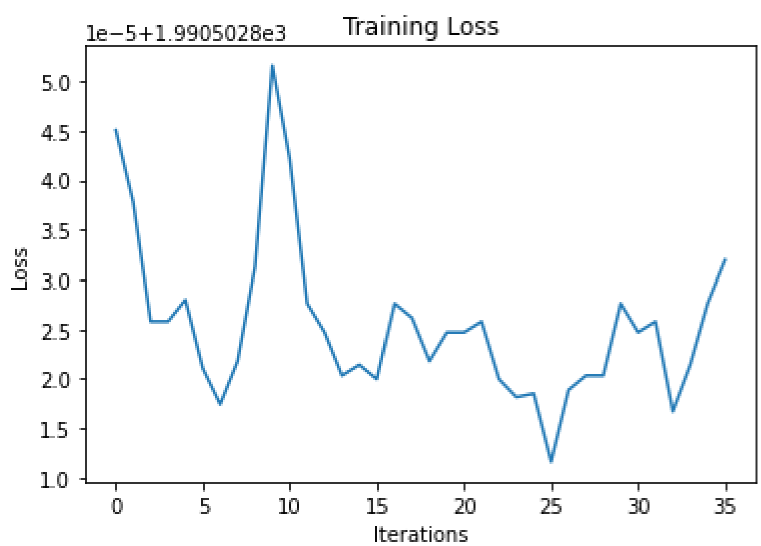

Training Loss
1e−5+1.9905028e3
Loss
Iterations

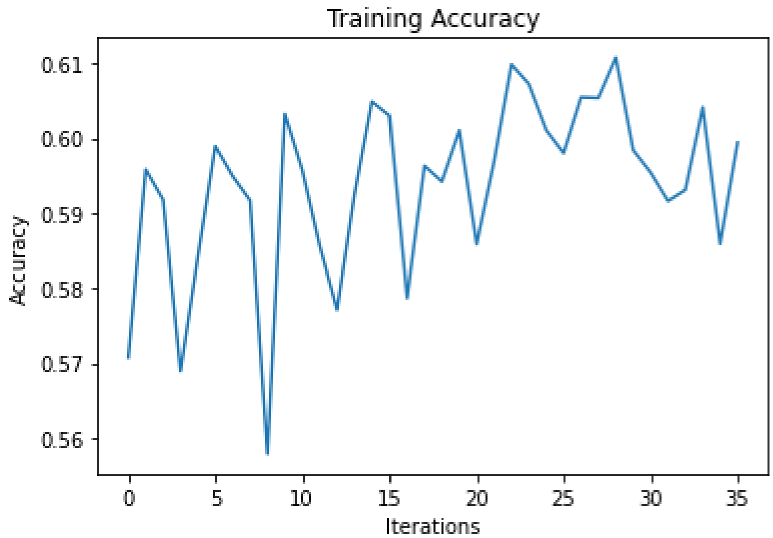

Training Accuracy
Accuracy
Iterations

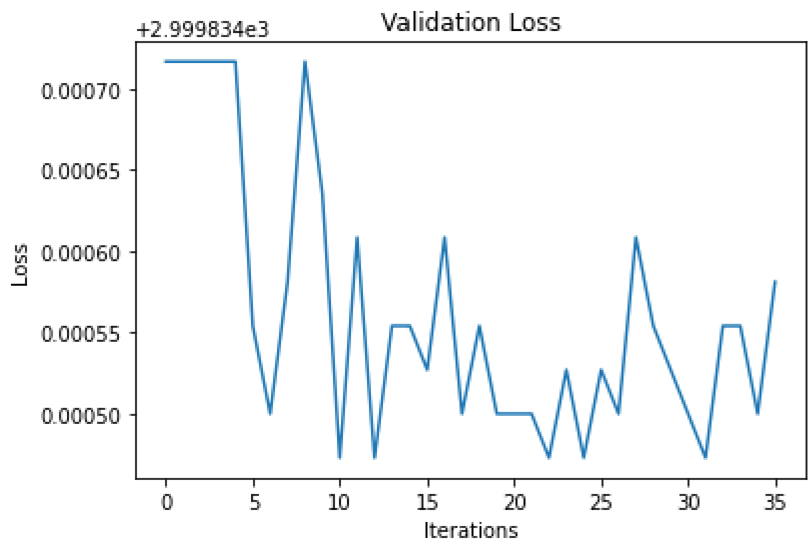

Validation Loss
+2.999834e3
Loss
Iterations

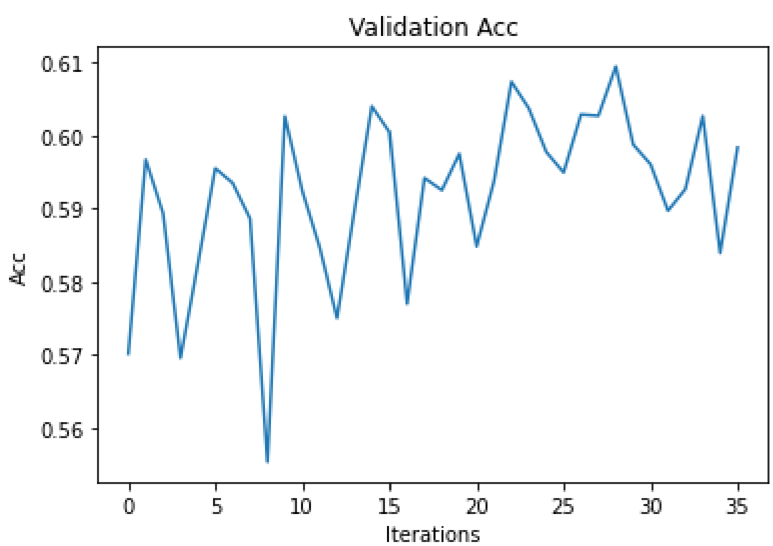



```
Epoch 30
Training: loss 1990.502824692499, training acc: 0.5953942268936223
Validation: loss 2999.8344997829863, validation acc: 0.5960648148148148
Epoch 31
Training: loss 1990.5028257824126, training acc: 0.5915651257247387
Validation: loss 2999.83447265625, validation acc: 0.5896990740740741
Epoch 32
Training: loss 1990.5028166998, training acc: 0.5930611105943633
Validation: loss 2999.8345540364585, validation acc: 0.5926649305555556
Epoch 33
Training: loss 1990.5028214227586, training acc: 0.6041298483861967
Validation: loss 2999.8345540364585, validation acc: 0.6026837384259259
Epoch 34
Training: loss 1990.502827598935, training acc: 0.5858369764052956
Validation: loss 2999.8344997829863, validation acc: 0.5839120370370371
```




Training Accuracy
Accuracy
Iterations
0.56
0.57
0.58
0.59
0.60
0.61
0
5
10
15
20
25
30
35


Validation Loss
+2.999834e3
Loss
Iterations
0.00050
0.00055
0.00060
0.00065
0.00070
0
5
10
15
20
25
30
35


Validation Acc
Acc
Iterations
0.56
0.57
0.58
0.59
0.60
0.61
0
5
10
15
20
25
30
35

```
Epoch 35
Training: loss 1990.502831958589, training acc: 0.599386103618268
Validation: loss 2999.8345811631943, validation acc: 0.5983796296296297
Epoch 36
Training: loss 1990.5028217860631, training acc: 0.6021300344154032
Validation: loss 2999.8344997829863, validation acc: 0.6003327546296297
Epoch 37
Training: loss 1990.502824692499, training acc: 0.6104703438439835
Validation: loss 2999.8344997829863, validation acc: 0.6100260416666666
Epoch 38
Training: loss 1990.502828325544, training acc: 0.5974095432982979
Validation: loss 2999.8345811631943, validation acc: 0.5953052662037037
Epoch 39
Training: loss 1990.5028254191081, training acc: 0.5952159489039779
Validation: loss 2999.834526909722, validation acc: 0.5948350694444444
Done
```

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.

Attempt 1: bs = 1024, lr = 0.001, 20 epochs, 2 hidden layers (35, 11) in encoder and decoder
No significant overfitting but slow increase in training and validation accuracy. Because the accuracies haven't flattened out, the model may not have trained enough to reach max accuracy. Due to this, I will try to increase the learning rate and increase the number of epochs to try and get the model to train more. The training accuracy was 0.48 validation accuracy was 0.46.

Attempt 2: bs = 1024, lr = 0.007, 30 epochs, 2 hidden layers (35, 11) in encoder and decoder
There still wasn't overfitting and the accuracy of both training and validation data increased significantly (0.59 training and 0.59 validation). There is still and upward trend in the accuracies so I'm going to try to increase learning rate and number of epochs a little more a little more. I'm also going to decrease batch size to add a little more noise into the training to counteract any overfitting that may be caused by increasing learning rate and number of epochs.

Attempt 3: bs = , lr = 0.01, 40 epochs, 2 hidden layers (35, 11) in encoder and decoder
The acccuracy was very similar to the previous attempt (0.58 training and 0.58 testing) but this time, the training curve was still very flat. To try and increase accuracy, I'm going to decrease batch_size even more since I think adding more noise and having some more oscillations in the training may lead to a better result. Furthermore, I'm going to increase the number of nodes in the inner hidden layer to allow the model to represent more features, which may help it to decode the encodings better.

Attempt 4: bs = 64, lr = 0.01, 40 epochs, 2 hidden layers (35, 13) in encoder and decoder

The accuracy of the model increased slightly to a maximum of 0.61 for both training and validation on epoch 28. If I were to continue training this model, I would try to decrease batch size further and also tune the hidden layers a little more since it seems like the architecture for encoding the features is limiting the model's accuracy (based on the increase in accuracy from the last attempt, although increasing number of features for an autoencoder feels like cheating).

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [ ]:
```
state = torch.load(f"autoencoder_epoch_{28}")
autoencoder.load_state_dict(state)

test_acc = get_accuracy(autoencoder, test_loader)
print(f"The test accuracy is: {test_acc}")
```

```
The test accuracy is: 0.6057581018518519
```

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 [ ]:
```
copy = df_not_missing.copy()
copy = copy.sample(frac=1).reset_index(drop=True)

most_common = {}
for i in df_not_missing.columns:
    if i in catcols:
        most_common[i] = df_not_missing[i][:int(len(df_not_missing[i])*0.7)].mode().i

print(most_common)
```

```
test_df = df_not_missing[:][int(len(df_not_missing[i])*0.7):]
total, correct = 0, 0
for i in test_df.columns:
    if i in catcols:
        values = test_df[i].tolist()
        for j in range(len(values)):
            if values[j] == most_common[i]:
                correct += 1
            total += 1

print(f"The accuracy is {correct/total}")
```

```
{'work': ' Private', 'marriage': ' Married-civ-spouse', 'occupation': ' Prof-specialt
y', 'edu': ' HS-grad', 'relationship': ' Husband', 'sex': ' Male'}
The accuracy is 0.4607566550925926
```

Part (c) [1 pt]

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

The test accuracy from part A (0.6057) is significantly higher than the test accuracy from part B (0.46075).

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 [ ]:
```
get_features(testing_set[0])

# Yes. While there is no conclusive evidence of the education level of the
# person, it is reasonable to suspect that they have an university level
# education based on their occupation.
```

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

Part (e) [2 pt]

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

In [ ]:
```
data = torch.Tensor(testing_set[0]).view(1, 57)
data = data.to(device)
out = autoencoder(zero_out_feature(data, "edu")).detach().cpu().numpy()
out = get_feature(out[0], "edu")
print(f"My model predicts this person is a {out}")
```

```
My model predicts this person is a HS-grad
```

Part (f) [2 pt]

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

The most common level of education is high school graduate, so the baseline model predicts that they are a high school graduate.