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://drive.google.com/file/d/1o_-CvM75xUKrnFLI2Z1OFB83WoPX1Ph9/view?usp=sharing

In [62]:
```
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 [63]:
```
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 [64]:
```
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)
```

```
C:\Users\hossa\AppData\Local\Temp\ipykernel_27088\1831985018.py:3: ParserWarning: Length of header or names does not match len
gth of data. This leads to a loss of data with index_col=False.
  df = pd.read_csv(
```

In [65]:
```
df.shape # there are 32561 rows (records) in the data frame, and 14 columns (features)
```

Out[65]:
```
(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 [66]:
```
df[:3] # show the first 3 records
```

Out[66]:

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationship	race	sex	capgain	caploss	workhr	country
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States

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

In [67]:
```
subdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
subdf[:3] # show the first 3 records
```

Out[67]:

	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 [68]:
```
np.sum(subdf["caploss"])
```

Out[68]:
```
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 [ ]:
```

```

In [69]:
```
# minimum, maximum, and average value
for cat in subdf:
  print(cat)
  print('min of',cat,":", (df[cat]).min())
```

```
  print('max of',cat,":", (df[cat]).max())
  print('avg of',cat,":", (df[cat]).mean())
```

```
age
min of age : 17
max of age : 90
avg of age : 38.58164675532078
yredu
min of yredu : 1
max of yredu : 16
avg of yredu : 10.0806793403151
capgain
min of capgain : 0
max of capgain : 99999
avg of capgain : 1077.6488437087312
caploss
min of caploss : 0
max of caploss : 4356
avg of caploss : 87.303829734959
workhr
min of workhr : 1
max of workhr : 99
avg of workhr : 40.437455852092995
```

In [70]:
```
for cat in subdf:
    df[cat] = (subdf[cat] - subdf[cat].min())/(subdf[cat].max() - subdf[cat].min())
df[:3]
```

Out[70]:

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationship	race	sex	capgain	caploss	workhr	country
0	0.301370	State-gov	77516	Bachelors	0.800000	Never-married	Adm-clerical	Not-in-family	White	Male	0.02174	0.0	0.397959	United-States
1	0.452055	Self-emp-not-inc	83311	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husband	White	Male	0.00000	0.0	0.122449	United-States
2	0.287671	Private	215646	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-in-family	White	Male	0.00000	0.0	0.397959	United-States

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 [71]:
```
print(sum(df["sex"] == " Male")/df["sex"].size*100, "% are male.")
print(sum(df["sex"] == " Female")/df["sex"].size*100, "% are female.")
```

```
66.92054912318419 % are male.
33.07945087681583 % are female.
```

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 [72]:
```
contcols = ["age", "yredu", "capgain", "caploss", "workhr"]
catcols = ["work", "marriage", "occupation", "edu", "relationship", "sex"]
features = contcols + catcols
df = df[features]
```

In [73]:
```
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 [74]:
```
print(df_with_missing.shape[0], "records that contain missing features.")
print("Therefore,", df_with_missing.shape[0]/df.shape[0]*100, "% of records were removed.")
```

```
1843 records that contain missing features.
Therefore, 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 [75]:
```
print("Values of \"work\":")
for work in set(df_not_missing["work"]):
    print(work)
```

```
Values of "work":
 Without-pay
 Federal-gov
 Local-gov
 State-gov
 Self-emp-inc
 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 [76]:
```
data = pd.get_dummies(df_not_missing)
```

In [77]:
```
data[:3]
```

Out[77]:

	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	relationship_ Husband	relationship_ Not-in-family	relationsh Oth relat
0	0.301370	0.800000	0.02174	0.0	0.397959	0	0	0	0	0	...	0	0	0	1	
1	0.452055	0.800000	0.00000	0.0	0.122449	0	0	0	0	1	...	0	0	1	0	
2	0.287671	0.533333	0.00000	0.0	0.397959	0	0	1	0	0	...	0	0	0	1	

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 [78]:
```
data.shape
# 57 columns
```

Out[78]:
```
(30718, 57)
```

In [79]:
```
# Each of the columns in "catcols" above have a specific number of possible values each which becomes their own column with va
# Then we add the columns in "contcols" above
```

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 [80]:
```
datanp = data.values.astype(np.float32)
```

In [81]:
```
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 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]
```

In [82]:
```
cat_index
```

Out[82]:
```
{'work': 5,
 'marriage': 12,
 'occupation': 19,
 'edu': 33,
 'relationship': 49,
 'sex': 55}
```

In [83]:
```
cat_values
```

Out[83]:
```
{'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']}
```

In [84]:
```python
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., 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.
    rounded_one_hot = np.argmax(onehot)
    return cat_values[feature][rounded_one_hot]
```

In [85]:
```python
# 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 [86]:
```
# set the numpy seed for reproducibility
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html
np.random.seed(50)
np.random.shuffle(datanp)

train_data, val_data, test_data = datanp[:int(len(datanp)*0.7)], datanp[int(len(datanp)*0.7):int(len(datanp)*0.85)], datanp[in

print("Training set:", train_data.shape[0])
print("Validation set:", val_data.shape[0])
print("Test set:", test_data.shape[0])
```

```
Training set: 21502
Validation set: 4608
Test set: 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 [87]:
```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self, name="AutoEncoder"):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(57, 28), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(28, 15)
        )
        self.decoder = nn.Sequential(
            nn.Linear(15, 28), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(28,57),
            nn.Sigmoid() # get to the range (0, 1)
        )
        self.name = name

    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 [88]:
```
# Since the values inside data are all between 0 and 1,
# the sigmoid activation function scales the output to between 0 to 1 which is all we need
```

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 [93]:
```
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 get_model_name(name, learning_rate, epoch):
    path = "model_{0}_lr{1}_epoch{2}".format(name,
                                            learning_rate,
                                            epoch)
    return path

# COPIED FROM BELOW
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
            inp = item.detach().numpy()
            out = model(zero_out_feature(item.clone(), col)).detach().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

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_accuracy = np.zeros(num_epochs)
    val_accuracy = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)

    for epoch in range(num_epochs):
        total_epoch_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_epoch_loss += loss.item()

        train_accuracy[epoch] = get_accuracy(model, train_loader)
        val_accuracy[epoch] = get_accuracy(model, valid_loader)
        train_loss[epoch] = float(total_epoch_loss) / len(train_loader)

        total_epoch_loss = 0.0
        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_epoch_loss += loss.item()
        val_loss[epoch] = float(total_epoch_loss) / len(valid_loader)

        print(("Epoch {}: Train accuracy: {} | Validation accuracy: {} | Train Loss: {} | Validation Loss: {}")
              .format(
                    epoch + 1,
                    train_accuracy[epoch],
                    val_accuracy[epoch],
                    train_loss[epoch],
                    val_loss[epoch]))
        # Save the current model (checkpoint) to a file
        model_path = get_model_name(model.name, learning_rate, epoch)
        torch.save(model.state_dict(), "./testing/"+model_path)
    print('Finished Training')

    import matplotlib.pyplot as plt

    plt.plot(np.arange(1, num_epochs + 1), train_accuracy, val_accuracy)
    plt.title("Training Curve (Default Parameters)")
    plt.xlabel("Epochs")
    plt.ylabel("Training Accuracy")
    plt.show()

    plt.plot(np.arange(1, num_epochs + 1), train_loss, val_loss)
    plt.title("Validation Curve (Default Parameters)")
    plt.xlabel("Epochs")
    plt.ylabel("Validation Accuracy")
    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 [90]:
```
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
            inp = item.detach().numpy()
            out = model(zero_out_feature(item.clone(), col)).detach().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]

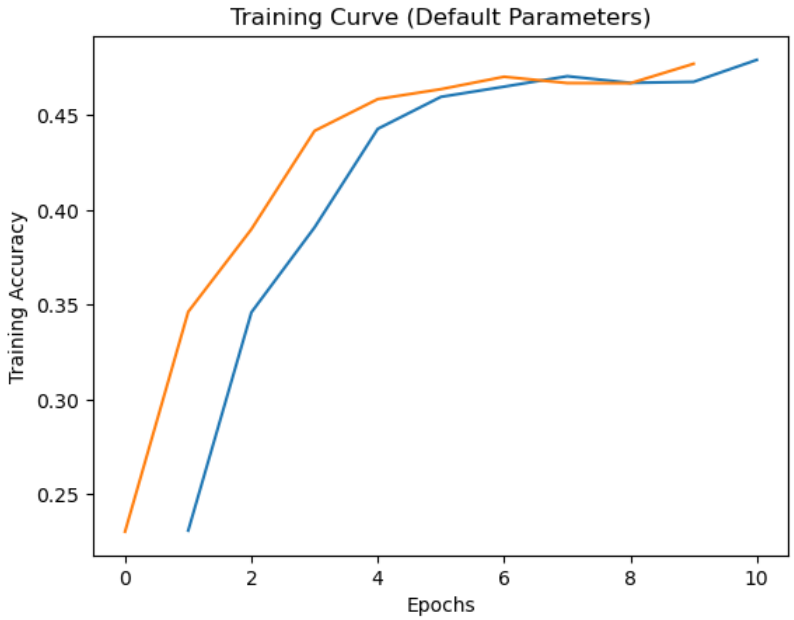
Run your updated training code, using reasonable initial hyperparameters.

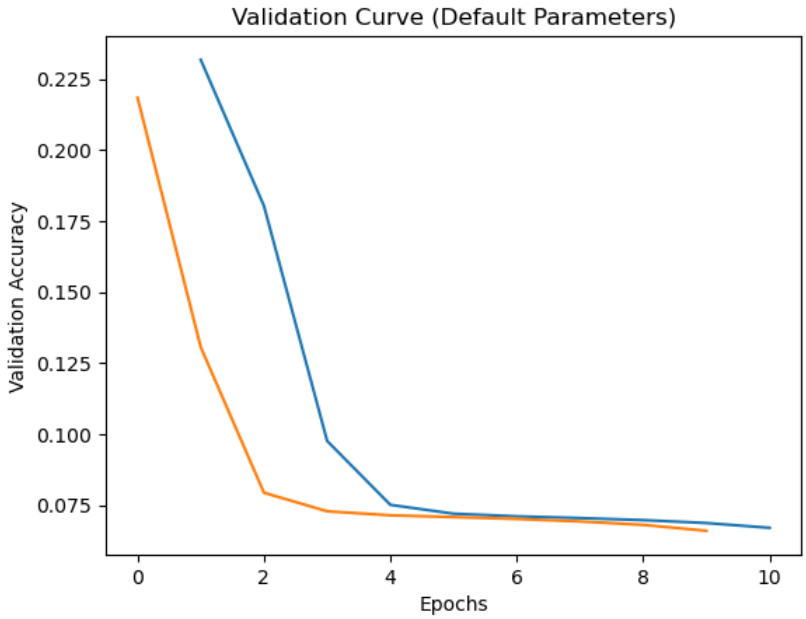
Include your training curve in your submission.

In [94]:
```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=128, shuffle=True)
model = AutoEncoder()

train(model, train_loader, valid_loader, num_epochs=10)
```

```
Epoch 1: Train accuracy: 0.23089325024028773 | Validation accuracy: 0.23036024305555555 | Train Loss: 0.23173854127526283 | Va
lidation Loss: 0.21840135380625725
Epoch 2: Train accuracy: 0.3459755681642018 | Validation accuracy: 0.3462818287037037 | Train Loss: 0.18030919152356328 | Vali
dation Loss: 0.13057960818211237
Epoch 3: Train accuracy: 0.39078535330046815 | Validation accuracy: 0.3898292824074074 | Train Loss: 0.09758800973317452 | Val
idation Loss: 0.07936270804040962
Epoch 4: Train accuracy: 0.4427495116733327 | Validation accuracy: 0.44169560185185186 | Train Loss: 0.0750944045416656 | Vali
dation Loss: 0.07282672284377946
Epoch 5: Train accuracy: 0.4596239109540198 | Validation accuracy: 0.4584418402777778 | Train Loss: 0.07197805436416752 | Vali
dation Loss: 0.07141350209712982
Epoch 6: Train accuracy: 0.46500325551111527 | Validation accuracy: 0.4636863425925926 | Train Loss: 0.0710528235705126 | Vali
dation Loss: 0.0707849216543966
Epoch 7: Train accuracy: 0.4705221219731498 | Validation accuracy: 0.47023292824074076 | Train Loss: 0.07043877328258186 | Val
idation Loss: 0.07013434771862295
Epoch 8: Train accuracy: 0.4670030694819087 | Validation accuracy: 0.4669053819444444 | Train Loss: 0.06972765860458215 | Vali
dation Loss: 0.06925964086420006
Epoch 9: Train accuracy: 0.4675534058847239 | Validation accuracy: 0.46683304398148145 | Train Loss: 0.0686967470461414 | Vali
dation Loss: 0.06804291531443596
Epoch 10: Train accuracy: 0.47913372399466714 | Validation accuracy: 0.47706886574074076 | Train Loss: 0.06701033835166267 | V
alidation Loss: 0.06597270019766358
Finished Training
```





Part (d) [5 pt]

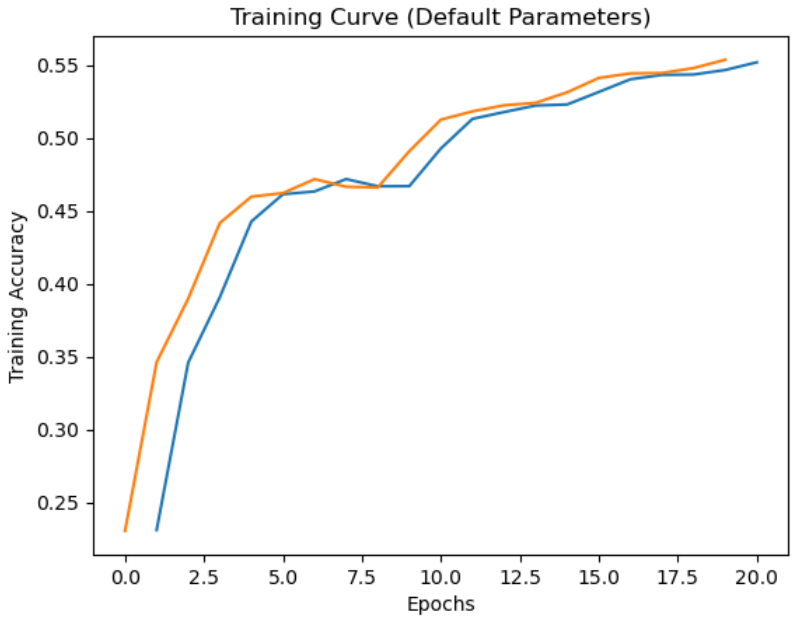
Tune your hyperparameters, training at least 4 different models (4 sets of hyperparameters).

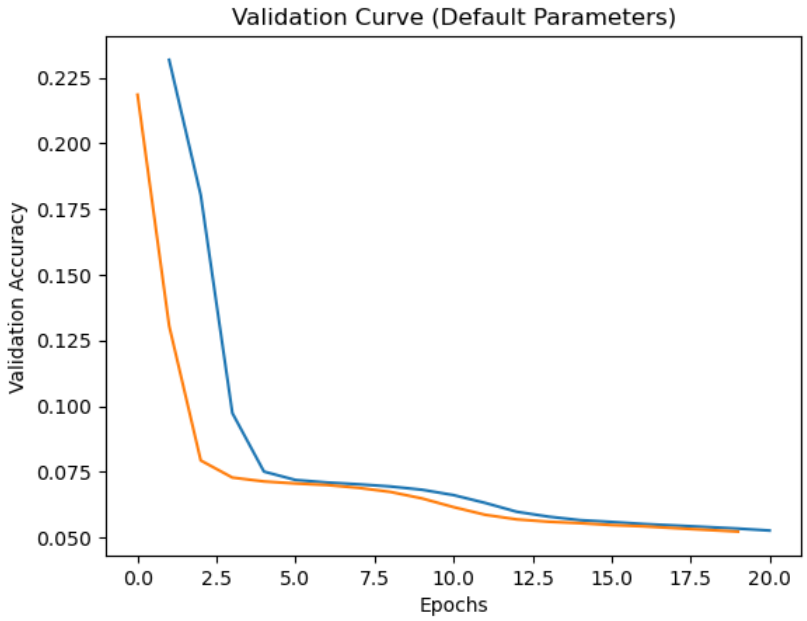
Do not include all your training curves. Instead, explain what hyperparameters you tried, what their effect was, and what your thought process was as you chose the next set of hyperparameters to try.

In [95]:
```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=128, shuffle=True)
model = AutoEncoder()

# Increasing num of epoch
train(model, train_loader, valid_loader, num_epochs=20)
```

```
Epoch 1: Train accuracy: 0.23112578674851952 | Validation accuracy: 0.23061342592592593 | Train Loss: 0.23179208771103904 | Validation Loss: 0.21853798710637623
Epoch 2: Train accuracy: 0.3459755681642018 | Validation accuracy: 0.3462818287037037 | Train Loss: 0.1803058941981622 | Validation Loss: 0.13058099523186684
Epoch 3: Train accuracy: 0.39078535330046815 | Validation accuracy: 0.3898292824074074 | Train Loss: 0.09737090074590274 | Validation Loss: 0.07932573701772425
Epoch 4: Train accuracy: 0.4427495116733327 | Validation accuracy: 0.44169560185185186 | Train Loss: 0.07508591668946403 | Validation Loss: 0.07281504530045721
Epoch 5: Train accuracy: 0.4614531981521099 | Validation accuracy: 0.4597800925925926 | Train Loss: 0.07189239769996632 | Validation Loss: 0.07133212809761365
Epoch 6: Train accuracy: 0.4633754999534927 | Validation accuracy: 0.4622395833333333 | Train Loss: 0.07093120233288833 | Validation Loss: 0.0705599890400966
Epoch 7: Train accuracy: 0.4718398288531299 | Validation accuracy: 0.4717881944444444 | Train Loss: 0.07025229141470932 | Validation Loss: 0.06996717676520348
Epoch 8: Train accuracy: 0.4669100548786159 | Validation accuracy: 0.4665798611111111 | Train Loss: 0.0694275860719028 | Validation Loss: 0.06888577548993959
Epoch 9: Train accuracy: 0.4669798158310855 | Validation accuracy: 0.46607349537037035 | Train Loss: 0.06818088192847513 | Validation Loss: 0.06735002994537354
Epoch 10: Train accuracy: 0.4928378755464608 | Validation accuracy: 0.4909939236111111 | Train Loss: 0.06613961638261874 | Validation Loss: 0.06487766125549872
Epoch 11: Train accuracy: 0.5131460639320373 | Validation accuracy: 0.5126229745370371 | Train Loss: 0.0631646327230902 | Validation Loss: 0.061565780287815466
Epoch 12: Train accuracy: 0.5178122965305553 | Validation accuracy: 0.5183015046296297 | Train Loss: 0.059795014848489134 | Validation Loss: 0.058645177736050554
Epoch 13: Train accuracy: 0.5222847487055468 | Validation accuracy: 0.5224609375 | Train Loss: 0.05794806969130323 | Validation Loss: 0.056899129413068295
Epoch 14: Train accuracy: 0.5230288655318885 | Validation accuracy: 0.5241247106481481 | Train Loss: 0.056609215769207195 | Validation Loss: 0.05600383163740238
Epoch 15: Train accuracy: 0.5316017114687006 | Validation accuracy: 0.5313946759259259 | Train Loss: 0.05592307539301969 | Validation Loss: 0.05545360646728012
Epoch 16: Train accuracy: 0.5402908256596286 | Validation accuracy: 0.5412326388888888 | Train Loss: 0.055152129204500286 | Validation Loss: 0.05473851195226113
Epoch 17: Train accuracy: 0.5433060490497008 | Validation accuracy: 0.5443070023148148 | Train Loss: 0.05456909661491712 | Validation Loss: 0.054252630513575345
Epoch 18: Train accuracy: 0.5435618392087558 | Validation accuracy: 0.5445963541666666 | Train Loss: 0.053992575810601316 | Validation Loss: 0.053559082456760936
Epoch 19: Train accuracy: 0.5466390723343565 | Validation accuracy: 0.5479962384259259 | Train Loss: 0.05339376029691526 | Validation Loss: 0.05289715218047301
Epoch 20: Train accuracy: 0.5519409047220414 | Validation accuracy: 0.5536747685185185 | Train Loss: 0.052650806649277605 | Validation Loss: 0.05224224925041199
Finished Training
```

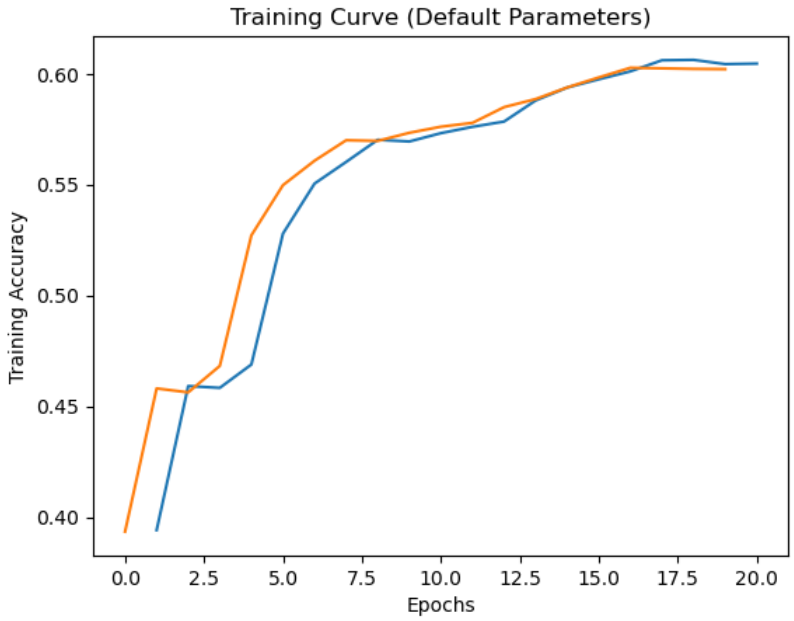


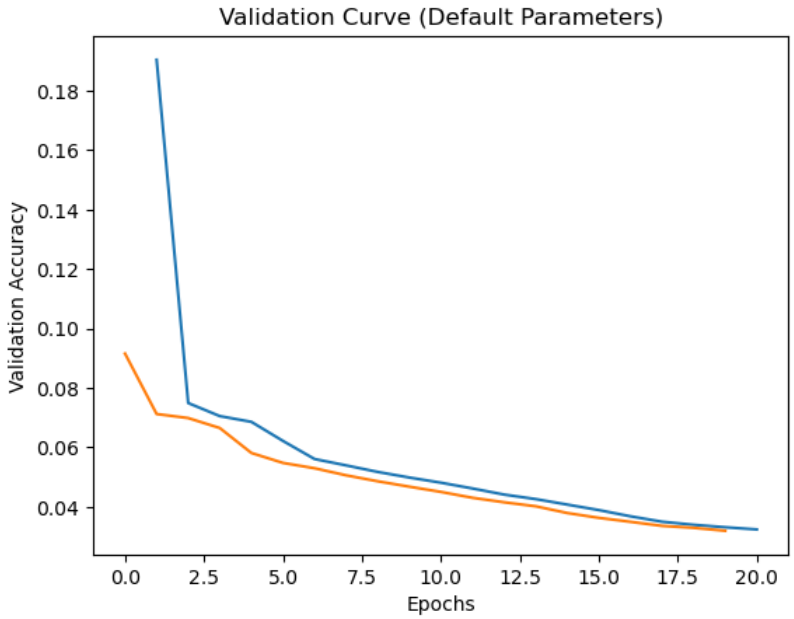


In [96]:
```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=128, shuffle=True)
model = AutoEncoder()

# Increasing learning rate
train(model, train_loader, valid_loader, num_epochs=20, learning_rate=3e-4)
```

```
Epoch 1: Train accuracy: 0.39408737171735964 | Validation accuracy: 0.39341001157407407 | Train Loss: 0.19045666760454574 | Validation Loss: 0.09156873760124047
Epoch 2: Train accuracy: 0.45907357455120457 | Validation accuracy: 0.4580078125 | Train Loss: 0.07493137107008979 | Validation Loss: 0.07123941948844327
Epoch 3: Train accuracy: 0.4582751992062754 | Validation accuracy: 0.4562355324074074 | Train Loss: 0.07053174282468501 | Validation Loss: 0.06987463289664851
Epoch 4: Train accuracy: 0.46881685424611663 | Validation accuracy: 0.4681712962962963 | Train Loss: 0.06859656587420475 | Validation Loss: 0.06651947668029203
Epoch 5: Train accuracy: 0.5278811273369919 | Validation accuracy: 0.5270905671296297 | Train Loss: 0.06217730242670292 | Validation Loss: 0.05812723655253649
Epoch 6: Train accuracy: 0.5504449198524168 | Validation accuracy: 0.5497323495370371 | Train Loss: 0.056087261553676354 | Validation Loss: 0.054776852019131184
Epoch 7: Train accuracy: 0.5602114531981521 | Validation accuracy: 0.5607638888888888 | Train Loss: 0.05394395919782775 | Validation Loss: 0.05298221514870723
Epoch 8: Train accuracy: 0.570195020618237 | Validation accuracy: 0.5699869791666666 | Train Loss: 0.051744665063562845 | Validation Loss: 0.050604814663529396
Epoch 9: Train accuracy: 0.5694276501410721 | Validation accuracy: 0.5696976273148148 | Train Loss: 0.04985856375701371 | Validation Loss: 0.048598116988109216
Epoch 10: Train accuracy: 0.5731869903574861 | Validation accuracy: 0.5733506944444444 | Train Loss: 0.04812573184747072 | Validation Loss: 0.04678121633413765
Epoch 11: Train accuracy: 0.5760394381917961 | Validation accuracy: 0.5760995370370371 | Train Loss: 0.046178109322985016 | Validation Loss: 0.04502148895214001
Epoch 12: Train accuracy: 0.5784268130096425 | Validation accuracy: 0.5778356481481481 | Train Loss: 0.04412376128935388 | Validation Loss: 0.04303891853325897
Epoch 13: Train accuracy: 0.5879453074132639 | Validation accuracy: 0.5849247685185185 | Train Loss: 0.042604611587843726 | Validation Loss: 0.04152821625272433
Epoch 14: Train accuracy: 0.5937742225529408 | Validation accuracy: 0.5885778356481481 | Train Loss: 0.04075547154726727 | Validation Loss: 0.04018411516315407
Epoch 15: Train accuracy: 0.5975025579015906 | Validation accuracy: 0.5938223379629629 | Train Loss: 0.03890005403774835 | Validation Loss: 0.037915668450295925
Epoch 16: Train accuracy: 0.6010758689114191 | Validation accuracy: 0.5983434606481481 | Train Loss: 0.03681233021918507 | Validation Loss: 0.03628441091212961
Epoch 17: Train accuracy: 0.6060754038384026 | Validation accuracy: 0.6026837384259259 | Train Loss: 0.03498791473075038 | Validation Loss: 0.03495208008421792
Epoch 18: Train accuracy: 0.606253681828047 | Validation accuracy: 0.6024305555555556 | Train Loss: 0.03395061387813517 | Validation Loss: 0.03358783039988743
Epoch 19: Train accuracy: 0.6043391312436053 | Validation accuracy: 0.6021773726851852 | Train Loss: 0.03312554791392315 | Validation Loss: 0.03292270325538185
Epoch 20: Train accuracy: 0.6045794189687781 | Validation accuracy: 0.6020688657407407 | Train Loss: 0.0323917680708248 | Validation Loss: 0.03194271038389868
Finished Training
```

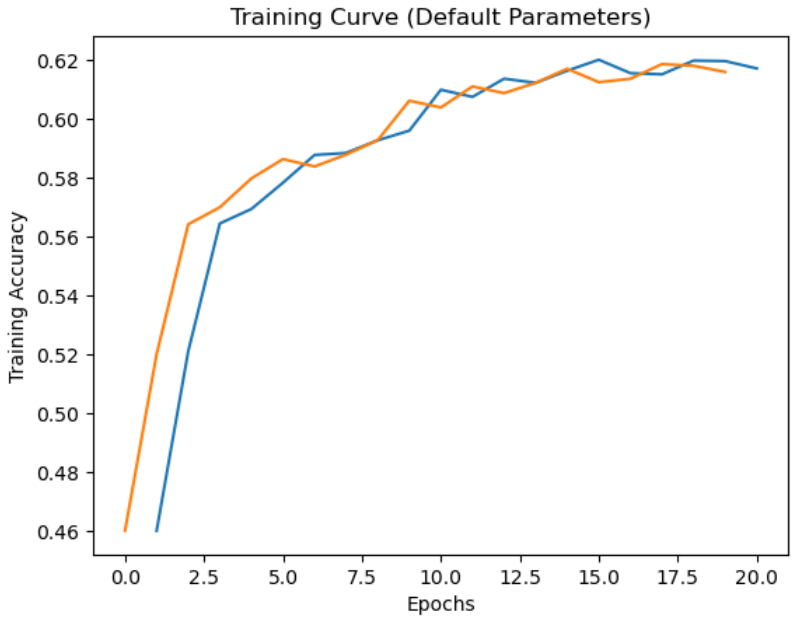


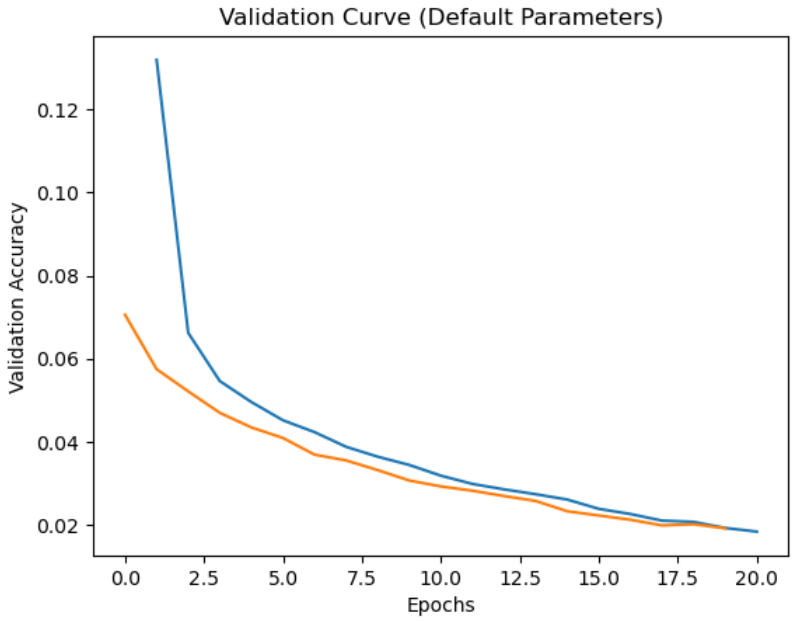


In [98]:
```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=128, shuffle=True)
model = AutoEncoder()

# Increasing learning rate further
train(model, train_loader, valid_loader, num_epochs=20, learning_rate=7e-4)
```

```
Epoch 1: Train accuracy: 0.4600812327535423 | Validation accuracy: 0.4601417824074074 | Train Loss: 0.1319815388747624 | Valid
ation Loss: 0.07058223233454758
Epoch 2: Train accuracy: 0.5210600564288593 | Validation accuracy: 0.5201099537037037 | Train Loss: 0.06623470397400004 | Vali
dation Loss: 0.05747833392686314
Epoch 3: Train accuracy: 0.5645443834682045 | Validation accuracy: 0.5642722800925926 | Train Loss: 0.054616974001484256 | Val
idation Loss: 0.05213920440938738
Epoch 4: Train accuracy: 0.5694741574427185 | Validation accuracy: 0.5700593171296297 | Train Loss: 0.0495654688926325 | Valid
ation Loss: 0.04697292029029793
Epoch 5: Train accuracy: 0.5784113105757603 | Validation accuracy: 0.5798611111111112 | Train Loss: 0.045175321045376006 | Val
idation Loss: 0.04345292473832766
Epoch 6: Train accuracy: 0.5878522928099712 | Validation accuracy: 0.5864438657407407 | Train Loss: 0.04231736587271804 | Vali
dation Loss: 0.04090573742157883
Epoch 7: Train accuracy: 0.5885266486838434 | Validation accuracy: 0.5839482060185185 | Train Loss: 0.03877931782266214 | Vali
dation Loss: 0.03690127510991362
Epoch 8: Train accuracy: 0.5928440765200137 | Validation accuracy: 0.5879629629629629 | Train Loss: 0.036402372643351555 | Val
idation Loss: 0.03549971260751287
Epoch 9: Train accuracy: 0.5961150900691409 | Validation accuracy: 0.5928819444444444 | Train Loss: 0.03440762028497245 | Vali
dation Loss: 0.0331688129550053
Epoch 10: Train accuracy: 0.610020773261402 | Validation accuracy: 0.6063006365740741 | Train Loss: 0.03182913701138681 | Vali
dation Loss: 0.030665226487649813
Epoch 11: Train accuracy: 0.6075946423588503 | Validation accuracy: 0.6039858217592593 | Train Loss: 0.02984260284297523 | Val
idation Loss: 0.02923801763811045
Epoch 12: Train accuracy: 0.613780113477816 | Validation accuracy: 0.6111472800925926 | Train Loss: 0.028531742991790884 | Val
idation Loss: 0.0282021495513618
Epoch 13: Train accuracy: 0.6123771432114842 | Validation accuracy: 0.6088686342592593 | Train Loss: 0.02736780467620563 | Val
idation Loss: 0.026925353695534997
Epoch 14: Train accuracy: 0.6165085418410691 | Validation accuracy: 0.6123770254629629 | Train Loss: 0.026079332566864434 | Va
lidation Loss: 0.02574351497201456
Epoch 15: Train accuracy: 0.6202136235388956 | Validation accuracy: 0.6171513310185185 | Train Loss: 0.023838721781170795 | Va
lidation Loss: 0.023263262481325202
Epoch 16: Train accuracy: 0.6156559079775524 | Validation accuracy: 0.6125940393518519 | Train Loss: 0.02259477793372103 | Val
idation Loss: 0.02223718859669235
Epoch 17: Train accuracy: 0.6152683471304995 | Validation accuracy: 0.6137514467592593 | Train Loss: 0.021006720489822328 | Va
lidation Loss: 0.021207814001374774
Epoch 18: Train accuracy: 0.6199345797290174 | Validation accuracy: 0.6187427662037037 | Train Loss: 0.020692463188121717 | Va
lidation Loss: 0.01986217053814067
Epoch 19: Train accuracy: 0.6197640529563141 | Validation accuracy: 0.6181640625 | Train Loss: 0.01923922857358342 | Validatio
n Loss: 0.020130404866197042
Epoch 20: Train accuracy: 0.6172604098843518 | Validation accuracy: 0.6160662615740741 | Train Loss: 0.018357482672269856 | Va
lidation Loss: 0.019154885395740468
Finished Training
```



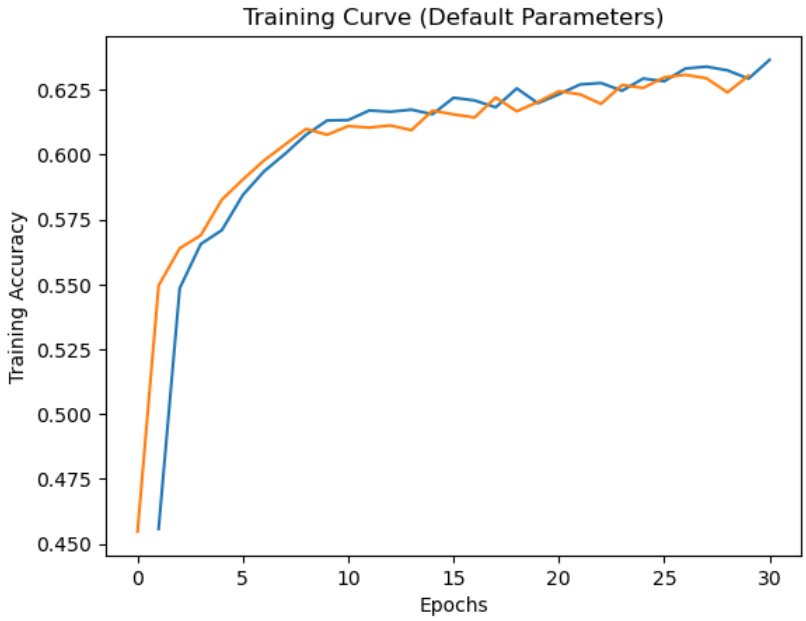


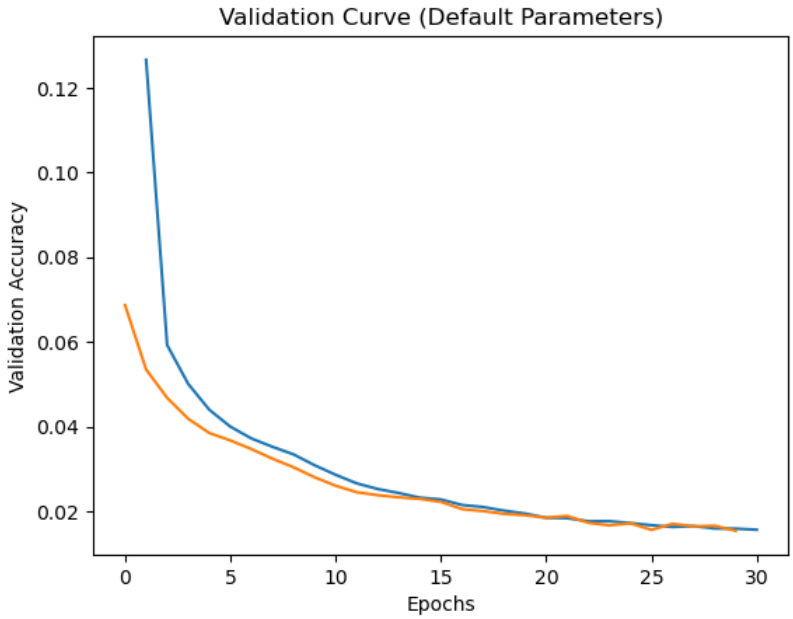
In [100…

```
train_loader = torch.utils.data.DataLoader(train_data, batch_size=128, shuffle=True)
valid_loader = torch.utils.data.DataLoader(val_data, batch_size=128, shuffle=True)
model = AutoEncoder()

# UNDER TIME CRUNCH FOR COMPLETING, HAD LITTLE TIME TO TRAIN, couldn't icrease num_epochs too much
# Increasing learning rate and increasing number of epoch
train(model, train_loader, valid_loader, num_epochs=30, learning_rate=8e-4)
```

Epoch 1: Train accuracy: 0.455763804917372 | Validation accuracy: 0.4547164351851852 | Train Loss: 0.12665772619878962 | Valid
ation Loss: 0.06867763068940905
Epoch 2: Train accuracy: 0.5484838619663287 | Validation accuracy: 0.5495876736111112 | Train Loss: 0.05923892118568931 | Vali
dation Loss: 0.05348140373826027
Epoch 3: Train accuracy: 0.5655597928874834 | Validation accuracy: 0.5639105902777778 | Train Loss: 0.0500744215124065 | Valid
ation Loss: 0.04676443193521765
Epoch 4: Train accuracy: 0.5709081325768146 | Validation accuracy: 0.5689380787037037 | Train Loss: 0.04396809007795084 | Vali
dation Loss: 0.04183275376756986
Epoch 5: Train accuracy: 0.5844882646575512 | Validation accuracy: 0.5826461226851852 | Train Loss: 0.03996919842791699 | Vali
dation Loss: 0.03846802986744377
Epoch 6: Train accuracy: 0.5935649396955321 | Validation accuracy: 0.5904586226851852 | Train Loss: 0.03718588977963442 | Vali
dation Loss: 0.03671184585740169
Epoch 7: Train accuracy: 0.6003317520850774 | Validation accuracy: 0.5977647569444444 | Train Loss: 0.035202259419574625 | Val
idation Loss: 0.03466910315263602
Epoch 8: Train accuracy: 0.6077186618299073 | Validation accuracy: 0.6039134837962963 | Train Loss: 0.03341568669392949 | Vali
dation Loss: 0.03239630628377199
Epoch 9: Train accuracy: 0.6131987722072365 | Validation accuracy: 0.6099537037037037 | Train Loss: 0.03085451726136463 | Vali
dation Loss: 0.030375706342359383
Epoch 10: Train accuracy: 0.6133537965460577 | Validation accuracy: 0.6077112268518519 | Train Loss: 0.028594520198003875 | Va
lidation Loss: 0.02800961107843452
Epoch 11: Train accuracy: 0.6170666294608254 | Validation accuracy: 0.6110387731481481 | Train Loss: 0.026564829927381305 | Va
lidation Loss: 0.026010162610974576
Epoch 12: Train accuracy: 0.6165783027935385 | Validation accuracy: 0.6104600694444444 | Train Loss: 0.025236079934984446 | Va
lidation Loss: 0.024476573957751196
Epoch 13: Train accuracy: 0.6173766781384677 | Validation accuracy: 0.6112919560185185 | Train Loss: 0.024281344609335065 | Va
lidation Loss: 0.02377844938180513
Epoch 14: Train accuracy: 0.6156094006759061 | Validation accuracy: 0.6094835069444444 | Train Loss: 0.023185176553115958 | Va
lidation Loss: 0.023264248803671863
Epoch 15: Train accuracy: 0.6219576473506341 | Validation accuracy: 0.6170066550925926 | Train Loss: 0.022759934310756978 | Va
lidation Loss: 0.02289356767303414
Epoch 16: Train accuracy: 0.6209344867144142 | Validation accuracy: 0.6155598958333334 | Train Loss: 0.021460892909782984 | Va
lidation Loss: 0.02219517653187116
Epoch 17: Train accuracy: 0.6182680680866897 | Validation accuracy: 0.6143663194444444 | Train Loss: 0.020976170426278952 | Va
lidation Loss: 0.020489001491417486
Epoch 18: Train accuracy: 0.62558521687905 | Validation accuracy: 0.6220703125 | Train Loss: 0.020116941759451515 | Validation
Loss: 0.02000681347110205
Epoch 19: Train accuracy: 0.6198880724273711 | Validation accuracy: 0.6167534722222222 | Train Loss: 0.01943407759868673 | Val
idation Loss: 0.019359065918251872
Epoch 20: Train accuracy: 0.6233218615322605 | Validation accuracy: 0.6203342013888888 | Train Loss: 0.018429836773845767 | Va
lidation Loss: 0.0190383475791249
Epoch 21: Train accuracy: 0.6270734505317335 | Validation accuracy: 0.6244574652777778 | Train Loss: 0.01834240368944371 | Val
idation Loss: 0.018448944384646084
Epoch 22: Train accuracy: 0.6276625430192541 | Validation accuracy: 0.6233000578703703 | Train Loss: 0.01761615437100686 | Val
idation Loss: 0.01886528251796133
Epoch 23: Train accuracy: 0.624670573280005 | Validation accuracy: 0.6196108217592593 | Train Loss: 0.017632747623359875 | Val
idation Loss: 0.017248938843193982
Epoch 24: Train accuracy: 0.6293755619632282 | Validation accuracy: 0.6268807870370371 | Train Loss: 0.01718431235557156 | Val
idation Loss: 0.016649142606183887
Epoch 25: Train accuracy: 0.6283291476761852 | Validation accuracy: 0.6257595486111112 | Train Loss: 0.01667684220731081 | Val
idation Loss: 0.017125199341939554
Epoch 26: Train accuracy: 0.6332279167829349 | Validation accuracy: 0.6298466435185185 | Train Loss: 0.016269437613941375 | Va
lidation Loss: 0.015570431533786986
Epoch 27: Train accuracy: 0.6339565311753945 | Validation accuracy: 0.6308232060185185 | Train Loss: 0.016467448712016147 | Va
lidation Loss: 0.017015874851495028
Epoch 28: Train accuracy: 0.6325380584751806 | Validation accuracy: 0.6294849537037037 | Train Loss: 0.015900955657430348 | Va
lidation Loss: 0.01640498366517325
Epoch 29: Train accuracy: 0.6293290546615818 | Validation accuracy: 0.6240234375 | Train Loss: 0.015865065052085334 | Validati
on Loss: 0.0165414914695753
Epoch 30: Train accuracy: 0.6365686912845316 | Validation accuracy: 0.6304615162037037 | Train Loss: 0.015611355708512877 | Va
lidation Loss: 0.015365466165045897
Finished Training





Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [101…

```
test_loader = torch.utils.data.DataLoader(test_data, batch_size=128, shuffle=True)
test_acc = get_accuracy(model, test_loader)
print("Test accuracy: ", test_acc)
```

```
Test accuracy:  0.6308955439814815
```

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…]
```
bl = {}
for col in df_not_missing:
    bl[col] = df[col].value_counts().idxmax()

bl_acc = sum(df_not_missing["marriage"] == bl["marriage"])/len(df_not_missing)
print("Baseline accuracy (\"marriage\" feature):", bl_acc)
```

```
Baseline accuracy ("marriage" feature): 0.4667947131974738
```

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 was much better (63% vs 47%)
```

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])
```

Out[104]:
```
{'work': 'Private',
 'marriage': 'Divorced',
 'occupation': 'Prof-specialty',
 'edu': 'Bachelors',
 'relationship': 'Not-in-family',
 'sex': 'Male'}
```

In [ ]:
```
# Some features can help with guessing the education value (e.g. work, occupation) but the others not so much.
```

Part (e) [2 pt]

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

In [105…]
```
test_edu = zero_out_feature(test_data[:1], "edu")[0]
predict = model(torch.from_numpy(test_edu))
get_feature(predict.detach().numpy(), "edu")
```

Out[105]:
```
'Bachelors'
```

Part (f) [2 pt]

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

In [106…]
```
bl["edu"]
```

Out[106]:
```
' HS-grad'
```