Lab 4: Data Imputation using an Autoencoder

Deadline: Mon, March 01, 5:00pm

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

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

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

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

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

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

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

In the process, you are expected to learn to:

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

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

What to submit

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

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Colab Link

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

Colab Link: https://drive.google.com/file/d/1uYo_I5C-BoDdryNGs7jNV257q4YoT7YE/view?usp=sharing (https://drive.google.com/file/d/1uYo_I5C-BoDdryNGs7jNV257q4YoT7YE/view?usp=sharing)

In [84]:
```
import csv
import numpy as np
import random
import torch
import torch.utils.data
import matplotlib.pyplot as plt
```

Part 0

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

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

In [ ]:
```
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 [ ]:
```
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)
```

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	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 [ ]:
```
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 age:", (df["age"]).max(), "min age:", (df["age"]).min(), "avg age:"
, (df["age"]).mean())
print("max yredu:", (df["yredu"]).max(), "min yredu:", (df["yredu"]).min(), "a
vg yredu:", (df["yredu"]).mean())
print("max capgain:", (df["capgain"]).max(), "min capgain:", (df["capgain"]).m
in(), "avg capgain:", (df["capgain"]).mean())
print("max caploss:", (df["caploss"]).max(), "min caploss:", (df["caploss"]).m
in(), "avg caploss:", (df["caploss"]).mean())
print("max workhr:", (df["workhr"]).max(), "min workhr:", (df["workhr"]).min
(), "avg workhr:", (df["workhr"]).mean())
```

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

In [ ]:
```
# Normalize by (value - min)/(max - min)
df["age"] = (df["age"]-(df["age"]).min()) / ((df["age"]).max() - (df["age"]).m
in())
df["yredu"] = (df["yredu"]-(df["yredu"]).min()) / ((df["yredu"]).max() - (df[
"yredu"]).min())
df["capgain"] = (df["capgain"]-(df["capgain"]).min()) / ((df["capgain"]).max()
- (df["capgain"]).min())
df["caploss"] = (df["caploss"]-(df["caploss"]).min()) / ((df["caploss"]).max()
- (df["caploss"]).min())
df["workhr"] = (df["workhr"]-(df["workhr"]).min()) / ((df["workhr"]).max() - (
df["workhr"]).min())
```

Part (b) Categorical Features [1 pt]

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

What percentage of people in our data set are female?

In [ ]:
```
# hint: you can do something like this in pandas
sum_male = sum(df["sex"] == " Male")
sum_female = sum(df["sex"] == " Female")
percentage = sum_female/(sum_female+sum_male)
print(percentage)
# There is 33.1% female in the data set, assuming that people are either male
 or female.
```

```
0.33079450876815825
```

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(len(df_with_missing))
percentage_missing = (len(df_with_missing))/(len(df_with_missing) + len(df_not
_missing))
print(percentage_missing)

# 5.66% of records are removed
```

```
1843
0.056601455729246644
```

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', ' State-gov', ' Without-pay', ' Private', ' Self-emp-not-in
c', ' Local-gov', ' Self-emp-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	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 [ ]:
```
print(len(data.columns))

#There are 57 features in the dataframe data
#This number comes from the transformation of the get_dummies function that tr
ansforms
#the categorical values into columns(features) with 1 or 0 indicating whether
 the person
#has or doesn't have the feature. Thus, this number is like the sum of non-cat
egorical features
#and categorical features that were expanded to different columns of features.
```

```
57
```

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 [85]:
```
cat_index = {}  # Mapping of feature -> start index of feature in a record
cat_values = {} # Mapping of feature -> list of categorical values the feature
can take

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

print(cat_index)
print(cat_values)

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

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

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

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

    Examples:

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

```
    cat_index_max = np.argmax(onehot)
    #print(cat_index_max)

    return cat_value[cat_index_max]
```

```
{'work': 5, 'marriage': 12, 'occupation': 19, 'edu': 33, 'relationship': 49,
'sex': 55}
{'work': ['Federal-gov', 'Local-gov', 'Private', 'Self-emp-inc', 'Self-emp-no
t-inc', 'State-gov', 'Without-pay'], 'marriage': ['Divorced', 'Married-AF-spo
use', 'Married-civ-spouse', 'Married-spouse-absent', 'Never-married', 'Separa
ted', 'Widowed'], 'occupation': ['Adm-clerical', 'Armed-Forces', 'Craft-repai
r', 'Exec-managerial', 'Farming-fishing', 'Handlers-cleaners', 'Machine-op-in
spct', 'Other-service', 'Priv-house-serv', 'Prof-specialty', 'Protective-ser
v', 'Sales', 'Tech-support', 'Transport-moving'], 'edu': ['10th', '11th', '12
th', '1st-4th', '5th-6th', '7th-8th', '9th', 'Assoc-acdm', 'Assoc-voc', 'Bach
elors', 'Doctorate', 'HS-grad', 'Masters', 'Preschool', 'Prof-school', 'Some-
college'], 'relationship': ['Husband', 'Not-in-family', 'Other-relative', 'Ow
n-child', 'Unmarried', 'Wife'], 'sex': ['Female', 'Male']}
```

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

```
2
```

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

# find the indices and shuffle them and split the dataset
relevant_indices = np.arange(len(data))
np.random.shuffle(relevant_indices)
split1 = int(len(relevant_indices) * 0.7) #split at 70%
split2 = int(len(relevant_indices) * 0.85) #split at 85%

# split into training and validation indices
relevant_train_indices, relevant_val_indices, relevant_test_indices = relevant
_indices[:split1], relevant_indices[split1:split2], relevant_indices[split2:]
train_sampler = datanp[relevant_train_indices]
val_sampler = datanp[relevant_val_indices]
test_sampler = datanp[relevant_test_indices]

print("train_sampler:",len(train_sampler))
print("val_sampler:",len(val_sampler))
print("test_sampler:",len(test_sampler))
```

```
train_sampler: 21502
val_sampler: 4608
test_sampler: 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 [75]:
```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.name = 'AutoEncoder'
        self.encoder = nn.Sequential(
            nn.Linear(57, 57), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(57,30),
            nn.ReLU(),
            nn.Linear(30,11)

        )
        self.decoder = nn.Sequential(
            nn.Linear(11,30),
            nn.ReLU(),
            nn.Linear(30,57),
            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.)

The sigmoid activation function will convert the output to a vector of values between 0 and 1. Because we have normalized values in the data frame data and then applied the get_dummies function, thus we want our autoencoder output to match the normalized date that are between 0 and 1. Moreover, the maximum value in the output after applying the softmax will be regarded as the predicted value.

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 [87]:
```
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_data,val_data, batch_size=64, num_epochs=10,lr = 0.001
):
    """ Training loop. You should update this."""
    torch.manual_seed(42)
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_si
ze, shuffle=True)
    valid_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size
, shuffle=True)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)


    epochs, trainloss, validloss, train_acc, val_acc = [], [], [], [], []

    n_t = 0 # number of iterations
    n_v = 0

    for epoch in range(num_epochs):
        for data in train_loader:
            datam = zero_out_random_feature(data.clone()) # zero out one categ
orical feature
            recon = model(datam)
            train_loss = criterion(recon, data)
            train_loss.backward()
            optimizer.step()
            optimizer.zero_grad()


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

        epochs.append(epoch)
        trainloss.append(float(train_loss)/batch_size)         # compute *aver
age* loss
        train_acc.append(get_accuracy(model, train_loader)) # compute training
accuracy
        print("epoch number", epoch+1, "accuracy: ",train_acc[epoch])
        validloss.append(float(valid_loss)/batch_size)
```

```
        val_acc.append(get_accuracy(model, valid_loader))  # compute validatio
n accuracy
        model_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(model.name, batch
_size, lr, num_epochs)
        torch.save(model.state_dict(), model_path)

    # plotting
    # Due to the large batch size that are used, I decide to plot the loss and
accuracy information after every epoch instead of every itertation that is sup
er slow to compute.
    plt.title("Training Curve")
    plt.plot(epochs, trainloss, label="Train")
    plt.plot(epochs, validloss, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training Curve")
    plt.plot(epochs, train_acc, label="Train")
    plt.plot(epochs, val_acc, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))
```

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 [74]:
```
def get_accuracy(model, data_loader):
    """Return the "accuracy" of the autoencoder model across a data set.
    That is, for each record and for each categorical feature,
    we determine whether the model can successfully predict the value
    of the categorical feature given all the other features of the
    record. The returned "accuracy" measure is the percentage of times
    that our model is successful.

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

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

        >>> model = AutoEncoder()
        >>> vdl = torch.utils.data.DataLoader(data_valid, batch_size=256, shuf
fle=True)
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            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]

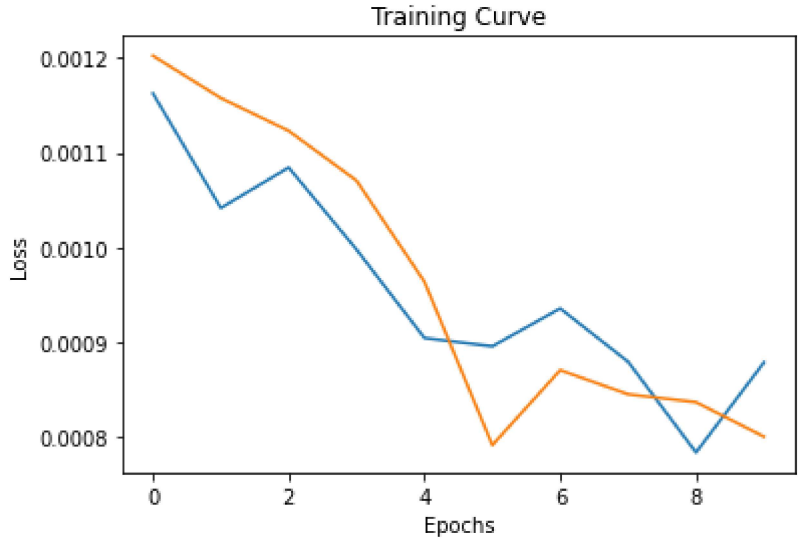
Run your updated training code, using reasonable initial hyperparameters.

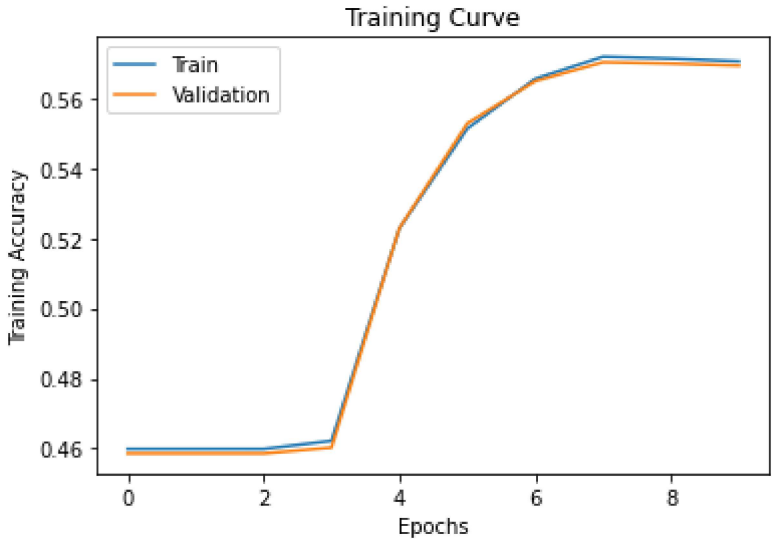
Include your training curve in your submission.

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

train(model, train_sampler, val_sampler, 64, 10, 0.0001)
```

```
epoch number 1 accuracy:  0.4598486962453105
epoch number 2 accuracy:  0.4598486962453105
epoch number 3 accuracy:  0.4598409450283695
epoch number 4 accuracy:  0.4621508076768053
epoch number 5 accuracy:  0.5228893436269494
epoch number 6 accuracy:  0.5513983195361671
epoch number 7 accuracy:  0.5656063001891297
epoch number 8 accuracy:  0.5719235419960934
epoch number 9 accuracy:  0.5713809568102192
epoch number 10 accuracy:  0.5705980838991722
```






```
Final Training Accuracy: 0.5705980838991722
Final Validation Accuracy: 0.5693721064814815
```

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 [90]:
```
def train(model, train_data,val_data, batch_size=64, num_epochs=10,lr = 0.001
):
    """ Training loop. You should update this."""
    torch.manual_seed(42)
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_si
ze, shuffle=True)
    valid_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size
, shuffle=True)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)


    epochs, trainloss, validloss, train_acc, val_acc = [], [], [], [], []

    n_t = 0 # number of iterations
    n_v = 0

    for epoch in range(num_epochs):
        for data in train_loader:
            datam = zero_out_random_feature(data.clone()) # zero out one categ
orical feature
            recon = model(datam)
            train_loss = criterion(recon, data)
            train_loss.backward()
            optimizer.step()
            optimizer.zero_grad()


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

        epochs.append(epoch)
        trainloss.append(float(train_loss)/batch_size)         # compute *aver
age* loss
        train_acc.append(get_accuracy(model, train_loader)) # compute training
accuracy
        print("epoch number", epoch+1, "accuracy: ",train_acc[epoch])
        validloss.append(float(valid_loss)/batch_size)
        val_acc.append(get_accuracy(model, valid_loader))  # compute validatio
n accuracy
        model_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(model.name, batch
_size, lr, num_epochs)
        torch.save(model.state_dict(), model_path)

    print("Final Training Accuracy: {}".format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))
```

In [92]:
```
model = AutoEncoder()
train(model, train_sampler, val_sampler, 64, 10, 0.001)
# Batch number = 64, number of epochs = 10, learning rate = 0.001
# Firstly, I increased the learning rate a bit to speed up the rate of converg
ence.
```

```
epoch number 1 accuracy:  0.5551499085356401
epoch number 2 accuracy:  0.5736520633739497
epoch number 3 accuracy:  0.6098967537903451
epoch number 4 accuracy:  0.60261836108269
epoch number 5 accuracy:  0.6107261340030384
epoch number 6 accuracy:  0.604176355687843
epoch number 7 accuracy:  0.5993938548352091
epoch number 8 accuracy:  0.611950826279726
epoch number 9 accuracy:  0.6117880507239637
epoch number 10 accuracy:  0.6171828977149413
Final Training Accuracy: 0.6171828977149413
Final Validation Accuracy: 0.6145109953703703
```

In [93]:
```
model = AutoEncoder()
train(model, train_sampler, val_sampler, 64, 10, 0.005)
# Batch number = 64, number of epochs = 10, learning rate = 0.005
# Secondly the above results show that the accuracies have been improved and t
herefore I continue to increase the learning rate
```

```
epoch number 1 accuracy:  0.6110749387653862
epoch number 2 accuracy:  0.6114082410938517
epoch number 3 accuracy:  0.6078504325179053
epoch number 4 accuracy:  0.6172991659690571
epoch number 5 accuracy:  0.6196710383530214
epoch number 6 accuracy:  0.6090983784454159
epoch number 7 accuracy:  0.6149195423681518
epoch number 8 accuracy:  0.6241047344433076
epoch number 9 accuracy:  0.6340417945617461
epoch number 10 accuracy:  0.6374833348835767
Final Training Accuracy: 0.6374833348835767
Final Validation Accuracy: 0.6360677083333334
```

In [95]:
```
model = AutoEncoder()
train(model, train_sampler, val_sampler, 64, 30, 0.005)
# Batch number = 64, number of epochs = 30, learning rate = 0.005
# The above results have shown that the accuracies have been further increased
and thus I increased the number of epochs
# to achieve more updates to the parameters for the model.
```

```
epoch number 1 accuracy:  0.6042461166403126
epoch number 2 accuracy:  0.5965724118686634
epoch number 3 accuracy:  0.6039670728304344
epoch number 4 accuracy:  0.6072535888134437
epoch number 5 accuracy:  0.610532353579512
epoch number 6 accuracy:  0.622081666821691
epoch number 7 accuracy:  0.6100982854308127
epoch number 8 accuracy:  0.6191749604687936
epoch number 9 accuracy:  0.6254844510588162
epoch number 10 accuracy:  0.6227870275633275
epoch number 11 accuracy:  0.6263293337053918
epoch number 12 accuracy:  0.6342123213344495
epoch number 13 accuracy:  0.6286857036554739
epoch number 14 accuracy:  0.6303289616469786
epoch number 15 accuracy:  0.6399947291724801
epoch number 16 accuracy:  0.6326930828140018
epoch number 17 accuracy:  0.6396304219762503
epoch number 18 accuracy:  0.6359408427123059
epoch number 19 accuracy:  0.6485598238923511
epoch number 20 accuracy:  0.6487148482311723
epoch number 21 accuracy:  0.6523269153257061
epoch number 22 accuracy:  0.6523424177595882
epoch number 23 accuracy:  0.6508464328899637
epoch number 24 accuracy:  0.633762750751868
epoch number 25 accuracy:  0.6717514649800018
epoch number 26 accuracy:  0.6618066536446222
epoch number 27 accuracy:  0.6666201593650203
epoch number 28 accuracy:  0.6736272594797383
epoch number 29 accuracy:  0.6619229218987381
epoch number 30 accuracy:  0.6659535547080891
Final Training Accuracy: 0.6659535547080891
Final Validation Accuracy: 0.6635199652777778
```

In [96]:
```
model = AutoEncoder()
train(model, train_sampler, val_sampler, 16, 30, 0.0001)
# Batch number = 16, number of epochs = 30, learning rate = 0.0001
# I also want to see the effects of reducing the batch number on the accuracies, since
# I have not tried the effects of adjusting the batch size on the accuracies.
# Technically when I reduced the batch size, I need to reduce the learning rate as well.
# Therefore this is what I did.
```

```
epoch number 1 accuracy:  0.4671348401699067
epoch number 2 accuracy:  0.5586379561591169
epoch number 3 accuracy:  0.5728691904629026
epoch number 4 accuracy:  0.5772098719498961
epoch number 5 accuracy:  0.5786671007348154
epoch number 6 accuracy:  0.5843487427526122
epoch number 7 accuracy:  0.5954252317613865
epoch number 8 accuracy:  0.6071838278609741
epoch number 9 accuracy:  0.6114935044802033
epoch number 10 accuracy:  0.6057886088115835
epoch number 11 accuracy:  0.606517223204043
epoch number 12 accuracy:  0.6047421945245404
epoch number 13 accuracy:  0.6029981707128019
epoch number 14 accuracy:  0.6040678386506682
epoch number 15 accuracy:  0.60261836108269
epoch number 16 accuracy:  0.6035950144172635
epoch number 17 accuracy:  0.6061606672247543
epoch number 18 accuracy:  0.6033392242582085
epoch number 19 accuracy:  0.6047344433075993
epoch number 20 accuracy:  0.6058118624624066
epoch number 21 accuracy:  0.6024478343099867
epoch number 22 accuracy:  0.6081372275447245
epoch number 23 accuracy:  0.6074318668030881
epoch number 24 accuracy:  0.604556165317955
epoch number 25 accuracy:  0.6038895606610237
epoch number 26 accuracy:  0.6075713887080272
epoch number 27 accuracy:  0.609245651567296
epoch number 28 accuracy:  0.6108501534740954
epoch number 29 accuracy:  0.6092223979164729
epoch number 30 accuracy:  0.6120205872321954
Final Training Accuracy: 0.6120205872321954
Final Validation Accuracy: 0.6084346064814815
```

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [98]:
```
best_mod = AutoEncoder()
best_mod_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format("AutoEncoder", 64, 0.0
05, 30)
state = torch.load(best_mod_path)
best_mod.load_state_dict(state)
test_loader = torch.utils.data.DataLoader(test_sampler, batch_size=64, shuffle
=True)
test_accuracy = get_accuracy(best_mod, test_loader)
print("Test accuracy:", test_accuracy)
```

```
Test accuracy: 0.6623987268518519
```

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 [99]:
```
print(type(df_not_missing))
```

```
<class 'pandas.core.frame.DataFrame'>
```

In [128]:
```
#df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
                    #'num_wings': [2, 0, 0, 0]},
                   #index=['falcon', 'falcon', 'cat', 'ant'])
#df.value_counts().idxmax()
```

Out[128]:
```
(4, 0)
```

In [108]:
```
most_common_values = {}  # Get the most common value for each feature
for col in df_not_missing.columns:
  # Find the feature that is repeated most often in each column and locate its
feature name with the counts
  most_common_values[col] = df_not_missing[col].value_counts().idxmax()

print(most_common_values)
accuracy = sum(df_not_missing['marriage'] == most_common_values['marriage'])/l
en(df_not_missing)
print("The accuracy for baseline model of missing 'marriage' test is:", accura
cy)
```

```
{'age': 0.2602739726027397, 'yredu': 0.5333333333333333, 'capgain': 0.0, 'cap
loss': 0.0, 'workhr': 0.3979591836734694, 'work': ' Private', 'marriage': ' M
arried-civ-spouse', 'occupation': ' Prof-specialty', 'edu': ' HS-grad', 'rela
tionship': ' Husband', 'sex': ' Male'}
The accuracy for baseline model of missing 'marriage' test is: 0.466794713197
4738
```

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 is 66.2% that is much higher than the baseline accuracy of 46.7%. This makes sense since we were using the autoencoder to learn the feautures and improve the embeddings unlike the simple 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 [135]:
```
get_features(test_sampler[0])
```

Out[135]:
```
{'edu': 'Bachelors',
 'marriage': 'Divorced',
 'occupation': 'Prof-specialty',
 'relationship': 'Not-in-family',
 'sex': 'Male',
 'work': 'Private'}
```

I think it is reasonable to guess the person's education level based on these features because this person's occupation is a prof-specialty, which means that this person must have received high level education.

Part (e) [2 pt]

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

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
start_index = cat_index[feature]
stop_index = cat_index[feature] + len(cat_values[feature])
records[:, start_index:stop_index] = 0
```

In [149]:
```
#zero out
test_edu = test_sampler[0]
start_index = cat_index["edu"]
stop_index = cat_index["edu"] + len(cat_values["edu"])
test_edu[start_index:stop_index] = 0
test_edu = torch.from_numpy(test_edu)
#print(test_edu)

best_mod = AutoEncoder()
best_mod_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format("AutoEncoder", 64, 0.0
05, 30)
state = torch.load(best_mod_path)
best_mod.load_state_dict(state)

prediction = best_mod(test_edu)
prediction = prediction.detach().numpy()
predicted_edu = get_feature(prediction, "edu")
print("Predicted_edu:", predicted_edu)
```

```
Predicted_edu: Masters
```

Part (f) [2 pt]

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

In [156]:
```
print(most_common_values['edu'])

# The baseline model's prediction is HS-grad
```

```
HS-grad
```