Lab 4: Data Imputation using an Autoencoder

Deadline: Thursday, Oct 29, 11:59pm

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

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 (.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://colab.research.google.com/drive/1-0g74jxRpWElQY8IqzRgUESkI_ZR6w5E?authuser=1

```
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

```
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`](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

```
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)
```

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

```
(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.

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

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationship	race
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White

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

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

	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:

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

```
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.

```
print("AGE")
print("max: %d" % np.max(subdf["age"]))
print("min: %d" % np.min(subdf["age"]))
print("avg: %.2f" % np.average(subdf["age"]))
print(" ")
print("YREDU")
print("max: %d" % np.max(subdf["yredu"]))
print("min: %d" %np.min(subdf["yredu"]))
print("avg: %.2f" % np.average(subdf["yredu"]))
print(" ")
print("CAPGAIN")
print("max: %d" % np.max(subdf["capgain"]))
print("min: %d" %np.min(subdf["capgain"]))
print("avg: %.2f" % np.average(subdf["capgain"]))
print(" ")
print("CAPLOSS")
print("max: %d" % np.max(subdf["caploss"]))
print("min: %d" %np.min(subdf["caploss"]))
print("avg: %.2f" % np.average(subdf["caploss"]))
print(" ")
print("WORKHR")
print("max: %d" % np.max(subdf["workhr"]))
print("min: %d" %np.min(subdf["workhr"]))
print("avg: %.2f" % np.average(subdf["workhr"]))
```

```
AGE
max: 90
min: 17
avg: 38.58

YREDU
max: 16
min: 1
avg: 10.08

CAPGAIN
```

```
max: 99999
min: 0
avg: 1077.65

CAPLOSS
max: 4356
min: 0
avg: 87.30

WORKHR
max: 99
min: 1
avg: 40.44
```

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?

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

percentage = (y) / (x+y)
print(percentage)

# 33.08 % of the people in the dataset are 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?

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

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

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.

```
set(df_not_missing)
```

```
{'age',
 'capgain',
 'caploss',
 'edu',
 'marriage',
 'occupation',
 'relationship',
 'sex',
 'work',
 'workhr',
 'yredu'}
```

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`.

```
data = pd.get_dummies(df_not_missing)
```

```
data[:3]
```

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

[ ] ↳ *1 cell hidden*

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.

```
datanp = data.values.astype(np.float32)


cat_index = {}  # Mapping of feature -> start index of feature in a record
cat_values = {} # Mapping of feature -> list of categorical values the feature can tal

# 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'
    """

    ind = np.argmax(onehot)
    values = cat_values[feature]
    return values[ind]


# 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.

```
# 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_index = int(datanp.shape[0] * 0.7)
val_index = int(datanp.shape[0] * 0.85)

train_set = datanp[:train_index]
val_set = datanp[train_index:val_index]
test_set = datanp[val_index:]

print("# Train Set: " + str(train_set.shape[0]))
print("# Test Set: " + str(val_set.shape[0]))
print("# Val Set: " + str(test_set.shape[0]))
```

```
    # Train Set: 21502
    # Test Set: 4608
    # Val 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.

```
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!
        )
        self.decoder = nn.Sequential(
            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 use a sigmoid activation as it allows us to normalize the output to values between 0 and 1, which will match the format of our input

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.)

```
def get_model_name(name, learning_rate, epoch):
    """ Generate a name for the model consisting of all the hyperparameter values

    Args:
        config: Configuration object containing the hyperparameters
    Returns:
        path: A string with the hyperparameter name and value concatenated
    """
    path = "model_{0}_lr{1}_epoch{2}".format(name,
                                               learning_rate,
                                               epoch)
    return path
```

```
    return path

def plot_training_curve(path):
    """ Plots the training curve for a model run, given the csv files
    containing the train/validation accuracy/loss.

    Args:
        path: The base path of the csv files produced during training
    """
    import matplotlib.pyplot as plt
    train_acc = np.loadtxt("{}_train_acc.csv".format(path))
    val_acc = np.loadtxt("{}_val_acc.csv".format(path))
    train_loss = np.loadtxt("{}_train_loss.csv".format(path))
    val_loss = np.loadtxt("{}_val_loss.csv".format(path))
    plt.title("Train vs Validation Accuracy")
    n = len(train_acc) # number of epochs
    plt.plot(range(1,n+1), train_acc, label="Train")
    plt.plot(range(1,n+1), val_acc, label="Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

def get_loss(model, data_loader, criterion):
    total_loss = 0.0
    for data in data_loader:
        datam = zero_out_random_feature(data.clone()) # zero out one categorical featu
        recon = model(datam)
        loss = criterion(recon, data)
        total_loss += loss.item()
    loss = float(total_loss) / (len(data_loader))
    return loss

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, val_loader, num_epochs=5, learning_rate=1e-4):
    """ Training loop. You should update this."""
    torch.manual_seed(42)
    criterion = nn.MSELoss()
```

```
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_acc = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_acc = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)

    for epoch in range(num_epochs):
        total_train_loss = 0.0
        for data in train_loader:
            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()
            total_train_loss += loss.item()

        train_acc[epoch] = get_accuracy(model, train_loader)
        train_loss[epoch] = float(total_train_loss) / (len(train_loader))
        val_acc[epoch] = get_accuracy(model, val_loader)
        val_loss[epoch] = get_loss(model, val_loader, criterion)

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

    print('Finished Training')
    # Write the train/test loss/err into CSV file for plotting later
    epochs = np.arange(1, num_epochs + 1)
    np.savetxt("{}_train_acc.csv".format(model_path), train_acc)
    np.savetxt("{}_train_loss.csv".format(model_path), train_loss)
    np.savetxt("{}_val_acc.csv".format(model_path), val_acc)
    np.savetxt("{}_val_loss.csv".format(model_path), val_loss)
    plot_training_curve(model_path)
```

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.

```
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]

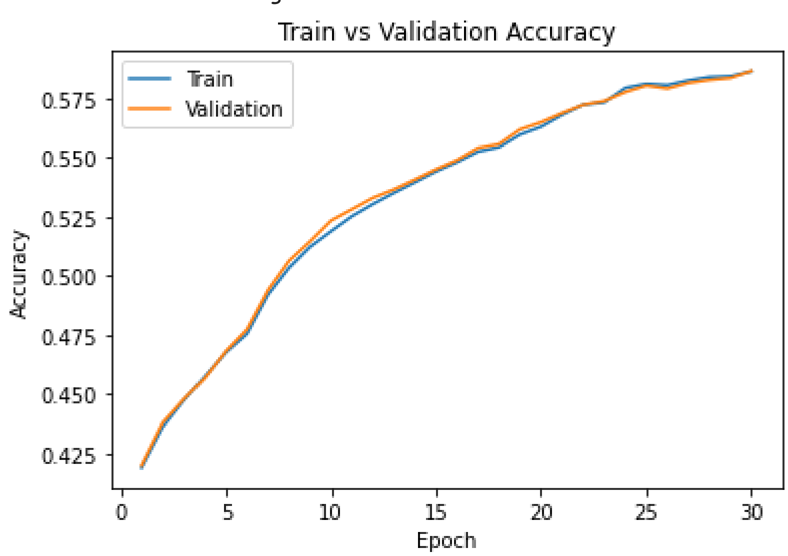
Run your updated training code, using reasonable initial hyperparameters.

Include your training curve in your submission.

```
batch_size=64
train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, num_worke
val_loader = torch.utils.data.DataLoader(val_set, batch_size=batch_size, num_workers=1
test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size, num_workers
```

```
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=0.0001)
```

```
Epoch 1: Train err: 0.4189687780981614, Train loss: 1019046.1291147867 |Validati
Epoch 2: Train err: 0.43608346510402135, Train loss: 1019045.9415537516 |Validat
Epoch 3: Train err: 0.4473614857532633, Train loss: 1019010.4902452742 |Validati
Epoch 4: Train err: 0.4574380677766409, Train loss: 1019003.3001207624 |Validati
Epoch 5: Train err: 0.46765417170495766, Train loss: 1019003.2962246849 |Validat
Epoch 6: Train err: 0.4755294081170744, Train loss: 1019003.2928633917 |Validati
Epoch 7: Train err: 0.49210926115400117, Train loss: 1019003.2893102737 |Validat
Epoch 8: Train err: 0.5034802964065358, Train loss: 1019003.2832765125 |Validati
Epoch 9: Train err: 0.5123399373701671, Train loss: 1019003.2827780588 |Validati
Epoch 10: Train err: 0.5189672278547732, Train loss: 1019003.2772572836 |Validat
Epoch 11: Train err: 0.5253619818311475, Train loss: 1019003.2738095238 |Validat
Epoch 12: Train err: 0.5305320435308344, Train loss: 1019003.2708987282 |Validat
Epoch 13: Train err: 0.5352602858648808, Train loss: 1019003.2660638718 |Validat
Epoch 14: Train err: 0.5397172356059902, Train loss: 1019003.2633666992 |Validat
Epoch 15: Train err: 0.5442516975165101, Train loss: 1019003.2610698881 |Validat
Epoch 16: Train err: 0.5481428084209221, Train loss: 1019003.257009597 |Validati
Epoch 17: Train err: 0.5525765045112082, Train loss: 1019003.2554917109 |Validat
Epoch 18: Train err: 0.5544367965770626, Train loss: 1019003.2536308651 |Validat
Epoch 19: Train err: 0.5598238923510991, Train loss: 1019003.2526375906 |Validat
Epoch 20: Train err: 0.5630949059002264, Train loss: 1019003.2517024449 |Validat
Epoch 21: Train err: 0.568109943261092, Train loss: 1019003.2316923595 |Validati
Epoch 22: Train err: 0.5723343564939696, Train loss: 1019003.1229175386 |Validat
Epoch 23: Train err: 0.5733962732148947, Train loss: 1019003.120623634 |Validati
Epoch 24: Train err: 0.5795042321644498, Train loss: 1019003.1188790457 |Validat
Epoch 25: Train err: 0.581209499891483, Train loss: 1019003.1153964088 |Validati
Epoch 26: Train err: 0.580690168356432, Train loss: 1019003.1132710775 |Validati
Epoch 27: Train err: 0.5826667286764022, Train loss: 1019003.1128663563 |Validat
Epoch 28: Train err: 0.5841084550274394, Train loss: 1019003.1112772623 |Validat
Epoch 29: Train err: 0.5844417573559049, Train loss: 1019003.1107126871 |Validat
Epoch 30: Train err: 0.5864803274114035, Train loss: 1019003.1107083275 |Validat
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.

```
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=40, learning_rate=0.001)
```

```
Epoch 1: Train err: 0.5175797600223235, Train loss: 1019004.1378144764 |Validati
Epoch 2: Train err: 0.5566768982730289, Train loss: 1019003.2597082229 |Validati
Epoch 3: Train err: 0.5885886584193718, Train loss: 1019003.2448047457 |Validati
Epoch 4: Train err: 0.5977660992775866, Train loss: 1019003.2325628372 |Validati
Epoch 5: Train err: 0.6058506185471119, Train loss: 1019003.226432437 |Validation
Epoch 6: Train err: 0.6096022075465848, Train loss: 1019003.2237345377 |Validati
Epoch 7: Train err: 0.6095091929432921, Train loss: 1019003.2217072987 |Validati
Epoch 8: Train err: 0.6147567668123896, Train loss: 1019003.2038283575 |Validati
Epoch 9: Train err: 0.6105090999286888, Train loss: 1019003.0837467739 |Validati
Epoch 10: Train err: 0.6166015564443618, Train loss: 1019003.0808933803 |Validat
```

The following four models were tried with tuned hyperparameters:

1. **train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=1e-3), bs=64** : The learning rate was increased to try to converge the training rate faster. This did not occur, so changing the batch size and number of epochs was tried instead.
2. **train(autoencoder, train_loader, val_loader, num_epochs=40, learning_rate=1e-3), bs=64** : A slight improvement in the accuracy was found when increasing the number of epochs. This was tried after with other changes to see if the two together had better improvements.
3. **train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=1e-3), bs=128** : Doubling the batch size saw little to no improvements, the results were relatively the same.
4. **train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=1e-2), bs=64** : Learning rate was increased again, even more, but no improvement in the training results were found.

After analyzing all four of the changes, the following model, the second one with increased epochs was found to have the best results, so this was chosen.

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

Train vs Validation Accuracy

```
model = get_model_name("autoencoder", learning_rate=0.001, epoch=32)
state = torch.load(model)
autoencoder.load_state_dict(state)
test_acc = get_accuracy(autoencoder, test_loader)
print(test_acc)
```

```
0.6253616898148148
```

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?

```
most_common = get_features(train_set.sum(axis = 0))

print('The most common values of each feature in train_set is: \n', most_common)

correct = 0.0
for record in test_set:
    test_values = get_features(record)
    for feature in most_common:
        if most_common[feature] == test_values[feature]:
            correct += 1

accuracy = correct / (len(test_set) * len(catcols))
print("The accuracy of this model is", round(accuracy * 100, 2) , "%")

# The test accuracy of this baseline model is
```

```
The most common values of each feature in train_set is:
 {'work': 'Private', 'marriage': 'Married-civ-spouse', 'occupation': 'Prof-speci
The accuracy of this model is 45.69 %
```

Part (c) [1 pt]

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

The test accuracy was **62.5%** compared with the baseline accuracy of **45.7%**.

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.

I think it would be quite difficult to guess the person's education level without the other features. Yes, we could make an educated guess, but quite often there may not be a correlation. For example, someone may have received PhD in Quantum Physics, but decided to become a hairdresser instead. If we are given that this person is a hairdresser, we are unlikely to guess that he/she is a PhD graduate.

Part (e) [2 pt]

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

```
out = autoencoder(zero_out_feature(torch.tensor(test_set[0]).view(1,57), "edu")).detac
print(get_feature(out[0],"edu"))
```

```
Some-college
```

The model predicts some college education

Part (f) [2 pt]

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

The baseline model predicts that the person is a high school graduate.