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.

Saved successfully!] 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/11URkDHsdoE5AQUhyBH1he9s9mgg9W6q-

```
import csv
import numpy as np
import random
import torch
import torch.utils.data
# added matplotlib
import matplotlib.pyplot as plt
from torch.utils.data.dataloader import DataLoader
```

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`

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)
```

```
  <ipython-input-3-837957db2593>:3: ParserWarning: Length of header or names does not match length of data. This leads to a loss of data with index_col=False.
    df = pd.read_csv(
```

```
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	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.

```
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.

```
#min/max/avg:

min = subdf.min()
max = subdf.max()
avg = subdf.mean()
print('The minimum values are: \n', min, '\n', 'The maximum values are: \n', max, '\n', 'The average values are: \n', avg)

#normalize:
df_min_max_scaled = subdf.copy()
for column in df_min_max_scaled.columns:
  df[column] = (df_min_max_scaled[column] - df_min_max_scaled[column].min()) / (df_min_max_scaled[column].max() - df_min_max_scaled[column].min())
print(df)
```

```
    capgain     0
    caploss     0
    workhr      1
    dtype: int64
     The maximum values are:
     age         90
    yredu        16
    capgain   99999
    caploss    4356
    workhr       99
    dtype: int64
     The average values are:
     age          38.581647
    yredu         10.080679
    capgain     1077.648844
    caploss       87.303830
    workhr        40.437456
    dtype: float64
                age                 work  fnlwgt          edu     yredu  \
    0      0.301370            State-gov   77516    Bachelors  0.800000
    1      0.452055       Self-emp-not-inc   83311    Bachelors  0.800000
    2      0.287671              Private  215646      HS-grad  0.533333
    3      0.493151              Private  234721         11th  0.400000
    4      0.150685              Private  338409    Bachelors  0.800000
    ...         ...                  ...     ...          ...       ...
    32556  0.136986              Private  257302  Assoc-acdm  0.733333
    32557  0.315068              Private  154374      HS-grad  0.533333
    32558  0.561644              Private  151910      HS-grad  0.533333
    32559  0.068493              Private  201490      HS-grad  0.533333
    32560  0.479452         Self-emp-inc  287927      HS-grad  0.533333

                    marriage          occupation    relationship    race  \
    0          Never-married        Adm-clerical   Not-in-family   White
    1     Married-civ-spouse     Exec-managerial         Husband   White
    2               Divorced   Handlers-cleaners   Not-in-family   White
    3     Married-civ-spouse   Handlers-cleaners         Husband   Black
    4     Married-civ-spouse      Prof-specialty            Wife   Black
    ...                  ...                 ...             ...     ...
    32556 Married-civ-spouse        Tech-support            Wife   White
    32557 Married-civ-spouse   Machine-op-inspct         Husband   White
    32558            Widowed        Adm-clerical       Unmarried   White
    32559      Never-married        Adm-clerical       Own-child   White
    32560 Married-civ-spouse     Exec-managerial            Wife   White

              sex   capgain  caploss    workhr         country
    0        Male  0.021740      0.0  0.397959   United-States
    1        Male  0.000000      0.0  0.122449   United-States
    2        Male  0.000000      0.0  0.397959   United-States
    3        Male  0.000000      0.0  0.397959   United-States
    4      Female  0.000000      0.0  0.397959            Cuba
    ...       ...       ...      ...       ...            ...
    32556  Female  0.000000      0.0  0.377551   United-States
    32557    Male  0.000000      0.0  0.397959   United-States
    32558  Female  0.000000      0.0  0.397959   United-States
    32559    Male  0.000000      0.0  0.193878   United-States
    32560  Female  0.150242      0.0  0.397959   United-States

    [32561 rows x 14 columns]
```

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
male = sum(df["sex"] == " Male")
#print(male)
tot = len(df['sex'])
#print(tot)
prc = male/tot*100
print(prc,'% of the population are male.')
```

```
    66.92054912318419 % of the population are male.
```

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]
print(len(df_with_missing), 'records contained missing features.', len(df_with_missing)/len(df_not_missing)*100,'% records were removed.')
```

```
    1843 records contained missing features. 5.999739556378819 % 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.

```
val = set(df_not_missing['work'])
#val
```

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	...	edu_ Prof-school	edu_ Some-college	relationship_ Husband	relationship_ Not-in-family	relationship_ Other-relative	relationship_ Own-child	relationship_ Unmarried	relationship_ Wife	sex_ Female	sex_ Male
0	0.301370	0.800000	0.02174	0.0	0.397959	0	0	0	0	0	...	0	0	0	1	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	0	0	0	0	0	1
2	0.287671	0.533333	0.00000	0.0	0.397959	0	0	1	0	0	...	0	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.

```
#There are 57 columns in the df data. This represents the total number of combinations of features,
#including all numerical features taking up 1 column and
#each unique categorical feature taking up 1 column.
```

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)
#print(datanp) #rows as arrays
#print(data)


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]

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

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

    Examples:

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

    for i in range(len(onehot)):
      if onehot[i] >= 1:
        #should return from cat_values['something']
        return (cat_values[feature][i])
        break


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

#print(cat_index)
#cat_index is a dictionary with keys that indicate which index (column) the feature is first observed. i.e) work_ is first found in column 6, so index 5.
#print(cat_values)
#cat_values is a dictionary with keys as arrays that return the values of the columns with the "prefix" of the feature. i.e) work will display all the values that start with "work_"
```

```
    'Private'
```

```
# 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)

#assign indices to all data points, then randomly shuffle them so data is split differently everytime
idx = np.arange(len(datanp))
np.random.shuffle(idx)

#determine the split (convert to integer to properly index)
train_split = int(len(data)*0.7)
val_split = int(train_split + len(data)*0.15)

#split data, determine indices
train_idx = idx[0:train_split]
val_idx = idx[train_split:val_split]
test_idx = idx[val_split:]
train_data = datanp[train_idx]
val_data = datanp[val_idx]
test_data = datanp[test_idx]

#print(train_data, val_data, test_data)

print("Size of the training dataset:", len(train_data))
print("Size of the validation dataset:", len(val_data))
print("Size of the testing dataset:", len(test_data))

# todo
```

```
    Size of the training dataset: 21502
    Size of the validation dataset: 4607
    Size of the testing dataset: 4609
```

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):
        #name AutoEncoder:
        self.name = "AutoEncoder"
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(57, 57), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(57, 36),
            nn.ReLU(),
            nn.Linear(36, 18) # reduce to contain info for ~11 features
        )
        self.decoder = nn.Sequential(
            nn.Linear(18, 36),
            nn.ReLU(),
            nn.Linear(36, 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.)

Saved successfully! ndardizes the data to a range of [0,1], rather than a large range of numbers as seen in the data df.

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 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=5, learning_rate=1e-4):
    """ Training loop. You should update this."""
    torch.manual_seed(42)
    #no data is loaded (updated)
    #load training and validation data
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,shuffle=True)
    val_loader = torch.utils.data.DataLoader(val_data, batch_size=batch_size,shuffle=True)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    # numpy arrays to store training/validation errors and loss, taken from Lab 3
    train_acc, train_loss_list, val_loss_list, val_acc, iters = [], [], [], [], []

    for epoch in range(num_epochs):
      #training model with training dataset first
        for data in train_loader:
            datam = zero_out_random_feature(data.clone()) # zero out one categorical feature
            recon = model(datam)
            train_loss = criterion(recon, data) # train loss
            train_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
      #testing model with validation dataset
        for data in val_loader:
            datam = zero_out_random_feature(data.clone())
            recon = model(datam)
            val_loss = criterion(recon, data)

        iters.append(epoch)
        train_loss_list.append(float(train_loss)/batch_size)   # compute *average* training loss
        val_loss_list.append(float(val_loss)/batch_size)       # compute *average* validation loss
        train_acc.append(get_accuracy(model, train_loader)) # compute training accuracy
        val_acc.append(get_accuracy(model, val_loader))  # compute validation accuracy
        print("epoch number", epoch+1, "accuracy: ",train_acc[epoch])
        #set the path of the csv files:
        path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(model.name, batch_size, learning_rate, epoch)
        torch.save(model.state_dict(), path)
    #plot the training (from lab 2)
    #Loss Curve
    plt.title("Training Curve")
    plt.plot(iters, train_loss_list, label="Train")
    plt.plot(iters, val_loss_list, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    #Accuracy Curve
    plt.title("Training Curve")
    plt.plot(iters, train_acc, label="Train")
    plt.plot(iters, val_acc, label="Validation")
    plt.xlabel("Iterations")
    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.

```
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()
        >>> val = torch.utils.data.DataLoader(data_valid, batch_size=256, shuffle=True)
        >>> get_accuracy(model, val)
    """
    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
```

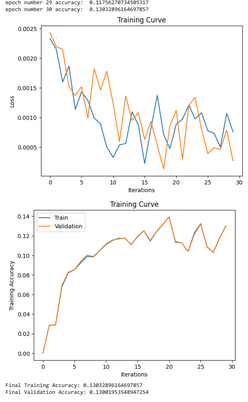
Part (c) [4 pt]

Run your updated training code, using reasonable initial hyperparameters.

Include your training curve in your submission.

```
model1 = AutoEncoder()
train(model1, train_data, val_data, batch_size=16, num_epochs=30, learning_rate=8e-04)
```

```
epoch number 1 accuracy:  0.0
epoch number 2 accuracy:  0.028710587549685
epoch number 3 accuracy:  0.028811273369919076
epoch number 4 accuracy:  0.06770087998015688
epoch number 5 accuracy:  0.08210088983970483
epoch number 6 accuracy:  0.08534866497380886
epoch number 7 accuracy:  0.09245651567290865
epoch number 8 accuracy:  0.0989752893203919
epoch number 9 accuracy:  0.09841728159806356
epoch number 10 accuracy:  0.10481195547700989
epoch number 11 accuracy:  0.11102843146373981
epoch number 12 accuracy:  0.11525284469661737
epoch number 13 accuracy:  0.11739993178929092
epoch number 14 accuracy:  0.117384429355488
epoch number 15 accuracy:  0.11123774321144841
epoch number 16 accuracy:  0.11912845316714724
epoch number 17 accuracy:  0.12521315846587913
epoch number 18 accuracy:  0.11465724490745047
epoch number 19 accuracy:  0.12412798899413078
epoch number 20 accuracy:  0.13163891731001767
epoch number 21 accuracy:  0.13935688004025423
epoch number 22 accuracy:  0.11337785819688091
epoch number 23 accuracy:  0.11276470405857372
epoch number 24 accuracy:  0.10392056552878882
epoch number 25 accuracy:  0.12290329581744334
epoch number 26 accuracy:  0.13246829752271105
epoch number 27 accuracy:  0.10867280151365764
epoch number 28 accuracy:  0.10320745157021052
epoch number 29 accuracy:  0.11756270734585137
epoch number 30 accuracy:  0.13032896164697857
```




```
Final Training Accuracy: 0.13032896164697857
Final Validation Accuracy: 0.13001953548947254
```

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.

```
#not sure why my model accuracy is so low when I am following the proper steps to creating my autoencoder and encoded dataset (if the marker
#could explain that would be great). I want to adjust my hyperparameters by increasing the learning rate of the models and lowering the number of
#epochs.
model = AutoEncoder()
train(model, train_data, val_data, batch_size=16, num_epochs=25, learning_rate=0.005)
```

```
epoch number 1 accuracy:  0.006611788050723964
epoch number 2 accuracy:  0.07991584666232599
epoch number 3 accuracy:  0.06690075341828668
epoch number 4 accuracy:  0.09767388647429386
epoch number 5 accuracy:  0.10533128701206089
epoch number 6 accuracy:  0.10933544125197656
epoch number 7 accuracy:  0.1038372368461848
epoch number 8 accuracy:  0.10004493705825815
epoch number 9 accuracy:  0.10946260856416574
epoch number 10 accuracy:  0.11339887834790375
epoch number 11 accuracy:  0.11075341828667102
epoch number 12 accuracy:  0.11144223464483924
epoch number 13 accuracy:  0.1418559212383344
epoch number 14 accuracy:  0.09947931821216079
epoch number 15 accuracy:  0.17192199175270517
epoch number 16 accuracy:  0.18348680743076633
epoch number 17 accuracy:  0.15858989861640824
epoch number 18 accuracy:  0.16758990145203528
epoch number 19 accuracy:  0.15586147025082094
epoch number 20 accuracy:  0.19057317092983198
epoch number 21 accuracy:  0.14056056800917743
epoch number 22 accuracy:  0.11863266120779462
epoch number 23 accuracy:  0.20056227383405 2894
epoch number 24 accuracy:  0.18626748657580783
epoch number 25 accuracy:  0.2293748117 1984
```



```
Final Training Accuracy: 0.2293748117 1984
Final Validation Accuracy: 0.22867375732580855
```

```
#After changing the learning rate to 0.005 and batch size to 25, the accuracy is much better. I will try increasing
#the learning rate and iterations again in the third attempt.
model3 = AutoEncoder()
train(model3, train_data, val_data, batch_size=16, num_epochs=30, learning_rate=0.01)
```

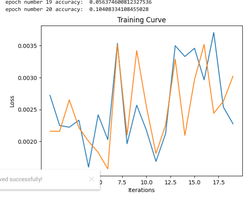
```
epoch number 1 accuracy:  0.015564443617647797
epoch number 2 accuracy:  0.039073884559882218
epoch number 3 accuracy:  0.037859435403358
epoch number 4 accuracy:  0.15111772754829008
epoch number 5 accuracy:  0.14222933284345422
epoch number 6 accuracy:  0.12481784640188509
epoch number 7 accuracy:  0.25190765820223378
epoch number 8 accuracy:  0.26252596657675253
epoch number 9 accuracy:  0.21407159323479985
epoch number 10 accuracy:  0.29854587317018572
epoch number 11 accuracy:  0.32857348861315322267
epoch number 12 accuracy:  0.40747372337450983
epoch number 13 accuracy:  0.42509513806467616 6
epoch number 14 accuracy:  0.4711948531919511
epoch number 15 accuracy:  0.46321272439577304
epoch number 16 accuracy:  0.4737777633088 3949
epoch number 17 accuracy:  0.5128747713393802
epoch number 18 accuracy:  0.5136343905993241
epoch number 19 accuracy:  0.4957928859180456 6
epoch number 20 accuracy:  0.5117663473165287
epoch number 21 accuracy:  0.5003835153343875
epoch number 22 accuracy:  0.5139060242458059
epoch number 23 accuracy:  0.5166617244990745
epoch number 24 accuracy:  0.4926983536415217
epoch number 25 accuracy:  0.4798047891917444
epoch number 26 accuracy:  0.4797228164838773
epoch number 27 accuracy:  0.4901488233552626 6
epoch number 28 accuracy:  0.4934967288586450 86
epoch number 29 accuracy:  0.4545701175 0844883
epoch number 30 accuracy:  0.2990729544513 84924
```



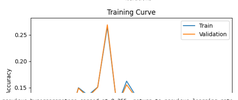
```
Final Training Accuracy: 0.2990729544513 84924
Final Validation Accuracy: 0.3025468489979874
```

```
#increased the learning rate to 0.02
model4 = AutoEncoder()
train(model4, train_data, val_data, batch_size=16, num_epochs=20, learning_rate=0.02)
```

```
epoch number 1 accuracy:  0.010789693981955168
epoch number 2 accuracy:  0.015827984993644004
epoch number 3 accuracy:  0.019347837484885126
epoch number 4 accuracy:  0.08777478064056057
epoch number 5 accuracy:  0.14597845161603854
epoch number 6 accuracy:  0.13549902134665541
epoch number 7 accuracy:  0.15111772548259008
epoch number 8 accuracy:  0.26499804478095586
epoch number 9 accuracy:  0.11491179115121074
epoch number 10 accuracy:  0.16229498031190898
epoch number 11 accuracy:  0.13155040461352431
epoch number 12 accuracy:  0.08113973891981343
epoch number 13 accuracy:  0.09926983536415217
epoch number 14 accuracy:  0.10567234055746752
epoch number 15 accuracy:  0.0220955632893422937
epoch number 16 accuracy:  0.023679967754937525
epoch number 17 accuracy:  0.06533366198493163
epoch number 18 accuracy:  0.060095184944036221
epoch number 19 accuracy:  0.056374600812327536
epoch number 20 accuracy:  0.10408834188455028
```

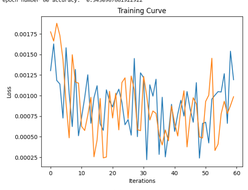


Saved successfully!




```
#the previous hyperparameters capped at 0.255, return to previous learning rate = 0.005, try epochs = 60
model5 = AutoEncoder()
train(model5, train_data, val_data, batch_size=16, num_epochs=60, learning_rate=0.005)
```

```
epoch number 1 accuracy:  0.006945090379189533
epoch number 2 accuracy:  0.026222366911610512
epoch number 3 accuracy:  0.013074442687501594
epoch number 4 accuracy:  0.04243791275230211
epoch number 5 accuracy:  0.0585362360759285028
epoch number 6 accuracy:  0.08585418349424859
epoch number 7 accuracy:  0.08405419650885189
epoch number 8 accuracy:  0.07964375406933889
epoch number 9 accuracy:  0.10041701547142902
epoch number 10 accuracy:  0.13688649311791151
epoch number 11 accuracy:  0.09223173038166992
epoch number 12 accuracy:  0.13467739415091388
epoch number 13 accuracy:  0.11753170247728893
epoch number 14 accuracy:  0.10884258828636095
epoch number 15 accuracy:  0.10185874182246613
epoch number 16 accuracy:  0.17938641366694572
epoch number 17 accuracy:  0.161113229779875
epoch number 18 accuracy:  0.08906148265277648
epoch number 19 accuracy:  0.13430533593774222
epoch number 20 accuracy:  0.11251666511642328
epoch number 21 accuracy:  0.13413480916503892
epoch number 22 accuracy:  0.11284996744488884
epoch number 23 accuracy:  0.12530841518775928
epoch number 24 accuracy:  0.06767587511239265
epoch number 25 accuracy:  0.11081139738939014
epoch number 26 accuracy:  0.09491365144432766
epoch number 27 accuracy:  0.11864787771608153
epoch number 28 accuracy:  0.14627210489473846
epoch number 29 accuracy:  0.15052881209538827
epoch number 30 accuracy:  0.17086007503178
epoch number 31 accuracy:  0.13595625213150465 9
epoch number 32 accuracy:  0.15224161519393545
epoch number 33 accuracy:  0.17541089393763285
epoch number 34 accuracy:  0.16170814634297584
epoch number 35 accuracy:  0.19250814727312881
epoch number 36 accuracy:  0.23865221839828854
epoch number 37 accuracy:  0.21134468113493504
epoch number 38 accuracy:  0.22411808663338015
epoch number 39 accuracy:  0.24450438718878864
epoch number 40 accuracy:  0.24446563110408334
epoch number 41 accuracy:  0.19673363718103742
epoch number 42 accuracy:  0.34517719281929726
epoch number 43 accuracy:  0.29423619588262628
epoch number 44 accuracy:  0.35792891934787485
epoch number 45 accuracy:  0.31893157225660443
epoch number 46 accuracy:  0.33473638059966516
epoch number 47 accuracy:  0.3775230211451496
epoch number 48 accuracy:  0.3629972405667 0896
epoch number 49 accuracy:  0.39585385390595216
epoch number 50 accuracy:  0.42618516107028803
epoch number 51 accuracy:  0.44877320723653616
epoch number 52 accuracy:  0.3792747961429945
epoch number 53 accuracy:  0.45687228939994356
epoch number 54 accuracy:  0.46811920447214225
epoch number 55 accuracy:  0.5206492419390832
epoch number 56 accuracy:  0.5005270827515921
epoch number 57 accuracy:  0.48207143523153288
epoch number 58 accuracy:  0.5265103571760767
epoch number 59 accuracy:  0.5198276129352308
epoch number 60 accuracy:  0.5430967661922922
```




```
Final Training Accuracy: 0.5430967661922922
Final Validation Accuracy: 0.5414224704048299
```

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

```
#load test dataset
test_loader = torch.utils.data.DataLoader(test_data, batch_size=16, shuffle=True)
#model
test_acc = get_accuracy(model5,test_loader)

#print results
print('The testing accuracy is', test_acc*100, '%.')
```

```
The testing accuracy is 53.887321904641499 %.
```

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?

```
#create baseline model
baseline = {}
for col in df_not_missing.columns:
  baseline[col] = df_not_missing[col].value_counts().idxmax()
#calculate the accuracy
accuracy = sum(df_not_missing['marriage'] == baseline['marriage'])/len(df_not_missing)
print("The accuracy of the model missing the marriage feature is:", accuracy*100, '%.')
```

```
The accuracy of the model missing the marriage feature is: 46.67947131974758 %.
```

Part (c) [1 pt]

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

```
#The accuracy of the baseline model in B is worse. It sits at 46.67% while the test accuracy from A sits at 53.88%.
```

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.

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

```
#Yes, a human should be able to correlate features like "work" and "occupation" to make a reasonable
#estimate to a person's education level. However, anomalies always exist, so you can never be sure.
```

```
{'work': 'Private',
 'marriage': 'Never-married',
 'occupation': 'Transport-moving',
 'edu': '11th',
 'relationship': 'Own-child',
 'sex': 'Female'}
```

Part (e) [2 pt]

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

```
test_edu = zero_out_feature(test_data[:1], 'edu')[0]
predict = model5(torch.from_numpy(test_edu))
#print(predict)
pred = predict.detach().numpy()
prediction = get_feature(pred, 'edu')
print(prediction)
```

```
None
```

```
#can't get get_feature() to print out the proper feature even though the model is being called properly
```

Part (f) [2 pt]

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

```
baseline['edu']
```

```
' HS-grad'
```