Lab 4: Data Imputation using an Autoencoder

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

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

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

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

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

In the process, you are expected to learn to:

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

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

What to submit

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

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Colab Link

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

Colab Link: https://colab.research.google.com/drive/1fQpHsdoGsg5_ovE5jMRHRr3vu7zZ4ODd?usp=sharing

In [49]:
```
import csv
import numpy as np
import random
import torch
import torch.utils.data
```

```
This application is used to convert notebook files (*.ipynb)
        to various other formats.

        WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.

Options
=======
The options below are convenience aliases to configurable class-options,
as listed in the "Equivalent to" description-line of the aliases.
To see all configurable class-options for some <cmd>, use:
    <cmd> --help-all

--debug
    set log level to logging.DEBUG (maximize logging output)
    Equivalent to: [--Application.log_level=10]
--show-config
    Show the application's configuration (human-readable format)
    Equivalent to: [--Application.show_config=True]
--show-config-json
    Show the application's configuration (json format)
    Equivalent to: [--Application.show_config_json=True]
--generate-config
    generate default config file
    Equivalent to: [--JupyterApp.generate_config=True]
-y
    Answer yes to any questions instead of prompting.
    Equivalent to: [--JupyterApp.answer_yes=True]
--execute
    Execute the notebook prior to export.
    Equivalent to: [--ExecutePreprocessor.enabled=True]
--allow-errors
    Continue notebook execution even if one of the cells throws an error and incl
ude the error message in the cell output (the default behaviour is to abort conve
rsion). This flag is only relevant if '--execute' was specified, too.
    Equivalent to: [--ExecutePreprocessor.allow_errors=True]
--stdin
    read a single notebook file from stdin. Write the resulting notebook with def
ault basename 'notebook.*'
    Equivalent to: [--NbConvertApp.from_stdin=True]
--stdout
    Write notebook output to stdout instead of files.
    Equivalent to: [--NbConvertApp.writer_class=StdoutWriter]
--inplace
    Run nbconvert in place, overwriting the existing notebook (only
            relevant when converting to notebook format)
    Equivalent to: [--NbConvertApp.use_output_suffix=False --NbConvertApp.export_
format=notebook --FilesWriter.build_directory=]
--clear-output
    Clear output of current file and save in place,
            overwriting the existing notebook.
    Equivalent to: [--NbConvertApp.use_output_suffix=False --NbConvertApp.export_
format=notebook --FilesWriter.build_directory= --ClearOutputPreprocessor.enabled=
True]
--coalesce-streams
    Coalesce consecutive stdout and stderr outputs into one stream (within each c
ell).
    Equivalent to: [--NbConvertApp.use_output_suffix=False --NbConvertApp.export_
format=notebook --FilesWriter.build_directory= --CoalesceStreamsPreprocessor.enab
led=True]
--no-prompt
```

```
    Exclude input and output prompts from converted document.
    Equivalent to: [--TemplateExporter.exclude_input_prompt=True --TemplateExport
er.exclude_output_prompt=True]
--no-input
    Exclude input cells and output prompts from converted document.
            This mode is ideal for generating code-free reports.
    Equivalent to: [--TemplateExporter.exclude_output_prompt=True --TemplateExpor
ter.exclude_input=True --TemplateExporter.exclude_input_prompt=True]
--allow-chromium-download
    Whether to allow downloading chromium if no suitable version is found on the
system.
    Equivalent to: [--WebPDFExporter.allow_chromium_download=True]
--disable-chromium-sandbox
    Disable chromium security sandbox when converting to PDF..
    Equivalent to: [--WebPDFExporter.disable_sandbox=True]
--show-input
    Shows code input. This flag is only useful for dejavu users.
    Equivalent to: [--TemplateExporter.exclude_input=False]
--embed-images
    Embed the images as base64 dataurls in the output. This flag is only useful f
or the HTML/WebPDF/Slides exports.
    Equivalent to: [--HTMLExporter.embed_images=True]
--sanitize-html
    Whether the HTML in Markdown cells and cell outputs should be sanitized..
    Equivalent to: [--HTMLExporter.sanitize_html=True]
--log-level=<Enum>
    Set the log level by value or name.
    Choices: any of [0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CR
ITICAL']
    Default: 30
    Equivalent to: [--Application.log_level]
--config=<Unicode>
    Full path of a config file.
    Default: ''
    Equivalent to: [--JupyterApp.config_file]
--to=<Unicode>
    The export format to be used, either one of the built-in formats
            ['asciidoc', 'custom', 'html', 'latex', 'markdown', 'notebook', 'pd
f', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides', 'webpdf']
            or a dotted object name that represents the import path for an
            ``Exporter`` class
    Default: ''
    Equivalent to: [--NbConvertApp.export_format]
--template=<Unicode>
    Name of the template to use
    Default: ''
    Equivalent to: [--TemplateExporter.template_name]
--template-file=<Unicode>
    Name of the template file to use
    Default: None
    Equivalent to: [--TemplateExporter.template_file]
--theme=<Unicode>
    Template specific theme(e.g. the name of a JupyterLab CSS theme distributed
    as prebuilt extension for the lab template)
    Default: 'light'
    Equivalent to: [--HTMLExporter.theme]
--sanitize_html=<Bool>
    Whether the HTML in Markdown cells and cell outputs should be sanitized.This
    should be set to True by nbviewer or similar tools.
    Default: False
```

```
    Equivalent to: [--HTMLExporter.sanitize_html]
--writer=<DottedObjectName>
    Writer class used to write the
                                    results of the conversion
    Default: 'FilesWriter'
    Equivalent to: [--NbConvertApp.writer_class]
--post=<DottedOrNone>
    PostProcessor class used to write the
                                    results of the conversion
    Default: ''
    Equivalent to: [--NbConvertApp.postprocessor_class]
--output=<Unicode>
    Overwrite base name use for output files.
                Supports pattern replacements '{notebook_name}'.
    Default: '{notebook_name}'
    Equivalent to: [--NbConvertApp.output_base]
--output-dir=<Unicode>
    Directory to write output(s) to. Defaults
                                  to output to the directory of each notebook. To
recover
                                  previous default behaviour (outputting to the c
urrent
                                  working directory) use . as the flag value.
    Default: ''
    Equivalent to: [--FilesWriter.build_directory]
--reveal-prefix=<Unicode>
    The URL prefix for reveal.js (version 3.x).
            This defaults to the reveal CDN, but can be any url pointing to a cop
y
            of reveal.js.
            For speaker notes to work, this must be a relative path to a local
            copy of reveal.js: e.g., "reveal.js".
            If a relative path is given, it must be a subdirectory of the
            current directory (from which the server is run).
            See the usage documentation
            (https://nbconvert.readthedocs.io/en/latest/usage.html#reveal-js-html
-slideshow)
            for more details.
    Default: ''
    Equivalent to: [--SlidesExporter.reveal_url_prefix]
--nbformat=<Enum>
    The nbformat version to write.
            Use this to downgrade notebooks.
    Choices: any of [1, 2, 3, 4]
    Default: 4
    Equivalent to: [--NotebookExporter.nbformat_version]

Examples
--------

    The simplest way to use nbconvert is

            > jupyter nbconvert mynotebook.ipynb --to html

            Options include ['asciidoc', 'custom', 'html', 'latex', 'markdown',
'notebook', 'pdf', 'python', 'qtpdf', 'qtpng', 'rst', 'script', 'slides', 'webpd
f'].

            > jupyter nbconvert --to latex mynotebook.ipynb
```

```
            Both HTML and LaTeX support multiple output templates. LaTeX includes
            'base', 'article' and 'report'.  HTML includes 'basic', 'lab' and
            'classic'. You can specify the flavor of the format used.

            > jupyter nbconvert --to html --template lab mynotebook.ipynb

            You can also pipe the output to stdout, rather than a file

            > jupyter nbconvert mynotebook.ipynb --stdout

            PDF is generated via latex

            > jupyter nbconvert mynotebook.ipynb --to pdf

            You can get (and serve) a Reveal.js-powered slideshow

            > jupyter nbconvert myslides.ipynb --to slides --post serve

            Multiple notebooks can be given at the command line in a couple of
            different ways:

            > jupyter nbconvert notebook*.ipynb
            > jupyter nbconvert notebook1.ipynb notebook2.ipynb

            or you can specify the notebooks list in a config file, containing::

                c.NbConvertApp.notebooks = ["my_notebook.ipynb"]

            > jupyter nbconvert --config mycfg.py

To see all available configurables, use `--help-all`.
```

In [59]:
```
jupyter nbconvert --to html "/content/Lab4 Data Imputation.ipynb"
```

```
  File "/tmp/ipython-input-59-207554202.py", line 1
    jupyter nbconvert --to html "/content/Lab4 Data Imputation.ipynb"
            ^
SyntaxError: invalid syntax
```

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://colab.research.google.com/drive/1fQpHsdoGsg5_ovE5jMRHRr3vu7zZ4ODd?usp=sharing

In [2]:
```
import pandas as pd
```

Part 1. Data Cleaning [15 pt]

The adult.data file is available at `https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data`

The function `pd.read_csv` loads the adult.data file into a pandas dataframe. You can read about the pandas documentation for `pd.read_csv` at https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

In [3]:
```
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)
```

```
/tmp/ipython-input-3-1831985018.py:3: ParserWarning: Length of header or names do
es not match length of data. This leads to a loss of data with index_col=False.
  df = pd.read_csv(
```

In [4]:
```
df.shape # there are 32561 rows (records) in the data frame, and 14 columns (fea
```

Out[4]: (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 [5]:
```
df[:3] # show the first 3 records
```

Out[5]:

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

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

In [7]:
```
subdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
subdf[:3] # show the first 3 records
```

Out[7]:

	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 [8]:
```
np.sum(subdf["caploss"])
```

Out[8]:
```
np.int64(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 [9]:
```
columns = ["age", "yredu", "capgain", "caploss", "workhr"]
for col in columns:
  print(f"min {col}: {np.min(df[col])}")
  print(f"max {col}: {np.max(df[col])}")
  print(f"average {col}: {np.average(df[col])}")
normcolumns =   ["age", "yredu", "capgain", "caploss", "workhr"]
for col in normcolumns:
  df[col] = (df[col]- np.min(df[col]))/(np.max(df[col])-np.min(df[col]))
df[:10]
```

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

Out[9]:

	age	work	fnlwgt	edu	yredu	marriage	occupation	relationsh
0	0.301370	State-gov	77516	Bachelors	0.800000	Never-married	Adm-clerical	Not-i fam
1	0.452055	Self-emp-not-inc	83311	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husbar
2	0.287671	Private	215646	HS-grad	0.533333	Divorced	Handlers-cleaners	Not-i fam
3	0.493151	Private	234721	11th	0.400000	Married-civ-spouse	Handlers-cleaners	Husbar
4	0.150685	Private	338409	Bachelors	0.800000	Married-civ-spouse	Prof-specialty	Wi
5	0.273973	Private	284582	Masters	0.866667	Married-civ-spouse	Exec-managerial	Wi
6	0.438356	Private	160187	9th	0.266667	Married-spouse-absent	Other-service	Not-i fam
7	0.479452	Self-emp-not-inc	209642	HS-grad	0.533333	Married-civ-spouse	Exec-managerial	Husbar
8	0.191781	Private	45781	Masters	0.866667	Never-married	Prof-specialty	Not-i fam
9	0.342466	Private	159449	Bachelors	0.800000	Married-civ-spouse	Exec-managerial	Husbar

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 [10]:

```
# hint: you can do something like this in pandas
male= sum(df["sex"] == " Male")
female = sum(df["sex"] == " Female")
male_percen = male/ (male+female)
male_percen
```

Out[10]: 0.6692054912318418

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 [11]:
```
contcols = ["age", "yredu", "capgain", "caploss", "workhr"]
catcols = ["work", "marriage", "occupation", "edu", "relationship", "sex"]
features = contcols + catcols
df = df[features]
```

In [12]:
```
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 [13]:
```
sum_missing = sum(missing)
print(f"there are {sum_missing} missing rows")
print(f"the percentage of missing data removed is {sum_missing/len(df)}")
```

```
there are 1843 missing rows
the percentage of missing data removed is 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 [14]:
```
work_values = set(df_not_missing["work"])
print(work_values)
np.shape(df_not_missing)
```

```
{' Federal-gov', ' State-gov', ' Private', ' Local-gov', ' Self-emp-inc', ' Witho
ut-pay', ' Self-emp-not-inc'}
```

Out[14]: `(30718, 11)`

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 [15]:
```
data = pd.get_dummies(df_not_missing)
```

In [16]:
```
data[:3]
```

Out[16]:

	age	yredu	capgain	caploss	workhr	work_ Federal-gov	work_ Local-gov	work_ Private	work_ Self-emp-inc
0	0.301370	0.800000	0.02174	0.0	0.397959	False	False	False	False
1	0.452055	0.800000	0.00000	0.0	0.122449	False	False	False	False
2	0.287671	0.533333	0.00000	0.0	0.397959	False	False	True	False

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 now 57 columns in the data, there was originally 11 columns in df. This indicates that there are 46 extra columns. Each original feature column is now spreaded into columns containing all its types, and for each group of feature columns on each row, there is only one true label and the other columns are false.

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 [17]:
```
datanp = data.values.astype(np.float32)
```

In [18]:
```
cat_index = {}  # Mapping of feature -> start index of feature in a record
cat_values = {} # Mapping of feature -> list of categorical values the feature c

# 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 underscor
        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.
    start = cat_index[feature]
    values = cat_values[feature]
    relative_index = np.argmax(onehot)
    actual_index = start + relative_index
    return values[relative_index]
```

In [19]:
```
# 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 [20]:
```
# set the numpy seed for reproducibility
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html
np.random.seed(50)

# todo
indices = np.random.permutation(len(df_not_missing))

n_total = len(indices)
n_train = int(n_total * 0.7)
n_val = int(n_total * 0.15)
n_test = n_total - n_train - n_val

train_idx = indices[:n_train]
val_idx = indices[n_train:n_train + n_val]
test_idx = indices[n_train + n_val:]

train_set = df_not_missing.iloc[train_idx]
val_set = df_not_missing.iloc[val_idx]
test_set = df_not_missing.iloc[test_idx]

print(f"Training set size: {len(train_set)}")
print(f"Validation set size: {len(val_set)}")
print(f"Test set size: {len(test_set)}")
```

```
Training set size: 21502
Validation set size: 4607
Test set size: 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.

In [21]:
```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(57,32),
            nn.ReLU(),
            nn.Linear(32,16),
            nn.ReLU(),
            nn.Linear(16,8),
        )
        self.decoder = nn.Sequential(
            nn.Linear(8, 16),
            nn.ReLU(),
            nn.Linear(16, 32),
            nn.ReLU(),
            nn.Linear(32, 57),
            nn.Sigmoid()
        )

    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.)

Sigmoid function does the same thing as normalization, mapping data from 0 to 1. Since data has been normalized before encoder, this helps the decoder to reconstruct it and match input distribution.

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 [38]:
```
def get_model_name(name, learning_rate, epoch):
    """ Generate a name for the model consisting of all the hyperparameter value

    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

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()

    plt.title("Train vs Validation Loss")
    plt.plot(range(1,n+1), train_loss, label="Train")
```

```
    plt.plot(range(1,n+1), val_loss, label="Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()


def get_loss(model, data_loader, criterion):
    total_loss = 0.0
    for data in data_loader:
        data = data[0] # unwrap the tensor from the list
        datam = zero_out_random_feature(data.clone()) # zero out one categorical
        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, valid_loader, num_epochs=5, learning_rate=1e-4):
    torch.manual_seed(42)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

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

    for epoch in range(num_epochs):
        model.train()
        total_train_loss = 0.0

        for data in train_loader:
            data = data[0] # unwrap the tensor from the list
            datam = zero_out_random_feature(data.clone())  # zero out one catego
            recon = model(datam)
            loss = criterion(recon, data)

            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            total_train_loss += loss.item()

        # Record losses & accuracy at the end of epoch
        model.eval()
```

```
        train_loss[epoch] = float(total_train_loss) / len(train_loader)
        val_loss[epoch] = get_loss(model, valid_loader, criterion)

        train_acc[epoch] = get_accuracy(model, train_loader)
        val_acc[epoch] = get_accuracy(model, valid_loader)

        print(("Epoch {}: Train acc: {:.4f}, Train loss: {:.4f} | "
               "Validation acc: {:.4f}, Validation loss: {:.4f}").format(
                   epoch + 1,
                   train_acc[epoch],
                   train_loss[epoch],
                   val_acc[epoch],
                   val_loss[epoch]))

        # Save model checkpoint every 5 epochs
        if (epoch + 1) % 5 == 0:
          model_path = get_model_name("autoencoder", learning_rate, epoch+1)
          torch.save(model.state_dict(), model_path)

    print('Finished Training')

    # Save metrics to CSV for plotting after training
    model_path = get_model_name("autoencoder", learning_rate, num_epochs)
    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)
```

In [ ]:

```
```

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 [36]:

```
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, shuffl
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            data = item[0] # unwrap the tensor from the list
            inp = data.detach().numpy()
            out = model(zero_out_feature(data.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.

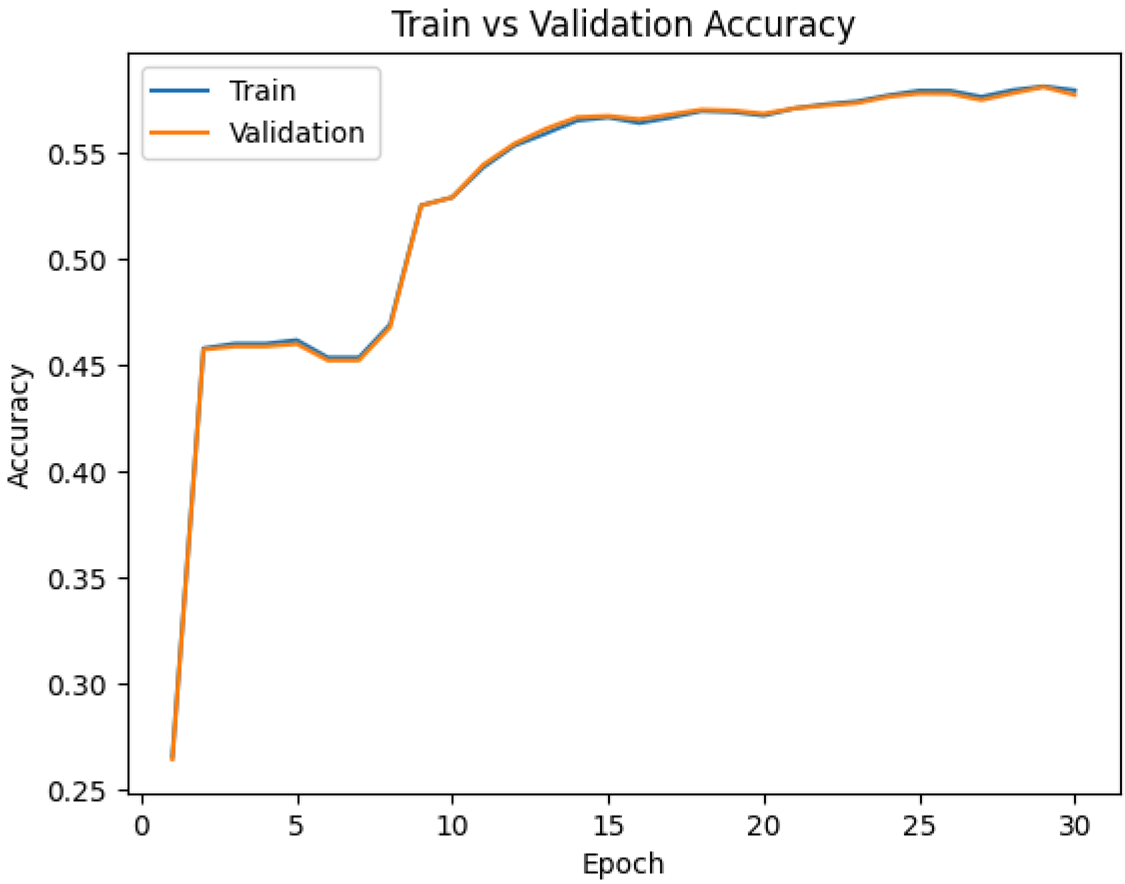
In [39]:

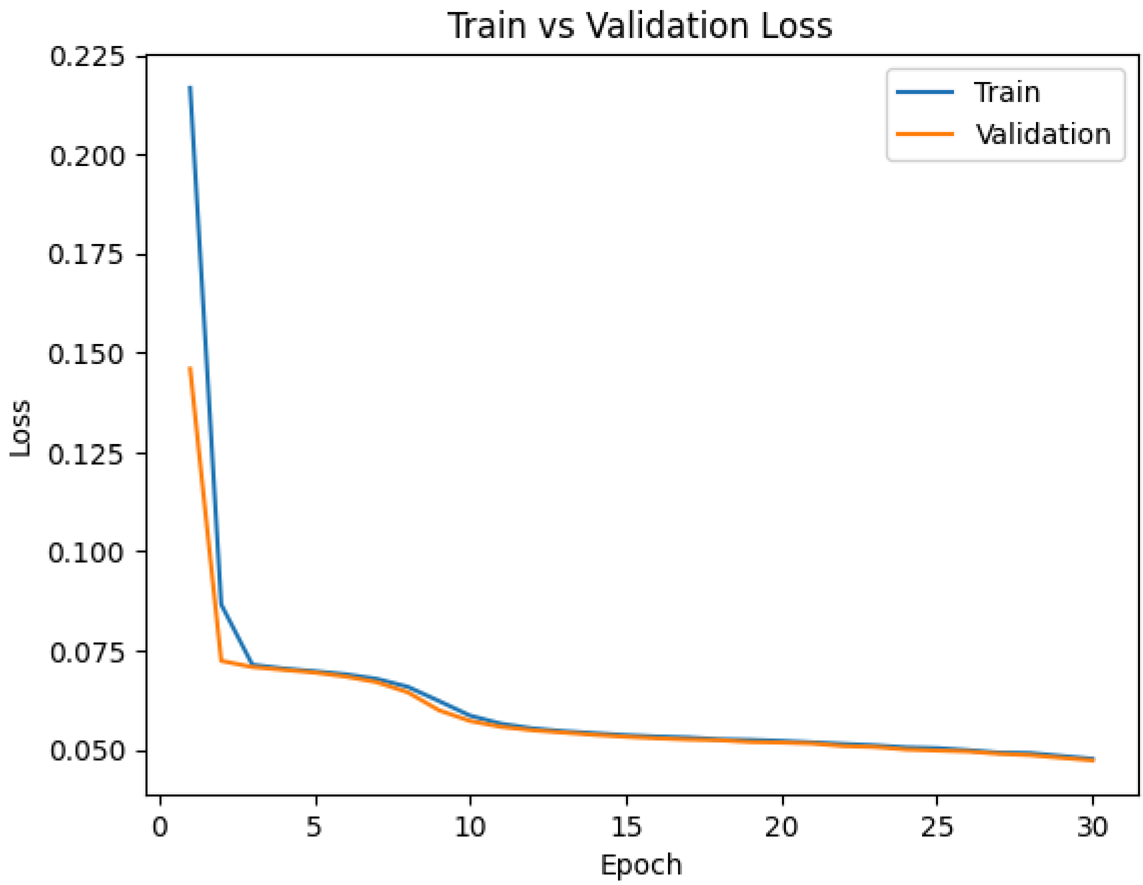
```
import torch
from torch.utils.data import TensorDataset, DataLoader

def get_data_loader(train_data, val_data, test_data, batch_size):
    """
    Convert tabular data into PyTorch DataLoaders, mimicking the CIFAR-style fun

    Args:
        train_data: numpy array for training
        val_data: numpy array for validation
        test_data: numpy array for testing
        batch_size: batch size to use

    Returns:
        train_loader, val_loader, test_loader
    """
    # Convert to torch tensors
    train_tensor = torch.tensor(train_data, dtype=torch.float32)
    val_tensor = torch.tensor(val_data, dtype=torch.float32)
    test_tensor = torch.tensor(test_data, dtype=torch.float32)

    # Wrap each tensor in a TensorDataset
    train_dataset = TensorDataset(train_tensor)
    val_dataset = TensorDataset(val_tensor)
    test_dataset = TensorDataset(test_tensor)

    # Create DataLoaders (no sampler needed)
```

```
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True
    val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False, n
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False,

    return train_loader, val_loader, test_loader

# Split the one-hot encoded data
train_data = datanp[train_idx]
val_data = datanp[val_idx]
test_data = datanp[test_idx]

train_loader, val_loader, test_loader = get_data_loader(train_data, val_data, te
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=1e-4)
```

```
Epoch 1: Train acc: 0.2657, Train loss: 0.2167 | Validation acc: 0.2644, Validati
on loss: 0.1459
Epoch 2: Train acc: 0.4577, Train loss: 0.0866 | Validation acc: 0.4571, Validati
on loss: 0.0724
Epoch 3: Train acc: 0.4598, Train loss: 0.0714 | Validation acc: 0.4586, Validati
on loss: 0.0708
Epoch 4: Train acc: 0.4599, Train loss: 0.0704 | Validation acc: 0.4586, Validati
on loss: 0.0701
Epoch 5: Train acc: 0.4616, Train loss: 0.0697 | Validation acc: 0.4598, Validati
on loss: 0.0694
Epoch 6: Train acc: 0.4533, Train loss: 0.0690 | Validation acc: 0.4520, Validati
on loss: 0.0684
Epoch 7: Train acc: 0.4533, Train loss: 0.0678 | Validation acc: 0.4520, Validati
on loss: 0.0670
Epoch 8: Train acc: 0.4689, Train loss: 0.0658 | Validation acc: 0.4676, Validati
on loss: 0.0644
Epoch 9: Train acc: 0.5250, Train loss: 0.0622 | Validation acc: 0.5249, Validati
on loss: 0.0599
Epoch 10: Train acc: 0.5288, Train loss: 0.0585 | Validation acc: 0.5288, Validat
ion loss: 0.0572
Epoch 11: Train acc: 0.5429, Train loss: 0.0565 | Validation acc: 0.5442, Validat
ion loss: 0.0557
Epoch 12: Train acc: 0.5532, Train loss: 0.0553 | Validation acc: 0.5541, Validat
ion loss: 0.0549
Epoch 13: Train acc: 0.5588, Train loss: 0.0546 | Validation acc: 0.5610, Validat
ion loss: 0.0543
Epoch 14: Train acc: 0.5650, Train loss: 0.0541 | Validation acc: 0.5665, Validat
ion loss: 0.0537
Epoch 15: Train acc: 0.5664, Train loss: 0.0536 | Validation acc: 0.5671, Validat
ion loss: 0.0532
Epoch 16: Train acc: 0.5638, Train loss: 0.0533 | Validation acc: 0.5656, Validat
ion loss: 0.0528
Epoch 17: Train acc: 0.5663, Train loss: 0.0530 | Validation acc: 0.5678, Validat
ion loss: 0.0525
Epoch 18: Train acc: 0.5694, Train loss: 0.0526 | Validation acc: 0.5701, Validat
ion loss: 0.0523
Epoch 19: Train acc: 0.5690, Train loss: 0.0524 | Validation acc: 0.5697, Validat
ion loss: 0.0519
Epoch 20: Train acc: 0.5673, Train loss: 0.0522 | Validation acc: 0.5682, Validat
ion loss: 0.0518
Epoch 21: Train acc: 0.5707, Train loss: 0.0518 | Validation acc: 0.5707, Validat
ion loss: 0.0515
Epoch 22: Train acc: 0.5726, Train loss: 0.0514 | Validation acc: 0.5720, Validat
ion loss: 0.0509
Epoch 23: Train acc: 0.5740, Train loss: 0.0510 | Validation acc: 0.5732, Validat
ion loss: 0.0506
Epoch 24: Train acc: 0.5768, Train loss: 0.0505 | Validation acc: 0.5761, Validat
ion loss: 0.0500
Epoch 25: Train acc: 0.5790, Train loss: 0.0503 | Validation acc: 0.5777, Validat
ion loss: 0.0497
Epoch 26: Train acc: 0.5790, Train loss: 0.0498 | Validation acc: 0.5776, Validat
ion loss: 0.0495
Epoch 27: Train acc: 0.5761, Train loss: 0.0492 | Validation acc: 0.5746, Validat
ion loss: 0.0489
Epoch 28: Train acc: 0.5792, Train loss: 0.0491 | Validation acc: 0.5778, Validat
ion loss: 0.0486
Epoch 29: Train acc: 0.5810, Train loss: 0.0484 | Validation acc: 0.5807, Validat
ion loss: 0.0479
Epoch 30: Train acc: 0.5792, Train loss: 0.0477 | Validation acc: 0.5770, Validat
```

```
ion loss: 0.0473
Finished Training
```





Part (d) [5 pt]

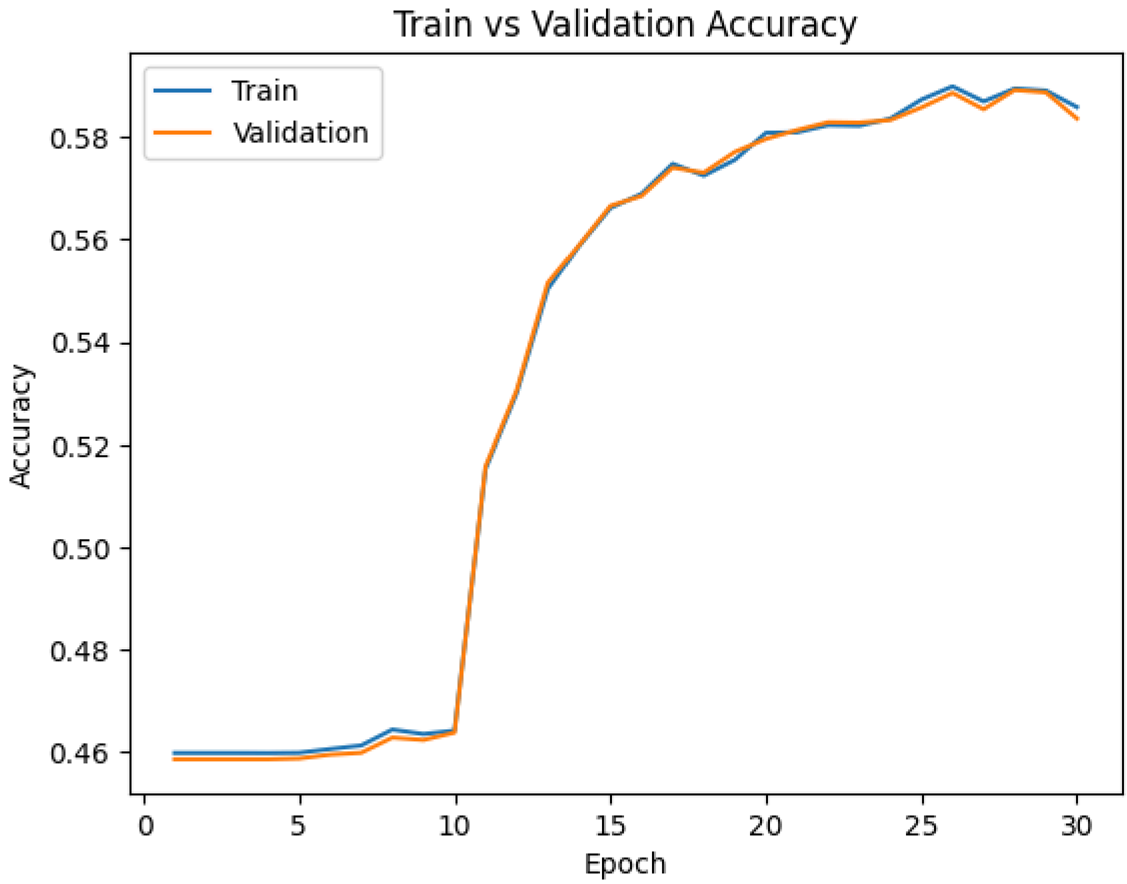
Tune your hyperparameters, training at least 4 different models (4 sets of hyperparameters).

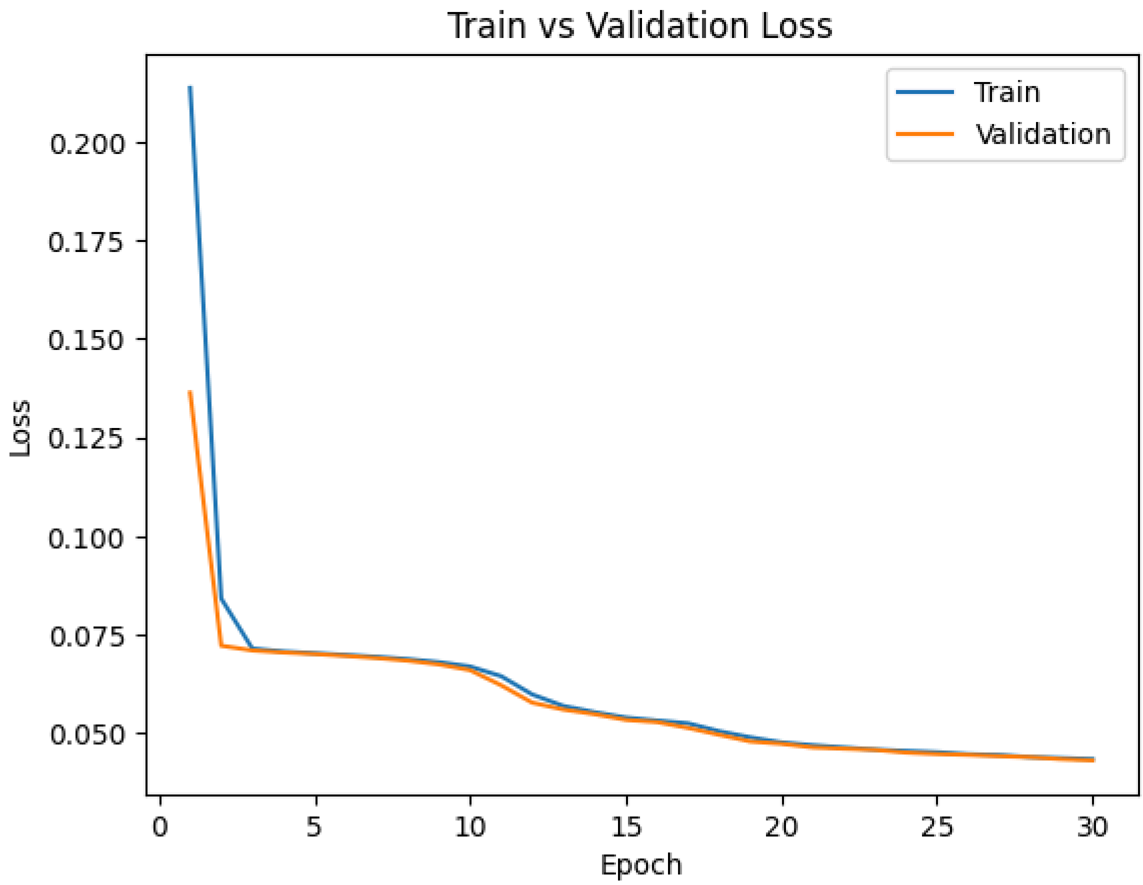
Do not include all your training curves. Instead, explain what hyperparameters you tried, what their effect was, and what your thought process was as you chose the next set of hyperparameters to try.

In [40]:
```
# Model 1: Baseline
train_loader, val_loader, test_loader = get_data_loader(train_data, val_data, te
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=1e-4)

# Model 2: Higher learning rate, smaller batch size
train_loader, val_loader, test_loader = get_data_loader(train_data, val_data, te
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=30, learning_rate=5e-4)

# Model 3: Lower learning rate, bigger batch size, more epochs
train_loader, val_loader, test_loader = get_data_loader(train_data, val_data, te
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=50, learning_rate=5e-5)

# Model 4: Shorter training, fastest learning
train_loader, val_loader, test_loader = get_data_loader(train_data, val_data, te
autoencoder = AutoEncoder()
train(autoencoder, train_loader, val_loader, num_epochs=20, learning_rate=1e-3)

#Model 1 (LR=1e-4, BS=64): Stable but slow learning, reached 58% accuracy.
#Model 2 (LR=5e-4, BS=32): Faster convergence, improved to 62%, but some fluctua
#Model 3 (LR=5e-5, BS=128, 50 epochs): Very slow learning, lower accuracy (55%),
#Model 4 (LR=1e-3, BS=16, 20 epochs): Best performance (63%), fast learning, sma
```

```
Epoch 1: Train acc: 0.4598, Train loss: 0.2137 | Validation acc: 0.4586, Validati
on loss: 0.1363
Epoch 2: Train acc: 0.4598, Train loss: 0.0842 | Validation acc: 0.4586, Validati
on loss: 0.0722
Epoch 3: Train acc: 0.4598, Train loss: 0.0715 | Validation acc: 0.4586, Validati
on loss: 0.0710
Epoch 4: Train acc: 0.4598, Train loss: 0.0707 | Validation acc: 0.4586, Validati
on loss: 0.0705
Epoch 5: Train acc: 0.4599, Train loss: 0.0703 | Validation acc: 0.4588, Validati
on loss: 0.0701
Epoch 6: Train acc: 0.4606, Train loss: 0.0699 | Validation acc: 0.4595, Validati
on loss: 0.0696
Epoch 7: Train acc: 0.4613, Train loss: 0.0694 | Validation acc: 0.4599, Validati
on loss: 0.0691
Epoch 8: Train acc: 0.4644, Train loss: 0.0688 | Validation acc: 0.4628, Validati
on loss: 0.0684
Epoch 9: Train acc: 0.4635, Train loss: 0.0680 | Validation acc: 0.4624, Validati
on loss: 0.0675
Epoch 10: Train acc: 0.4642, Train loss: 0.0669 | Validation acc: 0.4638, Validat
ion loss: 0.0660
Epoch 11: Train acc: 0.5153, Train loss: 0.0645 | Validation acc: 0.5158, Validat
ion loss: 0.0622
Epoch 12: Train acc: 0.5300, Train loss: 0.0598 | Validation acc: 0.5306, Validat
ion loss: 0.0577
Epoch 13: Train acc: 0.5504, Train loss: 0.0569 | Validation acc: 0.5516, Validat
ion loss: 0.0560
Epoch 14: Train acc: 0.5588, Train loss: 0.0553 | Validation acc: 0.5590, Validat
ion loss: 0.0548
Epoch 15: Train acc: 0.5662, Train loss: 0.0540 | Validation acc: 0.5666, Validat
ion loss: 0.0534
Epoch 16: Train acc: 0.5689, Train loss: 0.0532 | Validation acc: 0.5685, Validat
ion loss: 0.0528
Epoch 17: Train acc: 0.5748, Train loss: 0.0525 | Validation acc: 0.5741, Validat
ion loss: 0.0513
Epoch 18: Train acc: 0.5725, Train loss: 0.0505 | Validation acc: 0.5731, Validat
ion loss: 0.0496
Epoch 19: Train acc: 0.5756, Train loss: 0.0489 | Validation acc: 0.5771, Validat
ion loss: 0.0479
Epoch 20: Train acc: 0.5809, Train loss: 0.0476 | Validation acc: 0.5797, Validat
ion loss: 0.0473
Epoch 21: Train acc: 0.5810, Train loss: 0.0469 | Validation acc: 0.5814, Validat
ion loss: 0.0464
Epoch 22: Train acc: 0.5824, Train loss: 0.0464 | Validation acc: 0.5829, Validat
ion loss: 0.0461
Epoch 23: Train acc: 0.5822, Train loss: 0.0458 | Validation acc: 0.5828, Validat
ion loss: 0.0458
Epoch 24: Train acc: 0.5836, Train loss: 0.0455 | Validation acc: 0.5834, Validat
ion loss: 0.0451
Epoch 25: Train acc: 0.5873, Train loss: 0.0452 | Validation acc: 0.5858, Validat
ion loss: 0.0447
Epoch 26: Train acc: 0.5899, Train loss: 0.0447 | Validation acc: 0.5886, Validat
ion loss: 0.0445
Epoch 27: Train acc: 0.5870, Train loss: 0.0444 | Validation acc: 0.5854, Validat
ion loss: 0.0442
Epoch 28: Train acc: 0.5895, Train loss: 0.0439 | Validation acc: 0.5892, Validat
ion loss: 0.0439
Epoch 29: Train acc: 0.5891, Train loss: 0.0437 | Validation acc: 0.5887, Validat
ion loss: 0.0434
Epoch 30: Train acc: 0.5859, Train loss: 0.0434 | Validation acc: 0.5836, Validat
```

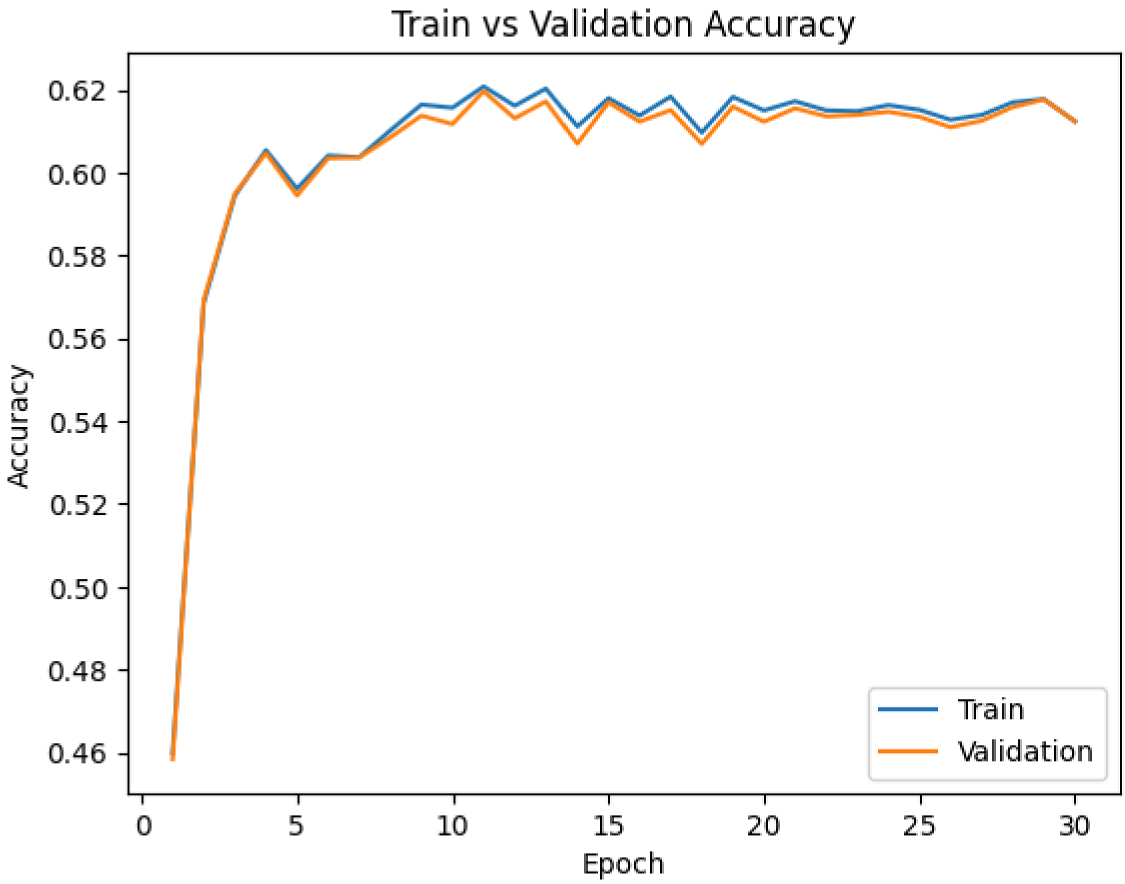
```
ion loss: 0.0431
Finished Training
```

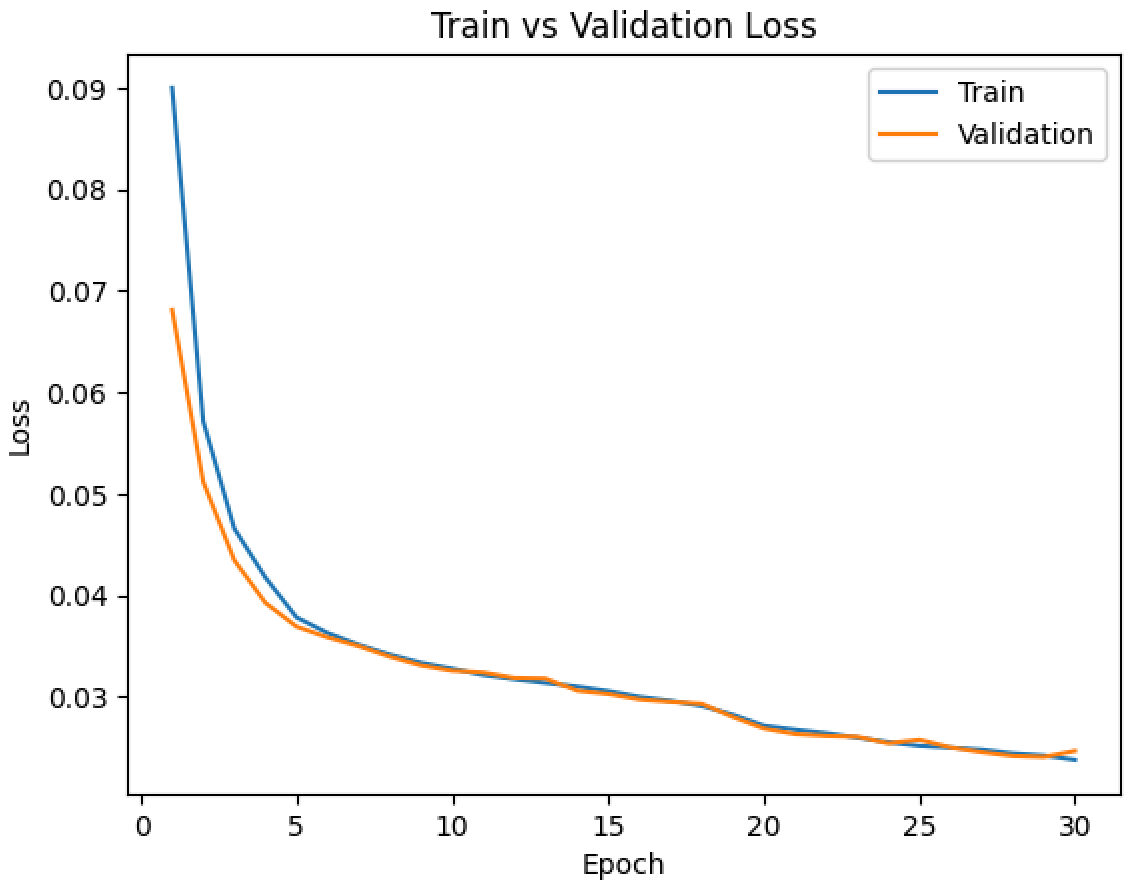




```
Epoch 1: Train acc: 0.4598, Train loss: 0.0900 | Validation acc: 0.4584, Validati
on loss: 0.0681
Epoch 2: Train acc: 0.5685, Train loss: 0.0572 | Validation acc: 0.5693, Validati
on loss: 0.0512
Epoch 3: Train acc: 0.5944, Train loss: 0.0466 | Validation acc: 0.5949, Validati
on loss: 0.0434
Epoch 4: Train acc: 0.6054, Train loss: 0.0417 | Validation acc: 0.6047, Validati
on loss: 0.0392
Epoch 5: Train acc: 0.5961, Train loss: 0.0378 | Validation acc: 0.5945, Validati
on loss: 0.0369
Epoch 6: Train acc: 0.6041, Train loss: 0.0363 | Validation acc: 0.6035, Validati
on loss: 0.0358
Epoch 7: Train acc: 0.6037, Train loss: 0.0351 | Validation acc: 0.6036, Validati
on loss: 0.0350
Epoch 8: Train acc: 0.6102, Train loss: 0.0341 | Validation acc: 0.6085, Validati
on loss: 0.0339
Epoch 9: Train acc: 0.6164, Train loss: 0.0333 | Validation acc: 0.6137, Validati
on loss: 0.0331
Epoch 10: Train acc: 0.6157, Train loss: 0.0328 | Validation acc: 0.6117, Validat
ion loss: 0.0326
Epoch 11: Train acc: 0.6208, Train loss: 0.0321 | Validation acc: 0.6196, Validat
ion loss: 0.0324
Epoch 12: Train acc: 0.6161, Train loss: 0.0317 | Validation acc: 0.6131, Validat
ion loss: 0.0318
Epoch 13: Train acc: 0.6203, Train loss: 0.0314 | Validation acc: 0.6172, Validat
ion loss: 0.0318
Epoch 14: Train acc: 0.6111, Train loss: 0.0310 | Validation acc: 0.6070, Validat
ion loss: 0.0306
Epoch 15: Train acc: 0.6180, Train loss: 0.0305 | Validation acc: 0.6170, Validat
ion loss: 0.0303
Epoch 16: Train acc: 0.6138, Train loss: 0.0300 | Validation acc: 0.6123, Validat
ion loss: 0.0297
Epoch 17: Train acc: 0.6183, Train loss: 0.0296 | Validation acc: 0.6151, Validat
ion loss: 0.0295
Epoch 18: Train acc: 0.6097, Train loss: 0.0291 | Validation acc: 0.6069, Validat
ion loss: 0.0293
Epoch 19: Train acc: 0.6183, Train loss: 0.0282 | Validation acc: 0.6159, Validat
ion loss: 0.0280
Epoch 20: Train acc: 0.6150, Train loss: 0.0271 | Validation acc: 0.6123, Validat
ion loss: 0.0269
Epoch 21: Train acc: 0.6172, Train loss: 0.0267 | Validation acc: 0.6155, Validat
ion loss: 0.0263
Epoch 22: Train acc: 0.6150, Train loss: 0.0264 | Validation acc: 0.6136, Validat
ion loss: 0.0261
Epoch 23: Train acc: 0.6148, Train loss: 0.0260 | Validation acc: 0.6139, Validat
ion loss: 0.0260
Epoch 24: Train acc: 0.6163, Train loss: 0.0255 | Validation acc: 0.6147, Validat
ion loss: 0.0254
Epoch 25: Train acc: 0.6152, Train loss: 0.0252 | Validation acc: 0.6135, Validat
ion loss: 0.0257
Epoch 26: Train acc: 0.6128, Train loss: 0.0250 | Validation acc: 0.6110, Validat
ion loss: 0.0250
Epoch 27: Train acc: 0.6139, Train loss: 0.0248 | Validation acc: 0.6125, Validat
ion loss: 0.0245
Epoch 28: Train acc: 0.6170, Train loss: 0.0244 | Validation acc: 0.6158, Validat
ion loss: 0.0242
Epoch 29: Train acc: 0.6177, Train loss: 0.0242 | Validation acc: 0.6176, Validat
ion loss: 0.0241
Epoch 30: Train acc: 0.6124, Train loss: 0.0238 | Validation acc: 0.6124, Validat
```

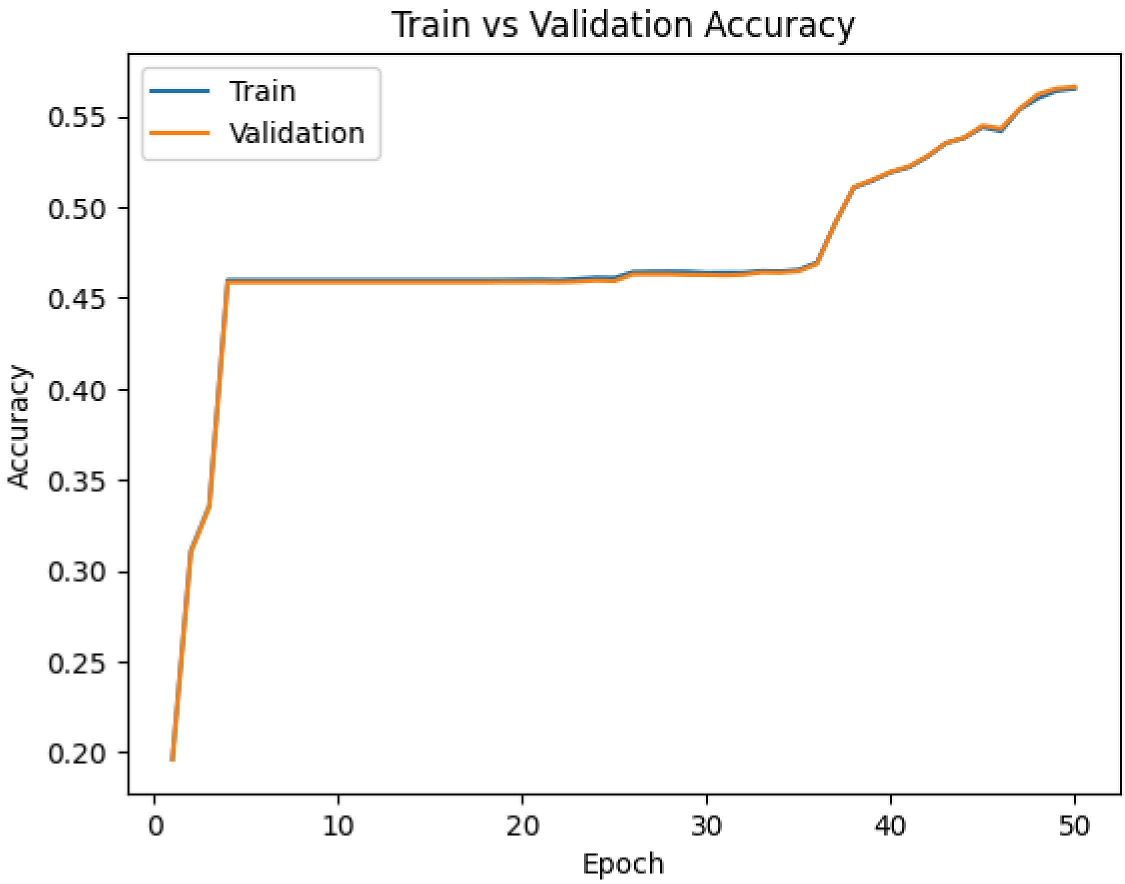
```
ion loss: 0.0246
Finished Training
```



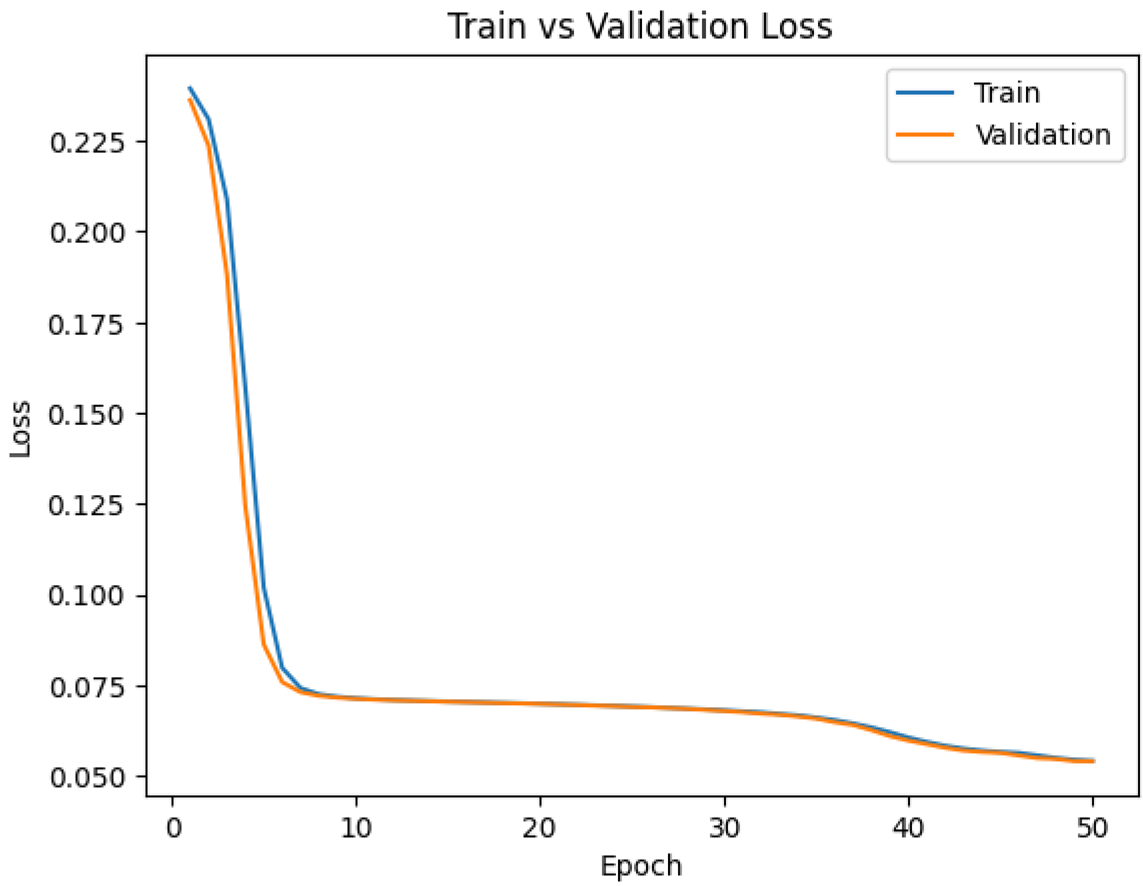


```
Epoch 1: Train acc: 0.1961, Train loss: 0.2395 | Validation acc: 0.1961, Validati
on loss: 0.2363
Epoch 2: Train acc: 0.3109, Train loss: 0.2310 | Validation acc: 0.3103, Validati
on loss: 0.2237
Epoch 3: Train acc: 0.3356, Train loss: 0.2089 | Validation acc: 0.3353, Validati
on loss: 0.1882
Epoch 4: Train acc: 0.4598, Train loss: 0.1570 | Validation acc: 0.4586, Validati
on loss: 0.1244
Epoch 5: Train acc: 0.4598, Train loss: 0.1020 | Validation acc: 0.4586, Validati
on loss: 0.0863
Epoch 6: Train acc: 0.4598, Train loss: 0.0799 | Validation acc: 0.4586, Validati
on loss: 0.0760
Epoch 7: Train acc: 0.4598, Train loss: 0.0743 | Validation acc: 0.4586, Validati
on loss: 0.0732
Epoch 8: Train acc: 0.4598, Train loss: 0.0726 | Validation acc: 0.4586, Validati
on loss: 0.0722
Epoch 9: Train acc: 0.4598, Train loss: 0.0719 | Validation acc: 0.4586, Validati
on loss: 0.0717
Epoch 10: Train acc: 0.4598, Train loss: 0.0715 | Validation acc: 0.4586, Validat
ion loss: 0.0714
Epoch 11: Train acc: 0.4598, Train loss: 0.0712 | Validation acc: 0.4586, Validat
ion loss: 0.0711
Epoch 12: Train acc: 0.4598, Train loss: 0.0710 | Validation acc: 0.4586, Validat
ion loss: 0.0710
Epoch 13: Train acc: 0.4598, Train loss: 0.0709 | Validation acc: 0.4586, Validat
ion loss: 0.0708
Epoch 14: Train acc: 0.4598, Train loss: 0.0707 | Validation acc: 0.4586, Validat
ion loss: 0.0707
Epoch 15: Train acc: 0.4598, Train loss: 0.0706 | Validation acc: 0.4586, Validat
ion loss: 0.0705
Epoch 16: Train acc: 0.4598, Train loss: 0.0705 | Validation acc: 0.4586, Validat
ion loss: 0.0704
Epoch 17: Train acc: 0.4598, Train loss: 0.0704 | Validation acc: 0.4586, Validat
ion loss: 0.0703
Epoch 18: Train acc: 0.4598, Train loss: 0.0702 | Validation acc: 0.4586, Validat
ion loss: 0.0702
Epoch 19: Train acc: 0.4599, Train loss: 0.0701 | Validation acc: 0.4588, Validat
ion loss: 0.0701
Epoch 20: Train acc: 0.4600, Train loss: 0.0700 | Validation acc: 0.4587, Validat
ion loss: 0.0700
Epoch 21: Train acc: 0.4601, Train loss: 0.0699 | Validation acc: 0.4588, Validat
ion loss: 0.0698
Epoch 22: Train acc: 0.4599, Train loss: 0.0697 | Validation acc: 0.4586, Validat
ion loss: 0.0696
Epoch 23: Train acc: 0.4605, Train loss: 0.0696 | Validation acc: 0.4590, Validat
ion loss: 0.0695
Epoch 24: Train acc: 0.4612, Train loss: 0.0694 | Validation acc: 0.4596, Validat
ion loss: 0.0693
Epoch 25: Train acc: 0.4611, Train loss: 0.0692 | Validation acc: 0.4593, Validat
ion loss: 0.0691
Epoch 26: Train acc: 0.4643, Train loss: 0.0691 | Validation acc: 0.4630, Validat
ion loss: 0.0690
Epoch 27: Train acc: 0.4645, Train loss: 0.0689 | Validation acc: 0.4631, Validat
ion loss: 0.0688
Epoch 28: Train acc: 0.4646, Train loss: 0.0686 | Validation acc: 0.4631, Validat
ion loss: 0.0686
Epoch 29: Train acc: 0.4645, Train loss: 0.0684 | Validation acc: 0.4627, Validat
ion loss: 0.0683
Epoch 30: Train acc: 0.4639, Train loss: 0.0682 | Validation acc: 0.4627, Validat
ion loss: 0.0680
```

```
Epoch 31: Train acc: 0.4640, Train loss: 0.0679 | Validation acc: 0.4624, Validat
ion loss: 0.0677
Epoch 32: Train acc: 0.4640, Train loss: 0.0676 | Validation acc: 0.4628, Validat
ion loss: 0.0673
Epoch 33: Train acc: 0.4649, Train loss: 0.0672 | Validation acc: 0.4641, Validat
ion loss: 0.0670
Epoch 34: Train acc: 0.4647, Train loss: 0.0668 | Validation acc: 0.4640, Validat
ion loss: 0.0665
Epoch 35: Train acc: 0.4655, Train loss: 0.0662 | Validation acc: 0.4647, Validat
ion loss: 0.0659
Epoch 36: Train acc: 0.4695, Train loss: 0.0655 | Validation acc: 0.4687, Validat
ion loss: 0.0649
Epoch 37: Train acc: 0.4914, Train loss: 0.0646 | Validation acc: 0.4914, Validat
ion loss: 0.0642
Epoch 38: Train acc: 0.5107, Train loss: 0.0635 | Validation acc: 0.5111, Validat
ion loss: 0.0627
Epoch 39: Train acc: 0.5145, Train loss: 0.0621 | Validation acc: 0.5150, Validat
ion loss: 0.0611
Epoch 40: Train acc: 0.5191, Train loss: 0.0607 | Validation acc: 0.5195, Validat
ion loss: 0.0598
Epoch 41: Train acc: 0.5219, Train loss: 0.0595 | Validation acc: 0.5224, Validat
ion loss: 0.0589
Epoch 42: Train acc: 0.5277, Train loss: 0.0584 | Validation acc: 0.5281, Validat
ion loss: 0.0579
Epoch 43: Train acc: 0.5353, Train loss: 0.0576 | Validation acc: 0.5353, Validat
ion loss: 0.0571
Epoch 44: Train acc: 0.5381, Train loss: 0.0571 | Validation acc: 0.5383, Validat
ion loss: 0.0567
Epoch 45: Train acc: 0.5441, Train loss: 0.0567 | Validation acc: 0.5449, Validat
ion loss: 0.0565
Epoch 46: Train acc: 0.5418, Train loss: 0.0565 | Validation acc: 0.5433, Validat
ion loss: 0.0558
Epoch 47: Train acc: 0.5538, Train loss: 0.0558 | Validation acc: 0.5541, Validat
ion loss: 0.0550
Epoch 48: Train acc: 0.5599, Train loss: 0.0551 | Validation acc: 0.5622, Validat
ion loss: 0.0548
Epoch 49: Train acc: 0.5641, Train loss: 0.0546 | Validation acc: 0.5653, Validat
ion loss: 0.0542
Epoch 50: Train acc: 0.5653, Train loss: 0.0543 | Validation acc: 0.5663, Validat
ion loss: 0.0541
Finished Training
```

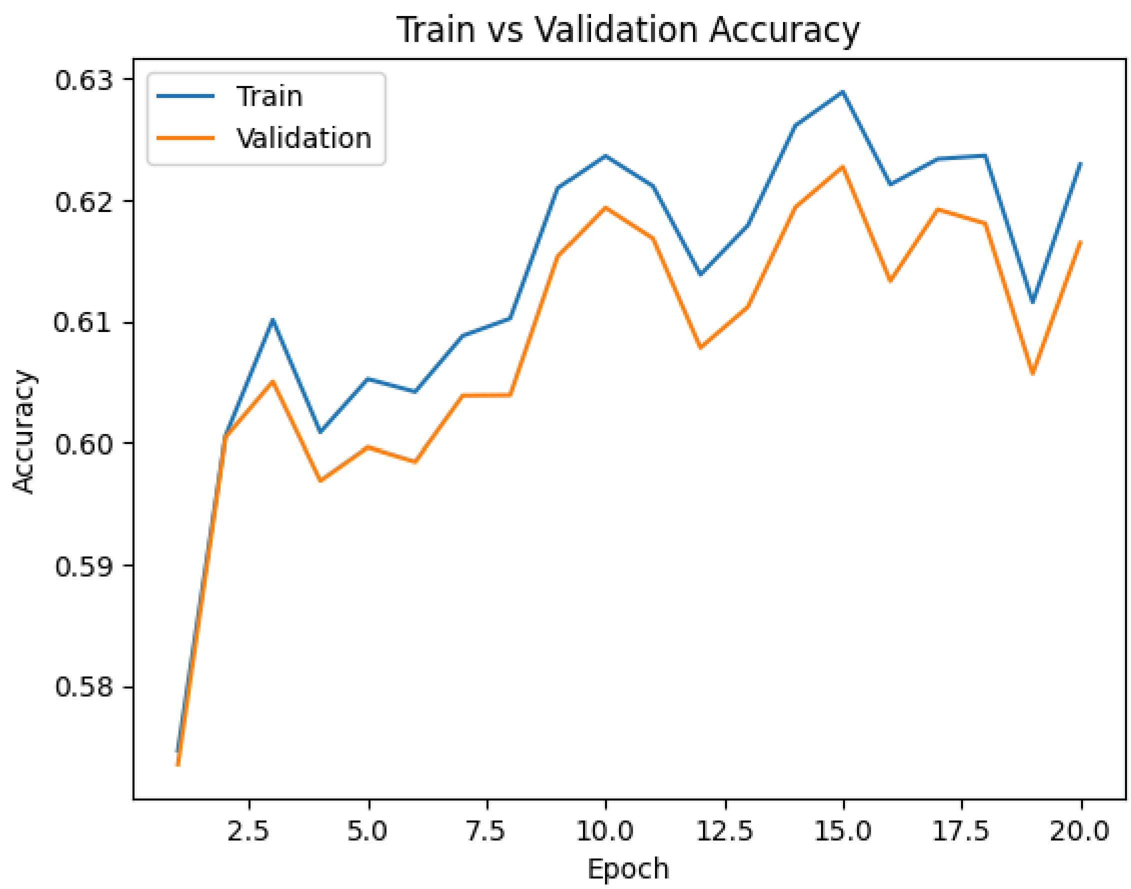


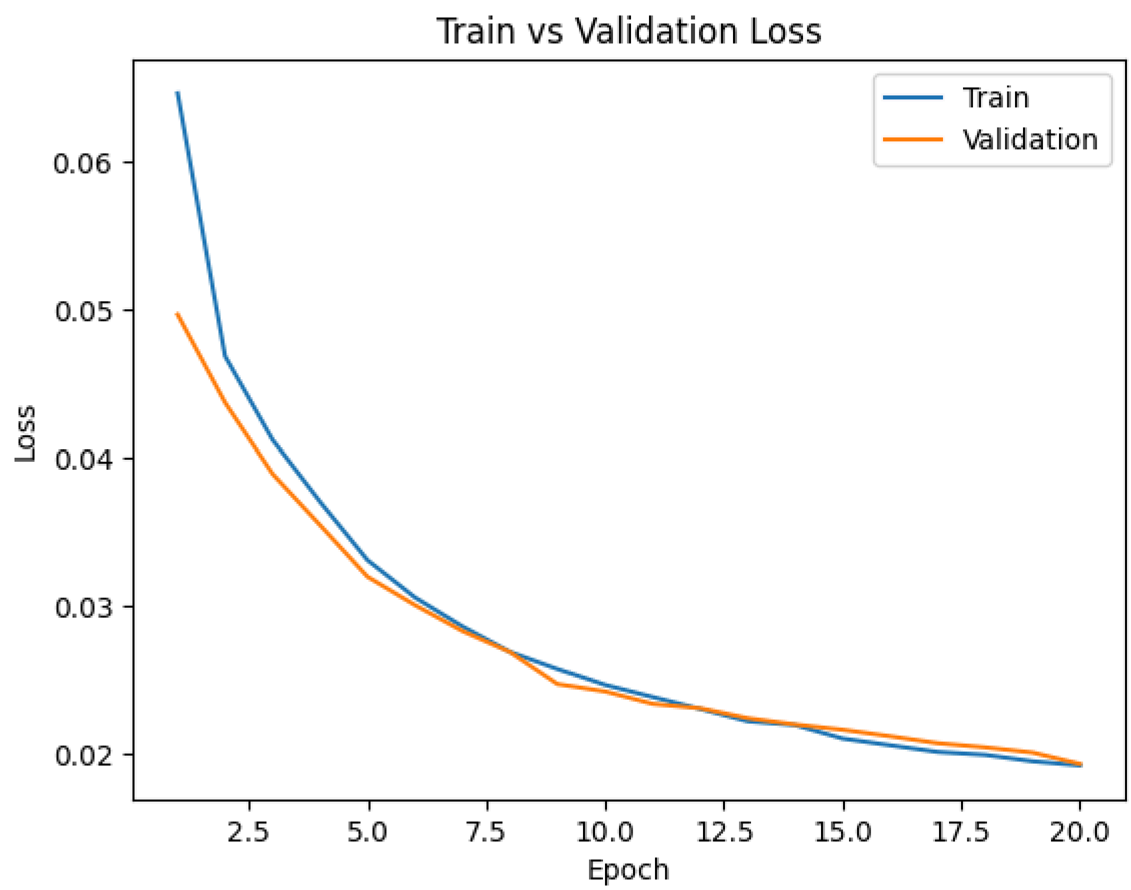
Train vs Validation Accuracy
Train
Validation
Accuracy
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0
10
20
30
40
50
Epoch



Train vs Validation Loss
Train
Validation
Loss
0.225
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0
10
20
30
40
50
Epoch

```
Epoch 1: Train acc: 0.5746, Train loss: 0.0646 | Validation acc: 0.5735, Validati
on loss: 0.0497
Epoch 2: Train acc: 0.6005, Train loss: 0.0469 | Validation acc: 0.6004, Validati
on loss: 0.0437
Epoch 3: Train acc: 0.6101, Train loss: 0.0412 | Validation acc: 0.6050, Validati
on loss: 0.0389
Epoch 4: Train acc: 0.6009, Train loss: 0.0371 | Validation acc: 0.5969, Validati
on loss: 0.0355
Epoch 5: Train acc: 0.6052, Train loss: 0.0331 | Validation acc: 0.5996, Validati
on loss: 0.0320
Epoch 6: Train acc: 0.6042, Train loss: 0.0306 | Validation acc: 0.5984, Validati
on loss: 0.0301
Epoch 7: Train acc: 0.6088, Train loss: 0.0286 | Validation acc: 0.6039, Validati
on loss: 0.0283
Epoch 8: Train acc: 0.6102, Train loss: 0.0269 | Validation acc: 0.6039, Validati
on loss: 0.0269
Epoch 9: Train acc: 0.6210, Train loss: 0.0258 | Validation acc: 0.6153, Validati
on loss: 0.0247
Epoch 10: Train acc: 0.6236, Train loss: 0.0247 | Validation acc: 0.6193, Validat
ion loss: 0.0242
Epoch 11: Train acc: 0.6211, Train loss: 0.0239 | Validation acc: 0.6168, Validat
ion loss: 0.0234
Epoch 12: Train acc: 0.6138, Train loss: 0.0231 | Validation acc: 0.6078, Validat
ion loss: 0.0231
Epoch 13: Train acc: 0.6179, Train loss: 0.0222 | Validation acc: 0.6112, Validat
ion loss: 0.0224
Epoch 14: Train acc: 0.6261, Train loss: 0.0220 | Validation acc: 0.6194, Validat
ion loss: 0.0220
Epoch 15: Train acc: 0.6289, Train loss: 0.0211 | Validation acc: 0.6227, Validat
ion loss: 0.0217
Epoch 16: Train acc: 0.6213, Train loss: 0.0206 | Validation acc: 0.6133, Validat
ion loss: 0.0212
Epoch 17: Train acc: 0.6234, Train loss: 0.0202 | Validation acc: 0.6192, Validat
ion loss: 0.0208
Epoch 18: Train acc: 0.6236, Train loss: 0.0200 | Validation acc: 0.6180, Validat
ion loss: 0.0205
Epoch 19: Train acc: 0.6116, Train loss: 0.0195 | Validation acc: 0.6057, Validat
ion loss: 0.0201
Epoch 20: Train acc: 0.6229, Train loss: 0.0193 | Validation acc: 0.6165, Validat
ion loss: 0.0194
Finished Training
```





Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [42]:
```
test_acc = get_accuracy(autoencoder, test_loader)
print(f"Test accuracy: {test_acc:.4f}")
```

```
Test accuracy: 0.6180
```

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 [45]:
```
acc_list = []
for feature in catcols:
    # Find the most common value for the feature in the training set
    mode_value = train_set[feature].mode()[0]
    # Calculate accuracy for this feature in the test set
    acc = sum(test_set[feature] == mode_value) / len(test_set)
    acc_list.append(acc)

# Calculate the average accuracy across all categorical features
baseline_acc = sum(acc_list) / len(acc_list)
print(f"Baseline accuracy: {baseline_acc:.4f}")
```

```
Baseline accuracy: 0.4568
```

Part (c) [1 pt]

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

In [ ]:
```
#The test accuracy comparing to the baseline is much better which is 62%
```

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.

It's hard to guess someone's education level from other features. For instance, someone with a master's degree might be working as a barista, which wouldn't clearly indicate their education level. So the correlation can be weak.

Part (e) [2 pt]

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

In [48]:
```
out = autoencoder(zero_out_feature(torch.tensor(test_data[0]).unsqueeze(0), "edu
print(get_feature(out[0],"edu"))
```

```
12th
```

Part (f) [2 pt]

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

In [ ]:
```
#Baseline predicted the person's education level is high school
```

In [54]:
```
import os

# Replace with your filename
filename = "Lab4 Data Imputation.ipynb"
full_path = os.path.abspath(filename)
print(full_path)
```

```
/content/Lab4 Data Imputation.ipynb
```

This will print the current working directory, which is usually `/content/` in Google Colab.

Your notebook file will be in this directory. You can find the exact name of your notebook file by looking at the tab in your browser or the file explorer on the left side of the Colab interface.

So, the path to your current notebook file will be `/content/` followed by the name of your notebook file (e.g., `/content/MyNotebook.ipynb`).