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/1OqUBMKFIMYXwGWn2gINu7LPGCREXDFdf?usp=sharing

In [1]:
```
import csv
import numpy as np
import random
import torch
import torch.utils.data
```

Part 0

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

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

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)
```

```
<ipython-input-3-037957db2593>: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(
```

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

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

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

In [6]:
```
subdf = df[["age", "yredu", "capgain", "caploss", "workhr"]]
subdf[:3] # show the first 3 records
```

Out[6]:

	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 [7]:
```
np.sum(subdf["caploss"])
```

Out[7]:
```
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 [8]:
```
#report min, max, average
columns = ["age", "yredu", "capgain", "caploss", "workhr"]
summary = df[columns].agg(['min', 'max', 'mean'])
summary
```

Out[8]:

	age	yredu	capgain	caploss	workhr
min	17.000000	1.000000	0.000000	0.00000	1.000000
max	90.000000	16.000000	99999.000000	4356.00000	99.000000
mean	38.581647	10.080679	1077.648844	87.30383	40.437456

In [9]:
```
#Min-Max Normalization
min_values = summary.loc['min']
max_values = summary.loc['max']
```

```
df[columns] = (df[columns]-min_values)/(max_values-min_values)
df[:3]
```

Out[9]:

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

Part (b) Categorical Features [1 pt]

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

What percentage of people in our data set are female?

In [10]:
```
# hint: you can do something like this in pandas
sum(df["sex"] == " Male")
```

Out[10]:
```
21790
```

In [11]:
```
people = len(df)
male = sum(df["sex"] == " Male")
female = sum(df["sex"] == " Female")
print(f"Percentage of Male: {(male/people)*100}%")
print(f"Percentage of Female: {(female/people)*100}%")
```

```
Percentage of Male: 66.92054912318419%
Percentage of Female: 33.07945087681583%
```

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 [12]:
```
contcols = ["age", "yredu", "capgain", "caploss", "workhr"]
catcols = ["work", "marriage", "occupation", "edu", "relationship", "sex"]
features = contcols + catcols
df = df[features]
```

In [13]:
```
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 [14]:
```
print(f"Total of {len(df_with_missing)} records contained missing features")
print(f"{(len(df_with_missing)/len(df))*100}% of records were removed")
```

```
Total of 1843 records contained missing features
5.660145572924664% of records were removed
```

Part (d) One-Hot Encoding [1 pt]

What are all the possible values of the feature "work" in `df_not_missing`? You may find the Python function `set` useful.

In [15]:
```
set(df_not_missing["work"])
```

Out[15]:
```
{' Federal-gov',
 ' Local-gov',
 ' Private',
 ' Self-emp-inc',
 ' Self-emp-not-inc',
 ' State-gov',
 ' Without-pay'}
```

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 [16]:
```python
data = pd.get_dummies(df_not_missing)
```

In [17]:
```python
data[:3]
```

Out[17]:

	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
0	0.301370	0.800000	0.02174	0.0	0.397959	False	False	False	False	False	...	False	False	False	True
1	0.452055	0.800000	0.00000	0.0	0.122449	False	False	False	False	True	...	False	False	True	False
2	0.287671	0.533333	0.00000	0.0	0.397959	False	False	True	False	False	...	False	False	False	True

3 rows × 57 columns

Part (e) One-Hot Encoding [2 pt]

The dataframe `data` contains the cleaned and normalized data that we will use to train our denoising autoencoder.

How many **columns** (features) are in the dataframe `data`?

Briefly explain where that number come from.

In [18]:
```python
print(f"data has a total of {len(data.columns)} columns")
```

```
data has a total of 57 columns
```

In [19]:
```python
#df_not_missing has 11 columns, but dataframe data used one hot encoding on some of the columns of df_not_missing; therefore,
#Ex. column "work" will be represented by 7 columns, each corresponds to a value of "work", in dataframe data.
```

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 [20]:
```python
datanp = data.values.astype(np.float32)
```

In [21]:
```python
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.
    ind = np.argmax(onehot)
    return (cat_values[feature])[ind]
```

In [22]:
```
#Testing get_categorical_value function
print(get_categorical_value(np.array([0., 0., 0., 0., 0., 1., 0.]), "work"))
print(get_categorical_value(np.array([0.1, 0., 1.1, 0.2, 0., 1., 0.]), "work"))
```

```
State-gov
Private
```

In [23]:
```
# 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 [31]:
```
# set the numpy seed for reproducibility
# https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.seed.html

np.random.seed(50)

# todo
import math
from torch.utils.data.sampler import SubsetRandomSampler
print(datanp.shape)

indices = list(range(len(datanp)))
np.random.shuffle(indices)

train_val_cut = math.floor(len(datanp)*0.7) #Between train and validation
val_test_cut = train_val_cut + math.floor(len(datanp)*0.15) #Between validation and test

train_indices = indices[:train_val_cut]
val_indices = indices[train_val_cut:val_test_cut]
test_indices = indices[val_test_cut:]

#Random inside train, validation, and test set
train_sampler = SubsetRandomSampler(train_indices)
val_sampler = SubsetRandomSampler(val_indices)
test_sampler = SubsetRandomSampler(test_indices)
train_loader = torch.utils.data.DataLoader(datanp, batch_size=32, num_workers=1, sampler=train_sampler)
val_loader = torch.utils.data.DataLoader(datanp, batch_size=32, num_workers=1, sampler=val_sampler)
test_loader = torch.utils.data.DataLoader(datanp, batch_size=32, num_workers=1, sampler=test_sampler)


def inspect_loader(loader):
    total_samples = 0

    for item in loader:
        batch_size = item.size(0)
        total_samples += batch_size
    return total_samples


print(f"Number of items in train: {inspect_loader(train_loader)}")
print(f"Number of items in validation: {inspect_loader(val_loader)}")
print(f"Number of items in test: {inspect_loader(test_loader)}")
```

```
(30718, 57)
Number of items in train: 21502
Number of items in validation: 4607
Number of items in test: 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 [25]:
```
from torch import nn

class AutoEncoder(nn.Module):
    def __init__(self):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(57, 24), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(24,12),
            nn.ReLU()
        )
        self.decoder = nn.Sequential(
            nn.Linear(12, 24), # TODO -- FILL OUT THE CODE HERE!
            nn.ReLU(),
            nn.Linear(24,57),
            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.)

In [26]:
```
#Because the data has been normalized to [0,1], the one-hot encoded columns can also take only value 0 and 1
```

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 [33]:
```
import matplotlib.pyplot as plt
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))

#Helper function
def plot(train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record, start = None, end = None):
    if start == None and end == None:
      keys = list(train_loss_record.keys())
      train_loss = list(train_loss_record.values())
      val_loss = list(val_loss_record.values())

      train_acc = list(train_accuracy_record.values())
      val_acc = list(val_accuracy_record.values())
    else:
      keys = list(train_loss_record.keys())[start:end]
      train_loss = list(train_loss_record.values())[start:end]
      val_loss = list(val_loss_record.values())[start:end]

      train_acc = list(train_accuracy_record.values())[start:end]
      val_acc = list(val_accuracy_record.values())[start:end]

    plt.figure(figsize=(10, 6))
    plt.plot(keys,train_loss, label='Training Loss')
    plt.plot(keys,val_loss, label='Validation Loss', linestyle='--')
    plt.xlabel('Iteration (x5000)')
    plt.ylabel('Loss')
    plt.title('Training and Validation Loss plot')
    plt.legend()
    plt.grid(True)
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(keys,train_acc, label='Training Accuracy')
    plt.plot(keys,val_acc, label='Validation Accuracy', linestyle='--')
    plt.xlabel('Iteration (x5000)')
    plt.ylabel('Accuracy')
    plt.title('Training and Validation Accuracy plot')
    plt.legend()
    plt.grid(True)
    plt.show()
def get_model_name(learning_rate, epoch, batch):
    """ Generate a name for the model consisting of all the hyperparameter values
    Args:
        config: Configuration object containing the hyperparameters
    Returns:
        path: A string with the hyperparameter name and value concatenated
    """
    path = "model_lr{0}_epoch{1}_batch{2}".format(learning_rate,
                                                   epoch,
                                                   batch)
    return path

def evaluate(net,loader,criterion):
    total_loss = 0
    total_num = 0

    for x in loader:
      datam = zero_out_random_feature(x.clone()) # zero out one categorical feature
      recon = net(datam)
      loss = criterion(recon,x)
      total_loss += loss.item()
      total_num += 1 #Average across every batch

    return total_loss/total_num

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

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

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

        >>> model = AutoEncoder()
        >>> vdl = torch.utils.data.DataLoader(data_valid, batch_size=256, shuffle=True)
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            inp = item.detach().numpy()
            out = model(zero_out_feature(item.clone(), col)).detach().numpy()
            for i in range(out.shape[0]): # record in minibatch
                acc += int(get_feature(out[i], col) == get_feature(inp[i], col))
                total += 1
    return acc / total

def train(model, train_loader, valid_loader, num_epochs=5, learning_rate=1e-4):
    """ Training loop. You should update this."""
    torch.manual_seed(42)
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_loss_record = dict()
    val_loss_record = dict()

    train_accuracy_record = dict()
    val_accuracy_record = dict()

    for epoch in range(num_epochs):
        train_loss = 0
        for i, data in enumerate(train_loader):

            datam = zero_out_random_feature(data.clone()) # zero out one categorical feature
            recon = model(datam)
            loss = criterion(recon, data)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            train_loss += loss.item()
            if (i+1) % 100 == 0:

                avg_train_loss = train_loss / 100
                train_loss = 0

                train_loss_record[f"({epoch},{i+1})"] = avg_train_loss
                val_loss = evaluate(model, valid_loader, nn.MSELoss())
                val_loss_record[f"({epoch},{i+1})"] = val_loss

                train_acc = get_accuracy(model, train_loader)
                val_acc = get_accuracy(model, valid_loader)
                train_accuracy_record[f"({epoch},{i+1})"] = train_acc
                val_accuracy_record[f"({epoch},{i+1})"] = val_acc

                print(f"At epoch:{epoch}, iteration:{i+1}; Train Loss = {avg_train_loss} Validation Loss = {val_loss}, Train A

                model_path = get_model_name(learning_rate, epoch, i+1)
                torch.save(model.state_dict(), model_path)
    return train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record
```

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 [28]:
```
def get_accuracy(model, data_loader):
    """Return the "accuracy" of the autoencoder model across a data set.
    That is, for each record and for each categorical feature,
    we determine whether the model can successfully predict the value
    of the categorical feature given all the other features of the
    record. The returned "accuracy" measure is the percentage of times
    that our model is successful.

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

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

```
        >>> model = AutoEncoder()
        >>> vdl = torch.utils.data.DataLoader(data_valid, batch_size=256, shuffle=True)
        >>> get_accuracy(model, vdl)
    """
    total = 0
    acc = 0
    for col in catcols:
        for item in data_loader: # minibatches
            inp = item.detach().numpy()
            out = model(zero_out_feature(item.clone(), col)).detach().numpy()
            for i in range(out.shape[0]): # record in minibatch
                acc += int(get_feature(out[i], col) == get_feature(inp[i], col))
                total += 1
    return acc / total
```

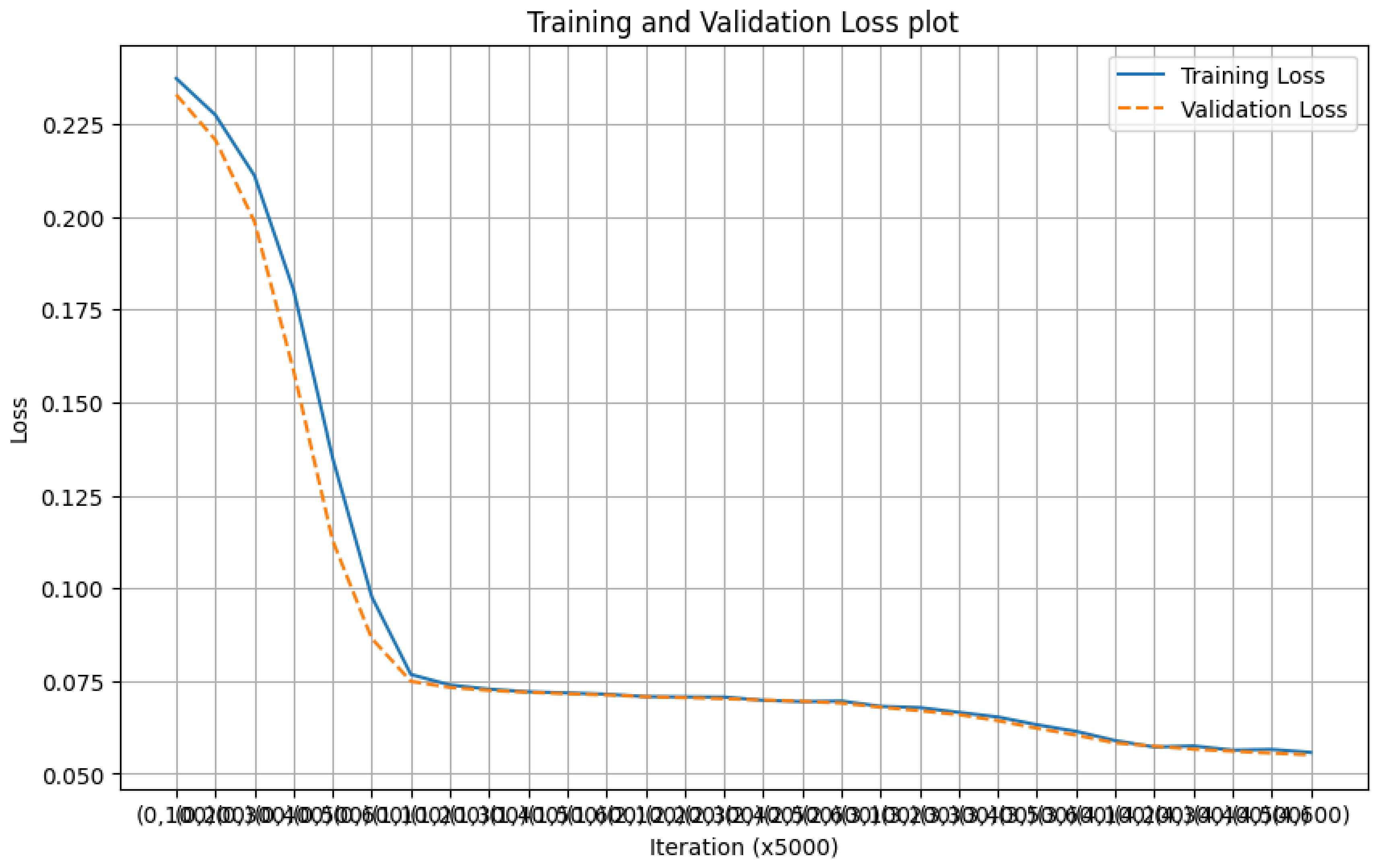
Part (c) [4 pt]

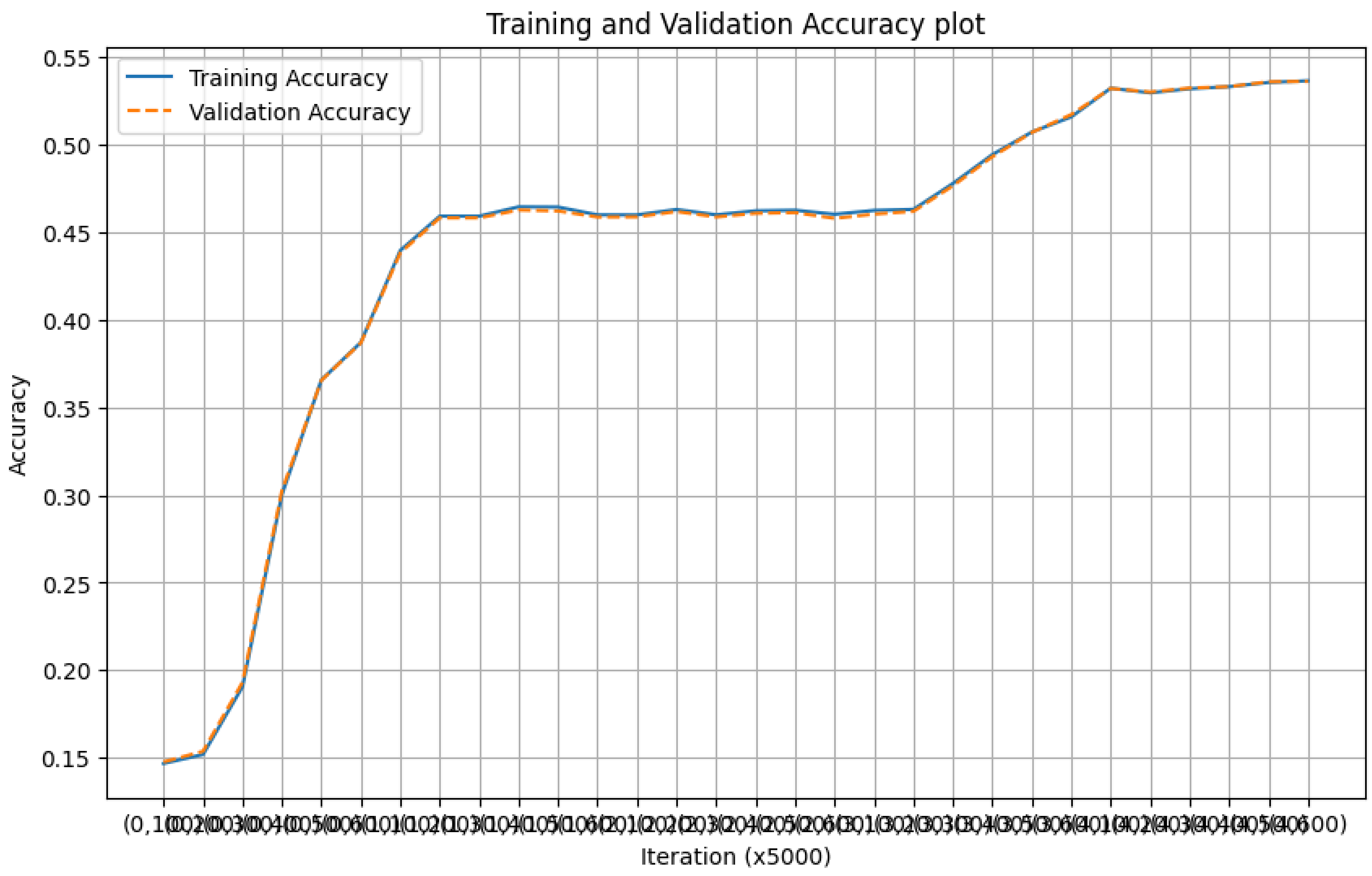
Run your updated training code, using reasonable initial hyperparameters.

Include your training curve in your submission.

In [34]:
```
ae = AutoEncoder()
train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record = train(ae, train_loader, val_loader)
plot(train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record)
```

```
At epoch:0, iteration:100; Train Loss = 0.23733536437153815 Validation Loss = 0.23294911792294848, Train Accuracy = 0.146769292
7789663 Validation Accuracy = 0.14763765284711672
At epoch:0, iteration:200; Train Loss = 0.22749810799956321 Validation Loss = 0.22079457694457638, Train Accuracy = 0.151885095
96006572 Validation Accuracy = 0.15371536068301858
At epoch:0, iteration:300; Train Loss = 0.21113947823643683 Validation Loss = 0.19870050779233375, Train Accuracy = 0.190563668
49595385 Validation Accuracy = 0.19307575428695464
At epoch:0, iteration:400; Train Loss = 0.18030465096235276 Validation Loss = 0.15858299440393844, Train Accuracy = 0.300220134
5611261 Validation Accuracy = 0.30254684899790174
At epoch:0, iteration:500; Train Loss = 0.13508102625608445 Validation Loss = 0.11284579186596805, Train Accuracy = 0.365531888
50649554 Validation Accuracy = 0.36531365313653136
At epoch:0, iteration:600; Train Loss = 0.09770404078066348 Validation Loss = 0.08644950384688047, Train Accuracy = 0.386956252
13158463 Validation Accuracy = 0.38662180739454455
At epoch:1, iteration:100; Train Loss = 0.07675911337137223 Validation Loss = 0.07490499912657672, Train Accuracy = 0.439470746
90726445 Validation Accuracy = 0.43813761667028434
At epoch:1, iteration:200; Train Loss = 0.07390911154448986 Validation Loss = 0.07325427332479093, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:300; Train Loss = 0.07279722720384597 Validation Loss = 0.07246092329215673, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:400; Train Loss = 0.07208708323538303 Validation Loss = 0.07194768611548676, Train Accuracy = 0.464398660
5897126 Validation Accuracy = 0.46259315534331813
At epoch:1, iteration:500; Train Loss = 0.07178405635058879 Validation Loss = 0.07155470922589302, Train Accuracy = 0.464282392
3355967 Validation Accuracy = 0.4620505028579698
At epoch:1, iteration:600; Train Loss = 0.07144362345337868 Validation Loss = 0.07124192878190014, Train Accuracy = 0.459825442
5944873 Validation Accuracy = 0.4585775269517401
At epoch:2, iteration:100; Train Loss = 0.07078143149614334 Validation Loss = 0.07076425596864687, Train Accuracy = 0.459809940
1606052 Validation Accuracy = 0.45864988061645323
At epoch:2, iteration:200; Train Loss = 0.07070910669863224 Validation Loss = 0.07049187065826522, Train Accuracy = 0.462879422
06926486 Validation Accuracy = 0.46172491136676075
At epoch:2, iteration:300; Train Loss = 0.07066054727882147 Validation Loss = 0.07022141137470801, Train Accuracy = 0.459848696
2453105 Validation Accuracy = 0.45861370378409666
At epoch:2, iteration:400; Train Loss = 0.0697942055761814 Validation Loss = 0.06986607492177023, Train Accuracy = 0.4621430564
598642 Validation Accuracy = 0.4607119600607771
At epoch:2, iteration:500; Train Loss = 0.06944153726100921 Validation Loss = 0.0695366157580995, Train Accuracy = 0.4624763587
8832977 Validation Accuracy = 0.4611099052166992
At epoch:2, iteration:600; Train Loss = 0.06958987269550562 Validation Loss = 0.06903403291168313, Train Accuracy = 0.460166496
13989397 Validation Accuracy = 0.45781781347225237
At epoch:3, iteration:100; Train Loss = 0.06815435070544482 Validation Loss = 0.06792226560517317, Train Accuracy = 0.462352339
3172728 Validation Accuracy = 0.4602416612401418
At epoch:3, iteration:200; Train Loss = 0.06781875167042017 Validation Loss = 0.06698986626644102, Train Accuracy = 0.462949183
02173443 Validation Accuracy = 0.4618334418638304
At epoch:3, iteration:300; Train Loss = 0.06653573703020811 Validation Loss = 0.06592152378935781, Train Accuracy = 0.477738504
9452764 Validation Accuracy = 0.47637652847116707
At epoch:3, iteration:400; Train Loss = 0.06529039841145277 Validation Loss = 0.06429580402457052, Train Accuracy = 0.494225343
3789105 Validation Accuracy = 0.49301787135518416
At epoch:3, iteration:500; Train Loss = 0.06319175116717815 Validation Loss = 0.06225362898678415, Train Accuracy = 0.507115617
1518928 Validation Accuracy = 0.5068735981477461
At epoch:3, iteration:600; Train Loss = 0.061431400999426844 Validation Loss = 0.06041567588949369, Train Accuracy = 0.51575047
28242334 Validation Accuracy = 0.5171116417046523
At epoch:4, iteration:100; Train Loss = 0.05892917886376381 Validation Loss = 0.058284921364651784, Train Accuracy = 0.53214429
66545747 Validation Accuracy = 0.5320164966355546
At epoch:4, iteration:200; Train Loss = 0.05720084849745035 Validation Loss = 0.05741574472954704, Train Accuracy = 0.529516634
1115555 Validation Accuracy = 0.5299544171912307
At epoch:4, iteration:300; Train Loss = 0.05748521581292152 Validation Loss = 0.05660972006929418, Train Accuracy = 0.531826496
7599913 Validation Accuracy = 0.5322335576296939
At epoch:4, iteration:400; Train Loss = 0.056361778452992436 Validation Loss = 0.056059870325649776, Train Accuracy = 0.5330201
841689145 Validation Accuracy = 0.5330294479415383
At epoch:4, iteration:500; Train Loss = 0.056542102210223676 Validation Loss = 0.05556223024096754, Train Accuracy = 0.53543081
26375841 Validation Accuracy = 0.5358150640329933
At epoch:4, iteration:600; Train Loss = 0.05574714545160532 Validation Loss = 0.05507760220724675, Train Accuracy = 0.536391963
5382756 Validation Accuracy = 0.5360683018594892
```





Part (d) [5 pt]

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

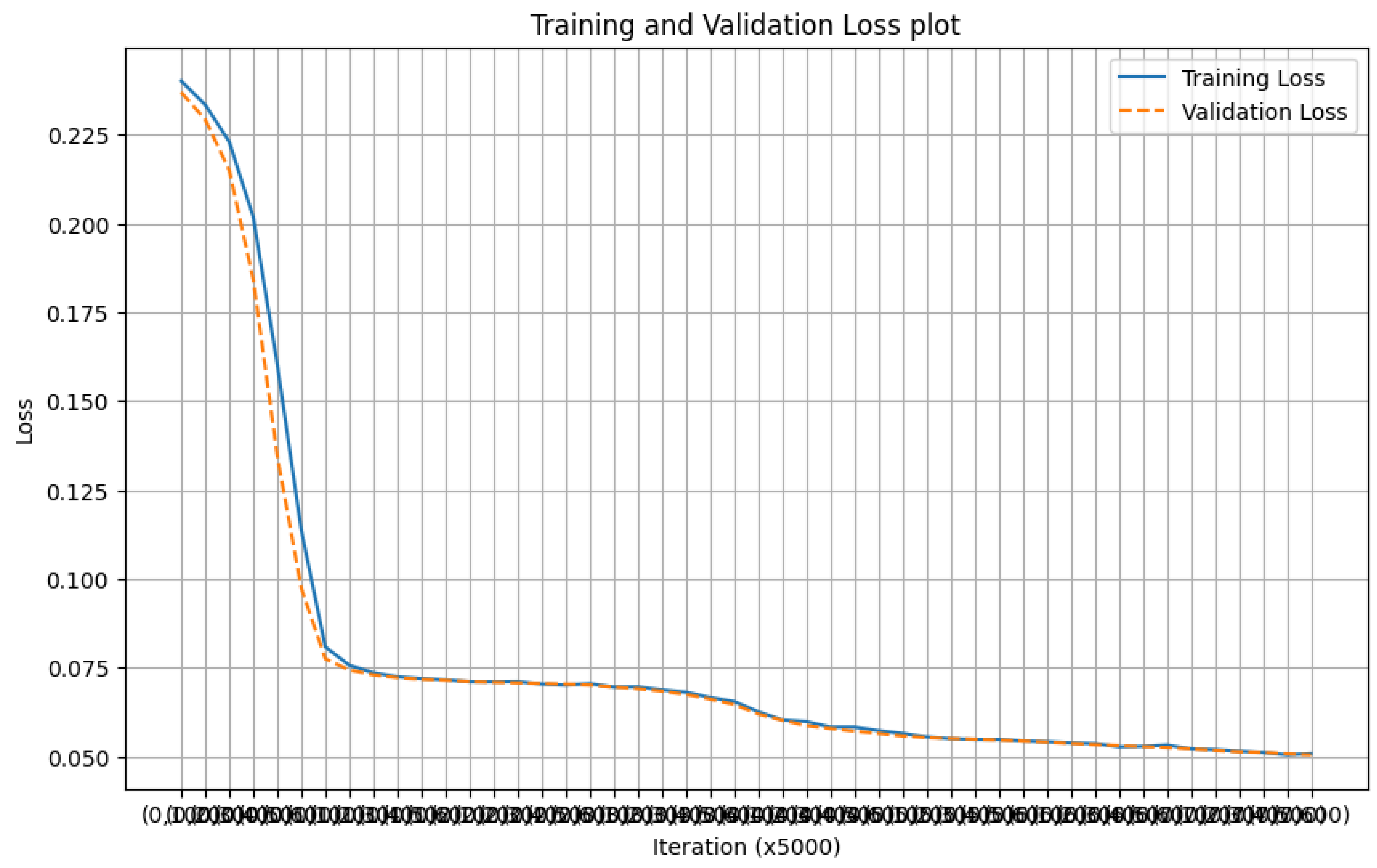
Do not include all your training curves. Instead, explain what hyperparameters you tried, what their effect was, and what your thought process was as you chose the next set of hyperparameters to try.

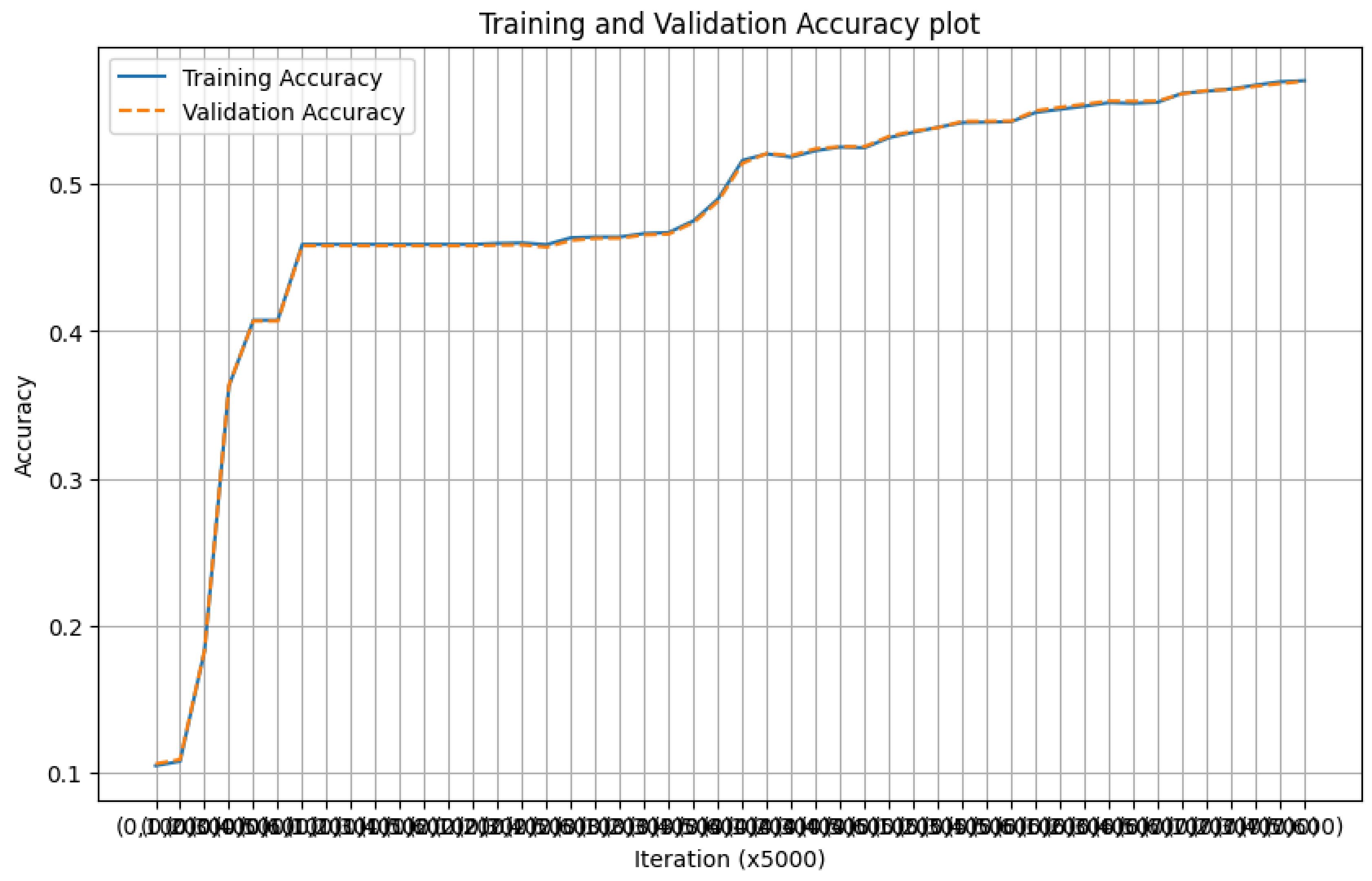
In [36]:
```
def get_best_model(val_accuracy_record):
      keys = list(val_accuracy_record.keys())
      acc = list(val_accuracy_record.values())
      ind = np.argmax(acc)

      return (keys[ind], acc[ind])
```

In [35]:
```
#increase num_epoch because the model hasn't passed the point of overfitting
ae = AutoEncoder()
train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record = train(ae, train_loader, val_loader,num_epochs
At epoch:0, iteration:100; Train Loss = 0.24014712125062943 Validation Loss = 0.23694984449280632, Train Accuracy = 0.104602672
61960128 Validation Accuracy = 0.10585341147529122
At epoch:0, iteration:200; Train Loss = 0.23346079111099244 Validation Loss = 0.22929812036454678, Train Accuracy = 0.107509378
97249868 Validation Accuracy = 0.10878373489617249
At epoch:0, iteration:300; Train Loss = 0.22303060457110405 Validation Loss = 0.21499332330293125, Train Accuracy = 0.182688432
08383717 Validation Accuracy = 0.18287388756240502
At epoch:0, iteration:400; Train Loss = 0.20184045568108558 Validation Loss = 0.18418862236042818, Train Accuracy = 0.362663938
2383034 Validation Accuracy = 0.3632877505245641
At epoch:0, iteration:500; Train Loss = 0.1602937351167202 Validation Loss = 0.13459523332615694, Train Accuracy = 0.4074814745
9151085 Validation Accuracy = 0.4068808335142175
At epoch:0, iteration:600; Train Loss = 0.11376267001032829 Validation Loss = 0.09735630887250106, Train Accuracy = 0.407481474
59151085 Validation Accuracy = 0.4068808335142175
At epoch:1, iteration:100; Train Loss = 0.0807813809812069 Validation Loss = 0.07743294883726372, Train Accuracy = 0.4590735745
5120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:200; Train Loss = 0.07562963128089904 Validation Loss = 0.07428466823572914, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:300; Train Loss = 0.07348445512354373 Validation Loss = 0.07291829130715793, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:400; Train Loss = 0.0723870337754488 Validation Loss = 0.07212191007824408, Train Accuracy = 0.4590735745
5120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:500; Train Loss = 0.07187672004103661 Validation Loss = 0.07166840958719452, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:1, iteration:600; Train Loss = 0.0715203557163477 Validation Loss = 0.07138843022079931, Train Accuracy = 0.4590735745
5120457 Validation Accuracy = 0.45807105129874826
At epoch:2, iteration:100; Train Loss = 0.07097952485084534 Validation Loss = 0.0710142517151932, Train Accuracy = 0.4590735745
5120457 Validation Accuracy = 0.45807105129874826
At epoch:2, iteration:200; Train Loss = 0.07097385443747044 Validation Loss = 0.07083111675456166, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:2, iteration:300; Train Loss = 0.07102006759494543 Validation Loss = 0.07064361347713405, Train Accuracy = 0.459716925
5573125 Validation Accuracy = 0.45843281962231386
At epoch:2, iteration:400; Train Loss = 0.07032505687326193 Validation Loss = 0.0704703224926359, Train Accuracy = 0.4600502278
8577804 Validation Accuracy = 0.4586860574488098
At epoch:2, iteration:500; Train Loss = 0.0700418633967638 Validation Loss = 0.07029158725506729, Train Accuracy = 0.4588797941
2767804 Validation Accuracy = 0.4571666304898343
At epoch:2, iteration:600; Train Loss = 0.07042980458587408 Validation Loss = 0.0700563711579889, Train Accuracy = 0.4635692803
7701917 Validation Accuracy = 0.46172491136676075
At epoch:3, iteration:100; Train Loss = 0.06949869871139526 Validation Loss = 0.06947782276094788, Train Accuracy = 0.464057607
04430593 Validation Accuracy = 0.46309963099630996
At epoch:3, iteration:200; Train Loss = 0.06954897187650204 Validation Loss = 0.06901130323401755, Train Accuracy = 0.464119616
77983443 Validation Accuracy = 0.4632081614933796
At epoch:3, iteration:300; Train Loss = 0.06869324564933776 Validation Loss = 0.06833377669358419, Train Accuracy = 0.466452733
0790934 Validation Accuracy = 0.46548730193184285
At epoch:3, iteration:400; Train Loss = 0.068035026229918 Validation Loss = 0.06741163106117812, Train Accuracy = 0.46708833286
82603 Validation Accuracy = 0.46602995441719125
At epoch:3, iteration:500; Train Loss = 0.06653843380510807 Validation Loss = 0.06607922821098731, Train Accuracy = 0.475072086
3175519 Validation Accuracy = 0.4738079733738514
At epoch:3, iteration:600; Train Loss = 0.06541667543351651 Validation Loss = 0.06457144355711837, Train Accuracy = 0.489977676
49520974 Validation Accuracy = 0.4883148831488315
At epoch:4, iteration:100; Train Loss = 0.06252483185380697 Validation Loss = 0.06188671668577525, Train Accuracy = 0.516231048
2745791 Validation Accuracy = 0.5142174951161276
At epoch:4, iteration:200; Train Loss = 0.06023130279034376 Validation Loss = 0.06012553312919206, Train Accuracy = 0.520618237
063219 Validation Accuracy = 0.5209825627668041
At epoch:4, iteration:300; Train Loss = 0.05974979981780052 Validation Loss = 0.05862667442609867, Train Accuracy = 0.518447896
3197222 Validation Accuracy = 0.5194631358078287
At epoch:4, iteration:400; Train Loss = 0.05822182137519121 Validation Loss = 0.05779919713839061, Train Accuracy = 0.522842836
325303 Validation Accuracy = 0.5242023008465378
At epoch:4, iteration:500; Train Loss = 0.05821312054991722 Validation Loss = 0.05704236408281657, Train Accuracy = 0.525299972
095619 Validation Accuracy = 0.525504666811374
At epoch:4, iteration:600; Train Loss = 0.05720999337732792 Validation Loss = 0.056402188467068806, Train Accuracy = 0.52477288
9343627 Validation Accuracy = 0.5256131973084437
At epoch:5, iteration:100; Train Loss = 0.056416237577795986 Validation Loss = 0.05569153962035974, Train Accuracy = 0.53167147
24211701 Validation Accuracy = 0.5325953259532595
At epoch:5, iteration:200; Train Loss = 0.05542332421988249 Validation Loss = 0.05523356021795836, Train Accuracy = 0.535337798
0342914 Validation Accuracy = 0.5361044786918457
At epoch:5, iteration:300; Train Loss = 0.05493936374783516 Validation Loss = 0.05498268685510589, Train Accuracy = 0.538732831
0544756 Validation Accuracy = 0.5384559727950221
At epoch:5, iteration:400; Train Loss = 0.0547284172475338 Validation Loss = 0.05478682671673596, Train Accuracy = 0.5419030787
83369 Validation Accuracy = 0.5427610158454526
At epoch:5, iteration:500; Train Loss = 0.054735633358359334 Validation Loss = 0.05456699914712873, Train Accuracy = 0.54221312
74610114 Validation Accuracy = 0.5428333695101657
At epoch:5, iteration:600; Train Loss = 0.054287602454423906 Validation Loss = 0.054264199687168, Train Accuracy = 0.5425386785
725359 Validation Accuracy = 0.5429419000072354
At epoch:6, iteration:100; Train Loss = 0.05406363807618618 Validation Loss = 0.05396057820568482, Train Accuracy = 0.548824915
5117353 Validation Accuracy = 0.5500325591491209
At epoch:6, iteration:200; Train Loss = 0.05375650804489851 Validation Loss = 0.05362179746023483, Train Accuracy = 0.550995256
255232 Validation Accuracy = 0.5523478764199407
At epoch:6, iteration:300; Train Loss = 0.05357294175773859 Validation Loss = 0.053240006479124226, Train Accuracy = 0.55308808
48293181 Validation Accuracy = 0.5544461326966211
At epoch:6, iteration:400; Train Loss = 0.05263401679694653 Validation Loss = 0.052967069670557976, Train Accuracy = 0.55551421
57318699 Validation Accuracy = 0.5566167426380146
At epoch:6, iteration:500; Train Loss = 0.05276991330087185 Validation Loss = 0.05269400237335099, Train Accuracy = 0.555018137
8476421 Validation Accuracy = 0.5564720353085884
At epoch:6, iteration:600; Train Loss = 0.053126024827361104 Validation Loss = 0.052456079257859126, Train Accuracy = 0.5557545
034570428 Validation Accuracy = 0.5566529194703712
At epoch:7, iteration:100; Train Loss = 0.05207889650017023 Validation Loss = 0.052053035961257085, Train Accuracy = 0.56190896
97082442 Validation Accuracy = 0.5615367918385066
At epoch:7, iteration:200; Train Loss = 0.051830868683755395 Validation Loss = 0.05164685920398268, Train Accuracy = 0.56328868
```

```
63237528 Validation Accuracy = 0.5637435786122567
At epoch:7, iteration:300; Train Loss = 0.051417570523917676 Validation Loss = 0.05122507019485864, Train Accuracy = 0.56483117
84950237 Validation Accuracy = 0.5643947615946747
At epoch:7, iteration:400; Train Loss = 0.051060112826526165 Validation Loss = 0.051059398929485016, Train Accuracy = 0.5674355
873872198 Validation Accuracy = 0.5666015483684249
At epoch:7, iteration:500; Train Loss = 0.05040313206613064 Validation Loss = 0.050686338269669146, Train Accuracy = 0.56969119
15170682 Validation Accuracy = 0.5684103899862528
At epoch:7, iteration:600; Train Loss = 0.05072520982474089 Validation Loss = 0.0503636688614885, Train Accuracy = 0.5704740644
281152 Validation Accuracy = 0.570038347442298
```

In [37]:
```
plot(train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record)
```





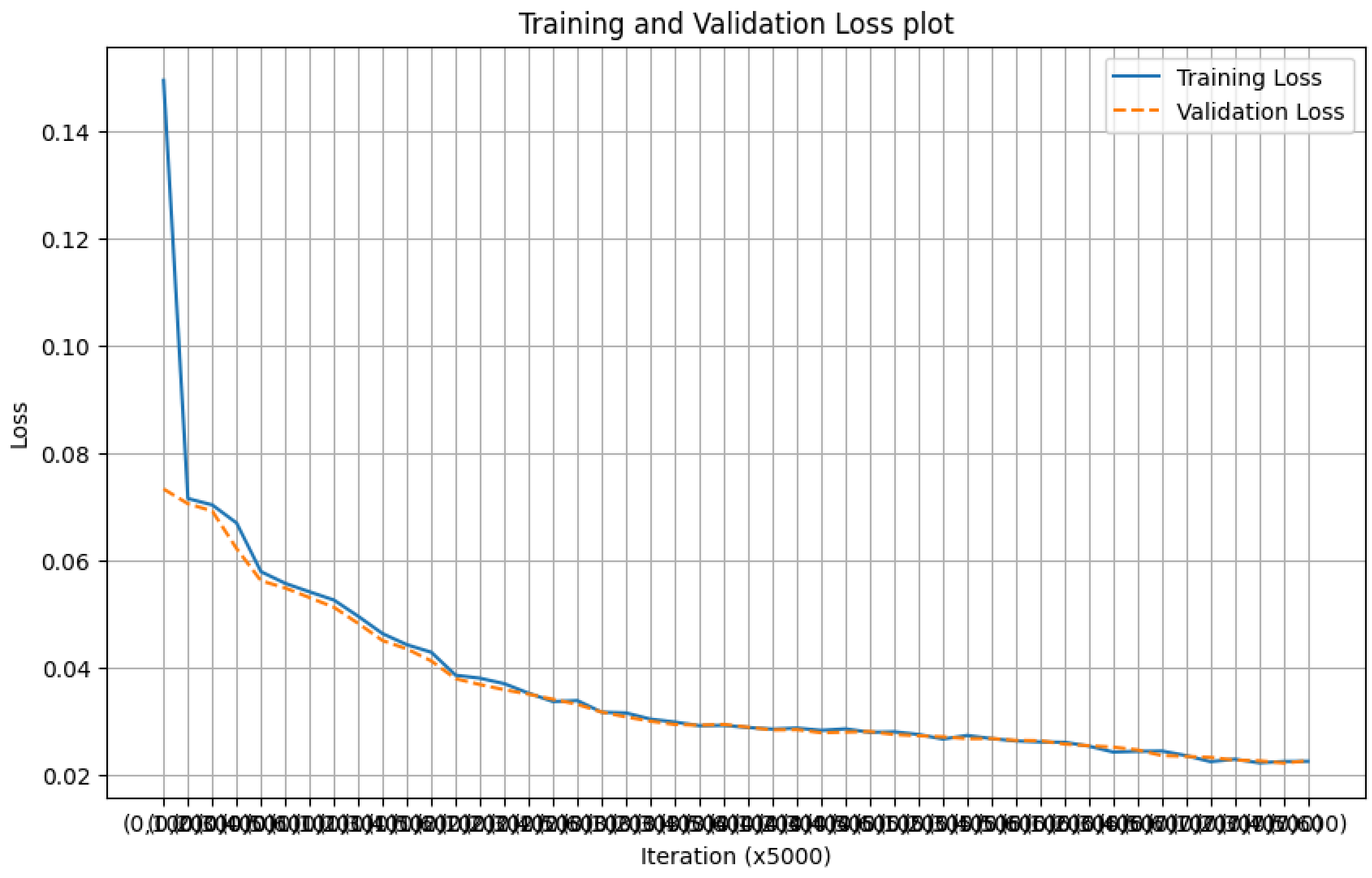
In [40]:
```
get_best_model(val_accuracy_record)
```

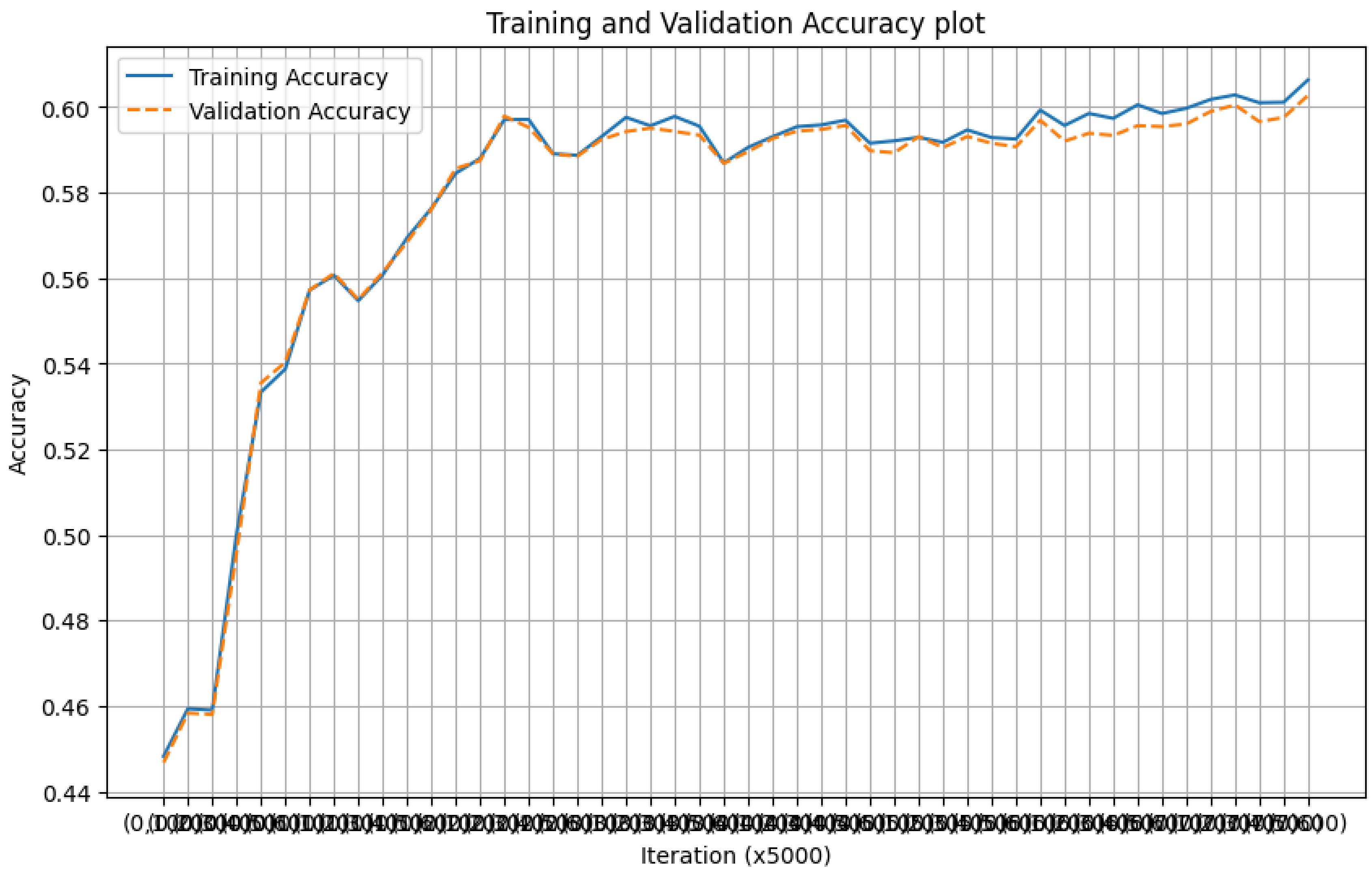
Out[40]:
```
('(7,600)', 0.570038347442298)
```

In [41]:
```
#Increasing num_epochs still has not make the model pass the point of overfitting; so, I will increase the learning rate, hopi
ae = AutoEncoder()
train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record = train(ae, train_loader, val_loader,num_epochs
plot(train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record)
```

```
At epoch:0, iteration:100; Train Loss = 0.14943280898034572 Validation Loss = 0.07331318299596508, Train Accuracy = 0.448260626
9184262 Validation Accuracy = 0.4468200564358585
At epoch:0, iteration:200; Train Loss = 0.07153491578996181 Validation Loss = 0.07057393225841224, Train Accuracy = 0.459406876
87967013 Validation Accuracy = 0.4583966427899573
At epoch:0, iteration:300; Train Loss = 0.07034367278218269 Validation Loss = 0.06920117926266459, Train Accuracy = 0.459073574
55120457 Validation Accuracy = 0.45807105129874826
At epoch:0, iteration:400; Train Loss = 0.06697291161864996 Validation Loss = 0.06220279481365449, Train Accuracy = 0.500263541
375996 Validation Accuracy = 0.49602054844077853
At epoch:0, iteration:500; Train Loss = 0.057903475537896154 Validation Loss = 0.056214917477013335, Train Accuracy = 0.5333302
328465569 Validation Accuracy = 0.5354894725417843
At epoch:0, iteration:600; Train Loss = 0.055720985755324366 Validation Loss = 0.054850738081667155, Train Accuracy = 0.5387328
310544756 Validation Accuracy = 0.5403009912452066
At epoch:1, iteration:100; Train Loss = 0.05413108460605145 Validation Loss = 0.05306348341724111, Train Accuracy = 0.557196229
8080799 Validation Accuracy = 0.5571955719557196
At epoch:1, iteration:200; Train Loss = 0.052610010728240016 Validation Loss = 0.05126184879595207, Train Accuracy = 0.56063777
01299105 Validation Accuracy = 0.5611026698502279
At epoch:1, iteration:300; Train Loss = 0.04957065679132938 Validation Loss = 0.048235021350491375, Train Accuracy = 0.55471584
03869408 Validation Accuracy = 0.5549887851819695
At epoch:1, iteration:400; Train Loss = 0.046344631798565385 Validation Loss = 0.04504934783714513, Train Accuracy = 0.56073853
59501442 Validation Accuracy = 0.5613197308443673
At epoch:1, iteration:500; Train Loss = 0.04427308939397335 Validation Loss = 0.04349004443631404, Train Accuracy = 0.569396645
2733079 Validation Accuracy = 0.5684103899862528
At epoch:1, iteration:600; Train Loss = 0.04287521466612816 Validation Loss = 0.04124744407211741, Train Accuracy = 0.576302979
5677922 Validation Accuracy = 0.5760437016134867
At epoch:2, iteration:100; Train Loss = 0.03858544928953052 Validation Loss = 0.03794121568919056, Train Accuracy = 0.584503767
0914334 Validation Accuracy = 0.5856305621879748
At epoch:2, iteration:200; Train Loss = 0.03803454434499145 Validation Loss = 0.03684427780616614, Train Accuracy = 0.587991814
7149102 Validation Accuracy = 0.5874394038058027
At epoch:2, iteration:300; Train Loss = 0.037014182340353724 Validation Loss = 0.03588235788306014, Train Accuracy = 0.59708399
21867733 Validation Accuracy = 0.5978945083568483
At epoch:2, iteration:400; Train Loss = 0.03523225052282214 Validation Loss = 0.035053939493890435, Train Accuracy = 0.59709949
46206555 Validation Accuracy = 0.5952535995948195
At epoch:2, iteration:500; Train Loss = 0.033684619944542644 Validation Loss = 0.034107612381275326, Train Accuracy = 0.5890847
363035997 Validation Accuracy = 0.5890311844294913
At epoch:2, iteration:600; Train Loss = 0.033859358336776495 Validation Loss = 0.03316505726737281, Train Accuracy = 0.58872042
91073699 Validation Accuracy = 0.588488531944143
At epoch:3, iteration:100; Train Loss = 0.03169946515932679 Validation Loss = 0.031721636381310724, Train Accuracy = 0.59311536
91129507 Validation Accuracy = 0.5924318066710079
At epoch:3, iteration:200; Train Loss = 0.03156202359125018 Validation Loss = 0.030820116775834724, Train Accuracy = 0.59760332
37218243 Validation Accuracy = 0.5942768251211924
At epoch:3, iteration:300; Train Loss = 0.03040456760674715 Validation Loss = 0.030010199976257153, Train Accuracy = 0.59562676
3401854 Validation Accuracy = 0.5950727154330366
At epoch:3, iteration:400; Train Loss = 0.02986354686319828 Validation Loss = 0.029412658445330128, Train Accuracy = 0.59782810
9013115 Validation Accuracy = 0.594313001953549
At epoch:3, iteration:500; Train Loss = 0.02918024057522416 Validation Loss = 0.029271368312442467, Train Accuracy = 0.59551049
51477382 Validation Accuracy = 0.593408581144635
At epoch:3, iteration:600; Train Loss = 0.02924670122563839 Validation Loss = 0.029420481748982437, Train Accuracy = 0.58694540
04278671 Validation Accuracy = 0.5868243976557412
At epoch:4, iteration:100; Train Loss = 0.028840661365538835 Validation Loss = 0.02890202806641658, Train Accuracy = 0.59058847
23901652 Validation Accuracy = 0.5896100137471963
At epoch:4, iteration:200; Train Loss = 0.028548926562070847 Validation Loss = 0.028391192951757047, Train Accuracy = 0.5930611
105943633 Validation Accuracy = 0.5926488676651472
At epoch:4, iteration:300; Train Loss = 0.028736649882048368 Validation Loss = 0.028440467224249408, Train Accuracy = 0.5954174
805444454 Validation Accuracy = 0.5943491787859055
At epoch:4, iteration:400; Train Loss = 0.028314453028142453 Validation Loss = 0.027871735212910507, Train Accuracy = 0.5958282
950423216 Validation Accuracy = 0.5947833007741842
At epoch:4, iteration:500; Train Loss = 0.028590786680579187 Validation Loss = 0.027982074857896402, Train Accuracy = 0.5969367
190648932 Validation Accuracy = 0.5956877215830981
At epoch:4, iteration:600; Train Loss = 0.027953048460185526 Validation Loss = 0.028089084326186113, Train Accuracy = 0.5915496
232908567 Validation Accuracy = 0.5897908979089791
At epoch:5, iteration:100; Train Loss = 0.028040960356593134 Validation Loss = 0.02753251950101306, Train Accuracy = 0.59209995
96936719 Validation Accuracy = 0.5893205990883438
At epoch:5, iteration:200; Train Loss = 0.027549592368304728 Validation Loss = 0.027296224583147302, Train Accuracy = 0.5929293
399063653 Validation Accuracy = 0.593082989653426
At epoch:5, iteration:300; Train Loss = 0.026688277535140515 Validation Loss = 0.027113276119861338, Train Accuracy = 0.5917666
573652063 Validation Accuracy = 0.5905506113884669
At epoch:5, iteration:400; Train Loss = 0.027358719017356634 Validation Loss = 0.026759479557060532, Train Accuracy = 0.5946578
612842216 Validation Accuracy = 0.5931191664857826
At epoch:5, iteration:500; Train Loss = 0.026775853913277387 Validation Loss = 0.02679080651917805, Train Accuracy = 0.59287508
13877779 Validation Accuracy = 0.591527385862094
At epoch:5, iteration:600; Train Loss = 0.026330397333949804 Validation Loss = 0.026447237378710672, Train Accuracy = 0.5925262
766254302 Validation Accuracy = 0.5907314955502496
At epoch:6, iteration:100; Train Loss = 0.026155092902481556 Validation Loss = 0.026381564701700375, Train Accuracy = 0.5992930
890149754 Validation Accuracy = 0.5969177338832212
At epoch:6, iteration:200; Train Loss = 0.026091351341456175 Validation Loss = 0.025757834354105096, Train Accuracy = 0.5957042
755712647 Validation Accuracy = 0.5920338615150857
At epoch:6, iteration:300; Train Loss = 0.025371133480221034 Validation Loss = 0.02544773690816429, Train Accuracy = 0.59850246
48869872 Validation Accuracy = 0.5938788799652702
At epoch:6, iteration:400; Train Loss = 0.024269180605188012 Validation Loss = 0.02518161989024116, Train Accuracy = 0.59736303
59966515 Validation Accuracy = 0.5933724043122784
At epoch:6, iteration:500; Train Loss = 0.024402814442291854 Validation Loss = 0.02466926169452361, Train Accuracy = 0.60053328
37255449 Validation Accuracy = 0.5956515447507417
At epoch:6, iteration:600; Train Loss = 0.024458511918783187 Validation Loss = 0.02361545390320114, Train Accuracy = 0.59849471
36700463 Validation Accuracy = 0.5954344837566022
At epoch:7, iteration:100; Train Loss = 0.023597871605306863 Validation Loss = 0.02345827512908727, Train Accuracy = 0.59971165
47297926 Validation Accuracy = 0.5960494899066637
At epoch:7, iteration:200; Train Loss = 0.022475549792870878 Validation Loss = 0.02326107444241643, Train Accuracy = 0.60178898
```

```
08699965 Validation Accuracy = 0.598979813327545
At epoch:7, iteration:300; Train Loss = 0.022933128820732236 Validation Loss = 0.022768944702369884, Train Accuracy = 0.6028431
463739807 Validation Accuracy = 0.6004992402865205
At epoch:7, iteration:400; Train Loss = 0.022256773859262467 Validation Loss = 0.022638664548544005, Train Accuracy = 0.6010138
591758906 Validation Accuracy = 0.5965921423920122
At epoch:7, iteration:500; Train Loss = 0.022506978642195464 Validation Loss = 0.022219401040476643, Train Accuracy = 0.6011223
762130654 Validation Accuracy = 0.5975689168656393
At epoch:7, iteration:600; Train Loss = 0.022544368831440807 Validation Loss = 0.022639227313144755, Train Accuracy = 0.6063466
964313398 Validation Accuracy = 0.6027422038926271
```





In [42]:
```
#The model performed better
get_best_model(val_accuracy_record)
```

Out[42]:
```
('(7,600)', 0.6027422038926271)
```

In [43]:
```
#It looks like the model is able to learn faster, and the graph does not look too aggressive; so, I will increase the learning
ae = AutoEncoder()
train_loss_record, val_loss_record, train_accuracy_record, val_accuracy_record = train(ae, train_loader, val_loader,num_epochs
get_best_model(val_accuracy_record)
```

```
At epoch:0, iteration:100; Train Loss = 0.09022060357034206 Validation Loss = 0.0644335610171159, Train Accuracy = 0.4799553529
9041947 Validation Accuracy = 0.476955357788872
At epoch:0, iteration:200; Train Loss = 0.05664133884012699 Validation Loss = 0.05422035599541333, Train Accuracy = 0.559676619
229219 Validation Accuracy = 0.559872657550105
At epoch:0, iteration:300; Train Loss = 0.05300431292504072 Validation Loss = 0.05201162762629489, Train Accuracy = 0.556793166
5271447 Validation Accuracy = 0.560198249041314
At epoch:0, iteration:400; Train Loss = 0.04758952517062426 Validation Loss = 0.04256729711778462, Train Accuracy = 0.566513192
5712337 Validation Accuracy = 0.5678315606685479
At epoch:0, iteration:500; Train Loss = 0.03956009702757001 Validation Loss = 0.03888166620809999, Train Accuracy = 0.582814001
7982824 Validation Accuracy = 0.5810361044786918
At epoch:0, iteration:600; Train Loss = 0.037988468669354916 Validation Loss = 0.036927708101251885, Train Accuracy = 0.5929913
496418938 Validation Accuracy = 0.5941682946241227
At epoch:1, iteration:100; Train Loss = 0.036247855070978406 Validation Loss = 0.036767016695294946, Train Accuracy = 0.5809847
146001922 Validation Accuracy = 0.579878445843282
At epoch:1, iteration:200; Train Loss = 0.03632888749241829 Validation Loss = 0.0352873969435071, Train Accuracy = 0.5941540321
830527 Validation Accuracy = 0.5936979958034875
At epoch:1, iteration:300; Train Loss = 0.03493742916733027 Validation Loss = 0.03472659501454069, Train Accuracy = 0.585743961
8020029 Validation Accuracy = 0.5856667390203314
At epoch:1, iteration:400; Train Loss = 0.033759630471467975 Validation Loss = 0.033633726687791445, Train Accuracy = 0.5925572
814931944 Validation Accuracy = 0.5933724043122784
At epoch:1, iteration:500; Train Loss = 0.033179165460169316 Validation Loss = 0.032822671229951084, Train Accuracy = 0.5851393
668806003 Validation Accuracy = 0.584870848708487
At epoch:1, iteration:600; Train Loss = 0.0321611712127924 Validation Loss = 0.031947517030251525, Train Accuracy = 0.591410101
3859175 Validation Accuracy = 0.5891758917589176
At epoch:2, iteration:100; Train Loss = 0.030404735170304776 Validation Loss = 0.03032917132238961, Train Accuracy = 0.59615384
61538461 Validation Accuracy = 0.593734172635844
At epoch:2, iteration:200; Train Loss = 0.030570791102945805 Validation Loss = 0.02959725422422505, Train Accuracy = 0.60090534
21387158 Validation Accuracy = 0.5976774473627089
At epoch:2, iteration:300; Train Loss = 0.030165012739598752 Validation Loss = 0.02945237681787047, Train Accuracy = 0.59320063
24993024 Validation Accuracy = 0.591852977353303
At epoch:2, iteration:400; Train Loss = 0.028833499625325203 Validation Loss = 0.028613620709317427, Train Accuracy = 0.5994946
206554429 Validation Accuracy = 0.5967730265537949
At epoch:2, iteration:500; Train Loss = 0.02826568441465497 Validation Loss = 0.02824429980117, Train Accuracy = 0.593650203081
8839 Validation Accuracy = 0.5919615078503726
At epoch:2, iteration:600; Train Loss = 0.02760209722444415 Validation Loss = 0.02626712524539067, Train Accuracy = 0.593929246
891762 Validation Accuracy = 0.5929744591563563
At epoch:3, iteration:100; Train Loss = 0.025723763536661864 Validation Loss = 0.026018481934443116, Train Accuracy = 0.5811087
340712492 Validation Accuracy = 0.582085232617032
At epoch:3, iteration:200; Train Loss = 0.02563454620540142 Validation Loss = 0.025630374186827492, Train Accuracy = 0.58779803
42913838 Validation Accuracy = 0.5902250198972578
At epoch:3, iteration:300; Train Loss = 0.025138333924114704 Validation Loss = 0.02523954819318735, Train Accuracy = 0.58361237
71432115 Validation Accuracy = 0.5795890311844295
At epoch:3, iteration:400; Train Loss = 0.02448456416837871 Validation Loss = 0.02487075597875648, Train Accuracy = 0.589456794
7167705 Validation Accuracy = 0.5912379712032414
At epoch:3, iteration:500; Train Loss = 0.024089974965900183 Validation Loss = 0.024449303429315075, Train Accuracy = 0.6020602
734629337 Validation Accuracy = 0.6000289414658853
At epoch:3, iteration:600; Train Loss = 0.024683804009109734 Validation Loss = 0.02536858709451432, Train Accuracy = 0.59409202
24475242 Validation Accuracy = 0.5935532884740612
At epoch:4, iteration:100; Train Loss = 0.02366473256610334 Validation Loss = 0.024351796607435163, Train Accuracy = 0.59887452
33001581 Validation Accuracy = 0.5977136241950655
At epoch:4, iteration:200; Train Loss = 0.024103643242269754 Validation Loss = 0.023572837992105633, Train Accuracy = 0.5987660
062629833 Validation Accuracy = 0.5978221546921352
At epoch:4, iteration:300; Train Loss = 0.024416889473795892 Validation Loss = 0.023645090087989554, Train Accuracy = 0.6133692
989799399 Validation Accuracy = 0.6119311193111932
At epoch:4, iteration:400; Train Loss = 0.024056877512484788 Validation Loss = 0.02350292548847695, Train Accuracy = 0.60368802
90205562 Validation Accuracy = 0.6025613197308444
At epoch:4, iteration:500; Train Loss = 0.023704181583598258 Validation Loss = 0.02373946681877391, Train Accuracy = 0.60952469
53771742 Validation Accuracy = 0.6081687287461109
At epoch:4, iteration:600; Train Loss = 0.023288673982024194 Validation Loss = 0.023389044360050723, Train Accuracy = 0.5825582
116392274 Validation Accuracy = 0.5822299399464583
At epoch:5, iteration:100; Train Loss = 0.022788823517039418 Validation Loss = 0.022597564882340118, Train Accuracy = 0.5954329
829783276 Validation Accuracy = 0.5938788799652702
At epoch:5, iteration:200; Train Loss = 0.02252761886455118 7 Validation Loss = 0.02297801058092672, Train Accuracy = 0.60915263
69640033 Validation Accuracy = 0.606034295637074
At epoch:5, iteration:300; Train Loss = 0.02161148976534605 Validation Loss = 0.02253412343432299, Train Accuracy = 0.609516944
1602332 Validation Accuracy = 0.6080240214166848
At epoch:5, iteration:400; Train Loss = 0.02272818804718554 Validation Loss = 0.022416350451142836, Train Accuracy = 0.60756363
74910861 Validation Accuracy = 0.6063960639606396
At epoch:5, iteration:500; Train Loss = 0.02193690505810082 Validation Loss = 0.02222518099523667, Train Accuracy = 0.61385762
56472266 Validation Accuracy = 0.6148614427320743
At epoch:5, iteration:600; Train Loss = 0.02146558390930295 Validation Loss = 0.022035332112055685, Train Accuracy = 0.60138591
75890615 Validation Accuracy = 0.6003545329570943
At epoch:6, iteration:100; Train Loss = 0.021955780498683453 Validation Loss = 0.02109572022325463, Train Accuracy = 0.61838433
63408055 Validation Accuracy = 0.6150785037262138
At epoch:6, iteration:200; Train Loss = 0.021505794981494547 Validation Loss = 0.021776696751152888, Train Accuracy = 0.6139118
84165814 Validation Accuracy = 0.6109905216699226
At epoch:6, iteration:300; Train Loss = 0.021000405587255955 Validation Loss = 0.021986196968807943, Train Accuracy = 0.5885886
584193718 Validation Accuracy = 0.5901164894001881
At epoch:6, iteration:400; Train Loss = 0.02160858261398971 Validation Loss = 0.02210518354938055 2, Train Accuracy = 0.61343905
99324094 Validation Accuracy = 0.6123652412994718
At epoch:6, iteration:500; Train Loss = 0.021409860774874688 Validation Loss = 0.02155077224597335, Train Accuracy = 0.60507549
68530059 Validation Accuracy = 0.6047681065045944
At epoch:6, iteration:600; Train Loss = 0.022152803130447863 Validation Loss = 0.021571068248401087, Train Accuracy = 0.6063777
01299104 Validation Accuracy = 0.604804283336951
At epoch:7, iteration:100; Train Loss = 0.02104872651398182 Validation Loss = 0.020690065155374922, Train Accuracy = 0.61243915
29470127 Validation Accuracy = 0.6127993632877505
At epoch:7, iteration:200; Train Loss = 0.02037515100091696 Validation Loss = 0.02116033336561587, Train Accuracy = 0.609958763
```

```
5258735 Validation Accuracy = 0.6086028507343897
At epoch:7, iteration:300; Train Loss = 0.020831130472943188 Validation Loss = 0.020476786400346707, Train Accuracy = 0.6065172
23204043 Validation Accuracy = 0.6050936979958035
At epoch:7, iteration:400; Train Loss = 0.02154139654710889 Validation Loss = 0.021356579246154677, Train Accuracy = 0.59891327
93848634 Validation Accuracy = 0.5988712828304754
At epoch:7, iteration:500; Train Loss = 0.02095452619716525 Validation Loss = 0.021519053843803704, Train Accuracy = 0.59128608
19148607 Validation Accuracy = 0.5893567759207003
At epoch:7, iteration:600; Train Loss = 0.02050634373910725 Validation Loss = 0.02039665513439104, Train Accuracy = 0.614229684
0603975 Validation Accuracy = 0.6134505462701686
```

Out[43]:
```
('(6,100)', 0.6150785037262138)
```

In [ ]:
```
#Now, we have tried 4 sets of hyperparameters (including the default hyperparameters from 3c). The best model comes from the l
#very high learning rate and num_epochs. It is able to achieve validation accuracy of 0.615.
```

Part 4. Testing [12 pt]

Part (a) [2 pt]

Compute and report the test accuracy.

In [45]:
```
#Load weights from the best model
best_ae = AutoEncoder()
model_path = get_model_name(5*1e-3, 6, 100)
state = torch.load(model_path)
best_ae.load_state_dict(state)
```

```
<ipython-input-45-1fd457372cd4>:4: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default val
ue), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbi
trary code during unpickling (See https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details).
In a future release, the default value for `weights_only` will be flipped to `True`. This limits the functions that could be ex
ecuted during unpickling. Arbitrary objects will no longer be allowed to be loaded via this mode unless they are explicitly all
owlisted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True` for any use
case where you don't have full control of the loaded file. Please open an issue on GitHub for any issues related to this experi
mental feature.
  state = torch.load(model_path)
```

Out[45]:
```
<All keys matched successfully>
```

In [48]:
```
print(f"Best Test Accuracy: {get_accuracy(best_ae, test_loader)}")
```

```
Best Test Accuracy: 0.6160772401822521
```

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 [49]:
```
catcols
```

Out[49]:
```
['work', 'marriage', 'occupation', 'edu', 'relationship', 'sex']
```

In [53]:
```
df_not_missing.reset_index(drop=True, inplace=True)
```

In [54]:
```
train_data = df_not_missing.iloc[train_indices]
test_data = df_not_missing.iloc[test_indices]

baseline = {}

# Get the mode for each column
for column in catcols:
    baseline[column] = train_data[column].mode()[0]
baseline
```

Out[54]:
```
{'work': ' Private',
 'marriage': ' Married-civ-spouse',
 'occupation': ' Prof-specialty',
 'edu': ' HS-grad',
 'relationship': ' Husband',
 'sex': ' Male'}
```

In [60]:
```
err = 0
for index, row in test_data.iterrows():
    missing = random.choice(catcols)
    if row[missing] != baseline[missing]:
        err += 1
err = err/len(test_data)
print(f"Test Accuracy = {(1-err)*100}%")
```

```
Test Accuracy = 46.582772835756124%
```

Part (c) [1 pt]

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

In [ ]:
```
#Test Accuracy from part a is 61.6 %, while test accuracy from baseline is 46.6%; therefore, the neural network is able to per
```

Part (d) [1 pt]

Look at the first item in your test data. Do you think it is reasonable for a human to be able to guess this person's education level based on their other features? Explain.

In [61]:
```
test_data[:1]
#Yes, you can probably guess the education level based on features like age and workhr. For example, 6th grade start at age 11
#Other features will also help with the guessing.
```

Out[61]:

	age	yredu	capgain	caploss	workhr	work	marriage	occupation	edu	relationship	sex
21870	0.0	0.4	0.0	0.0	0.153061	Private	Never-married	Transport-moving	11th	Own-child	Female

Part (e) [2 pt]

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

In [71]:
```
t = torch.tensor(datanp[test_indices[0]])
t = t.reshape(1,57)
t = zero_out_feature(t.clone(), "edu")

out = best_ae(t).detach().numpy()
get_feature(out[0], "edu")
```

Out[71]:
```
'HS-grad'
```

Part (f) [2 pt]

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

In [ ]:
```
#Baseline predicts HS-grad
```