Khansa-Saeed-311468-lab-08

January 15, 2023

[113]:
```
#Q1

#https://stackabuse.com/introduction-to-pytorch-for-classification/
#https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html

#Here importing the necessary modules,
#such as torch, torch.nn, torch.optim, TensorDataset,
#and DataLoader from PyTorch, and sklearn.model_selection.KFold, matplotlib.
 ↪pyplot,
#and SummaryWriter from TensorBoard.

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import KFold
import matplotlib.pyplot as plt
from torch.utils.tensorboard import SummaryWriter


# Then, it initializes a TensorBoard writer object, which will be used later to␣
 ↪log the training progress.


writer = SummaryWriter()

#disable extra warnings
pd.options.mode.chained_assignment = None # default='warn'

# Next, it defines the Model class, which inherits from the PyTorch nn.Module␣
 ↪class.
# The constructor of the Model class takes in several parameters
# such as embedding_size, num_numerical_cols, output_size, layers, and p␣
 ↪(dropout rate).

class Model(nn.Module):
```

```
#     The constructor of the Model class starts by initializing the parent class (nn.Module),
#     and then creates a list of embedding layers using the nn.Embedding module from PyTorch.
#     These embedding layers will be used to handle the categorical input variables.
#     The constructor also creates an embedding dropout layer using nn.Dropout and
#     a batch normalization layer for the numerical input variables using nn.BatchNorm1d.

    def __init__(self, embedding_size, num_numerical_cols, output_size, layers, p=0.4):
        super().__init__()
        self.all_embeddings = nn.ModuleList([nn.Embedding(ni, nf) for ni, nf in embedding_size])
        self.embedding_dropout = nn.Dropout(p)
        self.batch_norm_num = nn.BatchNorm1d(num_numerical_cols)

        all_layers = []
        num_categorical_cols = sum((nf for ni, nf in embedding_size))
        input_size = num_categorical_cols + num_numerical_cols

# The script then creates a list of fully connected (Linear) layers,
# with ReLU activation functions and batch normalization layers, followed by dropout layers.
# These layers will be used to create the multi-layer perceptron.
# The script then concatenates all the created layers using nn.Sequential.

        for i in layers:
            all_layers.append(nn.Linear(input_size, i))
            all_layers.append(nn.ReLU(inplace=True))
            all_layers.append(nn.BatchNorm1d(i))
            all_layers.append(nn.Dropout(p))
            input_size = i

        all_layers.append(nn.Linear(layers[-1], output_size))

        self.layers = nn.Sequential(*all_layers)

# Finally, the forward method of the Model class is defined, which takes in two inputs x_categorical,
# and x_numerical, and returns the output of the model. The forward method starts by passing
# the categorical input variables through the embedding layers and concatenating the results.
```

```
# The embeddings are then passed through an embedding dropout layer, and the␣
 ↪numerical input variables
# are passed through the batch normalization layer. The categorical and␣
 ↪numerical inputs are then concatenated
# and passed through the multi-layer perceptron.

    def forward(self, x_categorical, x_numerical):
        embeddings = []
        #print('\nforward, result: \n')
        for i,e in enumerate(self.all_embeddings):
            #print('i,e', i,e)
            embeddings.append(e(x_categorical[:,i]))
        x = torch.cat(embeddings, 1)
        x = self.embedding_dropout(x)

        x_numerical = self.batch_norm_num(x_numerical)
        x = torch.cat([x, x_numerical], 1)
        x = self.layers(x)
        return x


# Preprocessing

# The script then loads the data from a CSV file using pandas, preprocesses it␣
 ↪by one-hot encoding
# the categorical variables, normalizing the numerical variables, and mapping␣
 ↪the target variable to binary values (0,1).
# The preprocessed data is then used to train the model using K-fold␣
 ↪cross-validation
# and the training progress is logged using TensorBoard.

#Only will be using 1000 sample size
data = pd.read_csv('bank-additional-full.csv', sep=';')
df = data.iloc[:1000, :]
print('\n Original Data frame (with first 1000 samples) \n ')
display(df)

categorical_cols = df.select_dtypes(include='object').columns


#excluding y col
categorical_cols = categorical_cols[:-1]
print('\n Selected Categorical Cols\n', categorical_cols)

numerical_cols = df.select_dtypes(exclude='object').columns
scaler = StandardScaler()
df[numerical_cols] = scaler.fit_transform(df[numerical_cols])
```

```
encoder = OneHotEncoder()

df_encoded = pd.DataFrame(encoder.fit_transform(df[categorical_cols]).toarray())

#select only 15 caategorical data
df_encoded = df_encoded.iloc[:, :15]


df = pd.concat([df_encoded, df.drop(categorical_cols, axis=1)], axis=1)

print('\nTotal unique y outputs(before mapping)\n', df['y'].value_counts(),␣
 ↪df['y'])
#mapping yes and no into 0 and 1
mapping = {'yes':1, 'no':0}
df['y'] = df['y'].map(mapping)
print('\nTotal unique y outputs (after mapping)\n', df['y'].
 ↪value_counts(),df['y'] )

print('\n Final Scaled Data frame \n ')
display(df)
print('\nData type:', df.dtypes)

input_size = df_encoded.shape[1]

print('\n\nDivided Train/Test Data\n')
print('df_train, df_test', df_train.shape, df_test.shape)
print('X_train_categorical', X_train_categorical.shape)
print('X_train_numerical',X_train_numerical.shape)
print('y_train',y_train.shape)
print('X_test_categorical', X_test_categorical.shape)
print('X_test_numerical',X_test_numerical.shape )
print('y_test',y_test.shape)
print('\n\n')

epochs = 300

# following defines a function called "train_model" which takes in the number␣
 ↪of training epochs as input.
# The function then uses a for loop to iterate over the number of training␣
 ↪epochs. In each iteration,
# the function makes a forward pass through the model by passing in the input␣
 ↪variables (X_train_categorical, X_train_numerical)
# and receives the output of the model (y_pred). The function then calculates␣
 ↪the loss by comparing the output of
```

```
# the model (y_pred) to the actual target variable (y_train) using the loss␣
 ↪function (nn.CrossEntropyLoss()).
# The loss value is stored in the "single_loss" variable and appended to the␣
 ↪"aggregated_losses" list.

def train_model(epochs):
    for i in range(epochs):
        i += 1
        y_pred = model(X_train_categorical, X_train_numerical)
        single_loss = loss_function(y_pred, y_train)
        aggregated_losses.append(single_loss)

# The function also checks if the current iteration is a multiple of 25, and
# if so, it prints out the current epoch and the loss value. It also uses the␣
 ↪TensorBoard writer object
# to log the loss value for each iteration.
# Then the function uses the optimizer (torch.optim.Adam) to perform␣
 ↪backpropagation and update the model's parameters.

        if i%25 == 1:
            print(f'epoch: {i:3} loss: {single_loss.item():10.8f}')

        writer.add_scalar("Loss/train for 300 Epochs", single_loss, i)
        optimizer.zero_grad()
        single_loss.backward()
        optimizer.step()

# After that, the script uses the KFold class from sklearn.model_selection to␣
 ↪split the data into
# training and test sets using 3-fold cross-validation. It then iterates over␣
 ↪the splits and for each iteration,
# it initializes the model, loss function and optimizer and call the previously␣
 ↪defined train_model function for 300 epochs.
# Finally, it plots the aggregated losses using matplotlib.pyplot.

# #Evaluate the model using K-fold cross validation
kf = KFold(n_splits=3)
KFold_counter = 1

# The training set is created by selecting the rows specified by the␣
 ↪train_index for the
# input variables (X_train_categorical, X_train_numerical) and the target␣
 ↪variable (y_train).
# The test set is created by selecting the rows specified by the test_index for␣
 ↪the
```

```
# input variables (X_test_categorical, X_test_numerical) and the target␣
 ↪variable (y_test). This way, the script is able
# to use different subsets of the data for training and testing the model in␣
 ↪each iteration of the loop.

for train_index, test_index in kf.split(df):
    print('\n\nK-fold cross validation: ', KFold_counter)
    aggregated_losses = []
    df_train, df_test = df.iloc[train_index], df.iloc[test_index]
    X_train_categorical = df_train.iloc[:, :input_size]
    X_train_numerical = df_train.iloc[:, input_size:]
    y_train = df_train['y']
    X_test_categorical = df_test.iloc[:, :input_size]
    X_test_numerical = df_test.iloc[:, input_size:]
    y_test = df_test['y']


    X_train_categorical = X_train_categorical.values
    X_test_categorical = X_test_categorical.values
    X_train_categorical = torch.tensor(X_train_categorical, dtype=torch.int64)
    X_train_numerical = torch.tensor(X_train_numerical.values, dtype=torch.
 ↪float)
    y_train = torch.tensor(y_train.values).flatten()
    X_test_categorical = torch.tensor(X_test_categorical,dtype=torch.int64)
    X_test_numerical = torch.tensor(X_test_numerical.values, dtype=torch.float)
    y_test = torch.tensor(y_test.values).flatten()

    print('\n\nTensor Train_categorical\n', X_train_categorical)
    print('\nTensor Train_numerical\n', X_train_numerical)
    print('\nTensor Output Train\n', y_train)

    print('\nTrain_categorical Shape',X_train_categorical.shape)
    print('Train_numerical Shape',X_train_numerical.shape)
    print('Output Shape',y_train.shape)

# Below piece of code is used to compute the categorical_embedding_sizes for␣
 ↪the model.
# It starts by creating a list called "categorical_column_sizes"
# which contains the number of unique values for each column in the␣
 ↪"df_encoded" dataframe,
# using the len() function and the unique() method from pandas.

# Then it creates another list called "categorical_embedding_sizes", which is a␣
 ↪list of tuples
# where each tuple contains the size of the column and the minimum of 50 and␣
 ↪half of the column size plus 1.
```

```
# This is done using a list comprehension that iterates over the␣
 ↪"categorical_column_sizes" list and applies
# the specified calculations.

# The script then prints the "categorical_embedding_sizes" and its length, and␣
 ↪the shape of the
# "X_train_categorical" tensor. The categorical_embedding_sizes will be used␣
 ↪later in the script
# to create the embedding layers in the Model class.

    categorical_column_sizes = [len(df[column].unique()) for column in␣
 ↪df_encoded]
    categorical_embedding_sizes = [(col_size, min(50, (col_size+1)//2)) for␣
 ↪col_size in categorical_column_sizes]
    print('\n\nCategorical_embedding_sizes and shape\n',␣
 ↪categorical_embedding_sizes, len(categorical_embedding_sizes))
    print('\nX_train_categorical size',X_train_categorical.shape)


# Then creates an instance of the Model class by passing in the␣
 ↪"categorical_embedding_sizes",
# the number of columns in "X_train_numerical", the output size of 2 and a list␣
 ↪of three integers representing
# the number of neurons in three layers [200,100,50].
# The created object is named as 'model'.

    model = Model(categorical_embedding_sizes, X_train_numerical.shape[1], 2,␣
 ↪[200,100,50])
    print('\nModel Summary\n', model)

# Then, it creates an instance of the CrossEntropyLoss class from PyTorch's nn␣
 ↪module
# and assigns it to the variable "loss_function". This loss function will be␣
 ↪used to compute
# the difference between the predicted output of the model and the actual␣
 ↪output,
# and will be used to update the model's parameters during training.

    loss_function = nn.CrossEntropyLoss()

# Then, it creates an optimizer, in this case, an Adam optimizer from PyTorch's
# optim module and passing in the model's parameters and a learning rate of 0.
 ↪001.
# The optimizer will be used to update the model's parameters during training.

    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```
#Finally, it calls the previously defined train_model function for 300 epochs.

    train_model(epochs)

    print(f'epoch: {i:3} loss: {single_loss.item():10.10f}')

# The code above uses the plt.plot() method from the matplotlib library to␣
 ↪create a
# line plot of the training loss over the number of training epochs. The x-axis␣
 ↪of the plot represents
# the number of training epochs, which is specified by the range(epochs)␣
 ↪function. The y-axis of
# the plot represents the loss values, which is specified by the list␣
 ↪comprehension [l.detach().numpy()
# for l in aggregated_losses]. The list comprehension is iterating over the␣
 ↪list of losses stored in
# the "aggregated_losses" list and for each loss, it uses the detach() method␣
 ↪to remove the loss from the
# computation graph, and the numpy() method to convert the loss from a
# PyTorch tensor to a numpy array. This way, the loss values can be plotted␣
 ↪using the matplotlib library.

    plt.plot(range(epochs), [l.detach().numpy() for l in aggregated_losses])
    plt.ylabel('Loss')
    plt.xlabel('epoch');
    plt.show()
    KFold_counter = KFold_counter +1


print('\n\nPredicting with best model chosen\n')

# The code above is used to evaluate the trained model on the test set. The␣
 ↪torch.no_grad()
# context manager is used to indicate that no gradients should be computed␣
 ↪during the evaluation.
# Inside the no_grad context, the script uses the model to predict the outputs␣
 ↪for the test set
# inputs (X_test_categorical and X_test_numerical) and assigns the result to␣
 ↪the variable "y_val".
# Then it calculates the loss between "y_val" and the actual test set outputs␣
 ↪(y_test) using the previously
# defined loss_function.

# Then it prints the calculated loss value. This loss value represents how well␣
 ↪the model is performing on the test set.
```

```
with torch.no_grad():
    y_val = model(X_test_categorical, X_test_numerical)
    loss = loss_function(y_val, y_test)
print(f'Loss: {loss:.8f}')

# Finally, it uses the flush() method from the SummaryWriter class to flush the␣
 ↪event file to disk.
# This is done so that the data can be visualized in Tensorboard.

writer.flush()
```

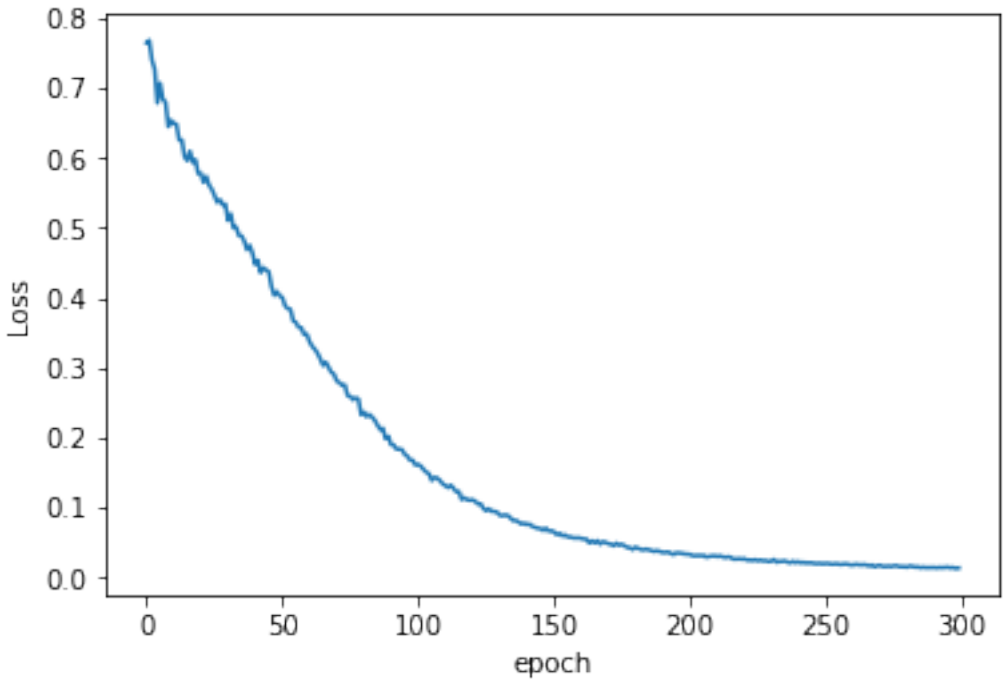
```
 Original Data frame (with first 1000 samples)


     age         job  marital    education  default housing loan    contact  \
0     56   housemaid  married     basic.4y       no      no   no  telephone
1     57    services  married  high.school  unknown      no   no  telephone
..   …         …      …           …      …     …  …        …
998   57  technician  married     basic.9y       no     yes   no  telephone
999   30    services  married      unknown       no      no   no  telephone

    month day_of_week  duration  campaign  pdays  previous     poutcome  \
0     may         mon       261         1    999         0  nonexistent
1     may         mon       149         1    999         0  nonexistent
..    …          …       …       …    …       …           …
998   may         wed        68         1    999         0  nonexistent
999   may         wed       143         3    999         0  nonexistent

     emp.var.rate  cons.price.idx  cons.conf.idx  euribor3m  nr.employed   y
0             1.1          93.994          -36.4      4.857       5191.0  no
1             1.1          93.994          -36.4      4.857       5191.0  no
..            …             …             …        …          …  ..
998           1.1          93.994          -36.4      4.856       5191.0  no
999           1.1          93.994          -36.4      4.856       5191.0  no

[1000 rows x 21 columns]


 Selected Categorical Cols
 Index(['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact',
       'month', 'day_of_week', 'poutcome'],
      dtype='object')

Total unique y outputs(before mapping)
 no     981
yes     19
Name: y, dtype: int64 0      no
```

```
1      no
       ..
998    no
999    no
Name: y, Length: 1000, dtype: object

Total unique y outputs (after mapping)
 0    981
1     19
Name: y, dtype: int64 0      0
1      0
       ..
998    0
999    0
Name: y, Length: 1000, dtype: int64

 Final Scaled Data frame


       0    1    2    3    4    5    6    7    8    9   10   11   12   13  \
0    0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0
1    0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  1.0
..   …  …  …  …  …  …  …  …  …  …  …  …  …  …  …
998  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  1.0
999  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0  1.0

      14       age  duration  campaign  pdays  previous  emp.var.rate  \
0    0.0  1.578333 -0.036086 -0.742897    0.0       0.0  2.220446e-16
1    0.0  1.694336 -0.517121 -0.742897    0.0       0.0  2.220446e-16
..   …       …       …       …    …       …           …
998  0.0  1.694336 -0.865012 -0.742897    0.0       0.0  2.220446e-16
999  0.0 -1.437738 -0.542890  1.484680    0.0       0.0  2.220446e-16

     cons.price.idx  cons.conf.idx  euribor3m  nr.employed  y
0      1.421085e-14  -1.421085e-14   0.568112          0.0  0
1      1.421085e-14  -1.421085e-14   0.568112          0.0  0
..               …             …          …          … ..
998    1.421085e-14  -1.421085e-14  -1.760216          0.0  0
999    1.421085e-14  -1.421085e-14  -1.760216          0.0  0

[1000 rows x 26 columns]


Data type: 0              float64
1              float64
               …
nr.employed    float64
y                int64
Length: 26, dtype: object
```

```
Divided Train/Test Data

df_train, df_test (667, 26) (333, 26)
X_train_categorical torch.Size([667, 15])
X_train_numerical torch.Size([667, 11])
y_train torch.Size([667])
X_test_categorical torch.Size([333, 15])
X_test_numerical torch.Size([333, 11])
y_test torch.Size([333])




K-fold cross validation:  1


Tensor Train_categorical
 tensor([[0, 0, 0,  …, 0, 1, 0],
        [0, 0, 1,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 0, 1],
        …,
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0]])

Tensor Train_numerical
 tensor([[ 0.4183,  0.1873,  1.4847,  …,  0.5681,  0.0000,  0.0000],
        [ 0.7663, -0.5773,  2.5985,  …,  0.5681,  0.0000,  0.0000],
        [-0.0457, -0.5042,  0.3709,  …,  0.5681,  0.0000,  0.0000],
        …,
        [ 1.9263,  0.2560, -0.7429,  …, -1.7602,  0.0000,  0.0000],
        [ 1.6943, -0.8650, -0.7429,  …, -1.7602,  0.0000,  0.0000],
        [-1.4377, -0.5429,  1.4847,  …, -1.7602,  0.0000,  0.0000]])

Tensor Output Train
 tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

Train_categorical Shape torch.Size([666, 15])
Train_numerical Shape torch.Size([666, 11])
Output Shape torch.Size([666])


Categorical_embedding_sizes and shape
 [(2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2,
1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1)] 15

X_train_categorical size torch.Size([666, 15])

Model Summary
 Model(
  (all_embeddings): ModuleList(
    (0): Embedding(2, 1)
    (1): Embedding(2, 1)
    (2): Embedding(2, 1)
    (3): Embedding(2, 1)
    (4): Embedding(2, 1)
    (5): Embedding(2, 1)
    (6): Embedding(2, 1)
    (7): Embedding(2, 1)
    (8): Embedding(2, 1)
    (9): Embedding(2, 1)
    (10): Embedding(2, 1)
    (11): Embedding(2, 1)
    (12): Embedding(2, 1)
    (13): Embedding(2, 1)
    (14): Embedding(2, 1)
```

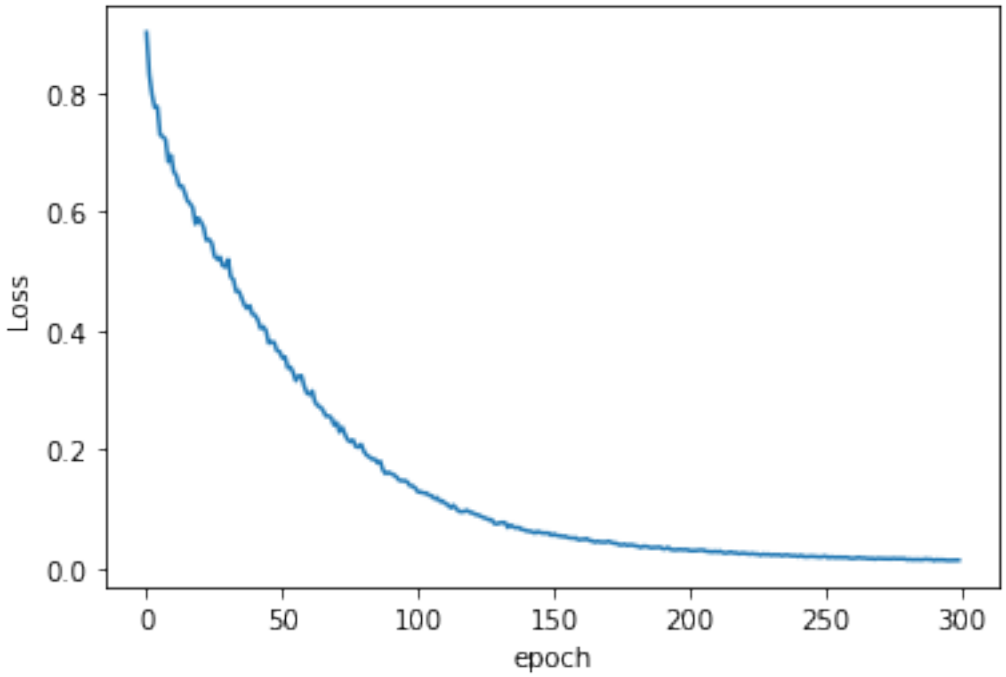
```
  )
  (embedding_dropout): Dropout(p=0.4, inplace=False)
  (batch_norm_num): BatchNorm1d(11, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (layers): Sequential(
    (0): Linear(in_features=26, out_features=200, bias=True)
    (1): ReLU(inplace=True)
    (2): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (3): Dropout(p=0.4, inplace=False)
    (4): Linear(in_features=200, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): Dropout(p=0.4, inplace=False)
    (8): Linear(in_features=100, out_features=50, bias=True)
    (9): ReLU(inplace=True)
    (10): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): Dropout(p=0.4, inplace=False)
    (12): Linear(in_features=50, out_features=2, bias=True)
  )
)
epoch:   1 loss: 0.76439404
epoch:  26 loss: 0.54644740
epoch:  51 loss: 0.40029764
epoch:  76 loss: 0.25874981
epoch: 101 loss: 0.16141389
epoch: 126 loss: 0.09517589
epoch: 151 loss: 0.06483967
epoch: 176 loss: 0.04717654
epoch: 201 loss: 0.03271381
epoch: 226 loss: 0.02370840
epoch: 251 loss: 0.01965448
epoch: 276 loss: 0.01618831
epoch: 300 loss: 0.0154857133
```




```
K-fold cross validation:  2


Tensor Train_categorical
 tensor([[0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        …,
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0]])

Tensor Train_numerical
 tensor([[ 1.5783, -0.0361, -0.7429,  …,  0.5681,  0.0000,  0.0000],
        [ 1.6943, -0.5171, -0.7429,  …,  0.5681,  0.0000,  0.0000],
        [-0.6257, -0.1864, -0.7429,  …,  0.5681,  0.0000,  0.0000],
        …,
        [ 1.9263,  0.2560, -0.7429,  …, -1.7602,  0.0000,  0.0000],
        [ 1.6943, -0.8650, -0.7429,  …, -1.7602,  0.0000,  0.0000],
        [-1.4377, -0.5429,  1.4847,  …, -1.7602,  0.0000,  0.0000]])

Tensor Output Train
 tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

Train_categorical Shape torch.Size([667, 15])
Train_numerical Shape torch.Size([667, 11])
Output Shape torch.Size([667])


Categorical_embedding_sizes and shape
 [(2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2,
1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1)] 15

X_train_categorical size torch.Size([667, 15])

Model Summary
 Model(
  (all_embeddings): ModuleList(
    (0): Embedding(2, 1)
    (1): Embedding(2, 1)
    (2): Embedding(2, 1)
    (3): Embedding(2, 1)
    (4): Embedding(2, 1)
    (5): Embedding(2, 1)
```

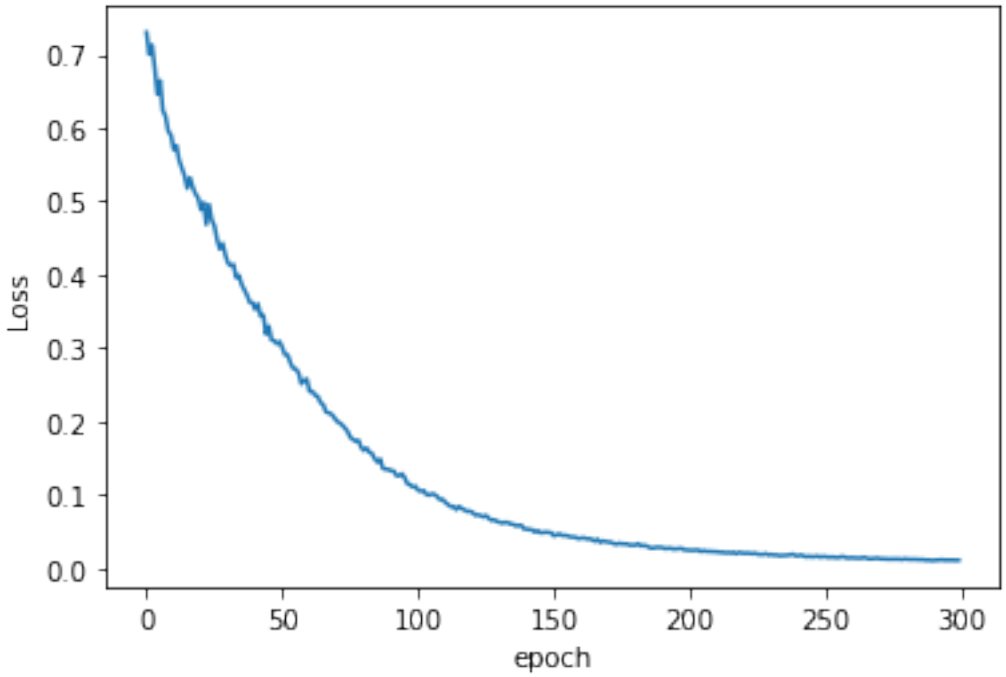
```
    (6): Embedding(2, 1)
    (7): Embedding(2, 1)
    (8): Embedding(2, 1)
    (9): Embedding(2, 1)
    (10): Embedding(2, 1)
    (11): Embedding(2, 1)
    (12): Embedding(2, 1)
    (13): Embedding(2, 1)
    (14): Embedding(2, 1)
  )
  (embedding_dropout): Dropout(p=0.4, inplace=False)
  (batch_norm_num): BatchNorm1d(11, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (layers): Sequential(
    (0): Linear(in_features=26, out_features=200, bias=True)
    (1): ReLU(inplace=True)
    (2): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (3): Dropout(p=0.4, inplace=False)
    (4): Linear(in_features=200, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): Dropout(p=0.4, inplace=False)
    (8): Linear(in_features=100, out_features=50, bias=True)
    (9): ReLU(inplace=True)
    (10): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): Dropout(p=0.4, inplace=False)
    (12): Linear(in_features=50, out_features=2, bias=True)
  )
)
epoch:   1 loss: 0.90104657
epoch:  26 loss: 0.52493924
epoch:  51 loss: 0.35371324
epoch:  76 loss: 0.21229772
epoch: 101 loss: 0.12750976
epoch: 126 loss: 0.08358144
epoch: 151 loss: 0.05711089
epoch: 176 loss: 0.03874510
epoch: 201 loss: 0.02886807
epoch: 226 loss: 0.02316569
epoch: 251 loss: 0.01971798
epoch: 276 loss: 0.01590399
epoch: 300 loss: 0.0154857133
```




```
K-fold cross validation:  3


Tensor Train_categorical
 tensor([[0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        …,
        [0, 1, 0,  …, 0, 1, 0],
        [0, 0, 0,  …, 0, 1, 0],
        [1, 0, 0,  …, 1, 0, 0]])

Tensor Train_numerical
 tensor([[ 1.5783, -0.0361, -0.7429,  …,  0.5681,  0.0000,  0.0000],
        [ 1.6943, -0.5171, -0.7429,  …,  0.5681,  0.0000,  0.0000],
        [-0.6257, -0.1864, -0.7429,  …,  0.5681,  0.0000,  0.0000],
        …,
        [ 0.4183,  0.8487,  0.3709,  …,  0.5681,  0.0000,  0.0000],
        [-1.7857, -0.1048,  0.3709,  …,  0.5681,  0.0000,  0.0000],
        [ 0.5343, -0.4699,  1.4847,  …,  0.5681,  0.0000,  0.0000]])

Tensor Output Train
 tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])

Train_categorical Shape torch.Size([667, 15])
Train_numerical Shape torch.Size([667, 11])
Output Shape torch.Size([667])


Categorical_embedding_sizes and shape
 [(2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1), (2,
1), (2, 1), (2, 1), (2, 1), (2, 1), (2, 1)] 15

X_train_categorical size torch.Size([667, 15])

Model Summary
 Model(
  (all_embeddings): ModuleList(
    (0): Embedding(2, 1)
    (1): Embedding(2, 1)
    (2): Embedding(2, 1)
    (3): Embedding(2, 1)
    (4): Embedding(2, 1)
    (5): Embedding(2, 1)
```

```
    (6): Embedding(2, 1)
    (7): Embedding(2, 1)
    (8): Embedding(2, 1)
    (9): Embedding(2, 1)
    (10): Embedding(2, 1)
    (11): Embedding(2, 1)
    (12): Embedding(2, 1)
    (13): Embedding(2, 1)
    (14): Embedding(2, 1)
  )
  (embedding_dropout): Dropout(p=0.4, inplace=False)
  (batch_norm_num): BatchNorm1d(11, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (layers): Sequential(
    (0): Linear(in_features=26, out_features=200, bias=True)
    (1): ReLU(inplace=True)
    (2): BatchNorm1d(200, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (3): Dropout(p=0.4, inplace=False)
    (4): Linear(in_features=200, out_features=100, bias=True)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): Dropout(p=0.4, inplace=False)
    (8): Linear(in_features=100, out_features=50, bias=True)
    (9): ReLU(inplace=True)
    (10): BatchNorm1d(50, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (11): Dropout(p=0.4, inplace=False)
    (12): Linear(in_features=50, out_features=2, bias=True)
  )
)
epoch:   1 loss: 0.73008651
epoch:  26 loss: 0.46638602
epoch:  51 loss: 0.30036083
epoch:  76 loss: 0.17836973
epoch: 101 loss: 0.10614106
epoch: 126 loss: 0.07176578
epoch: 151 loss: 0.04407472
epoch: 176 loss: 0.03315185
epoch: 201 loss: 0.02446492
epoch: 226 loss: 0.01917595
epoch: 251 loss: 0.01572125
epoch: 276 loss: 0.01228074
epoch: 300 loss: 0.0154857133
```




```
Predicting with best model chosen

Loss: 0.01078232
```

using following command :tensorboard –logdir=/Users/gadgetshop/Desktop/Semester 4-Winter 2022:23/ML-Lab-Machine learning Lab/Lab-08/runs

http://localhost:6006/

[112]:
```
#Q2

#https://github.com/FraLotito/pytorch-continuous-bag-of-words

# This code uses the PyTorch library to define and train a
# Continuous Bag-of-Words (CBOW) model for word embeddings.
# The script first reads in a text file "raw_text.txt" and preprocesses the␣
 ↪text by removing special characters, punctuations,
# numbers and single characters using NLTK and RE library. This step is done by␣
 ↪the prepare_data method.

import torch
import torch.nn as nn
import nltk
```

```
# The script then defines the CBOW model which is a subclass of the PyTorch nn.
 ↪Module class.
# The constructor of the model takes in five parameters: vocab_size,␣
 ↪embedding_dim, hidden_size, epoch_size and word_to_ix.
# The vocab_size is the number of unique words in the vocabulary,
# the embedding_dim is the dimension of the embedding vectors, the hidden_size␣
 ↪is the number of neurons in the hidden layer,
# the epoch_size is the number of times the model will see the data during␣
 ↪training,
# and the word_to_ix is the mapping of words to integer index.

class CBOW(torch.nn.Module):

    def __init__(self, vocab_size, embedding_dim, hidden_size, epoch_size,␣
 ↪test, word_to_ix):

        super(CBOW, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim
        self.hidden_size = hidden_size
        self.epoch_size = epoch_size
        self.test = test
        self.word_to_ix = word_to_ix

# The model has an embeddings layer which is an instance of the nn.Embedding␣
 ↪class,
# this layer maps the input words to their embedding vectors.
# The model also has a linear1 layer, which is a fully connected layer with the␣
 ↪embedding_dim
# as the number of input features and hidden_size as the number of output␣
 ↪features.
# The activation_function1 is ReLU activation function which is applied to the␣
 ↪output of the linear1 layer.
# The linear2 layer is also a fully connected layer with the hidden_size as the␣
 ↪number of input features and vocab_size
# as the number of output features.
# The activation_function2 is LogSoftmax activation function which is applied␣
 ↪to the output of the linear2 layer.

        self.embeddings = nn.Embedding(self.vocab_size, self.embedding_dim)
        self.linear1 = nn.Linear(self.embedding_dim, self.hidden_size)
        self.activation_function1 = nn.ReLU()
        self.linear2 = nn.Linear(hidden_size, vocab_size)
        self.activation_function2 = nn.LogSoftmax(dim = -1)
```

```
# The forward method of the model takes in the inputs, which are the context␣
 ↪words,
# and returns the probability distribution over the vocabulary for the target␣
 ↪word.
# The embeddings of the input words are looked up and summed,
# and the resulting embedding vector is passed through the linear1 layer, ReLU␣
 ↪activation function, linear2 layer
# and LogSoftmax activation function.

    def forward(self, inputs):
        embeds = sum(self.embeddings(inputs)).view(1,-1)
        out = self.linear1(embeds)
        out = self.activation_function1(out)
        out = self.linear2(out)
        out = self.activation_function2(out)
        return out

# Class fininshed
# main code started


embedding_size = 100
hidden_size = 128
epoch_size = 50

# The prepare_data method tokenizes the raw text and converts it to lowercase,
# and removes special characters, punctuations, numbers and single characters.
# Then it creates a vocabulary set and maps each word to a unique integer index.
 ↪
# Next, it prepares the training data by creating a context window of 2 words␣
 ↪to the left and 2 words to the right
# for each target word in the text.
# It then returns the vocab_size, data, word_to_ix, ix_to_word.

with open("raw_text.txt") as f:
    input_text = f.read()

vocab_size = 0
data = []
word_to_ix = 0


# This is a function called prepare_data which takes in a raw_text as an input.
# It starts by tokenizing the text using the NLTK library's word_tokenize␣
 ↪function,
# and converts all the words to lowercase. Then it prints out the original text.
```

```
# Then it removes special characters and punctuations from the words using RE library's sub function.
# It also removes any digits from the words and any word that has a length of less than 2 characters.
# The filtered text is printed out.

# It then creates a vocabulary set from the filtered text, this set will contain all the unique words from the text.
# The size of the vocabulary set is stored in vocab_size variable.
# It creates a word_to_ix dictionary which maps each word to a unique integer index,
# and an ix_to_word dictionary which maps the integer index back to the corresponding word.

# Next, it creates an empty list called data. The script then iterates through the filtered text,
# and for each word, it creates a context window of 2 words to the left and 2 words to the right.
# The current word is stored as the target.
# It then appends the tuple of (context, target) to the data list, resulting in a list of tuples
# where each tuple contains a list of context words and a target word.
# Finally, the function returns vocab_size, data, word_to_ix, ix_to_word.
# The returned data, word_to_ix and ix_to_word will be used to train the CBOW model.


def prepare_data(raw_text):
    tokenizer = nltk.tokenize.word_tokenize
    words = [word.lower() for word in tokenizer(raw_text)]
    print('original text :', words)
    print('\n')
    words = [re.sub(r'[^\w\s]','',word) for word in words] # remove special characters and punctuations
    words = [word for word in words if not word.isdigit()] # remove numbers
    words = [word for word in words if len(word) > 1] # remove single characters
    print('Filtered text :', words)
    print('\n')
    raw_text = words
    vocab = set(raw_text)
    vocab_size = len(vocab)
    word_to_ix = {word:ix for ix, word in enumerate(vocab)}
    ix_to_word = {ix:word for ix, word in enumerate(vocab)}
    data = []
    for i in range(2, len(raw_text) - 2):
```

```
        context = [raw_text[i - 2], raw_text[i - 1],
                   raw_text[i + 1], raw_text[i + 2]]
        target = raw_text[i]
        data.append((context, target))
    return vocab_size, data, word_to_ix, ix_to_word

vocab_size, data, word_to_ix, ix_to_word  = prepare_data(input_text)

test = ['leaving', 'the' , 'question', 'blank']

model = CBOW(vocab_size = vocab_size, embedding_dim =␣
 ↪embedding_size,hidden_size = hidden_size, epoch_size = epoch_size, test =␣
 ↪test,word_to_ix = word_to_ix)


# This is a function called create_input_vector which takes in two inputs,␣
 ↪context_words and word_to_index.

# The function starts by creating a list called indexes, which will store the␣
 ↪integer indexes of the context_words.
# It does this by iterating over the context_words and for each word,
# it looks up the word_to_index dictionary to get the integer index␣
 ↪corresponding to that word,
# and appends it to the indexes list.

# It then converts the indexes list into a PyTorch tensor using the torch.
 ↪tensor() function.
# The dtype is set to torch.long, which specifies that the tensor should be of␣
 ↪type long.

# Finally, it returns the input vector which is a tensor of integer indexes␣
 ↪representing the context words,
# this will be passed as input to the model during the training.

def create_input_vector(context_words, word_to_index):
    indexes = [word_to_index[w] for w in context_words]
    return torch.tensor(indexes, dtype=torch.long)

# This is a function called train which takes in a data as an input.
# It starts by creating a loss function which is an instance of the PyTorch nn.
 ↪NLLLoss class,
# this loss function will be used to compute the negative log likelihood of the␣
 ↪predicted target word.

# It then creates an optimizer which is an instance of the PyTorch optim.SGD␣
 ↪class,
```

```
# this optimizer will be used to update the model's parameters during training.
# The optimizer is initialized with the parameters of the model and a learning␣
 ↪rate of 0.001.

# The script then starts a for loop which will iterate for 50 times.
# This loop represents the training process. It initializes a variable␣
 ↪total_loss to 0,
# which will keep track of the total loss over all the data during each epoch.

# It then starts another for loop which will iterate over the data,
# where each iteration corresponds to a single training example.
# The loop takes in a context and target from the data, where context is a list␣
 ↪of context words and target is the corresponding target word. It then calls␣
 ↪the make_context_vector function, passing in the context and word_to_ix as␣
 ↪arguments. The make_context_vector function is not shown in this code, but␣
 ↪it is expected to convert the context words into a context vector which can␣
 ↪be passed into the model.

# The script then calls the forward method of the model, passing in the context␣
 ↪vector as the input.
# This returns the log probabilities of the predicted target word over the␣
 ↪vocabulary.
# It then computes the loss by passing in the log probabilities and
# the target word's index to the loss function. The target word's index is␣
 ↪obtained by looking up the word_to_ix dictionary.
# The loss is added to the total_loss variable.

# At the end of each epoch, the script optimizes the model by calling optimizer.
 ↪zero_grad()
# which sets the gradients of the model's parameters to zero, and then␣
 ↪backward()
# which computes the gradients of the total loss with respect to the model's␣
 ↪parameters.

def train(data):

    loss_function = nn.NLLLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

    for epoch in range(epoch_size):
        total_loss = 0

        for context, target in data:
            context_vector = create_input_vector(context, word_to_ix)

            log_probs = model(context_vector)
```

```
            total_loss += loss_function(log_probs, torch.
 ↪tensor([word_to_ix[target]]))

        optimizer.zero_grad()
        total_loss.backward()
        optimizer.step()

train(data)

context_vector = create_input_vector(test, word_to_ix)

predicted_word = model(inputs = context_vector)

#Print result
print(f'Raw text: {" ".join(raw_text)}\n')
print(f'Test Data: {test}\n')

# Following line of code is printing the predicted word from the input vector.

# It starts with the variable "predicted_word", which is the output of the␣
 ↪model after passing in the input vector.
# This output is in the form of a tensor, with the predicted probability of␣
 ↪each word in the vocabulary.

# The function torch.argmax(predicted_word[0]) is used to find the index of the␣
 ↪highest probability in the tensor.
# This returns the index of the word that the model predicts is most likely to␣
 ↪be the target word.

# The .item() function is then called on this index to convert the single value␣
 ↪tensor to a python number.

# Finally, we use the ix_to_word dictionary to look up the word corresponding␣
 ↪to this index
# and print it out as the predicted word, using string formatting.

print(f'Prediction: {ix_to_word[torch.argmax(predicted_word[0]).item()]}')
```

```
original text : ['during', 'my', 'second', 'month', 'of', 'nursing', 'school',
',', 'our', 'professor', 'gave', 'us', 'a', 'pop', 'quiz', '.', 'i', 'was', 'a',
'conscientious', 'student', 'and', 'had', 'breezed', 'through', 'the',
'questions', ',', 'until', 'i', 'read', 'the', 'last', 'one', ':', '"', 'what',
'is', 'the', 'first', 'name', 'of', 'the', 'woman', 'who', 'cleans', 'the',
'school', '?', '"', 'surely', 'this', 'was', 'some', 'kind', 'of', 'joke', '.',
'i', 'had', 'seen', 'the', 'cleaning', 'woman', 'several', 'times', '.', 'she',
'was', 'tall', ',', 'dark-haired', 'and', 'in', 'her', '50s', ',', 'but', 'how',
```

```
'would', 'i', 'know', 'her', 'name', '?', 'i', 'handed', 'in', 'my', 'paper',
',', 'leaving', 'the', 'last', 'question', 'blank', '.', 'before', 'class',
'ended', ',', 'one', 'student', 'asked', 'if', 'the', 'last', 'question',
'would', 'count', 'toward', 'our', 'quiz', 'grade', '.', '"', 'absolutely', ',',
'"', 'said', 'the', 'professor', '.', '"', 'in', 'your', 'careers', 'you',
'will', 'meet', 'many', 'people', '.', 'all', 'are', 'significant', '.', 'they',
'deserve', 'your', 'attention', 'and', 'care', ',', 'even', 'if', 'all', 'you',
'do', 'is', 'smile', 'and', 'say', '‘', 'hello', '’', '.', 'i', '’', 've',
'never', 'forgotten', 'that', 'lesson', '.', 'i', 'also', 'learned', 'her',
'name', 'was', 'dorothy', '.']
```

```
Filtered text : ['during', 'my', 'second', 'month', 'of', 'nursing', 'school',
'our', 'professor', 'gave', 'us', 'pop', 'quiz', 'was', 'conscientious',
'student', 'and', 'had', 'breezed', 'through', 'the', 'questions', 'until',
'read', 'the', 'last', 'one', 'what', 'is', 'the', 'first', 'name', 'of', 'the',
'woman', 'who', 'cleans', 'the', 'school', 'surely', 'this', 'was', 'some',
'kind', 'of', 'joke', 'had', 'seen', 'the', 'cleaning', 'woman', 'several',
'times', 'she', 'was', 'tall', 'darkhaired', 'and', 'in', 'her', '50s', 'but',
'how', 'would', 'know', 'her', 'name', 'handed', 'in', 'my', 'paper', 'leaving',
'the', 'last', 'question', 'blank', 'before', 'class', 'ended', 'one',
'student', 'asked', 'if', 'the', 'last', 'question', 'would', 'count', 'toward',
'our', 'quiz', 'grade', 'absolutely', 'said', 'the', 'professor', 'in', 'your',
'careers', 'you', 'will', 'meet', 'many', 'people', 'all', 'are', 'significant',
'they', 'deserve', 'your', 'attention', 'and', 'care', 'even', 'if', 'all',
'you', 'do', 'is', 'smile', 'and', 'say', 'hello', 've', 'never', 'forgotten',
'that', 'lesson', 'also', 'learned', 'her', 'name', 'was', 'dorothy']
```

```
Raw text: during my second month of nursing school our professor gave us pop
quiz was conscientious student and had breezed through the questions until read
the last one what is the first name of the woman who cleans the school surely
this was some kind of joke had seen the cleaning woman several times she was
tall darkhaired and in her 50s but how would know her name handed in my paper
leaving the last question blank before class ended one student asked if the last
question would count toward our quiz grade absolutely said the professor in your
careers you will meet many people all are significant they deserve your
attention and care even if all you do is smile and say hello ve never forgotten
that lesson also learned her name was dorothy

Test Data: ['leaving', 'the', 'question', 'blank']

Prediction: last
```

[ ]: