Drew Lickman

Dr. Barbosa

Project 4, due 10/30/24

AI Disclaimer:

Sentiment Analysis with Recurrent Neural Network (RNN Unit)

Dr. Sal Barbosa, Department of Computer Science, Middle Tennessee State University

In [1]:
```
# Adjustable variables
max_len                 = 512   # default: 512, best: 512
splitPercent    = 0.9   # default: 0.5, best: 0.99
batch_size              = 4     # default: 8, best: 4
output_dim              = 6     # 6 classes
hidden_dim              = 128   # Default: 64, best: 128
rtype                   = 'gru' # Default: rnn, best: gru
n_layers                = 1     # Default: 1, best: 2
dropout_rate    = 0.2   # Default: 0.1, best: 0.2
bidirectional   = True  # Default: False, best: True
epochs                  = 5             # Trial and error: evaluate when validation loss stop decreasing consis
```

In [2]:
```
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

from string import punctuation

import json

# import Word2Vec gensim utilities
from gensim.models import KeyedVectors
```

References

> Personal Notes/Projects
> Book Chapters
>
> - D. Jurafsky and J. Martin "Speech and Language Processing (3 Feb 2024 Draft)"
> - A. Bansal "Advanced Natural Language Processing with TensorFlow 2" (Chapter 2)
> - D. Rao and B. McMahan "Natural Language Processing with Pytorch" (Chapter 6)
>
> Some approaches and ideas for pre-processing and training/testing were gathered from:
> - https://github.com/bentrevett/pytorch-sentiment-analysis/blob/main/2%20-%20Recurrent%20Neural%20Networks.ipynb
> - https://github.com/cezannec/CNN_Text_Classification/blob/master/CNN_Text_Classification.ipynb

---

Load the Data

In [3]:
```
# Data directory
#DATADIR = "/scratch/user/u.sb157846/data/datasets/cb_speeches/"
DATADIR = "./" # Local speech data
```

In [4]:
```
# Read data file
with open(DATADIR + "cb_speeches.txt", encoding='utf8') as f:
```

```
    documents = [(story.strip().split('\t')[0].split(), story.strip().split('\t')[1]) for story in f] # Label i

# documents is a list of 2-tuples consisting of a list of the speech's tokens and its class

# Shuffle the data
#np.random.seed(42)
np.random.shuffle(documents)
```

In [5]:

```
# Split documents into speeches and labels
speeches, labels = zip(*documents)
#print(speeches[3735], labels[3735])
```

Encode labels

As was the case in the custom logistic regression classifier, to use speech labels in the neural network they must be converted from text to integers.

In [6]:

```
countries = {'canada': 0, 'euro area': 1, 'japan': 2, 'sweden': 3, 'united kingdom': 4, 'united states': 5}
encoded_labels = np.array([countries[label] for label in labels])
print(len(speeches), len(encoded_labels))
```

```
4240 4240
```

Using Pre-Trained Word Embeddings (word2vec)

The 300-dimentional word2vec pre-trained embeddings (`GoogleNews-vectors-negative300-SLIM.bin`) are used in this example. However, other pre-trained vectors like *Glove* or *FastText* can easily be substituted in their place.

> Set the directory location of vectors in the `EMB_DIR` variable as shown below.

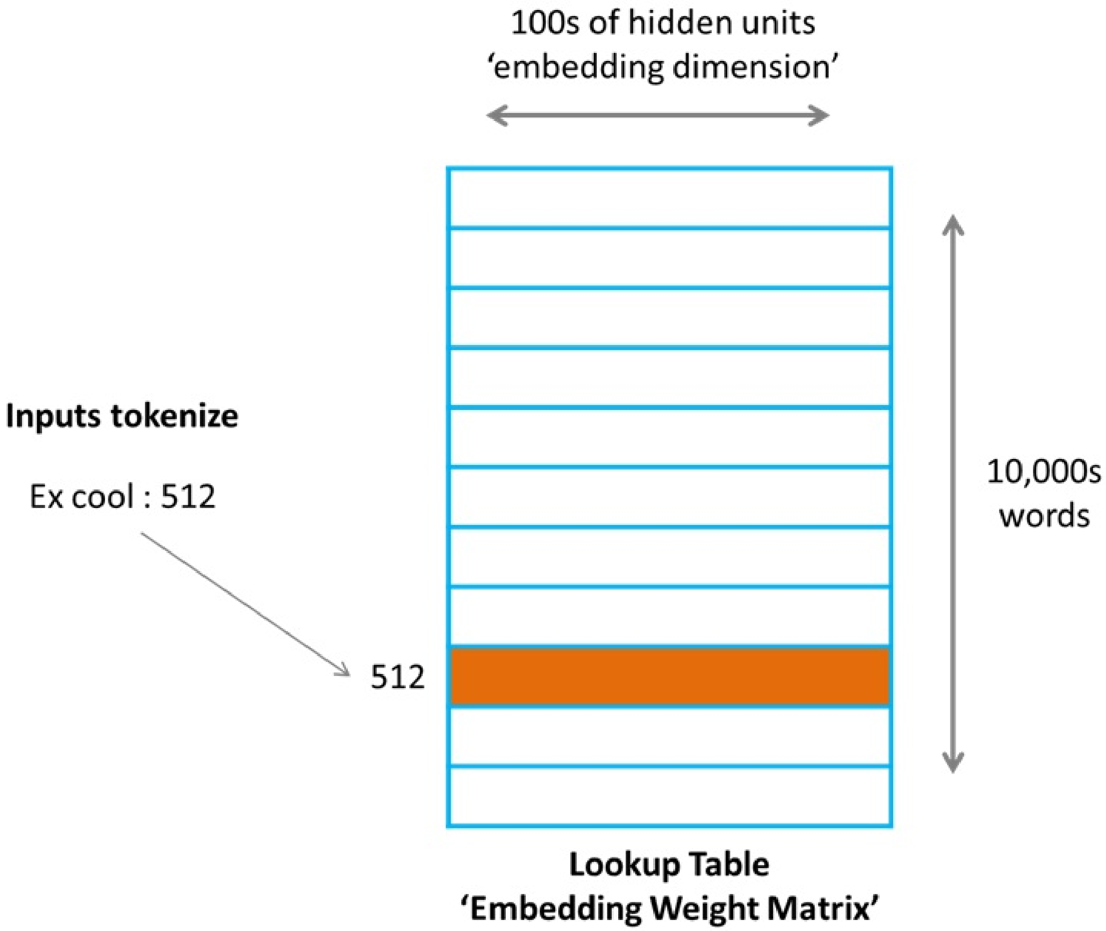
In [7]:

```
# Pre-trained embedding file location
#EMB_DIR = "/scratch/user/u.sb157846/data/word_vectors/GoogleNews-vectors-negative300-SLIM/"
EMB_DIR = "./" # Local directory

# Creating the embeddings matrix
embeddings_table = KeyedVectors.load_word2vec_format(EMB_DIR+'GoogleNews-vectors-negative300-SLIM.bin', binary=
```

Embeddings Matrix

The embeddings matrix in a table, are indexed by a word and contain that word's vector representation. In this example the embedding dimension is 300. The below example show the vector for the word *cool* is stored at index 512 of the lookup matrix.



In [8]:
```
# store pretrained vocabulary
pretrained_words = []
for word in embeddings_table.index_to_key:
    pretrained_words.append(word)
```

In [9]:
```
row = 0

# vocabulary information
print(f"Vocabulary Size: {len(pretrained_words)}\n")

# word/embedding information
word = pretrained_words[row]                    # words from index
embedding = embeddings_table[word]              # embedding from word
print(f"Embedding Size: {len(embedding)}\n")
print(f"Word in vocab: {word}\n")
#print('Associated embedding: \n', embedding)
```

```
Vocabulary Size: 299567

Embedding Size: 300

Word in vocab: in
```

Converting word tokens to index values

The input to neural networks for a 64-token sequence, for example, is not 64 x embedding dimension. Instead of sending the vectors to words as input, each token's embedding matrix index is sent. The entire embedding matrix is supplied to the network at initialization. During training, when the input is received by the neural network, the translation from indexes (indices) to vectors is done in the embedding layer. The below function converts lists of tokenized speeches to index values.

In [10]:
```
# convert tokens to index values
def tokens_to_index(embeddings_table, speeches):
    indexed_input = [[ embeddings_table.key_to_index[speech_tok] if speech_tok in embeddings_table.key_to_index
                      for speech_tok in speech] for speech in speeches]
    return indexed_input
```

In [11]:
```
indexed_speeches = tokens_to_index(embeddings_table, speeches)
```

Indexing of a short speech

In [12]:
```
# indexed tokens for a short speech
rev = None
while rev is None:
    rnd_idx = np.random.randint(len(speeches)) # Select a random speech from corpus
    rev = rnd_idx if len(speeches[rnd_idx]) < 150 else None

# print(' '.join(speeches[rev]))
# print()

# for i in range(len(speeches[rev])):
#     print(f"{speeches[rev][i]:<10}{indexed_speeches[rev][i]:>7}", end='\t')
#     if i % 5 == 4:
#         print()
# print()
```

Padding Speeches

Neural networks generally require that all input have the same dimensions. The speeches, however, have varying lengths. A maximum length of speeches must be chosen after analyzing average and maximum lengths of input as well as model constraints (architecture, memory usage, etc.). This demonstration uses 256 tokens as the maximum length. Any speech longer than this is truncated (tokens beyond 256 are discarded) and shorter speeches are left padded with zeros).

The below function pads all speeches to the maximum length.

In [13]:
```
# This function pads all speeches
def pad_input(tokenized_input, max_len):

    # Get a zero tensor of the correct shape
    features = np.zeros((len(tokenized_input), max_len), dtype=int)

    # Go through each speech and copy the indices into the features list (of padded speeches)
    for i, row in enumerate(tokenized_input):
        try:
            features[i, -len(row):] = np.array(row)[:max_len]
        except:
            print(i, row)

    return features

# Instead of padding to a max of 256, could I get the average speech length and pad up to that and truncate dowi
```

In [14]:
```
#max_len = 512 # Adjustable variable

features = pad_input(indexed_speeches, max_len=max_len)

# Check dimension lengths in resulsing data
assert len(features)==len(features), "Features must have same length as speeches."
assert len(features[0])==max_len, "Each feature row should contain max_len values."

#print(features[:25,:10])
```

Split Data into Training, Validation, and Test

The data must be split into training and test data minimally. Many training loops can also use validation data at the end of each epoch, allowing a comparison between training and validation losses (if this value is high or growing it may indicate overfitting).

The split for this demonstration will be 80% training and 10% each for test and validation.

In [15]:
```
# training/test split (validation will come from test portion)
tt_split = int(len(features) * splitPercent)

train_x, valtest_x = features[:tt_split], features[tt_split:]
train_y, valtest_y = encoded_labels[:tt_split], encoded_labels[tt_split:]
```

```
# Validation/test split (further split test data into validation and test)
vt_split = int(len(valtest_x)*0.5)
val_x, test_x = valtest_x[:vt_split], valtest_x[vt_split:]
val_y, test_y = valtest_y[:vt_split], valtest_y[vt_split:]

## Show shapes of data
print("\t\t\tFeature Shapes:")
print("Train set: \t\t{}".format(train_x.shape),
      "\nValidation set: \t{}".format(val_x.shape),
      "\nTest set: \t\t{}".format(test_x.shape))
```

```
                        Feature Shapes:
Train set:              (3816, 512)
Validation set:         (212, 512)
Test set:               (212, 512)
```

Batching and DataLoaders

Neural networks work best when data is processed in batches. This reduces convergence time through calculating and applying the average of losses and gradients of properly sized batches (compared to single input processing or full batch processing, which may not fit in memory).

PyTorch provides utilities for creating and managing batched data. The data is placed into Datasets, and Dataloaders are used to shuffle the data (if desired) and break it up into batches. This is carried out below for the traininng, validation, and test splits.

In [16]:

```
# create Tensor datasets
train_data              = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y).long())
valid_data              = TensorDataset(torch.from_numpy(val_x), torch.from_numpy(val_y).long())
test_data               = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y).long())
# Had to convert to .long() type so the code would function

# dataloaders
#batch_size             = 4 # default: 8, best: 4

# shuffling and batching data
train_loader    = DataLoader(train_data, shuffle=True, batch_size=batch_size)
valid_loader    = DataLoader(valid_data, shuffle=True, batch_size=batch_size)
test_loader     = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

The model

The neural network model below is a simple RNN-module based recurrent neural network.

In [17]:

```
class MulticlasRNN(nn.Module):
        def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers, dropout_rate,
                                bidirectional=False, embed_model=None, freeze_embeddings=True):

                super(MulticlasRNN, self).__init__()

                self.bidirectional = bidirectional
                self.hidden_dim = hidden_dim

                # Network Layers

                # Embedding Layer
                self.embedding = nn.Embedding(vocab_size, embedding_dim)

                # If pre-trained vectors are defined load the weights
                if embed_model is not None:
                        print("Loading pre-trained vectors")
                        self.embedding.weight = nn.Parameter(torch.from_numpy(embed_model.vectors)) # all vecto

                        # freeze embedding weights (since we're not fine-tuning them)
                        if freeze_embeddings:
                                print("Freezing pre-trained vectors")
                                self.embedding.requires_grad = False

                # RNN Layer
                if rtype == 'gru':
                        self.rnn = nn.GRU(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional, dro
                else:
                        self.rnn = nn.RNN(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional, dro

                # Adjust hidden_dim for bidirectional RNN
```

```
        self.attention_dim = hidden_dim * 2 if bidirectional else hidden_dim

        # Linear layer to project RNN output to attention input dimension
        self.project = nn.Linear(self.attention_dim, hidden_dim)

        self.attention = nn.MultiheadAttention(hidden_dim, num_heads=8)

        # Add multiple dense layers with ReLU and BatchNorm
        self.fc1 = nn.Linear(hidden_dim, hidden_dim//2)
        self.bn1 = nn.BatchNorm1d(hidden_dim//2)
        self.relu1 = nn.ReLU()

        self.fc2 = nn.Linear(hidden_dim//2, hidden_dim//4)
        self.bn2 = nn.BatchNorm1d(hidden_dim//4)
        self.relu2 = nn.ReLU()

        # Final classification layer
        self.fc3 = nn.Linear(hidden_dim//4, output_dim)

        # Residual connections
        self.residual = nn.Linear(hidden_dim, output_dim)

        # Layer normalization
        self.layer_norm = nn.LayerNorm(hidden_dim)

        # Dropout Layer
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x):

        batch_size = x.size(0)

        # Convert token index values to vector embeddings
        embedded = self.embedding(x)

        # Pass embeddings to RNN and get back output and hidden state
        out, hidden = self.rnn(embedded)

        # Project RNN output to match attention input dimension
        projected = self.project(out)

        # Apply attention
        attn_output, _ = self.attention(projected.transpose(0, 1), projected.transpose(0, 1), projected
        attn_output = attn_output.transpose(0, 1)

        # Apply layer normalization
        normalized = self.layer_norm(attn_output)

        # Get the hidden state of the last time step
        if self.bidirectional:
                last_hidden = torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim=1)
                last_hidden = self.project(last_hidden)
        else:
                last_hidden = hidden[-1,:,:]

        # Apply dropout
        x_do_out = self.dropout(last_hidden)

        # Residual connection
        residual = self.residual(x_do_out)

        # Multiple dense layers with BatchNorm and ReLU
        x = self.fc1(x_do_out)
        x = self.bn1(x)
        x = self.relu1(x)
        x = self.dropout(x)

        x = self.fc2(x)
        x = self.bn2(x)
        x = self.relu2(x)
        x = self.dropout(x)

        x = self.fc3(x)

        # Add residual connection
        x = x + residual

        return x
```

Model Parameters

In [18]:
```
# General parameters
vocab_size = len(pretrained_words)
#output_size = 1 # binary class (1 or 0)
embedding_dim = len(embeddings_table[pretrained_words[0]]) # 300-dim vectors
```

Neural Network Hyperparameters

In [19]:
```
# RNN-specific parameters
# output_dim = 6
# hidden_dim = 128              # Default: 64, best: 128
# rtype = 'gru'                         # Default: rnn, best: gru
# n_layers = 2                  # Default: 1, best: 2
# dropout_rate = 0.2            # Default: 0.1, best: 0.2
# bidirectional = True  # Default: False, best: True
emb = embeddings_table  # Default: embeddings_table, best: embeddings_table     # embeddings_table or None
freeze = True                   # Default: True, best: True                                 # Only i

# Instantiate RNN model # send embeddings_table in place of None
rnn_model = MulticlasRNN(vocab_size, embedding_dim, hidden_dim, output_dim, rtype, n_layers, dropout_rate,
                         bidirectional=bidirectional, embed_model=emb, freeze_embeddings=freeze)
```

```
Loading pre-trained vectors
Freezing pre-trained vectors
```

```
C:\Users\drew1\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n2kfra8p0\LocalCache\local-packag
es\Python311\site-packages\torch\nn\modules\rnn.py:123: UserWarning: dropout option adds dropout after all but l
ast recurrent layer, so non-zero dropout expects num_layers greater than 1, but got dropout=0.2 and num_layers=1
  warnings.warn(
```

Training

In [20]:
```
# Device
import torch.version

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
device
```

Out[20]:
```
device(type='cuda')
```

In [21]:
```
# Model loss and optimizer
rnn_lr = 5e-4
rnn_criterion = nn.CrossEntropyLoss() #Multiclass so can't use nn.BCELoss()
rnn_optimizer = optim.Adam(rnn_model.parameters(), lr=rnn_lr)
```

In [22]:
```
# Instantiate the model
rnn_model = rnn_model.to(device)
```

Training Loop

In [23]:
```
# Training loop
def rnn_train(net, train_loader, epochs, print_every=100):

    # Move model to GPU/CPU
    net.to(device)

    counter = 0 # for printing

    # Place model in training mode
    net.train()

    # Train for some number of epochs
    for e in range(epochs):

        # batch loop
        for inputs, labels in train_loader:
            counter += 1

            # Move training data batch to GPU/CPU
            inputs, labels = inputs.to(device), labels.to(device)

            # Zero out gradients
```

```python
            net.zero_grad()

            # Get the output from the model
            output = net(inputs)

            # Calculate the loss
            loss = rnn_criterion(output, labels)

            # Backpropagate
            loss.backward()

            # Update weights
            rnn_optimizer.step()

            # Validation
            if counter % print_every == 0:

                # Get validation loss
                val_losses = []

                # Place model in evaluation mode (weights are not updated)
                net.eval()

                # Go through validation set
                for inputs, labels in valid_loader:

                    # Move validation data batch to GPU/CPU
                    inputs, labels = inputs.to(device), labels.to(device)
                    # get predicted label
                    output = net(inputs)

                    # Calculate validation loss
                    #val_loss = rnn_criterion(output.squeeze(), labels.float())
                    val_loss = rnn_criterion(output, labels)

                    # Retain for average calculation
                    val_losses.append(val_loss.item())

                # Place model back in training mode
                net.train()

                # Output losses
                print("Epoch: {}/{}...".format(e+1, epochs),
                      "Step: {}...".format(counter),
                      "Train Loss: {:.6f}...".format(loss.item()),
                      "Val Loss: {:.6f}".format(np.mean(val_losses)))
```

In [24]:

```python
# Clear GPU memory before taining
if device == 'cuda':
    torch.cuda.empty_cache()
```

In [25]:

```python
# Training parameters and invoking training
#epochs = 5        # Trial and error: evaluate when validation loss stop decreasing consistently
print_every = 100
rnn_train(rnn_model, train_loader, epochs, print_every=print_every)
```

```
Epoch: 1/5... Step: 100... Train Loss: 1.858565... Val Loss: 1.751625
Epoch: 1/5... Step: 200... Train Loss: 1.574252... Val Loss: 1.720800
Epoch: 1/5... Step: 300... Train Loss: 1.581407... Val Loss: 1.676655
Epoch: 1/5... Step: 400... Train Loss: 1.971929... Val Loss: 1.597194
Epoch: 1/5... Step: 500... Train Loss: 1.689115... Val Loss: 1.562420
Epoch: 1/5... Step: 600... Train Loss: 1.786641... Val Loss: 1.523804
Epoch: 1/5... Step: 700... Train Loss: 1.340854... Val Loss: 1.484850
Epoch: 1/5... Step: 800... Train Loss: 1.355609... Val Loss: 1.442950
Epoch: 1/5... Step: 900... Train Loss: 1.784382... Val Loss: 1.417203
Epoch: 2/5... Step: 1000... Train Loss: 1.631276... Val Loss: 1.350674
Epoch: 2/5... Step: 1100... Train Loss: 1.231816... Val Loss: 1.331890
Epoch: 2/5... Step: 1200... Train Loss: 1.487851... Val Loss: 1.307150
Epoch: 2/5... Step: 1300... Train Loss: 1.439057... Val Loss: 1.172071
Epoch: 2/5... Step: 1400... Train Loss: 0.787474... Val Loss: 1.200143
Epoch: 2/5... Step: 1500... Train Loss: 1.347981... Val Loss: 1.117826
Epoch: 2/5... Step: 1600... Train Loss: 0.254589... Val Loss: 1.053495
Epoch: 2/5... Step: 1700... Train Loss: 1.071380... Val Loss: 0.936303
Epoch: 2/5... Step: 1800... Train Loss: 1.140666... Val Loss: 1.140688
Epoch: 2/5... Step: 1900... Train Loss: 0.859251... Val Loss: 1.016542
Epoch: 3/5... Step: 2000... Train Loss: 1.610591... Val Loss: 0.771779
Epoch: 3/5... Step: 2100... Train Loss: 0.459311... Val Loss: 0.789339
Epoch: 3/5... Step: 2200... Train Loss: 0.942752... Val Loss: 0.770263
Epoch: 3/5... Step: 2300... Train Loss: 1.120932... Val Loss: 0.723770
Epoch: 3/5... Step: 2400... Train Loss: 0.215841... Val Loss: 0.654607
Epoch: 3/5... Step: 2500... Train Loss: 0.183158... Val Loss: 0.713914
Epoch: 3/5... Step: 2600... Train Loss: 0.277138... Val Loss: 0.603362
Epoch: 3/5... Step: 2700... Train Loss: 0.865367... Val Loss: 0.609496
Epoch: 3/5... Step: 2800... Train Loss: 0.075917... Val Loss: 0.519117
Epoch: 4/5... Step: 2900... Train Loss: 0.348420... Val Loss: 0.589155
Epoch: 4/5... Step: 3000... Train Loss: 0.212735... Val Loss: 0.706127
Epoch: 4/5... Step: 3100... Train Loss: 1.388475... Val Loss: 0.461997
Epoch: 4/5... Step: 3200... Train Loss: 0.526071... Val Loss: 0.715499
Epoch: 4/5... Step: 3300... Train Loss: 0.388487... Val Loss: 0.526483
Epoch: 4/5... Step: 3400... Train Loss: 0.101951... Val Loss: 0.545650
Epoch: 4/5... Step: 3500... Train Loss: 0.019969... Val Loss: 0.412305
Epoch: 4/5... Step: 3600... Train Loss: 0.041025... Val Loss: 0.406672
Epoch: 4/5... Step: 3700... Train Loss: 0.023922... Val Loss: 0.459219
Epoch: 4/5... Step: 3800... Train Loss: 0.848623... Val Loss: 0.427831
Epoch: 5/5... Step: 3900... Train Loss: 0.001911... Val Loss: 0.395914
Epoch: 5/5... Step: 4000... Train Loss: 0.012347... Val Loss: 0.376494
Epoch: 5/5... Step: 4100... Train Loss: 0.268951... Val Loss: 0.355560
Epoch: 5/5... Step: 4200... Train Loss: 0.032402... Val Loss: 0.406768
Epoch: 5/5... Step: 4300... Train Loss: 0.002095... Val Loss: 0.390715
Epoch: 5/5... Step: 4400... Train Loss: 0.303330... Val Loss: 0.343336
Epoch: 5/5... Step: 4500... Train Loss: 0.003611... Val Loss: 0.369627
Epoch: 5/5... Step: 4600... Train Loss: 0.040754... Val Loss: 0.388942
Epoch: 5/5... Step: 4700... Train Loss: 0.004724... Val Loss: 0.408038
```

Testing the Model

In [26]:

```
#Testing loop
def rnn_test(test_loader):
    # Turn off gradient calculations (saves time and compute resources)
    with torch.no_grad():

        # Variables for tracking losses
        test_losses = []
        num_correct = 0

        true_list = []
        pred_list = []

        # Place model in evaluation mode
        rnn_model.eval()

        # Run test data through model
        for inputs, labels in test_loader:

            # Move test data batch to GPU/CPU
            inputs, labels = inputs.to(device), labels.to(device)

            # Get predicted output
            output = rnn_model(inputs)

            # Calculate the loss
            # test_loss = rnn_criterion(output.squeeze(), labels.float())
            test_loss = rnn_criterion(output.squeeze(), labels)
            test_losses.append(test_loss.item())
```

```python
            # Convert output sigmoid probabilities to predicted classes (0 or 1)
            #pred = torch.round(output.squeeze())  # rounds to the nearest integer
            pred = torch.argmax(output, dim=1)

            # Place true and predicted labels in list
            true_list += list(labels.cpu().numpy())
            pred_list += list(pred.cpu().numpy())

            # Compare predicted and true labels and count number of correct prediction
            correct_tensor = pred.eq(labels.float().view_as(pred))
            correct = np.squeeze(correct_tensor.numpy()) if device=='cpu' else np.squeeze(correct_tensor.cpu().n
            num_correct += np.sum(correct)

    pred_list = [a.squeeze().tolist() for a in pred_list]
    print(confusion_matrix(true_list, pred_list))
    print()
    print(classification_report(true_list, pred_list))
    print()
    print(f"Accuracy {accuracy_score(true_list, pred_list):.2%}")

    # Output average test loss
    print("Test loss: {:.3f}".format(np.mean(test_losses)))

    # Output average accuracy
    test_acc = num_correct/len(test_loader.dataset)
    print("Test accuracy: {:.3f}".format(test_acc))

rnn_test(test_loader)
```

```
[[31  0  0  0  0  1]
 [ 1 33  0  0  4  1]
 [ 2  0 39  1  0  1]
 [ 0  1  0 16  2  1]
 [ 4  0  0  1 32  0]
 [ 4  0  2  2  3 30]]

              precision    recall  f1-score   support

           0       0.74      0.97      0.84        32
           1       0.97      0.85      0.90        39
           2       0.95      0.91      0.93        43
           3       0.80      0.80      0.80        20
           4       0.78      0.86      0.82        37
           5       0.88      0.73      0.80        41

    accuracy                           0.85       212
   macro avg       0.85      0.85      0.85       212
weighted avg       0.87      0.85      0.85       212


Accuracy 85.38%
Test loss: 0.513
Test accuracy: 0.854
```