Drew Lickman

CSCI 4820-001

Project #4

Due 10/30/24

AI Disclaimer: A.I. Disclaimer: Work for this assignment was completed with the aid of artificial intelligence tools and comprehensive documentation of the names of, input provided to, and output obtained from, these tools is included as part of my assignment submission.

Sentiment Analysis with Recurrent Neural Network (RNN Unit)

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

In [1]:
```
# Adjustable variables all in one place
TAMU                                    = True  # determines if code is being run locally or on TAMU
max_len                                 = 512   # default: 512, best: 512
trainSplitPercent               = 0.95  # default: 0.5, best: 0.99
validationSplitPercent  = 0.5   # default: 0.5, best: 0.5
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 de
padding                                 = "avg" # Default: "max", best: "avg"
```

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
if TAMU:
        DATADIR = "/scratch/user/u.sb157846/data/datasets/cb_speeches/"
else:
        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.

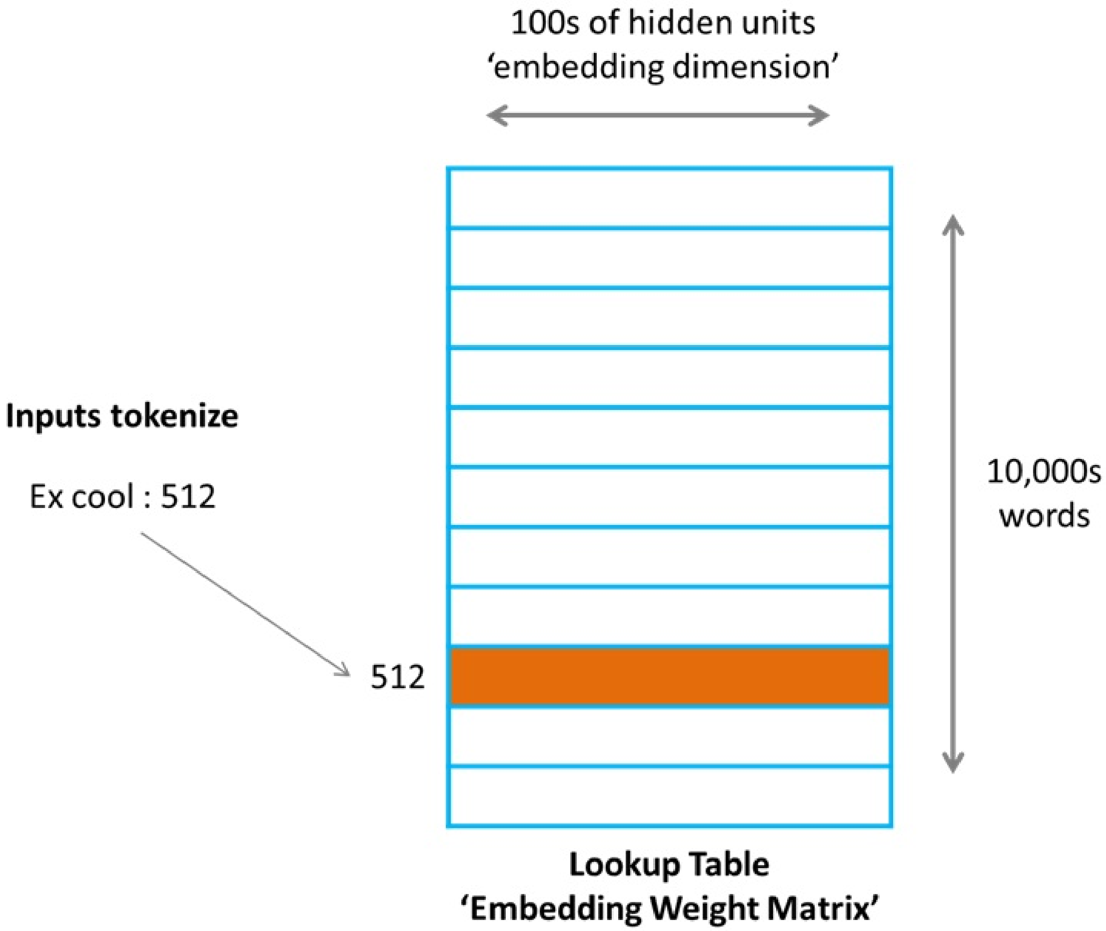
> Set the directory location of vectors in the `EMB_DIR` variable as shown below.

In [7]:
```
# Pre-trained embedding file location
if TAMU:
        EMB_DIR = "/scratch/user/u.sb157846/data/word_vectors/GoogleNews-vectors-negative300-SLIM/"
else:
        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)
```

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]:
```python
# 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

# Debug
# 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]:
```python
# This function pads all speeches
def pad_input(tokenized_input, max_len):
        # Max padding:
        if padding == "max":
                # 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)
        # Avg padding:
        else: #padding == "avg":
                # Calculate average length of speeches
                avg_len = int(np.mean([len(speech) for speech in tokenized_input]))

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

                # Go through each speech and copy the indices into the features list
                for i, row in enumerate(tokenized_input):
                        if len(row) <= avg_len:
                                # Pad shorter speeches
                                features[i, -len(row):] = np.array(row)
                        else:
                                # Truncate longer speeches to average length
                                features[i, :] = np.array(row[:avg_len])

        return features
```

In [14]:
```python
#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) * trainSplitPercent)

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) * validationSplitPercent) # Default 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:              (4028, 507)
Validation set:         (106, 507)
Test set:               (106, 507)
```

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
# Had to convert to .long() type so the code would function
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())

# 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__()

                # 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
```

```python
        elif rtype == 'lstm':
            self.rnn = nn.LSTM(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional, dr
        else:
            self.rnn = nn.RNN(embedding_dim, hidden_dim, n_layers, bidirectional=bidirectional, drop

        # ReLU Activation Layer
        self.relu = nn.ReLU()

        # First fully connected layer
        self.fc = nn.Linear(hidden_dim, hidden_dim)

        # Adding a second FC or ReLU layer was not beneficial
        # Second fully connected layer
        #self.fc = nn.Linear(hidden_dim, output_dim)

        # Secondary ReLU Activation Layer
        #self.relu = nn.ReLU()

        # 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)

        # Handle different hidden state formats for LSTM vs GRU/RNN
        if isinstance(hidden, tuple):  # LSTM case
            hidden_state = hidden[0][-1]  # Get just the hidden state, ignore cell state
        else:  # GRU/RNN case
            hidden_state = hidden[-1]

        # Apply dropout if multiple layers
        if n_layers > 1:
            hidden_state = self.dropout(hidden_state)

        # Then the fully-connected layer
        logit = self.fc(hidden_state)


        return logit
```

Model Parameters

In [18]:

```python
# 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]:

```python
# 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 

# 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
```

```
/opt/conda/lib/python3.11/site-packages/torch/nn/modules/rnn.py:82: UserWarning: dropout option adds dropout aft
er all but last recurrent layer, so non-zero dropout expects num_layers greater than 1, but got dropout=0.2 and
num_layers=1
  warnings.warn("dropout option adds dropout after all but last "
```

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 # Default: 5e-4, best: 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
            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]:
```
# Clear GPU memory before taining
if device == 'cuda':
    torch.cuda.empty_cache()
```

In [25]:
```
# 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: 1.749536... Val Loss: 1.771475
Epoch: 1/5... Step: 200... Train Loss: 1.898388... Val Loss: 1.703583
Epoch: 1/5... Step: 300... Train Loss: 1.419356... Val Loss: 1.610656
Epoch: 1/5... Step: 400... Train Loss: 1.334938... Val Loss: 1.555765
Epoch: 1/5... Step: 500... Train Loss: 1.333096... Val Loss: 1.517664
Epoch: 1/5... Step: 600... Train Loss: 2.037431... Val Loss: 1.441092
Epoch: 1/5... Step: 700... Train Loss: 1.172251... Val Loss: 1.402166
Epoch: 1/5... Step: 800... Train Loss: 0.868873... Val Loss: 1.358337
Epoch: 1/5... Step: 900... Train Loss: 2.123123... Val Loss: 1.293456
Epoch: 1/5... Step: 1000... Train Loss: 0.370663... Val Loss: 1.290338
Epoch: 2/5... Step: 1100... Train Loss: 0.751328... Val Loss: 1.029060
Epoch: 2/5... Step: 1200... Train Loss: 0.357092... Val Loss: 1.050985
Epoch: 2/5... Step: 1300... Train Loss: 1.828427... Val Loss: 0.885065
Epoch: 2/5... Step: 1400... Train Loss: 0.106139... Val Loss: 0.729005
Epoch: 2/5... Step: 1500... Train Loss: 0.412039... Val Loss: 0.828968
Epoch: 2/5... Step: 1600... Train Loss: 0.590250... Val Loss: 0.644859
Epoch: 2/5... Step: 1700... Train Loss: 0.083381... Val Loss: 0.581799
Epoch: 2/5... Step: 1800... Train Loss: 0.098366... Val Loss: 0.578468
Epoch: 2/5... Step: 1900... Train Loss: 0.471938... Val Loss: 0.461524
Epoch: 2/5... Step: 2000... Train Loss: 0.567279... Val Loss: 0.462718
Epoch: 3/5... Step: 2100... Train Loss: 0.129246... Val Loss: 0.378843
Epoch: 3/5... Step: 2200... Train Loss: 0.410881... Val Loss: 0.347036
Epoch: 3/5... Step: 2300... Train Loss: 0.018055... Val Loss: 0.382884
Epoch: 3/5... Step: 2400... Train Loss: 0.297949... Val Loss: 0.338478
Epoch: 3/5... Step: 2500... Train Loss: 0.496206... Val Loss: 0.383975
Epoch: 3/5... Step: 2600... Train Loss: 0.053829... Val Loss: 0.398016
Epoch: 3/5... Step: 2700... Train Loss: 0.151418... Val Loss: 0.232856
Epoch: 3/5... Step: 2800... Train Loss: 0.007825... Val Loss: 0.374497
Epoch: 3/5... Step: 2900... Train Loss: 0.013840... Val Loss: 0.290463
Epoch: 3/5... Step: 3000... Train Loss: 0.032089... Val Loss: 0.268147
Epoch: 4/5... Step: 3100... Train Loss: 0.041566... Val Loss: 0.276795
Epoch: 4/5... Step: 3200... Train Loss: 0.012599... Val Loss: 0.381343
Epoch: 4/5... Step: 3300... Train Loss: 0.008384... Val Loss: 0.266132
Epoch: 4/5... Step: 3400... Train Loss: 0.104277... Val Loss: 0.279654
Epoch: 4/5... Step: 3500... Train Loss: 0.010183... Val Loss: 0.436247
Epoch: 4/5... Step: 3600... Train Loss: 0.011259... Val Loss: 0.273775
Epoch: 4/5... Step: 3700... Train Loss: 0.015133... Val Loss: 0.386960
Epoch: 4/5... Step: 3800... Train Loss: 0.962769... Val Loss: 0.399231
Epoch: 4/5... Step: 3900... Train Loss: 0.008216... Val Loss: 0.273092
Epoch: 4/5... Step: 4000... Train Loss: 0.256409... Val Loss: 0.281134
Epoch: 5/5... Step: 4100... Train Loss: 0.004281... Val Loss: 0.190030
Epoch: 5/5... Step: 4200... Train Loss: 0.009098... Val Loss: 0.196764
Epoch: 5/5... Step: 4300... Train Loss: 0.004270... Val Loss: 0.327351
Epoch: 5/5... Step: 4400... Train Loss: 0.004430... Val Loss: 0.319841
Epoch: 5/5... Step: 4500... Train Loss: 0.006259... Val Loss: 0.349054
Epoch: 5/5... Step: 4600... Train Loss: 0.027720... Val Loss: 0.432380
Epoch: 5/5... Step: 4700... Train Loss: 0.007344... Val Loss: 0.273633
Epoch: 5/5... Step: 4800... Train Loss: 0.021657... Val Loss: 0.280670
Epoch: 5/5... Step: 4900... Train Loss: 0.006826... Val Loss: 0.251449
Epoch: 5/5... Step: 5000... Train Loss: 0.017747... Val Loss: 0.307524
```

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():
```

```python
        # 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())

            # 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().
            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)
```

```
[[16  0  0  0  1  0]
 [ 1 20  1  1  0  0]
 [ 0  0 22  0  1  0]
 [ 0  0  0  9  0  0]
 [ 0  0  0  1 18  0]
 [ 1  0  0  0  0 14]]

              precision    recall  f1-score   support

           0       0.89      0.94      0.91        17
           1       1.00      0.87      0.93        23
           2       0.96      0.96      0.96        23
           3       0.82      1.00      0.90         9
           4       0.90      0.95      0.92        19
           5       1.00      0.93      0.97        15

    accuracy                           0.93       106
   macro avg       0.93      0.94      0.93       106
weighted avg       0.94      0.93      0.93       106


Accuracy 93.40%
Test loss: 0.201
Test accuracy: 0.934
```