1.1 A Baseline Neural Network Tagger (40 points)

Train a feed-forward neural network classifier to predict the POS tag of a word in its context. The input should be the word embedding for the center word concatenated with the word embeddings for words in a context window. We'll define a context window as the sequence of words containing w words to either side of the center word and including the center word itself, so the context window contains 1 + 2w words in total. For example, if w = 1 and the word embedding dimensionality is d, the total dimensionality of the input will be 3d. For words near the sentence boundaries, pad the sentence with beginning-of-sentence and end-of-sentence characters. The word embeddings should be randomly initialized and learned along with all other parameters in the model.

functional architecture: The input is the concatenation of word embeddings in the context window, with the word to be tagged in the center. Use a single hidden layer of width 128 with a tanh nonlinearity. The hidden layer should then be fed to an affine transformation which will produce scores for all possible POS tags. Use a softmax transformation on the scores to produce a probability distribution over tags.

learning: Use log loss as the objective function (log loss is often called "cross entropy" or "negative log-likelihood" when training neural networks, so those terms may be useful when searching for the right loss function in toolkits). Use SGD or any other optimizer you wish. Toolkits typically have many optimizers already implemented.

initialization: Randomly initialize all parameters, including word embeddings, and train them. Note that embeddings for words that only appear in DEV/DEVTEST will not be trained at all. So, you need to be careful about how those embeddings are set in order to get good results. We suggest an initialization range of -0.01 to 0.01 for all word embedding parameters. (You could alternatively try setting embeddings for unknown words to all zeros or try learning an unknown word embedding during training.) Train on TRAIN, perform early stopping and preliminary testing on DEV, and report your final tagging accuracy on DEVTEST. Report results with both w = 0 and w = 1. Submit your code.

Notes: With w = 0, I was seeing a best DEV accuracy of 77-78% and with w = 1 it improved to 80-81%. I set the size (dimensionality) of word embeddings to 50, and used SGD with a fixed step size of 0.02 and each mini-batch contained one word to be tagged. I trained for 10 epochs and evaluated on DEV once per epoch. It took approximately 10 seconds per epoch using PyTorch on a 3.3 GHz Intel Core i5.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Here Initializing the embeddings randomly within range (-0.01, 0.01)
def initialize_embeddings(vocab_size, embedding_dim):
    embeddings = nn.Embedding(vocab_size, embedding_dim, padding_idx=0)
    embeddings.weight.data.uniform_(-0.01, 0.01)
    return embeddings

# Here Loading the data from the files
def load_data(file_path):
    data = []
    with open(file_path, 'r') as f:
        tweet = []
        for line in f:
            if line.strip():
                word, pos = line.strip().split('\t')
                tweet.append((word, pos))
            else:
                if tweet:
                    data.append(tweet)
                tweet = []
        if tweet:
            data.append(tweet)
    return data

# Here indexing the different POS tags
def pos_to_index():
    pos_tags = ["N", "O", "S", "L", "^", "Z", "M", "V", "A", "R", "!", "D", "P", "&", "T", "X", "Y", "#", "@", "~", "U", "E", "$", ",",
    return {tag: idx for idx, tag in enumerate(pos_tags)}

# Here Converting words to indices
def get_word_index(word, word_to_idx):
    return word_to_idx.get(word, word_to_idx['UUUNKKK'])

# Code for getting context window for a base word in a tweet
def get_context(tweet, index, window_size=1):
    padded_tweet = ['<s>'] * window_size + [word for word, pos in tweet] + ['</s>'] * window_size
    context_start = index
    context_end = index + 2 * window_size + 1
    return padded_tweet[context_start:context_end]

# Code for creating input vectors by concatenating word indices in the context window
def get_input_vector(context, word_to_idx):
    return torch.tensor([get_word_index(word, word_to_idx) for word in context], dtype=torch.long)
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Define the model class for Feed Forward N.N.
class POSModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, window_size):
        super(POSModel, self).__init__()
        self.embedding = initialize_embeddings(vocab_size, embedding_dim)
        input_dim = embedding_dim * (2 * window_size + 1)
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.tanh = nn.Tanh()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        embeds = self.embedding(x)
        x = embeds.view(1, -1)
        x = self.tanh(self.fc1(x))
        scores = self.fc2(x)
        return scores
```

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Code for training the Model
def train_model(model, train_data, dev_data, word_to_idx, pos_dict, window_size, epochs=10, lr=0.02):
    optimizer = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        total_loss = 0
        model.train()

        for tweet in train_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = get_input_vector(context, word_to_idx)
                target = torch.tensor([pos_dict[pos]], dtype=torch.long)

                optimizer.zero_grad()
                output = model(input_vector)

                loss = criterion(output, target)

                loss.backward()
                optimizer.step()
                total_loss += loss.item()

        print(f'Epoch {epoch+1}/{epochs}, Loss: {total_loss:.4f}')
        evaluate_model(model, dev_data, word_to_idx, pos_dict, window_size, 4821)

# Code for evaluating the model
def evaluate_model(model, test_data, word_to_idx, pos_dict, window_size, expected_count):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for tweet in test_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = get_input_vector(context, word_to_idx)
                target = pos_dict[pos]

                output = model(input_vector)
                predicted = torch.argmax(output).item()
                if predicted == target:
                    correct += 1
                total += 1

    assert total == expected_count, f"Expected {expected_count} predictions, but got {total}"

    accuracy = correct / total
    print(f'Accuracy: {accuracy:.4f}')
    return accuracy
```

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

```
import torch.optim as optim
import numpy as np

# A utility function to read files and setting dimensions and then calling the model to run on this data
def result(w):
    vocab = set()
    train_data = load_data('/twpos-train.tsv')
    dev_data = load_data('/twpos-dev.tsv')
    devtest_data = load_data('/twpos-devtest.tsv')

    for tweet in train_data:
        for word, _ in tweet:
            vocab.add(word)

    word_to_idx = {word: idx for idx, word in enumerate(vocab, start=1)}
    word_to_idx['<s>'] = 0
    word_to_idx['</s>'] = 0
    word_to_idx['UUUNKKK'] = len(vocab) + 1


    pos_dict = pos_to_index()


    vocab_size = len(word_to_idx)
    embedding_dim = 50
    hidden_dim = 128
    output_dim = len(pos_dict)
    window_size = w


    model = POSModel(vocab_size, embedding_dim, hidden_dim, output_dim, window_size)


    model.embedding.weight.data[word_to_idx['UUUNKKK']] = 0
    model.embedding.weight.data[word_to_idx['<s>']] = 0
    model.embedding.weight.data[word_to_idx['</s>']] = 0

    # Code to call Train and evaluate function for the model
    train_model(model, train_data, dev_data, word_to_idx, pos_dict, window_size, epochs=10)
    print("Evaluating on DEVTEST set...")
    evaluate_model(model, devtest_data, word_to_idx, pos_dict, window_size, 4639)


import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np


def main():
    context_window = [0, 1]
    for w in context_window:
        result(w)

if __name__ == "__main__":
    main()
```

```
Epoch 1/10, Loss: 26006.6902
Accuracy: 0.7094
Epoch 2/10, Loss: 12139.1604
Accuracy: 0.7693
Epoch 3/10, Loss: 7735.5150
Accuracy: 0.7725
Epoch 4/10, Loss: 6091.0452
Accuracy: 0.7737
Epoch 5/10, Loss: 5323.4264
Accuracy: 0.7743
Epoch 6/10, Loss: 4886.4637
Accuracy: 0.7729
Epoch 7/10, Loss: 4596.4812
Accuracy: 0.7727
Epoch 8/10, Loss: 4382.1922
Accuracy: 0.7727
Epoch 9/10, Loss: 4208.7536
Accuracy: 0.7731
Epoch 10/10, Loss: 4059.0412
Accuracy: 0.7745
Evaluating on DEVTEST set...
Accuracy: 0.7894
Epoch 1/10, Loss: 26248.9624
Accuracy: 0.7264
Epoch 2/10, Loss: 10571.9282
Accuracy: 0.7930
Epoch 3/10, Loss: 5715.2257
Accuracy: 0.7953
```

```
Epoch 4/10, Loss: 3615.7730
Accuracy: 0.7955
Epoch 5/10, Loss: 2390.1751
Accuracy: 0.7944
Epoch 6/10, Loss: 1653.6959
Accuracy: 0.7994
Epoch 7/10, Loss: 1250.9881
Accuracy: 0.8007
Epoch 8/10, Loss: 946.4691
Accuracy: 0.8017
Epoch 9/10, Loss: 769.2347
Accuracy: 0.7957
Epoch 10/10, Loss: 602.4454
Accuracy: 0.7982
Evaluating on DEVTEST set...
Accuracy: 0.8176
```

1.2 Feature Engineering (15 points)

Add features to the model by concatenating your own feature function outputs to the word embedding concatenation used above. Define feature functions based on looking at the training data, based on looking at the errors your tagger makes on DEV, or simply based on your intuitions about the task. For example, you could add binary features if the center word contains certain special characters or capitalization patterns, a feature that returns the number of characters in the center word, features for particular prefixes, suffixes, and other character patterns in the center word, etc. These sorts of features could also be defined for context words. You may find it helpful to use the orig-* files when computing features. (You will probably still want to use the twpos-* files for the word embeddings, though.) Develop and experiment with features and describe your results. You should be able to improve upon the accuracies you were seeing in Section 1.1.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np


# Code for feature extraction for a single word
def extract_features(word):

    capitalization = 1 if word[0].isupper() else 0

    special_char = 1 if any(not char.isalnum() for char in word) else 0

    word_length = len(word) / 10.0
    prefix = word[:3]
    suffix = word[-3:]
    prefix_idx = sum(ord(c) for c in prefix) % 1000
    suffix_idx = sum(ord(c) for c in suffix) % 1000
    prefix_norm = prefix_idx / 1000.0
    suffix_norm = suffix_idx / 1000.0

    return torch.tensor([capitalization, special_char, word_length, prefix_norm, suffix_norm], dtype=torch.float)

# Code to create create input vector by concatenating word embeddings with features
def get_input_vector(context, word_to_idx, embedding_layer):
    context_embeddings = []
    context_features = []

    for word in context:
        word_idx = get_word_index(word, word_to_idx)
        word_embedding = embedding_layer(torch.tensor([word_idx], dtype=torch.long)).squeeze(0)

        word_features = extract_features(word)

        combined_vector = torch.cat((word_embedding, word_features), dim=0)
        context_embeddings.append(combined_vector)

    return torch.cat(context_embeddings).view(1, -1)

# Define the model class for Feed Forward N.N. by adding feature dimension too
class POSModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, feature_dim, hidden_dim, output_dim, window_size):
        super(POSModel, self).__init__()
        self.embedding = initialize_embeddings(vocab_size, embedding_dim)

        input_dim = (embedding_dim + feature_dim) * (2 * window_size + 1)
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.tanh = nn.Tanh()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.tanh(self.fc1(x))
        scores = self.fc2(x)
```

```
        return scores

# Code for training the Model
def train_model(model, train_data, dev_data, word_to_idx, pos_dict, window_size, epochs=10, lr=0.02):
    optimizer = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        total_loss = 0
        model.train()

        for tweet in train_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = get_input_vector(context, word_to_idx, model.embedding)
                target = torch.tensor([pos_dict[pos]], dtype=torch.long)

                optimizer.zero_grad()
                output = model(input_vector)

                loss = criterion(output, target)

                loss.backward()
                optimizer.step()
                total_loss += loss.item()

        print(f'Epoch {epoch+1}/{epochs}, Loss: {total_loss:.4f}')
        evaluate_model(model, dev_data, word_to_idx, pos_dict, window_size, 4821)


# Code for evaluating the Model
def evaluate_model(model, test_data, word_to_idx, pos_dict, window_size, expected_count):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for tweet in test_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = get_input_vector(context, word_to_idx, model.embedding)
                target = pos_dict[pos]

                output = model(input_vector)
                predicted = torch.argmax(output).item()
                if predicted == target:
                    correct += 1
                total += 1

    assert total == expected_count, f"Expected {expected_count} predictions, but got {total}"

    accuracy = correct / total
    print(f'Accuracy: {accuracy:.4f}')
    return accuracy

# A utility function to read files and setting dimensions and then calling the model to run on this data
def result(w):
    vocab = set()
    train_data = load_data('/twpos-train.tsv')
    dev_data = load_data('/twpos-dev.tsv')
    devtest_data = load_data('/twpos-devtest.tsv')

    for tweet in train_data:
        for word, _ in tweet:
            vocab.add(word)

    word_to_idx = {word: idx for idx, word in enumerate(vocab, start=1)}
    word_to_idx['<s>'] = 0
    word_to_idx['</s>'] = 0
    word_to_idx['UUUNKKK'] = len(vocab) + 1

    pos_dict = pos_to_index()

    vocab_size = len(word_to_idx)
    embedding_dim = 50
    feature_dim = 5
    hidden_dim = 128
    output_dim = len(pos_dict)
    window_size = w

    model = POSModel(vocab_size, embedding_dim, feature_dim, hidden_dim, output_dim, window_size)
```

```
    model.embedding.weight.data[word_to_idx['UUUNKKK']] = 0
    model.embedding.weight.data[word_to_idx['<s>']] = 0

    # Code to call Train and evaluate function for the model
    train_model(model, train_data, dev_data, word_to_idx, pos_dict, window_size, epochs=10)
    print("Evaluating on DEVTEST set...")
    evaluate_model(model, devtest_data, word_to_idx, pos_dict, window_size, 4639)

def main():
    context_window = [0, 1]
    for w in context_window:
        result(w)

if __name__ == "__main__":
    main()
```

```
Epoch 1/10, Loss: 22638.2898
Accuracy: 0.7185
Epoch 2/10, Loss: 11121.6414
Accuracy: 0.7741
Epoch 3/10, Loss: 7399.7439
Accuracy: 0.7770
Epoch 4/10, Loss: 5998.6403
Accuracy: 0.7795
Epoch 5/10, Loss: 5255.0829
Accuracy: 0.7793
Epoch 6/10, Loss: 4798.8504
Accuracy: 0.7785
Epoch 7/10, Loss: 4493.9533
Accuracy: 0.7785
Epoch 8/10, Loss: 4269.9913
Accuracy: 0.7791
Epoch 9/10, Loss: 4094.4390
Accuracy: 0.7787
Epoch 10/10, Loss: 3947.8282
Accuracy: 0.7787
Evaluating on DEVTEST set...
Accuracy: 0.7948
Epoch 1/10, Loss: 21479.1382
Accuracy: 0.7434
Epoch 2/10, Loss: 9435.1631
Accuracy: 0.8048
Epoch 3/10, Loss: 5385.6582
Accuracy: 0.8079
Epoch 4/10, Loss: 3513.5458
Accuracy: 0.8119
Epoch 5/10, Loss: 2397.7036
Accuracy: 0.8191
Epoch 6/10, Loss: 1657.3680
Accuracy: 0.8189
Epoch 7/10, Loss: 1181.6974
Accuracy: 0.8177
Epoch 8/10, Loss: 904.4941
Accuracy: 0.8166
Epoch 9/10, Loss: 724.6088
Accuracy: 0.8171
Epoch 10/10, Loss: 609.8508
Accuracy: 0.8154
Evaluating on DEVTEST set...
Accuracy: 0.8263
```

1.3 Pretrained Embeddings (10 points) Initialize your word embeddings using the pretrained embeddings from twitter-embeddings.txt. For words in the tagging datasets that are not in the pretrained embeddings, use the unknown word embedding (i.e., the embedding for the word “UUUNKKK”). The pretrained embeddings contain an embedding for the sentence end symbol , but not the sentence start symbol.

1.3.1 Experiment with updating (fine-tuning) the pretrained embeddings for both w = 0 and w = 1 and report your results. You should see improvements over the randomly-initialized word embedding experiments from Section 1.1.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import defaultdict

# Code to load word embeddings from file containing pretrained embeddings
def load_word_embeddings(twitter_file):
    embeddings = {}
    with open(twitter_file, 'r') as f:
        for line in f:
            values = line.strip().split()
            word = values[0]
            word_vec = np.asarray(values[1:], dtype='float32')
            embeddings[word] = word_vec
```

```
    embeddings['<s>'] = np.zeros(50)
    embeddings['</s>'] = np.zeros(50)
    return embeddings

# COde for getting embedding vector for a word
def get_embedding_vector(word, embeddings):
    return torch.tensor(embeddings.get(word, embeddings['UUUNKKK']), dtype=torch.float32)

# Code for creating input vector by concatenating word embeddings in the context window
def get_input_vector(context, embeddings):
    return np.concatenate([get_embedding_vector(word, embeddings) for word in context])


import torch
import torch.nn as nn
import torch.optim as optim

# Feed Forward N.N. Model for Pretrained Embeddings
class FeedforwardNeuralNetModel(nn.Module):
     def __init__(self, embedding_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        self.fc1 = nn.Linear(embedding_dim, hidden_dim)
        self.tanh = nn.Tanh()
        self.fc2 = nn.Linear(hidden_dim, output_dim)

     def forward(self, x):
        x = self.tanh(self.fc1(x))
        scores = self.fc2(x)
        return scores


import torch
import torch.nn as nn
import torch.optim as optim

# Code for training the Model
def train_model(model, train_data, dev_data, embeddings, pos_dict, window_size, epochs=10, lr=0.02):
    optimizer = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        total_loss = 0
        model.train()

        for tweet in train_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = torch.tensor(get_input_vector(context, embeddings), dtype=torch.float32)
                target = torch.tensor([pos_dict[pos]], dtype=torch.long)

                optimizer.zero_grad()
                output = model(input_vector)
                loss = criterion(output.unsqueeze(0), target)

                loss.backward()
                optimizer.step()
                total_loss += loss.item()

        print(f'Epoch {epoch+1}/{epochs}, Loss: {total_loss:.4f}')
        evaluate_model(model, dev_data, embeddings, pos_dict, window_size, 4821)

# Code for evaluating the Model
def evaluate_model(model, test_data, embeddings, pos_dict, window_size, expected_count):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for tweet in test_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = torch.tensor(get_input_vector(context, embeddings), dtype=torch.float32)
                target = pos_dict[pos]

                output = model(input_vector)
                predicted = torch.argmax(output).item()
                if predicted == target:
                    correct += 1
                total += 1

    assert total == expected_count, f"Expected {expected_count} predictions, but got {total}"

    accuracy = correct / total
```

```
        print(f'Accuracy: {accuracy:.4f}')
        return accuracy


import torch
import torch.nn as nn
import torch.optim as optim

# Utility Funtion
def result(w):
    embeddings = load_word_embeddings('/twitter-embeddings.txt')

    train_data = load_data('/twpos-train.tsv')
    dev_data = load_data('/twpos-dev.tsv')
    devtest_data = load_data('/twpos-devtest.tsv')

    pos_dict = pos_to_index()

    input_dim = 50 * (2*w + 1)
    hidden_dim = 128
    output_dim = len(pos_dict)
    window_size = w

    model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

    train_model(model, train_data, dev_data, embeddings, pos_dict, window_size, epochs=10)
    print("Evaluating on DEVTEST set...")
    evaluate_model(model, devtest_data, embeddings, pos_dict, window_size, 4639)


def main():
    context_window = [0, 1]
    for w in context_window:
        result(w)

if __name__ == "__main__":
    main()
```

```
Epoch 1/10, Loss: 14436.3236
Accuracy: 0.8102
Epoch 2/10, Loss: 10423.5516
Accuracy: 0.8168
Epoch 3/10, Loss: 10012.9584
Accuracy: 0.8177
Epoch 4/10, Loss: 9749.0976
Accuracy: 0.8195
Epoch 5/10, Loss: 9518.9686
Accuracy: 0.8200
Epoch 6/10, Loss: 9292.5743
Accuracy: 0.8245
Epoch 7/10, Loss: 9071.1312
Accuracy: 0.8247
Epoch 8/10, Loss: 8866.4081
Accuracy: 0.8224
Epoch 9/10, Loss: 8685.2950
Accuracy: 0.8222
Epoch 10/10, Loss: 8527.8389
Accuracy: 0.8245
Evaluating on DEVTEST set...
Accuracy: 0.8230
Epoch 1/10, Loss: 13674.0126
Accuracy: 0.8243
Epoch 2/10, Loss: 8810.1603
Accuracy: 0.8345
Epoch 3/10, Loss: 8107.2609
Accuracy: 0.8351
Epoch 4/10, Loss: 7649.6606
Accuracy: 0.8355
Epoch 5/10, Loss: 7267.2322
Accuracy: 0.8365
Epoch 6/10, Loss: 6925.3962
Accuracy: 0.8403
Epoch 7/10, Loss: 6609.0819
Accuracy: 0.8428
Epoch 8/10, Loss: 6309.7090
Accuracy: 0.8471
Epoch 9/10, Loss: 6020.9735
Accuracy: 0.8488
Epoch 10/10, Loss: 5737.4976
Accuracy: 0.8490
Evaluating on DEVTEST set...
Accuracy: 0.8534
```

1.3.2

With w = 1, empirically compare updating the pretrained word embeddings during training and keeping them fixed.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Code for training the model by keeping embeddings fixed and updating
def train_model(model, train_data, dev_data, embeddings, pos_dict, window_size, epochs=10, lr=0.02, update_embeddings=True):
    optimizer = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        total_loss = 0
        model.train()

        for tweet in train_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = torch.tensor(get_input_vector(context, embeddings), dtype=torch.float32)

                if update_embeddings:
                    input_vector.requires_grad = True
                else:
                    input_vector.requires_grad = False

                target = torch.tensor([pos_dict[pos]], dtype=torch.long)

                optimizer.zero_grad()
                output = model(input_vector)

                loss = criterion(output.unsqueeze(0), target)

                loss.backward()
                optimizer.step()
                total_loss += loss.item()

        print(f'Epoch {epoch + 1}/{epochs}, Loss: {total_loss:.4f}')
        evaluate_model(model, dev_data, embeddings, pos_dict, window_size, 4821)


# Utility Function
def result(w, update_embeddings):
    embeddings = load_word_embeddings('/twitter-embeddings.txt')

    train_data = load_data('/twpos-train.tsv')
    dev_data = load_data('/twpos-dev.tsv')
    devtest_data = load_data('/twpos-devtest.tsv')

    pos_dict = pos_to_index()

    input_dim = 50 * (2 * w + 1)
    hidden_dim = 128
    output_dim = len(pos_dict)

    model = FeedforwardNeuralNetModel(input_dim, hidden_dim, output_dim)

    print(f"\nTraining with {'updating' if update_embeddings else 'fixed'} embeddings:")
    train_model(model, train_data, dev_data, embeddings, pos_dict, w, epochs=10, update_embeddings=update_embeddings)
    print("Evaluating on DEVTEST set...")
    evaluate_model(model, devtest_data, embeddings, pos_dict, w, 4639)

def main():
    context_window = [1]
    for w in context_window:
        result(w, update_embeddings=True)
        result(w, update_embeddings=False)

if __name__ == "__main__":
    main()
```

```
Training with updating embeddings:
Epoch 1/10, Loss: 13640.3876
Accuracy: 0.8251
Epoch 2/10, Loss: 8768.6606
Accuracy: 0.8330
Epoch 3/10, Loss: 8081.7865
Accuracy: 0.8359
Epoch 4/10, Loss: 7640.4794
Accuracy: 0.8376
Epoch 5/10, Loss: 7272.5088
Accuracy: 0.8365
Epoch 6/10, Loss: 6929.9067
```

```
Accuracy: 0.8390
Epoch 7/10, Loss: 6594.4760
Accuracy: 0.8411
Epoch 8/10, Loss: 6262.6499
Accuracy: 0.8434
Epoch 9/10, Loss: 5936.6733
Accuracy: 0.8477
Epoch 10/10, Loss: 5619.5313
Accuracy: 0.8494
Evaluating on DEVTEST set...
Accuracy: 0.8467

Training with fixed embeddings:
Epoch 1/10, Loss: 13596.9028
Accuracy: 0.8251
Epoch 2/10, Loss: 8778.0269
Accuracy: 0.8324
Epoch 3/10, Loss: 8111.7985
Accuracy: 0.8343
Epoch 4/10, Loss: 7687.1712
Accuracy: 0.8343
Epoch 5/10, Loss: 7327.7407
Accuracy: 0.8357
Epoch 6/10, Loss: 6988.6198
Accuracy: 0.8365
Epoch 7/10, Loss: 6654.0582
Accuracy: 0.8401
Epoch 8/10, Loss: 6325.0811
Accuracy: 0.8446
Epoch 9/10, Loss: 6003.6220
Accuracy: 0.8471
Epoch 10/10, Loss: 5690.7811
Accuracy: 0.8463
Evaluating on DEVTEST set...
Accuracy: 0.8506
```

1.3.3

Combine your features from Section 1.2 with the use of pretrained embeddings. Do the features still help?

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
from collections import defaultdict

# Code to get embedding vector for a word
def get_embedding_vector(word, embeddings):
    return torch.tensor(embeddings.get(word, embeddings['UUUNKKK']), dtype=torch.float32)

# Code to normalize features to be in the same range as word embeddings
def extract_features(word):
    features = []
    features.append(1 if word[0].isupper() else 0)

    features.append(len(word) / 10.0)
    features.append(1 if "@" in word or "#" in word else 0)
    prefix = word[:2]
    suffix = word[-2:]
    features.append((hash(prefix) % 100) / 100.0)
    features.append((hash(suffix) % 100) / 100.0)

    return np.array(features, dtype='float32')

# Code to create input vector by concatenating word embeddings and normalized additional features
def get_input_vector(context, embeddings):
    embedding_vector = np.concatenate([get_embedding_vector(word, embeddings) for word in context])
    features = extract_features(context[len(context)//2])
    return np.concatenate([embedding_vector, features])
```

```python
import torch
import torch.nn as nn
import torch.optim as optim

# Code to implement Model
class FeedforwardNeuralNetModel(nn.Module):
     def __init__(self, embedding_dim, feature_dim, hidden_dim, output_dim):
        super(FeedforwardNeuralNetModel, self).__init__()
        input_dim = embedding_dim + feature_dim
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.tanh = nn.Tanh()
        self.fc2 = nn.Linear(hidden_dim, output_dim)
```

```
    def forward(self, x):
        x = self.tanh(self.fc1(x))
        scores = self.fc2(x)
        return scores


# Utility Function
def result(w):
    embeddings = load_word_embeddings('/twitter-embeddings.txt')
    train_data = load_data('/twpos-train.tsv')
    dev_data = load_data('/twpos-dev.tsv')
    devtest_data = load_data('/twpos-devtest.tsv')

    pos_dict = pos_to_index()

    input_dim = 50 * (2*w + 1)
    feature_dim = 5
    hidden_dim = 128
    output_dim = len(pos_dict)
    window_size = w

    model = FeedforwardNeuralNetModel(input_dim, feature_dim, hidden_dim, output_dim)

    train_model(model, train_data, dev_data, embeddings, pos_dict, window_size, epochs=10)
    print("Evaluating on DEVTEST set...")
    evaluate_model(model, devtest_data, embeddings, pos_dict, window_size, 4639)


def main():
    context_window = [0, 1]
    for w in context_window:
        result(w)

if __name__ == "__main__":
    main()
```

```
Epoch 1/10, Loss: 13832.6978
Accuracy: 0.8197
Epoch 2/10, Loss: 9782.1717
Accuracy: 0.8227
Epoch 3/10, Loss: 9373.8821
Accuracy: 0.8243
Epoch 4/10, Loss: 9122.7273
Accuracy: 0.8256
Epoch 5/10, Loss: 8911.2973
Accuracy: 0.8278
Epoch 6/10, Loss: 8711.0557
Accuracy: 0.8270
Epoch 7/10, Loss: 8516.9677
Accuracy: 0.8283
Epoch 8/10, Loss: 8332.3968
Accuracy: 0.8305
Epoch 9/10, Loss: 8162.2448
Accuracy: 0.8314
Epoch 10/10, Loss: 8008.4303
Accuracy: 0.8349
Evaluating on DEVTEST set...
Accuracy: 0.8319
Epoch 1/10, Loss: 12976.2816
Accuracy: 0.8293
Epoch 2/10, Loss: 8230.7429
Accuracy: 0.8405
Epoch 3/10, Loss: 7572.0660
Accuracy: 0.8426
Epoch 4/10, Loss: 7149.6051
Accuracy: 0.8444
Epoch 5/10, Loss: 6799.0688
Accuracy: 0.8461
Epoch 6/10, Loss: 6477.5862
Accuracy: 0.8475
Epoch 7/10, Loss: 6168.9701
Accuracy: 0.8490
Epoch 8/10, Loss: 5866.2953
Accuracy: 0.8525
Epoch 9/10, Loss: 5566.7842
Accuracy: 0.8544
Epoch 10/10, Loss: 5270.8547
Accuracy: 0.8583
Evaluating on DEVTEST set...
Accuracy: 0.8629
```

1.4

Architecture Engineering (10 points) Using the best configuration from above, explore the space of neural architectures to see if you can improve your tagger further. Some suggestions are below: • Compare the use of 0, 1, and 2 hidden layers. For each number of hidden layers, try two different layer widths that differ by a factor of 2 (e.g., 256 and 512). • Keeping the number of layers and layer sizes fixed, experiment with different nonlinearities, e.g., identity (g(a) = a), tanh, ReLU, and logistic sigmoid. • Experiment with w = 2 and compare the results to w = 0 and 1.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Class for N.N. model for different sctivation Functions
class DynamicFeedforwardModel(nn.Module):
    def __init__(self, input_dim, hidden_layers, layer_widths, activation_function, output_dim):
        super(DynamicFeedforwardModel, self).__init__()

        # Initialize layers
        layers = []
        current_input_dim = input_dim

        for i in range(hidden_layers):
            layers.append(nn.Linear(current_input_dim, layer_widths[i]))
            if activation_function == 'tanh':
                layers.append(nn.Tanh())
            elif activation_function == 'relu':
                layers.append(nn.ReLU())
            elif activation_function == 'sigmoid':
                layers.append(nn.Sigmoid())
            elif activation_function == 'identity':
                layers.append(nn.Identity())
            current_input_dim = layer_widths[i]

        layers.append(nn.Linear(current_input_dim, output_dim))
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)

# Code for training the model
def train_model(model, train_data, dev_data, embeddings, pos_dict, window_size, epochs=10, lr=0.02, update_embeddings=True):
    optimizer = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for epoch in range(epochs):
        total_loss = 0
        model.train()

        for tweet in train_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = torch.tensor(get_input_vector(context, embeddings), dtype=torch.float32)

                if update_embeddings:
                    input_vector.requires_grad = True
                else:
                    input_vector.requires_grad = False

                target = torch.tensor([pos_dict[pos]], dtype=torch.long)

                optimizer.zero_grad()
                output = model(input_vector)
                loss = criterion(output.unsqueeze(0), target)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()

        print(f'Epoch {epoch + 1}/{epochs}, Loss: {total_loss:.4f}')
        evaluate_model(model, dev_data, embeddings, pos_dict, window_size, 4821)

# Code for evaluating the Model
def evaluate_model(model, test_data, embeddings, pos_dict, window_size, expected_count):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for tweet in test_data:
            for i, (word, pos) in enumerate(tweet):
                context = get_context(tweet, i, window_size)
                input_vector = torch.tensor(get_input_vector(context, embeddings), dtype=torch.float32)
                target = pos_dict[pos]
                output = model(input_vector)
                predicted = torch.argmax(output).item()
                if predicted == target:
```

```
                correct += 1
            total += 1

    assert total == expected_count, f"Expected {expected_count} predictions, but got {total}"

    accuracy = correct / total
    print(f'Accuracy: {accuracy:.4f}')
    return accuracy

# Function to run different experiments by changing the hidden layers and activation functions
def run_experiments():
    embeddings = load_word_embeddings('/twitter-embeddings.txt')
    train_data = load_data('/twpos-train.tsv')
    dev_data = load_data('/twpos-dev.tsv')
    devtest_data = load_data('/twpos-devtest.tsv')
    pos_dict = pos_to_index()

    window_sizes = [0, 1, 2]
    layer_combinations = [
        (0, [128]),  # 0 hidden layer
        (1, [128]),  # 1 hidden layer
        (1, [256]),  # 1 hidden layer
        (1, [512]),  # 1 hidden layer
        (2, [128, 256]),  # 2 hidden layers
        (2, [256, 512]),  # 2 hidden layers
    ]
    activations = ['tanh', 'relu', 'sigmoid', 'identity']
    epochs = 10

    for window_size in window_sizes:
        for hidden_layers, layer_widths in layer_combinations:
            for activation in activations:
                input_dim = 50 * (2 * window_size + 1)
                output_dim = len(pos_dict)

                print(f"\nRunning experiment with window_size={window_size}, hidden_layers={hidden_layers}, "
                      f"layer_widths={layer_widths}, activation={activation}")

                model = DynamicFeedforwardModel(input_dim, hidden_layers, layer_widths, activation, output_dim)
                train_model(model, train_data, dev_data, embeddings, pos_dict, window_size, epochs)

if __name__ == "__main__":
    run_experiments()
```

```
Running experiment with window_size=0, hidden_layers=0, layer_widths=[128], activation=tanh
Epoch 1/10, Loss: 20067.9755
Accuracy: 0.7764
Epoch 2/10, Loss: 12808.6664
Accuracy: 0.8075
Epoch 3/10, Loss: 11624.6174
Accuracy: 0.8129
Epoch 4/10, Loss: 11073.4463
Accuracy: 0.8175
Epoch 5/10, Loss: 10740.1414
Accuracy: 0.8181
Epoch 6/10, Loss: 10511.0815
Accuracy: 0.8214
Epoch 7/10, Loss: 10341.3145
Accuracy: 0.8210
Epoch 8/10, Loss: 10209.0107
Accuracy: 0.8214
Epoch 9/10, Loss: 10102.1276
Accuracy: 0.8212
Epoch 10/10, Loss: 10013.4223
Accuracy: 0.8239

Running experiment with window_size=0, hidden_layers=0, layer_widths=[128], activation=relu
Epoch 1/10, Loss: 20037.1366
Accuracy: 0.7766
Epoch 2/10, Loss: 12805.4272
Accuracy: 0.8073
Epoch 3/10, Loss: 11622.7861
Accuracy: 0.8127
Epoch 4/10, Loss: 11071.6904
Accuracy: 0.8179
Epoch 5/10, Loss: 10738.4446
Accuracy: 0.8179
Epoch 6/10, Loss: 10509.4469
Accuracy: 0.8206
Epoch 7/10, Loss: 10339.7156
Accuracy: 0.8208
Epoch 8/10, Loss: 10207.4179
Accuracy: 0.8214
Epoch 9/10, Loss: 10100.5181
Accuracy: 0.8214
```

```
Epoch 10/10, Loss: 10011.7867
Accuracy: 0.8243

Running experiment with window_size=0, hidden_layers=0, layer_widths=[128], activation=sigmoid
Epoch 1/10, Loss: 19985.5570
Accuracy: 0.7768
Epoch 2/10, Loss: 12802.8071
Accuracy: 0.8077
Epoch 3/10, Loss: 11622.1811
Accuracy: 0.8135
Epoch 4/10, Loss: 11072.5509
Accuracy: 0.8168
Epoch 5/10, Loss: 10739.9986
Accuracy: 0.8181
Epoch 6/10, Loss: 10511.3348
```