https://github.com/bentrevett/pytorch-sentiment-analysis

1 - Simple Sentiment Analysis

September 29, 2020

1 1 - Simple Sentiment Analysis

In this series we'll be building a machine learning model to detect sentiment (i.e. detect if a sentence is positive or negative) using PyTorch and TorchText. This will be done on movie reviews, using the IMDb dataset.

In this first notebook, we'll start very simple to understand the general concepts whilst not really caring about good results. Further notebooks will build on this knowledge and we'll actually get good results.

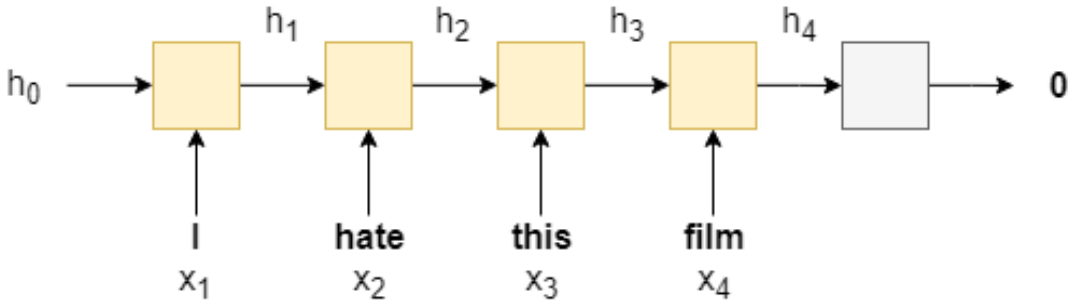
1.0.1 Introduction

We'll be using a **recurrent neural network** (RNN) as they are commonly used in analysing sequences. An RNN takes in sequence of words, $X = \{x_1, ..., x_T\}$, one at a time, and produces a *hidden state*, h, for each word. We use the RNN *recurrently* by feeding in the current word x_t as well as the hidden state from the previous word, h_{t-1}, to produce the next hidden state, h_t.

$$h_t = \text{RNN}(x_t, h_{t-1})$$

Once we have our final hidden state, h_T, (from feeding in the last word in the sequence, x_T) we feed it through a linear layer, f, (also known as a fully connected layer), to receive our predicted sentiment, $\hat{y} = f(h_T)$.

Below shows an example sentence, with the RNN predicting zero, which indicates a negative sentiment. The RNN is shown in orange and the linear layer shown in silver. Note that we use the same RNN for every word, i.e. it has the same parameters. The initial hidden state, h_0, is a tensor initialized to all zeros.



Note: some layers and steps have been omitted from the diagram, but these will be explained later.

1.1 Preparing Data

One of the main concepts of TorchText is the `Field`. These define how your data should be processed. In our sentiment classification task the data consists of both the raw string of the review and the sentiment, either "pos" or "neg".

The parameters of a `Field` specify how the data should be processed.

We use the `TEXT` field to define how the review should be processed, and the `LABEL` field to process the sentiment.

Our `TEXT` field has `tokenize='spacy'` as an argument. This defines that the "tokenization" (the act of splitting the string into discrete "tokens") should be done using the spaCy tokenizer. If no `tokenize` argument is passed, the default is simply splitting the string on spaces.

`LABEL` is defined by a `LabelField`, a special subset of the `Field` class specifically used for handling labels. We will explain the `dtype` argument later.

For more on `Fields`, go here.

We also set the random seeds for reproducibility.

[1]:
```
import torch
from torchtext import data

SEED = 1234

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

TEXT = data.Field(tokenize = 'spacy')
LABEL = data.LabelField(dtype = torch.float)
```

Another handy feature of TorchText is that it has support for common datasets used in natural language processing (NLP).

The following code automatically downloads the IMDb dataset and splits it into the canonical train/test splits as `torchtext.datasets` objects. It process the data using the `Fields` we have previously defined. The IMDb dataset consists of 50,000 movie reviews, each marked as being a positive or negative review.

[2]:
```
from torchtext import datasets

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
```

We can see how many examples are in each split by checking their length.

[3]:
```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 25000
Number of testing examples: 25000
```

We can also check an example.

[4]:
```
print(vars(train_data.examples[0]))
```

```
{'text': ['elvira', 'mistress', 'of', 'the', 'dark', 'is', 'one', 'of', 'my',
'fav', 'movies', ',', 'it', 'has', 'every', 'thing', 'you', 'would', 'want',
'in', 'a', 'film', ',', 'like', 'great', 'one', 'liners', ',', 'sexy', 'star',
'and', 'a', 'Outrageous', 'story', '!', 'if', 'you', 'have', 'not', 'seen',
'it', ',', 'you', 'are', 'missing', 'out', 'on', 'one', 'of', 'the', 'greatest',
'films', 'made', '.', 'i', 'ca', "n't", 'wait', 'till', 'her', 'new', 'movie',
'comes', 'out', '!'], 'label': 'pos'}
```

The IMDb dataset only has train/test splits, so we need to create a validation set. We can do this with the `.split()` method.

By default this splits 70/30, however by passing a `split_ratio` argument, we can change the ratio of the split, i.e. a `split_ratio` of 0.8 would mean 80% of the examples make up the training set and 20% make up the validation set.

We also pass our random seed to the `random_state` argument, ensuring that we get the same train/validation split each time.

[5]:
```
import random

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

Again, we'll view how many examples are in each split.

[6]:
```
print(f'Number of training examples: {len(train_data)}')
print(f'Number of validation examples: {len(valid_data)}')
print(f'Number of testing examples: {len(test_data)}')
```

```
Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
```

Next, we have to build a *vocabulary*. This is a effectively a look up table where every unique word in your data set has a corresponding *index* (an integer).

We do this as our machine learning model cannot operate on strings, only numbers. Each *index* is used to construct a *one-hot* vector for each word. A one-hot vector is a vector where all of the elements are 0, except one, which is 1, and dimensionality is the total number of unique words in your vocabulary, commonly denoted by V.

word	index	one-hot vector
I	0	[1, 0, 0, 0]
hate	1	[0, 1, 0, 0]
this	2	[0, 0, 1, 0]
film	3	[0, 0, 0, 1]

The number of unique words in our training set is over 100,000, which means that our one-hot vectors will have over 100,000 dimensions! This will make training slow and possibly won't fit onto your GPU (if you're using one).

There are two ways effectively cut down our vocabulary, we can either only take the top n most common words or ignore words that appear less than m times. We'll do the former, only keeping the top 25,000 words.

What do we do with words that appear in examples but we have cut from the vocabulary? We replace them with a special *unknown* or `<unk>` token. For example, if the sentence was "This film is great and I love it" but the word "love" was not in the vocabulary, it would become "This film is great and I `<unk>` it".

The following builds the vocabulary, only keeping the most common `max_size` tokens.

[7]:
```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data, max_size = MAX_VOCAB_SIZE)
LABEL.build_vocab(train_data)
```

Why do we only build the vocabulary on the training set? When testing any machine learning system you do not want to look at the test set in any way. We do not include the validation set as we want it to reflect the test set as much as possible.

[8]:
```
print(f"Unique tokens in TEXT vocabulary: {len(TEXT.vocab)}")
print(f"Unique tokens in LABEL vocabulary: {len(LABEL.vocab)}")
```

```
Unique tokens in TEXT vocabulary: 25002
Unique tokens in LABEL vocabulary: 2
```

Why is the vocab size 25002 and not 25000? One of the addition tokens is the `<unk>` token and the other is a `<pad>` token.

When we feed sentences into our model, we feed a *batch* of them at a time, i.e. more than one at a time, and all sentences in the batch need to be the same size. Thus, to ensure each sentence in the batch is the same size, any shorter than the longest within the batch are padded.

sent1	sent2
I	This
hate	film
this	sucks
film	<pad>

We can also view the most common words in the vocabulary and their frequencies.

[9]:
```
print(TEXT.vocab.freqs.most_common(20))
```

```
[('the', 202789), (',', 192769), ('.', 165632), ('and', 109469), ('a', 109242),
('of', 100791), ('to', 93641), ('is', 76253), ('in', 61374), ('I', 54030),
('it', 53487), ('that', 49111), ('"', 44657), ("'s", 43331), ('this', 42385),
('-', 36979), ('/><br', 35822), ('was', 35035), ('as', 30388), ('with', 29940)]
```

We can also see the vocabulary directly using either the `stoi` (**s**tring **to** **i**nt) or `itos` (**i**nt **to** **s**tring) method.

[10]:
```
print(TEXT.vocab.itos[:10])
```

```
['<unk>', '<pad>', 'the', ',', '.', 'and', 'a', 'of', 'to', 'is']
```

We can also check the labels, ensuring 0 is for negative and 1 is for positive.

[11]:
```
print(LABEL.vocab.stoi)
```

```
defaultdict(<function _default_unk_index at 0x7ff0c24dbf28>, {'neg': 0, 'pos':
1})
```

The final step of preparing the data is creating the iterators. We iterate over these in the training/evaluation loop, and they return a batch of examples (indexed and converted into tensors) at each iteration.

We'll use a `BucketIterator` which is a special type of iterator that will return a batch of examples where each example is of a similar length, minimizing the amount of padding per example.

We also want to place the tensors returned by the iterator on the GPU (if you're using one). PyTorch handles this using `torch.device`, we then pass this device to the iterator.

[12]:
```
BATCH_SIZE = 64

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

```
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

1.2 Build the Model

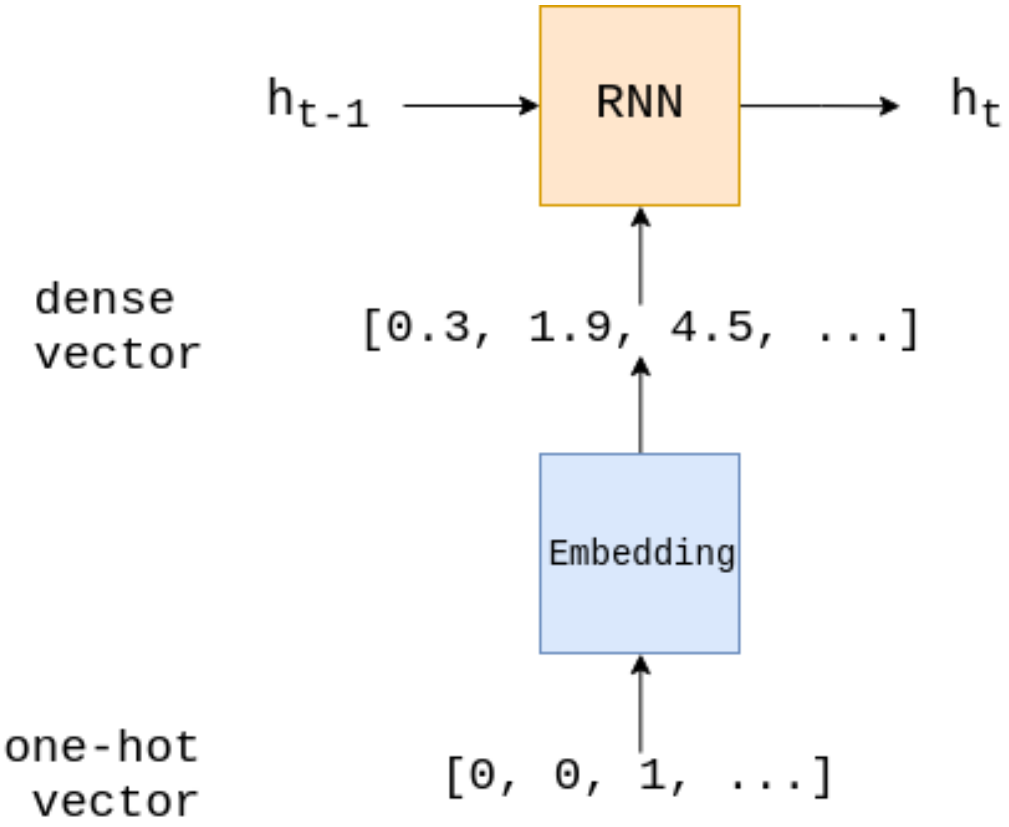
The next stage is building the model that we'll eventually train and evaluate.

There is a small amount of boilerplate code when creating models in PyTorch, note how our `RNN` class is a sub-class of `nn.Module` and the use of `super`.

Within the `__init__` we define the *layers* of the module. Our three layers are an *embedding* layer, our RNN, and a *linear* layer. All layers have their parameters initialized to random values, unless explicitly specified.

The embedding layer is used to transform our sparse one-hot vector (sparse as most of the elements are 0) into a dense embedding vector (dense as the dimensionality is a lot smaller and all the elements are real numbers). This embedding layer is simply a single fully connected layer. As well as reducing the dimensionality of the input to the RNN, there is the theory that words which have similar impact on the sentiment of the review are mapped close together in this dense vector space. For more information about word embeddings, see here.

The RNN layer is our RNN which takes in our dense vector and the previous hidden state h_{t-1}, which it uses to calculate the next hidden state, h_t.



Finally, the linear layer takes the final hidden state and feeds it through a fully connected layer, $f(h_T)$, transforming it to the correct output dimension.

The `forward` method is called when we feed examples into our model.

Each batch, `text`, is a tensor of size ***[sentence length, batch size]***. That is a batch of sentences, each having each word converted into a one-hot vector.

You may notice that this tensor should have another dimension due to the one-hot vectors, however PyTorch conveniently stores a one-hot vector as it's index value, i.e. the tensor representing a sentence is just a tensor of the indexes for each token in that sentence. The act of converting a list of tokens into a list of indexes is commonly called *numericalizing*.

The input batch is then passed through the embedding layer to get `embedded`, which gives us a dense vector representation of our sentences. `embedded` is a tensor of size ***[sentence length, batch size, embedding dim]***.

`embedded` is then fed into the RNN. In some frameworks you must feed the initial hidden state, h_0, into the RNN, however in PyTorch, if no initial hidden state is passed as an argument it defaults to a tensor of all zeros.

The RNN returns 2 tensors, `output` of size ***[sentence length, batch size, hidden dim]*** and `hidden` of size ***[1, batch size, hidden dim]***. `output` is the concatenation of the hidden state from every time step, whereas `hidden` is simply the final hidden state. We verify this using the `assert` statement. Note the `squeeze` method, which is used to remove a dimension of size 1.

Finally, we feed the last hidden state, `hidden`, through the linear layer, `fc`, to produce a prediction.

[13]:
```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.rnn = nn.RNN(embedding_dim, hidden_dim)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):

        #text = [sent len, batch size]

        embedded = self.embedding(text)

        #embedded = [sent len, batch size, emb dim]

        output, hidden = self.rnn(embedded)

        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]

        assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))
```

We now create an instance of our RNN class.

The input dimension is the dimension of the one-hot vectors, which is equal to the vocabulary size.

The embedding dimension is the size of the dense word vectors. This is usually around 50-250 dimensions, but depends on the size of the vocabulary.

The hidden dimension is the size of the hidden states. This is usually around 100-500 dimensions, but also depends on factors such as on the vocabulary size, the size of the dense vectors and the complexity of the task.

The output dimension is usually the number of classes, however in the case of only 2 classes the output value is between 0 and 1 and thus can be 1-dimensional, i.e. a single scalar real number.

[14]:
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
```

```
model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

Let's also create a function that will tell us how many trainable parameters our model has so we can compare the number of parameters across different models.

[15]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 2,592,105 trainable parameters
```

1.3 Train the Model

Now we'll set up the training and then train the model.

First, we'll create an optimizer. This is the algorithm we use to update the parameters of the module. Here, we'll use *stochastic gradient descent* (SGD). The first argument is the parameters will be updated by the optimizer, the second is the learning rate, i.e. how much we'll change the parameters by when we do a parameter update.

[16]:
```
import torch.optim as optim

optimizer = optim.SGD(model.parameters(), lr=1e-3)
```

Next, we'll define our loss function. In PyTorch this is commonly called a criterion.

The loss function here is *binary cross entropy with logits*.

Our model currently outputs an unbound real number. As our labels are either 0 or 1, we want to restrict the predictions to a number between 0 and 1. We do this using the *sigmoid* or *logit* functions.

We then use this this bound scalar to calculate the loss using binary cross entropy.

The `BCEWithLogitsLoss` criterion carries out both the sigmoid and the binary cross entropy steps.

[17]:
```
criterion = nn.BCEWithLogitsLoss()
```

Using `.to`, we can place the model and the criterion on the GPU (if we have one).

[18]:
```
model = model.to(device)
criterion = criterion.to(device)
```

Our criterion function calculates the loss, however we have to write our function to calculate the accuracy.

This function first feeds the predictions through a sigmoid layer, squashing the values between 0 and 1, we then round them to the nearest integer. This rounds any value greater than 0.5 to 1 (a positive sentiment) and the rest to 0 (a negative sentiment).

We then calculate how many rounded predictions equal the actual labels and average it across the batch.

[19]:
```
def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8,␣
↪NOT 8
    """

    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc
```

The `train` function iterates over all examples, one batch at a time.

`model.train()` is used to put the model in "training mode", which turns on *dropout* and *batch normalization*. Although we aren't using them in this model, it's good practice to include it.

For each batch, we first zero the gradients. Each parameter in a model has a `grad` attribute which stores the gradient calculated by the `criterion`. PyTorch does not automatically remove (or "zero") the gradients calculated from the last gradient calculation, so they must be manually zeroed.

We then feed the batch of sentences, `batch.text`, into the model. Note, you do not need to do `model.forward(batch.text)`, simply calling the model works. The `squeeze` is needed as the predictions are initially size ***[batch size, 1]***, and we need to remove the dimension of size 1 as PyTorch expects the predictions input to our criterion function to be of size ***[batch size]***.

The loss and accuracy are then calculated using our predictions and the labels, `batch.label`, with the loss being averaged over all examples in the batch.

We calculate the gradient of each parameter with `loss.backward()`, and then update the parameters using the gradients and optimizer algorithm with `optimizer.step()`.

The loss and accuracy is accumulated across the epoch, the `.item()` method is used to extract a scalar from a tensor which only contains a single value.

Finally, we return the loss and accuracy, averaged across the epoch. The `len` of an iterator is the number of batches in the iterator.

You may recall when initializing the `LABEL` field, we set `dtype=torch.float`. This is because TorchText sets tensors to be `LongTensor`s by default, however our criterion expects both inputs to be `FloatTensor`s. Setting the `dtype` to be `torch.float`, did this for us. The alternative method of doing this would be to do the conversion inside the `train` function by passing `batch.label.float()` instad of `batch.label` to the criterion.

[20]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0
```

```
    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

`evaluate` is similar to `train`, with a few modifications as you don't want to update the parameters when evaluating.

`model.eval()` puts the model in "evaluation mode", this turns off *dropout* and *batch normalization*. Again, we are not using them in this model, but it is good practice to include them.

No gradients are calculated on PyTorch operations inside the `with no_grad()` block. This causes less memory to be used and speeds up computation.

The rest of the function is the same as `train`, with the removal of `optimizer.zero_grad()`, `loss.backward()` and `optimizer.step()`, as we do not update the model's parameters when evaluating.

[21]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)
```

```
            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

We'll also create a function to tell us how long an epoch takes to compare training times between models.

[22]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

We then train the model through multiple epochs, an epoch being a complete pass through all examples in the training and validation sets.

At each epoch, if the validation loss is the best we have seen so far, we'll save the parameters of the model and then after training has finished we'll use that model on the test set.

[23]:
```
N_EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut1-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 0m 17s
```

```
        Train Loss: 0.694 | Train Acc: 50.12%
         Val. Loss: 0.696 |  Val. Acc: 50.17%
Epoch: 02 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 49.72%
         Val. Loss: 0.696 |  Val. Acc: 51.01%
Epoch: 03 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 50.22%
         Val. Loss: 0.696 |  Val. Acc: 50.87%
Epoch: 04 | Epoch Time: 0m 16s
        Train Loss: 0.693 | Train Acc: 49.94%
         Val. Loss: 0.696 |  Val. Acc: 49.91%
Epoch: 05 | Epoch Time: 0m 17s
        Train Loss: 0.693 | Train Acc: 50.07%
         Val. Loss: 0.696 |  Val. Acc: 51.00%
```

You may have noticed the loss is not really decreasing and the accuracy is poor. This is due to several issues with the model which we'll improve in the next notebook.

Finally, the metric we actually care about, the test loss and accuracy, which we get from our parameters that gave us the best validation loss.

[24]:
```
model.load_state_dict(torch.load('tut1-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.708 | Test Acc: 47.87%
```

1.4 Next Steps

In the next notebook, the improvements we will make are: - packed padded sequences - pre-trained word embeddings - different RNN architecture - bidirectional RNN - multi-layer RNN - regularization - a different optimizer

This will allow us to achieve ~84% accuracy.

3 - Faster Sentiment Analysis

September 29, 2020

1 3 - Faster Sentiment Analysis

In the previous notebook we managed to achieve a decent test accuracy of ~84% using all of the common techniques used for sentiment analysis. In this notebook, we'll implement a model that gets comparable results whilst training significantly faster and using around half of the parameters. More specifically, we'll be implementing the "FastText" model from the paper Bag of Tricks for Efficient Text Classification.

1.1 Preparing Data

One of the key concepts in the FastText paper is that they calculate the n-grams of an input sentence and append them to the end of a sentence. Here, we'll use bi-grams. Briefly, a bi-gram is a pair of words/tokens that appear consecutively within a sentence.

For example, in the sentence "how are you ?", the bi-grams are: "how are", "are you" and "you ?".

The `generate_bigrams` function takes a sentence that has already been tokenized, calculates the bi-grams and appends them to the end of the tokenized list.

[1]:
```
def generate_bigrams(x):
    n_grams = set(zip(*[x[i:] for i in range(2)]))
    for n_gram in n_grams:
        x.append(' '.join(n_gram))
    return x
```

As an example:

[2]:
```
generate_bigrams(['This', 'film', 'is', 'terrible'])
```

[2]:
```
['This', 'film', 'is', 'terrible', 'This film', 'film is', 'is terrible']
```

TorchText `Field`s have a `preprocessing` argument. A function passed here will be applied to a sentence after it has been tokenized (transformed from a string into a list of tokens), but before it has been numericalized (transformed from a list of tokens to a list of indexes). This is where we'll pass our `generate_bigrams` function.

As we aren't using an RNN we can't use packed padded sequences, thus we do not need to set `include_lengths = True`.

[3]:
```
import torch
from torchtext import data
from torchtext import datasets

SEED = 1234

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

TEXT = data.Field(tokenize = 'spacy', preprocessing = generate_bigrams)
LABEL = data.LabelField(dtype = torch.float)
```

As before, we load the IMDb dataset and create the splits.

[4]:
```
import random

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

Build the vocab and load the pre-trained word embeddings.

[5]:
```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data,
                 max_size = MAX_VOCAB_SIZE,
                 vectors = "glove.6B.100d",
                 unk_init = torch.Tensor.normal_)

LABEL.build_vocab(train_data)
```

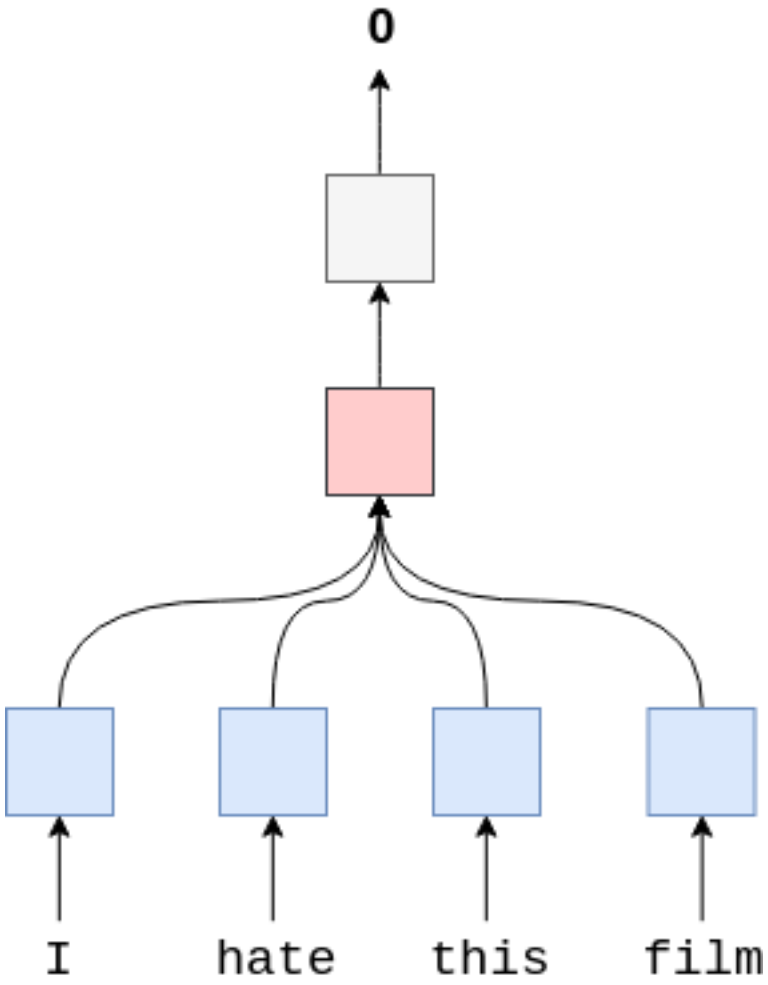
And create the iterators.

[6]:
```
BATCH_SIZE = 64

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

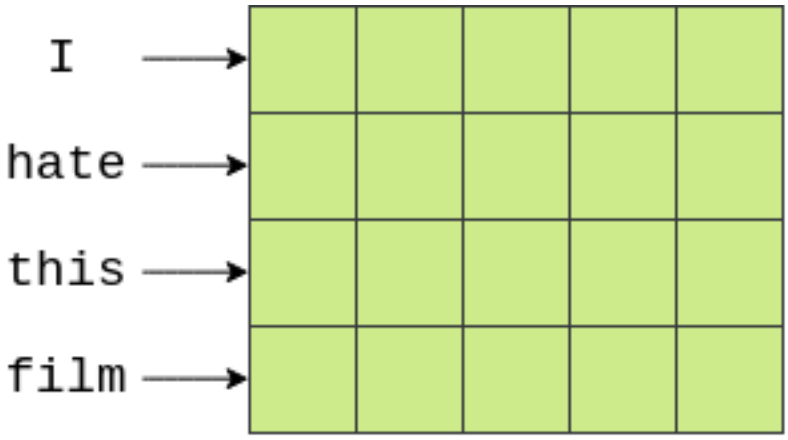
1.2 Build the Model

This model has far fewer parameters than the previous model as it only has 2 layers that have any parameters, the embedding layer and the linear layer. There is no RNN component in sight!

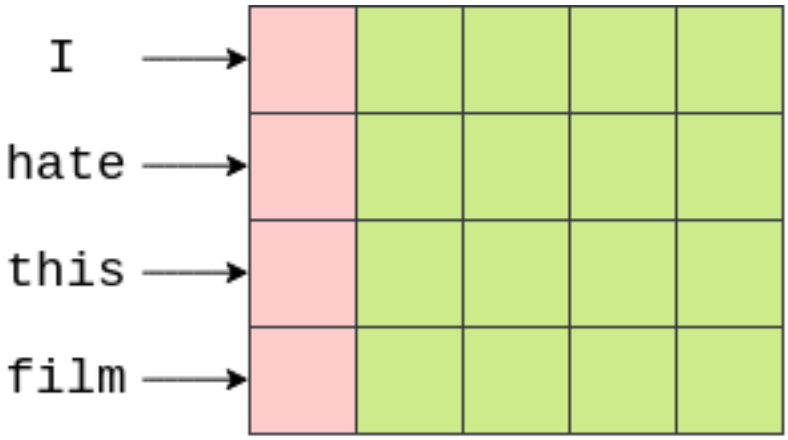
Instead, it first calculates the word embedding for each word using the `Embedding` layer (blue), then calculates the average of all of the word embeddings (pink) and feeds this through the `Linear` layer (silver), and that's it!



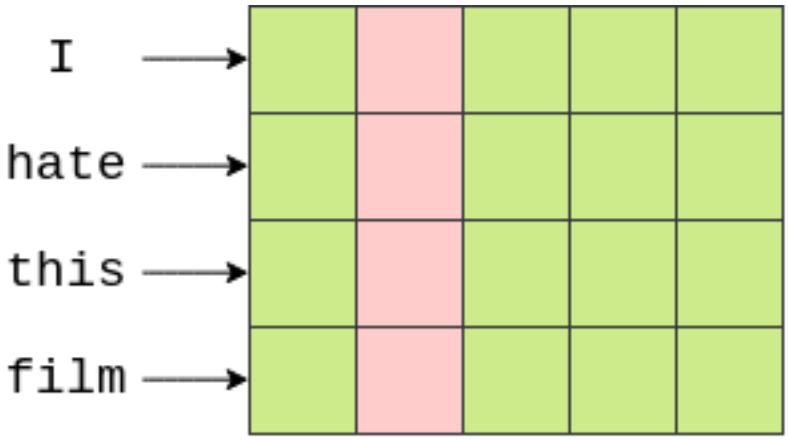
We implement the averaging with the `avg_pool2d` (average pool 2-dimensions) function. Initially, you may think using a 2-dimensional pooling seems strange, surely our sentences are 1-dimensional, not 2-dimensional? However, you can think of the word embeddings as a 2-dimensional grid, where the words are along one axis and the dimensions of the word embeddings are along the other. The image below is an example sentence after being converted into 5-dimensional word embeddings, with the words along the vertical axis and the embeddings along the horizontal axis. Each element in this [4x5] tensor is represented by a green block.



The `avg_pool2d` uses a filter of size `embedded.shape[1]` (i.e. the length of the sentence) by 1. This is shown in pink in the image below.



We calculate the average value of all elements covered by the filter, then the filter then slides to the right, calculating the average over the next column of embedding values for each word in the sentence.



Each filter position gives us a single value, the average of all covered elements. After the filter has

covered all embedding dimensions we get a [1x5] tensor. This tensor is then passed through the linear layer to produce our prediction.

[7]:
```
import torch.nn as nn
import torch.nn.functional as F

class FastText(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim,␣
↪padding_idx=pad_idx)

        self.fc = nn.Linear(embedding_dim, output_dim)

    def forward(self, text):

        #text = [sent len, batch size]

        embedded = self.embedding(text)

        #embedded = [sent len, batch size, emb dim]

        embedded = embedded.permute(1, 0, 2)

        #embedded = [batch size, sent len, emb dim]

        pooled = F.avg_pool2d(embedded, (embedded.shape[1], 1)).squeeze(1)

        #pooled = [batch size, embedding_dim]

        return self.fc(pooled)
```

As previously, we'll create an instance of our `FastText` class.

[8]:
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
OUTPUT_DIM = 1
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = FastText(INPUT_DIM, EMBEDDING_DIM, OUTPUT_DIM, PAD_IDX)
```

Looking at the number of parameters in our model, we see we have about the same as the standard RNN from the first notebook and half the parameters of the previous model.

[9]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
```

```
print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 2,500,301 trainable parameters
```

And copy the pre-trained vectors to our embedding layer.

[10]:
```
pretrained_embeddings = TEXT.vocab.vectors

model.embedding.weight.data.copy_(pretrained_embeddings)
```

[10]:
```
tensor([[-0.1117, -0.4966,  0.1631,  …,  1.2647, -0.2753, -0.1325],
        [-0.8555, -0.7208,  1.3755,  …,  0.0825, -1.1314,  0.3997],
        [-0.0382, -0.2449,  0.7281,  …, -0.1459,  0.8278,  0.2706],
        …,
        [ 0.3199,  0.0746,  0.0231,  …, -0.3609,  1.1303,  0.5668],
        [-1.0530, -1.0757,  0.3903,  …,  0.0792, -0.3059,  1.9734],
        [-0.1734, -0.3195,  0.3694,  …, -0.2435,  0.4767,  0.1151]])
```

Not forgetting to zero the initial weights of our unknown and padding tokens.

[11]:
```
UNK_IDX = TEXT.vocab.stoi[TEXT.unk_token]

model.embedding.weight.data[UNK_IDX] = torch.zeros(EMBEDDING_DIM)
model.embedding.weight.data[PAD_IDX] = torch.zeros(EMBEDDING_DIM)
```

1.3 Train the Model

Training the model is the exact same as last time.

We initialize our optimizer...

[12]:
```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())
```

We define the criterion and place the model and criterion on the GPU (if available)...

[13]:
```
criterion = nn.BCEWithLogitsLoss()

model = model.to(device)
criterion = criterion.to(device)
```

We implement the function to calculate accuracy...

[14]:
```
def binary_accuracy(preds, y):
    """
```

```
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8, NOT 8
    """

    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc
```

We define a function for training our model...

Note: we are no longer using dropout so we do not need to use `model.train()`, but as mentioned in the 1st notebook, it is good practice to use it.

[15]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

We define a function for testing our model...

Note: again, we leave `model.eval()` even though we do not use dropout.

[16]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
```

```
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

As before, we'll implement a useful function to tell us how long an epoch takes.

[17]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Finally, we train our model.

[18]:
```
N_EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
```

```
        torch.save(model.state_dict(), 'tut3-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 0m 6s
        Train Loss: 0.688 | Train Acc: 57.23%
         Val. Loss: 0.642 |  Val. Acc: 71.23%
Epoch: 02 | Epoch Time: 0m 5s
        Train Loss: 0.653 | Train Acc: 71.09%
         Val. Loss: 0.521 |  Val. Acc: 75.28%
Epoch: 03 | Epoch Time: 0m 5s
        Train Loss: 0.582 | Train Acc: 78.88%
         Val. Loss: 0.449 |  Val. Acc: 79.64%
Epoch: 04 | Epoch Time: 0m 5s
        Train Loss: 0.505 | Train Acc: 83.15%
         Val. Loss: 0.426 |  Val. Acc: 82.12%
Epoch: 05 | Epoch Time: 0m 5s
        Train Loss: 0.439 | Train Acc: 85.99%
         Val. Loss: 0.397 |  Val. Acc: 85.02%
```

...and get the test accuracy!

The results are comparable to the results in the last notebook, but training takes considerably less time!

[19]:
```
model.load_state_dict(torch.load('tut3-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.391 | Test Acc: 85.11%
```

1.4 User Input

And as before, we can test on any input the user provides making sure to generate bigrams from our tokenized sentence.

[20]:
```
import spacy
nlp = spacy.load('en')

def predict_sentiment(model, sentence):
    model.eval()
    tokenized = generate_bigrams([tok.text for tok in nlp.tokenizer(sentence)])
    indexed = [TEXT.vocab.stoi[t] for t in tokenized]
    tensor = torch.LongTensor(indexed).to(device)
```

```
    tensor = tensor.unsqueeze(1)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()
```

An example negative review...

[21]:
```
predict_sentiment(model, "This film is terrible")
```

[21]:
```
1.621993561684576e-07
```

An example positive review...

[22]:
```
predict_sentiment(model, "This film is great")
```

[22]:
```
1.0
```

1.5 Next Steps

In the next notebook we'll use convolutional neural networks (CNNs) to perform sentiment analysis.

4 - Convolutional Sentiment Analysis

September 29, 2020

1 4 - Convolutional Sentiment Analysis

In the previous notebooks, we managed to achieve a test accuracy of ~85% using RNNs and an implementation of the Bag of Tricks for Efficient Text Classification model. In this notebook, we will be using a *convolutional neural network* (CNN) to conduct sentiment analysis, implementing the model from Convolutional Neural Networks for Sentence Classification.

Note: This tutorial is not aiming to give a comprehensive introduction and explanation of CNNs. For a better and more in-depth explanation check out here and here.

Traditionally, CNNs are used to analyse images and are made up of one or more *convolutional* layers, followed by one or more linear layers. The convolutional layers use filters (also called *kernels* or *receptive fields*) which scan across an image and produce a processed version of the image. This processed version of the image can be fed into another convolutional layer or a linear layer. Each filter has a shape, e.g. a 3x3 filter covers a 3 pixel wide and 3 pixel high area of the image, and each element of the filter has a weight associated with it, the 3x3 filter would have 9 weights. In traditional image processing these weights were specified by hand by engineers, however the main advantage of the convolutional layers in neural networks is that these weights are learned via backpropagation.

The intuitive idea behind learning the weights is that your convolutional layers act like *feature extractors*, extracting parts of the image that are most important for your CNN's goal, e.g. if using a CNN to detect faces in an image, the CNN may be looking for features such as the existance of a nose, mouth or a pair of eyes in the image.

So why use CNNs on text? In the same way that a 3x3 filter can look over a patch of an image, a 1x2 filter can look over a 2 sequential words in a piece of text, i.e. a bi-gram. In the previous tutorial we looked at the FastText model which used bi-grams by explicitly adding them to the end of a text, in this CNN model we will instead use multiple filters of different sizes which will look at the bi-grams (a 1x2 filter), tri-grams (a 1x3 filter) and/or n-grams (a 1xn filter) within the text.

The intuition here is that the appearance of certain bi-grams, tri-grams and n-grams within the review will be a good indication of the final sentiment.

1.1 Preparing Data

As in the previous notebooks, we'll prepare the data.

Unlike the previous notebook with the FastText model, we no longer explicitly need to create the bi-grams and append them to the end of the sentence.

As convolutional layers expect the batch dimension to be first we can tell TorchText to return the data already permuted using the `batch_first = True` argument on the field.

[1]:
```
import torch
from torchtext import data
from torchtext import datasets
import random
import numpy as np

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

TEXT = data.Field(tokenize = 'spacy', batch_first = True)
LABEL = data.LabelField(dtype = torch.float)

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

Build the vocab and load the pre-trained word embeddings.

[2]:
```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data,
                 max_size = MAX_VOCAB_SIZE,
                 vectors = "glove.6B.100d",
                 unk_init = torch.Tensor.normal_)

LABEL.build_vocab(train_data)
```

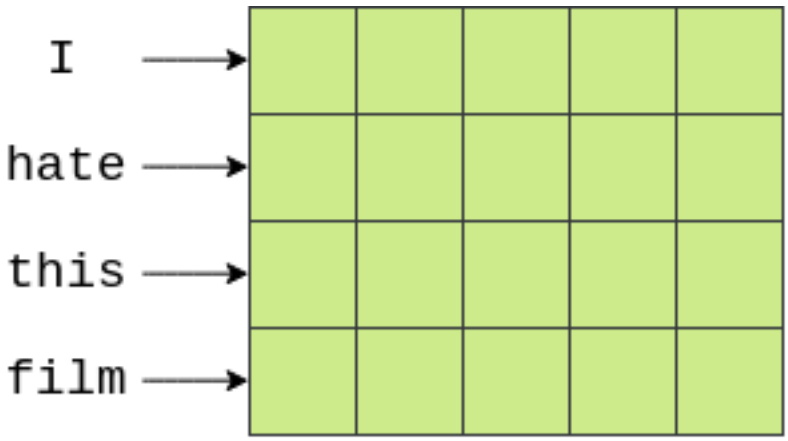
As before, we create the iterators.

[3]:
```
BATCH_SIZE = 64

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

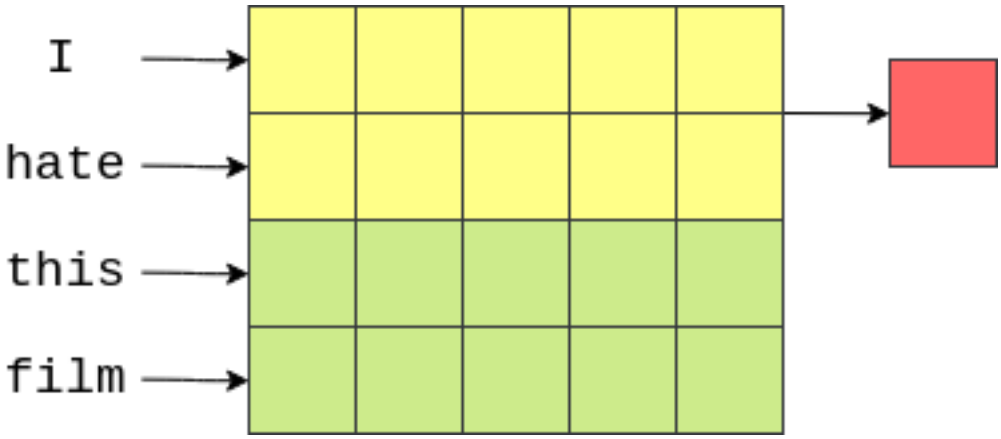
1.2 Build the Model

Now to build our model.

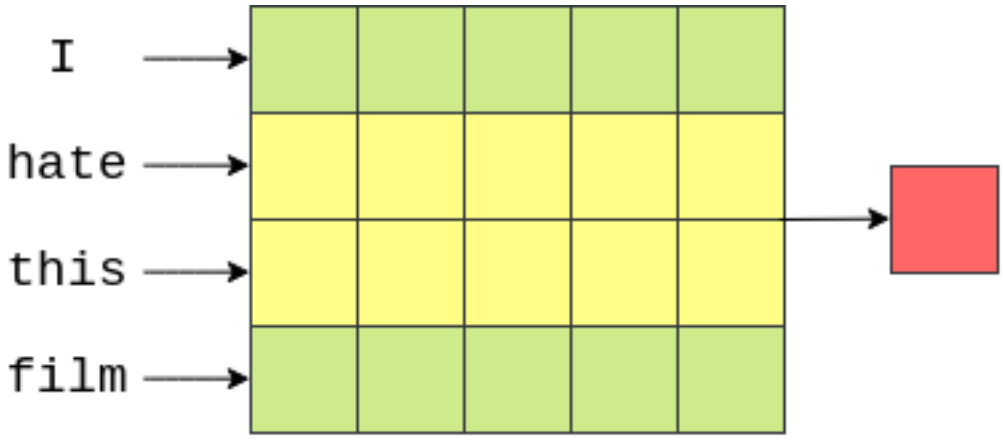
The first major hurdle is visualizing how CNNs are used for text. Images are typically 2 dimensional (we'll ignore the fact that there is a third "colour" dimension for now) whereas text is 1 dimensional. However, we know that the first step in almost all of our previous tutorials (and pretty much all NLP pipelines) is converting the words into word embeddings. This is how we can visualize our words in 2 dimensions, each word along one axis and the elements of vectors aross the other dimension. Consider the 2 dimensional representation of the embedded sentence below:



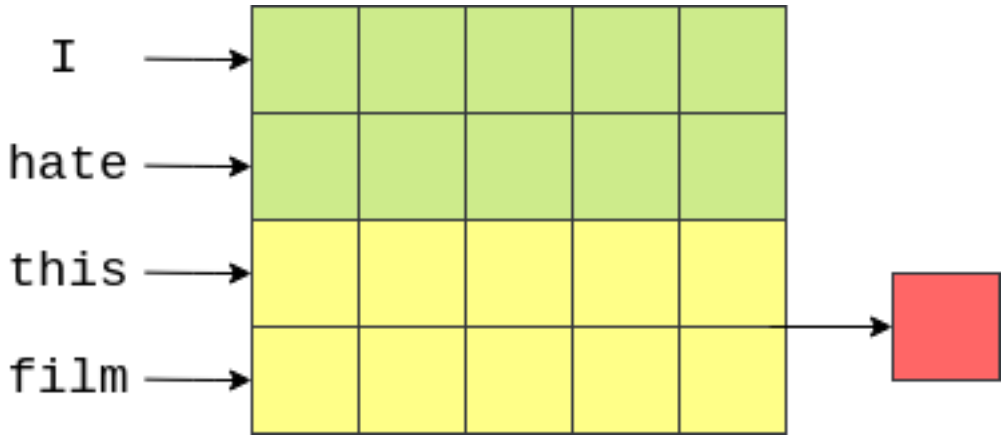
We can then use a filter that is **[n x emb_dim]**. This will cover n sequential words entirely, as their width will be `emb_dim` dimensions. Consider the image below, with our word vectors are represented in green. Here we have 4 words with 5 dimensional embeddings, creating a [4x5] "image" tensor. A filter that covers two words at a time (i.e. bi-grams) will be **[2x5]** filter, shown in yellow, and each element of the filter with have a *weight* associated with it. The output of this filter (shown in red) will be a single real number that is the weighted sum of all elements covered by the filter.



The filter then moves "down" the image (or across the sentence) to cover the next bi-gram and another output (weighted sum) is calculated.



Finally, the filter moves down again and the final output for this filter is calculated.

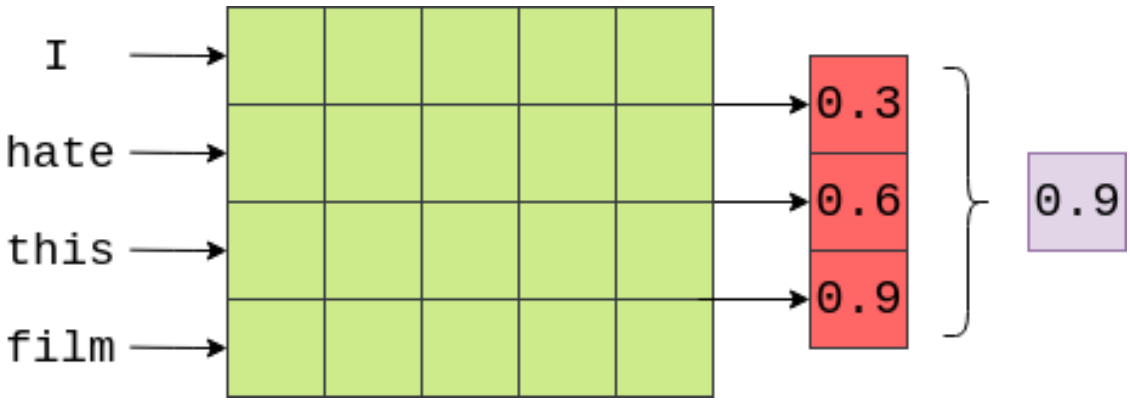


In our case (and in the general case where the width of the filter equals the width of the "image"), our output will be a vector with number of elements equal to the height of the image (or lenth of the word) minus the height of the filter plus one, $4 - 2 + 1 = 3$ in this case.

This example showed how to calculate the output of one filter. Our model (and pretty much all CNNs) will have lots of these filters. The idea is that each filter will learn a different feature to extract. In the above example, we are hoping each of the **[2 x emb_dim]** filters will be looking for the occurence of different bi-grams.

In our model, we will also have different sizes of filters, heights of 3, 4 and 5, with 100 of each of them. The intuition is that we will be looking for the occurence of different tri-grams, 4-grams and 5-grams that are relevant for analysing sentiment of movie reviews.

The next step in our model is to use *pooling* (specifically *max pooling*) on the output of the convolutional layers. This is similar to the FastText model where we performed the average over each of the word vectors, implemented by the `F.avg_pool2d` function, however instead of taking the average over a dimension, we are taking the maximum value over a dimension. Below an example of taking the maximum value (0.9) from the output of the convolutional layer on the example sentence (not shown is the activation function applied to the output of the convolutions).



The idea here is that the maximum value is the "most important" feature for determining the sentiment of the review, which corresponds to the "most important" n-gram within the review. How do we know what the "most important" n-gram is? Luckily, we don't have to! Through backpropagation, the weights of the filters are changed so that whenever certain n-grams that are highly indicative of the sentiment are seen, the output of the filter is a "high" value. This "high" value then passes through the max pooling layer if it is the maximum value in the output.

As our model has 100 filters of 3 different sizes, that means we have 300 different n-grams the model thinks are important. We concatenate these together into a single vector and pass them through a linear layer to predict the sentiment. We can think of the weights of this linear layer as "weighting up the evidence" from each of the 300 n-grams and making a final decision.

1.2.1 Implementation Details

We implement the convolutional layers with `nn.Conv2d`. The `in_channels` argument is the number of "channels" in your image going into the convolutional layer. In actual images this is usually 3 (one channel for each of the red, blue and green channels), however when using text we only have a single channel, the text itself. The `out_channels` is the number of filters and the `kernel_size` is the size of the filters. Each of our `kernel_size`s is going to be **[n x emb_dim]** where n is the size of the n-grams.

In PyTorch, RNNs want the input with the batch dimension second, whereas CNNs want the batch dimension first - we do not have to permute the data here as we have already set `batch_first = True` in our `TEXT` field. We then pass the sentence through an embedding layer to get our embeddings. The second dimension of the input into a `nn.Conv2d` layer must be the channel dimension. As text technically does not have a channel dimension, we `unsqueeze` our tensor to create one. This matches with our `in_channels=1` in the initialization of our convolutional layers.

We then pass the tensors through the convolutional and pooling layers, using the `ReLU` activation function after the convolutional layers. Another nice feature of the pooling layers is that they handle sentences of different lengths. The size of the output of the convolutional layer is dependent on the size of the input to it, and different batches contain sentences of different lengths. Without the max pooling layer the input to our linear layer would depend on the size of the input sentence (not what we want). One option to rectify this would be to trim/pad all sentences to the same length, however with the max pooling layer we always know the input to the linear layer will be the total number of filters. **Note**: there an exception to this if your sentence(s) are shorter than

the largest filter used. You will then have to pad your sentences to the length of the largest filter. In the IMDb data there are no reviews only 5 words long so we don't have to worry about that, but you will if you are using your own data.

Finally, we perform dropout on the concatenated filter outputs and then pass them through a linear layer to make our predictions.

[4]:
```
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes,␣
↪output_dim,
                 dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx =␣
↪pad_idx)

        self.conv_0 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[0], embedding_dim))

        self.conv_1 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[1], embedding_dim))

        self.conv_2 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (filter_sizes[2], embedding_dim))

        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [batch size, sent len]

        embedded = self.embedding(text)

        #embedded = [batch size, sent len, emb dim]

        embedded = embedded.unsqueeze(1)

        #embedded = [batch size, 1, sent len, emb dim]
```

```
        conved_0 = F.relu(self.conv_0(embedded).squeeze(3))
        conved_1 = F.relu(self.conv_1(embedded).squeeze(3))
        conved_2 = F.relu(self.conv_2(embedded).squeeze(3))

        #conved_n = [batch size, n_filters, sent len - filter_sizes[n] + 1]

        pooled_0 = F.max_pool1d(conved_0, conved_0.shape[2]).squeeze(2)
        pooled_1 = F.max_pool1d(conved_1, conved_1.shape[2]).squeeze(2)
        pooled_2 = F.max_pool1d(conved_2, conved_2.shape[2]).squeeze(2)

        #pooled_n = [batch size, n_filters]

        cat = self.dropout(torch.cat((pooled_0, pooled_1, pooled_2), dim = 1))

        #cat = [batch size, n_filters * len(filter_sizes)]

        return self.fc(cat)
```

Currently the `CNN` model can only use 3 different sized filters, but we can actually improve the code of our model to make it more generic and take any number of filters.

We do this by placing all of our convolutional layers in a `nn.ModuleList`, a function used to hold a list of PyTorch `nn.Modules`. If we simply used a standard Python list, the modules within the list cannot be "seen" by any modules outside the list which will cause us some errors.

We can now pass an arbitrary sized list of filter sizes and the list comprehension will create a convolutional layer for each of them. Then, in the `forward` method we iterate through the list applying each convolutional layer to get a list of convolutional outputs, which we also feed through the max pooling in a list comprehension before concatenating together and passing through the dropout and linear layers.

[5]:
```
class CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes,␣
↪output_dim,
                 dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx =␣
↪pad_idx)

        self.convs = nn.ModuleList([
                                    nn.Conv2d(in_channels = 1,
                                              out_channels = n_filters,
                                              kernel_size = (fs, embedding_dim))
                                    for fs in filter_sizes
                                    ])
```

```
        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [batch size, sent len]

        embedded = self.embedding(text)

        #embedded = [batch size, sent len, emb dim]

        embedded = embedded.unsqueeze(1)

        #embedded = [batch size, 1, sent len, emb dim]

        conved = [F.relu(conv(embedded)).squeeze(3) for conv in self.convs]

        #conved_n = [batch size, n_filters, sent len - filter_sizes[n] + 1]

        pooled = [F.max_pool1d(conv, conv.shape[2]).squeeze(2) for conv in␣
↪conved]

        #pooled_n = [batch size, n_filters]

        cat = self.dropout(torch.cat(pooled, dim = 1))

        #cat = [batch size, n_filters * len(filter_sizes)]

        return self.fc(cat)
```

We can also implement the above model using 1-dimensional convolutional layers, where the embedding dimension is the "depth" of the filter and the number of tokens in the sentence is the width.

We'll run our tests in this notebook using the 2-dimensional convolutional model, but leave the implementation for the 1-dimensional model below for anyone interested.

[6]:
```
class CNN1d(nn.Module):
    def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes,␣
↪output_dim,
                 dropout, pad_idx):

        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx =␣
↪pad_idx)
```

```
        self.convs = nn.ModuleList([
                                    nn.Conv1d(in_channels = embedding_dim,
                                              out_channels = n_filters,
                                              kernel_size = fs)
                                    for fs in filter_sizes
                                    ])

        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [batch size, sent len]

        embedded = self.embedding(text)

        #embedded = [batch size, sent len, emb dim]

        embedded = embedded.permute(0, 2, 1)

        #embedded = [batch size, emb dim, sent len]

        conved = [F.relu(conv(embedded)) for conv in self.convs]

        #conved_n = [batch size, n_filters, sent len - filter_sizes[n] + 1]

        pooled = [F.max_pool1d(conv, conv.shape[2]).squeeze(2) for conv in␣
↪conved]

        #pooled_n = [batch size, n_filters]

        cat = self.dropout(torch.cat(pooled, dim = 1))

        #cat = [batch size, n_filters * len(filter_sizes)]

        return self.fc(cat)
```

We create an instance of our `CNN` class.

We can change `CNN` to `CNN1d` if we want to run the 1-dimensional convolutional model, noting that both models give almost identical results.

[7]:
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
N_FILTERS = 100
FILTER_SIZES = [3,4,5]
OUTPUT_DIM = 1
```

```
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = CNN(INPUT_DIM, EMBEDDING_DIM, N_FILTERS, FILTER_SIZES, OUTPUT_DIM,␣
 ↪DROPOUT, PAD_IDX)
```

Checking the number of parameters in our model we can see it has about the same as the FastText model.

Both the `CNN` and the `CNN1d` models have the exact same number of parameters.

[8]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 2,620,801 trainable parameters
```

Next, we'll load the pre-trained embeddings

[9]:
```
pretrained_embeddings = TEXT.vocab.vectors

model.embedding.weight.data.copy_(pretrained_embeddings)
```

[9]:
```
tensor([[-0.1117, -0.4966,  0.1631,  …,  1.2647, -0.2753, -0.1325],
        [-0.8555, -0.7208,  1.3755,  …,  0.0825, -1.1314,  0.3997],
        [-0.0382, -0.2449,  0.7281,  …, -0.1459,  0.8278,  0.2706],
        …,
        [-0.0614, -0.0516, -0.6159,  …, -0.0354,  0.0379, -0.1809],
        [ 0.1885, -0.1690,  0.1530,  …, -0.2077,  0.5473, -0.4517],
        [-0.1182, -0.4701, -0.0600,  …,  0.7991, -0.0194,  0.4785]])
```

Then zero the initial weights of the unknown and padding tokens.

[10]:
```
UNK_IDX = TEXT.vocab.stoi[TEXT.unk_token]

model.embedding.weight.data[UNK_IDX] = torch.zeros(EMBEDDING_DIM)
model.embedding.weight.data[PAD_IDX] = torch.zeros(EMBEDDING_DIM)
```

1.3 Train the Model

Training is the same as before. We initialize the optimizer, loss function (criterion) and place the model and criterion on the GPU (if available)

[11]:
```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())
```

```
criterion = nn.BCEWithLogitsLoss()

model = model.to(device)
criterion = criterion.to(device)
```

We implement the function to calculate accuracy...

[12]:
```
def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8,␣
 ↪NOT 8
    """

    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc
```

We define a function for training our model...

Note: as we are using dropout again, we must remember to use `model.train()` to ensure the dropout is "turned on" while training.

[13]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()
```

```
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

We define a function for testing our model...

Note: again, as we are now using dropout, we must remember to use `model.eval()` to ensure the dropout is "turned off" while evaluating.

[14]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Let's define our function to tell us how long epochs take.

[15]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Finally, we train our model...

[16]:
```
N_EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()
```

```
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut4-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 0m 13s
        Train Loss: 0.645 | Train Acc: 62.08%
         Val. Loss: 0.488 |  Val. Acc: 78.64%
Epoch: 02 | Epoch Time: 0m 11s
        Train Loss: 0.418 | Train Acc: 81.14%
         Val. Loss: 0.361 |  Val. Acc: 84.59%
Epoch: 03 | Epoch Time: 0m 11s
        Train Loss: 0.300 | Train Acc: 87.33%
         Val. Loss: 0.348 |  Val. Acc: 85.06%
Epoch: 04 | Epoch Time: 0m 11s
        Train Loss: 0.217 | Train Acc: 91.49%
         Val. Loss: 0.320 |  Val. Acc: 86.71%
Epoch: 05 | Epoch Time: 0m 11s
        Train Loss: 0.156 | Train Acc: 94.22%
         Val. Loss: 0.334 |  Val. Acc: 87.06%
```

We get test results comparable to the previous 2 models!

[17]:
```
model.load_state_dict(torch.load('tut4-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.339 | Test Acc: 85.39%
```

1.4 User Input

And again, as a sanity check we can check some input sentences

Note: As mentioned in the implementation details, the input sentence has to be at least as long as the largest filter height used. We modify our `predict_sentiment` function to also accept a

minimum length argument. If the tokenized input sentence is less than `min_len` tokens, we append padding tokens (`<pad>`) to make it `min_len` tokens.

[20]:
```
import spacy
nlp = spacy.load('en')

def predict_sentiment(model, sentence, min_len = 5):
    model.eval()
    tokenized = [tok.text for tok in nlp.tokenizer(sentence)]
    if len(tokenized) < min_len:
        tokenized += ['<pad>'] * (min_len - len(tokenized))
    indexed = [TEXT.vocab.stoi[t] for t in tokenized]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(0)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()
```

An example negative review...

[21]:
```
predict_sentiment(model, "This film is terrible")
```

[21]:
```
0.11022213101387024
```

An example positive review...

[22]:
```
predict_sentiment(model, "This film is great")
```

[22]:
```
0.9785954356193542
```

5 - Multi-class Sentiment Analysis

September 29, 2020

1 5 - Multi-class Sentiment Analysis

In all of the previous notebooks we have performed sentiment analysis on a dataset with only two classes, positive or negative. When we have only two classes our output can be a single scalar, bound between 0 and 1, that indicates what class an example belongs to. When we have more than 2 examples, our output must be a C dimensional vector, where C is the number of classes.

In this notebook, we'll be performing classification on a dataset with 6 classes. Note that this dataset isn't actually a sentiment analysis dataset, it's a dataset of questions and the task is to classify what category the question belongs to. However, everything covered in this notebook applies to any dataset with examples that contain an input sequence belonging to one of C classes.

Below, we setup the fields, and load the dataset.

The first difference is that we do not need to set the `dtype` in the `LABEL` field. When doing a mutli-class problem, PyTorch expects the labels to be numericalized `LongTensor`s.

The second different is that we use `TREC` instead of `IMDB` to load the `TREC` dataset. The `fine_grained` argument allows us to use the fine-grained labels (of which there are 50 classes) or not (in which case they'll be 6 classes). You can change this how you please.

[1]:
```
import torch
from torchtext import data
from torchtext import datasets
import random

SEED = 1234

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

TEXT = data.Field(tokenize = 'spacy')
LABEL = data.LabelField()

train_data, test_data = datasets.TREC.splits(TEXT, LABEL, fine_grained=False)

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

Let's look at one of the examples in the training set.

[2]:
```
vars(train_data[-1])
```

[2]:
```
{'text': ['What', 'is', 'a', 'Cartesian', 'Diver', '?'], 'label': 'DESC'}
```

Next, we'll build the vocabulary. As this dataset is small (only ~3800 training examples) it also has a very small vocabulary (~7500 unique tokens), this means we do not need to set a `max_size` on the vocabulary as before.

[3]:
```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data,
                 max_size = MAX_VOCAB_SIZE,
                 vectors = "glove.6B.100d",
                 unk_init = torch.Tensor.normal_)

LABEL.build_vocab(train_data)
```

Next, we can check the labels.

The 6 labels (for the non-fine-grained case) correspond to the 6 types of questions in the dataset: - `HUM` for questions about humans - `ENTY` for questions about entities - `DESC` for questions asking you for a description - `NUM` for questions where the answer is numerical - `LOC` for questions where the answer is a location - `ABBR` for questions asking about abbreviations

[4]:
```
print(LABEL.vocab.stoi)
```

```
defaultdict(<function _default_unk_index at 0x7f0a50190d08>, {'HUM': 0, 'ENTY':
1, 'DESC': 2, 'NUM': 3, 'LOC': 4, 'ABBR': 5})
```

As always, we set up the iterators.

[5]:
```
BATCH_SIZE = 64

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

We'll be using the CNN model from the previous notebook, however any of the models covered in these tutorials will work on this dataset. The only difference is now the `output_dim` will be C instead of 1.

[6]:
```
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
```

```
   def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes,␣
↪output_dim,
                dropout, pad_idx):

       super().__init__()

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

       self.convs = nn.ModuleList([
                                   nn.Conv2d(in_channels = 1,
                                             out_channels = n_filters,
                                             kernel_size = (fs, embedding_dim))
                                   for fs in filter_sizes
                                   ])

       self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)

       self.dropout = nn.Dropout(dropout)

   def forward(self, text):

       #text = [sent len, batch size]

       text = text.permute(1, 0)

       #text = [batch size, sent len]

       embedded = self.embedding(text)

       #embedded = [batch size, sent len, emb dim]

       embedded = embedded.unsqueeze(1)

       #embedded = [batch size, 1, sent len, emb dim]

       conved = [F.relu(conv(embedded)).squeeze(3) for conv in self.convs]

       #conv_n = [batch size, n_filters, sent len - filter_sizes[n]]

       pooled = [F.max_pool1d(conv, conv.shape[2]).squeeze(2) for conv in␣
↪conved]

       #pooled_n = [batch size, n_filters]

       cat = self.dropout(torch.cat(pooled, dim = 1))

       #cat = [batch size, n_filters * len(filter_sizes)]
```

```
        return self.fc(cat)
```

We define our model, making sure to set `OUTPUT_DIM` to C. We can get C easily by using the size of the `LABEL` vocab, much like we used the length of the `TEXT` vocab to get the size of the vocabulary of the input.

The examples in this dataset are generally a lot smaller than those in the IMDb dataset, so we'll use smaller filter sizes.

[7]:
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
N_FILTERS = 100
FILTER_SIZES = [2,3,4]
OUTPUT_DIM = len(LABEL.vocab)
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = CNN(INPUT_DIM, EMBEDDING_DIM, N_FILTERS, FILTER_SIZES, OUTPUT_DIM,␣
 ↪DROPOUT, PAD_IDX)
```

Checking the number of parameters, we can see how the smaller filter sizes means we have about a third of the parameters than we did for the CNN model on the IMDb dataset.

[8]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 842,406 trainable parameters
```

Next, we'll load our pre-trained embeddings.

[9]:
```
pretrained_embeddings = TEXT.vocab.vectors

model.embedding.weight.data.copy_(pretrained_embeddings)
```

[9]:
```
tensor([[-0.1117, -0.4966,  0.1631,  …,  1.2647, -0.2753, -0.1325],
        [-0.8555, -0.7208,  1.3755,  …,  0.0825, -1.1314,  0.3997],
        [ 0.1638,  0.6046,  1.0789,  …, -0.3140,  0.1844,  0.3624],
        …,
        [-0.3110, -0.3398,  1.0308,  …,  0.5317,  0.2836, -0.0640],
        [ 0.0091,  0.2810,  0.7356,  …, -0.7508,  0.8967, -0.7631],
        [ 0.4306,  1.2011,  0.0873,  …,  0.8817,  0.3722,  0.3458]])
```

Then zero the initial weights of the unknown and padding tokens.

[10]:
```
UNK_IDX = TEXT.vocab.stoi[TEXT.unk_token]
```

```
model.embedding.weight.data[UNK_IDX] = torch.zeros(EMBEDDING_DIM)
model.embedding.weight.data[PAD_IDX] = torch.zeros(EMBEDDING_DIM)
```

Another different to the previous notebooks is our loss function (aka criterion). Before we used `BCEWithLogitsLoss`, however now we use `CrossEntropyLoss`. Without going into too much detail, `CrossEntropyLoss` performs a *softmax* function over our model outputs and the loss is given by the *cross entropy* between that and the label.

Generally: - `CrossEntropyLoss` is used when our examples exclusively belong to one of C classes - `BCEWithLogitsLoss` is used when our examples exclusively belong to only 2 classes (0 and 1) and is also used in the case where our examples belong to between 0 and C classes (aka multilabel classification).

[11]:
```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())

criterion = nn.CrossEntropyLoss()

model = model.to(device)
criterion = criterion.to(device)
```

Before, we had a function that calculated accuracy in the binary label case, where we said if the value was over 0.5 then we would assume it is positive. In the case where we have more than 2 classes, our model outputs a C dimensional vector, where the value of each element is the beleief that the example belongs to that class.

For example, in our labels we have: 'HUM' = 0, 'ENTY' = 1, 'DESC' = 2, 'NUM' = 3, 'LOC' = 4 and 'ABBR' = 5. If the output of our model was something like: **[5.1, 0.3, 0.1, 2.1, 0.2, 0.6]** this means that the model strongly believes the example belongs to class 0, a question about a human, and slightly believes the example belongs to class 3, a numerical question.

We calculate the accuracy by performing an `argmax` to get the index of the maximum value in the prediction for each element in the batch, and then counting how many times this equals the actual label. We then average this across the batch.

[12]:
```
def categorical_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8,␣
↪NOT 8
    """
    max_preds = preds.argmax(dim = 1, keepdim = True) # get the index of the␣
↪max probability
    correct = max_preds.squeeze(1).eq(y)
    return correct.sum() / torch.FloatTensor([y.shape[0]])
```

The training loop is similar to before, without the need to `squeeze` the model predictions as `CrossEntropyLoss` expects the input to be [**batch size, n classes**] and the label to be [**batch size**].

The label needs to be a `LongTensor`, which it is by default as we did not set the `dtype` to a `FloatTensor` as before.

[13]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text)

        loss = criterion(predictions, batch.label)

        acc = categorical_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

The evaluation loop is, again, similar to before.

[14]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text)

            loss = criterion(predictions, batch.label)

            acc = categorical_accuracy(predictions, batch.label)
```

```
            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[15]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Next, we train our model.

[16]:
```
N_EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut5-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 0m 0s
        Train Loss: 1.310 | Train Acc: 47.99%
         Val. Loss: 0.947 |  Val. Acc: 66.81%
Epoch: 02 | Epoch Time: 0m 0s
        Train Loss: 0.869 | Train Acc: 69.09%
         Val. Loss: 0.746 |  Val. Acc: 74.18%
Epoch: 03 | Epoch Time: 0m 0s
        Train Loss: 0.665 | Train Acc: 76.94%
         Val. Loss: 0.627 |  Val. Acc: 78.03%
Epoch: 04 | Epoch Time: 0m 0s
```

```
        Train Loss: 0.503 | Train Acc: 83.42%
         Val. Loss: 0.548 |  Val. Acc: 79.73%
Epoch: 05 | Epoch Time: 0m 0s
        Train Loss: 0.376 | Train Acc: 87.88%
         Val. Loss: 0.506 |  Val. Acc: 81.40%
```

Finally, let's run our model on the test set!

[17]:
```
model.load_state_dict(torch.load('tut5-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.411 | Test Acc: 87.15%
```

Similar to how we made a function to predict sentiment for any given sentences, we can now make a function that will predict the class of question given.

The only difference here is that instead of using a sigmoid function to squash the input between 0 and 1, we use the `argmax` to get the highest predicted class index. We then use this index with the label vocab to get the human readable label.

[18]:
```
import spacy
nlp = spacy.load('en')

def predict_class(model, sentence, min_len = 4):
    model.eval()
    tokenized = [tok.text for tok in nlp.tokenizer(sentence)]
    if len(tokenized) < min_len:
        tokenized += ['<pad>'] * (min_len - len(tokenized))
    indexed = [TEXT.vocab.stoi[t] for t in tokenized]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(1)
    preds = model(tensor)
    max_preds = preds.argmax(dim = 1)
    return max_preds.item()
```

Now, let's try it out on a few different questions...

[19]:
```
pred_class = predict_class(model, "Who is Keyser Söze?")
print(f'Predicted class is: {pred_class} = {LABEL.vocab.itos[pred_class]}')
```

```
Predicted class is: 0 = HUM
```

[20]:
```
pred_class = predict_class(model, "How many minutes are in six hundred and␣
 ↪eighteen hours?")
print(f'Predicted class is: {pred_class} = {LABEL.vocab.itos[pred_class]}')
```

```
Predicted class is: 3 = NUM
```

[21]:
```
pred_class = predict_class(model, "What continent is Bulgaria in?")
print(f'Predicted class is: {pred_class} = {LABEL.vocab.itos[pred_class]}')
```

```
Predicted class is: 4 = LOC
```

[22]:
```
pred_class = predict_class(model, "What does WYSIWYG stand for?")
print(f'Predicted class is: {pred_class} = {LABEL.vocab.itos[pred_class]}')
```

```
Predicted class is: 5 = ABBR
```

6 - Transformers for Sentiment Analysis

September 29, 2020

1 6 - Transformers for Sentiment Analysis

In this notebook we will be using the transformer model, first introduced in this paper. Specifically, we will be using the BERT (Bidirectional Encoder Representations from Transformers) model from this paper.

Transformer models are considerably larger than anything else covered in these tutorials. As such we are going to use the transformers library to get pre-trained transformers and use them as our embedding layers. We will freeze (not train) the transformer and only train the remainder of the model which learns from the representations produced by the transformer. In this case we will be using a multi-layer bi-directional GRU, however any model can learn from these representations.

1.1 Preparing Data

First, as always, let's set the random seeds for deterministic results.

[1]:
```
import torch

import random
import numpy as np

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

The transformer has already been trained with a specific vocabulary, which means we need to train with the exact same vocabulary and also tokenize our data in the same way that the transformer did when it was initially trained.

Luckily, the transformers library has tokenizers for each of the transformer models provided. In this case we are using the BERT model which ignores casing (i.e. will lower case every word). We get this by loading the pre-trained `bert-base-uncased` tokenizer.

[2]:
```
from transformers import BertTokenizer

```

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
I1106 14:55:11.110527 139759243081536 file_utils.py:39] PyTorch version 1.3.0
available.
I1106 14:55:11.917650 139759243081536 tokenization_utils.py:374] loading file
https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-uncased-vocab.txt
from cache at /home/ben/.cache/torch/transformers/26bc1ad6c0ac742e9b52263248f6d0
f00068293b33709fae12320c0e35ccfbbb.542ce4285a40d23a559526243235df47c5f75c197f04f
37d1a0c124c32c9a084
```

The `tokenizer` has a `vocab` attribute which contains the actual vocabulary we will be using. We can check how many tokens are in it by checking its length.

[3]:
```
len(tokenizer.vocab)
```

[3]:
```
30522
```

Using the tokenizer is as simple as calling `tokenizer.tokenize` on a string. This will tokenize and lower case the data in a way that is consistent with the pre-trained transformer model.

[4]:
```
tokens = tokenizer.tokenize('Hello WORLD how ARE yoU?')

print(tokens)
```

```
['hello', 'world', 'how', 'are', 'you', '?']
```

We can numericalize tokens using our vocabulary using `tokenizer.convert_tokens_to_ids`.

[5]:
```
indexes = tokenizer.convert_tokens_to_ids(tokens)

print(indexes)
```

```
[7592, 2088, 2129, 2024, 2017, 1029]
```

The transformer was also trained with special tokens to mark the beginning and end of the sentence, detailed here. As well as a standard padding and unknown token. We can also get these from the tokenizer.

Note: the tokenizer does have a beginning of sequence and end of sequence attributes (`bos_token` and `eos_token`) but these are not set and should not be used for this transformer.

[6]:
```
init_token = tokenizer.cls_token
eos_token = tokenizer.sep_token
pad_token = tokenizer.pad_token
unk_token = tokenizer.unk_token

print(init_token, eos_token, pad_token, unk_token)
```

```
[CLS] [SEP] [PAD] [UNK]
```

We can get the indexes of the special tokens by converting them using the vocabulary...

[7]:
```
init_token_idx = tokenizer.convert_tokens_to_ids(init_token)
eos_token_idx = tokenizer.convert_tokens_to_ids(eos_token)
pad_token_idx = tokenizer.convert_tokens_to_ids(pad_token)
unk_token_idx = tokenizer.convert_tokens_to_ids(unk_token)

print(init_token_idx, eos_token_idx, pad_token_idx, unk_token_idx)
```

```
101 102 0 100
```

...or by explicitly getting them from the tokenizer.

[8]:
```
init_token_idx = tokenizer.cls_token_id
eos_token_idx = tokenizer.sep_token_id
pad_token_idx = tokenizer.pad_token_id
unk_token_idx = tokenizer.unk_token_id

print(init_token_idx, eos_token_idx, pad_token_idx, unk_token_idx)
```

```
101 102 0 100
```

Another thing we need to handle is that the model was trained on sequences with a defined maximum length - it does not know how to handle sequences longer than it has been trained on. We can get the maximum length of these input sizes by checking the `max_model_input_sizes` for the version of the transformer we want to use. In this case, it is 512 tokens.

[9]:
```
max_input_length = tokenizer.max_model_input_sizes['bert-base-uncased']

print(max_input_length)
```

```
512
```

Previously we have used the `spaCy` tokenizer to tokenize our examples. However we now need to define a function that we will pass to our `TEXT` field that will handle all the tokenization for us. It will also cut down the number of tokens to a maximum length. Note that our maximum length is 2 less than the actual maximum length. This is because we need to append two tokens to each sequence, one to the start and one to the end.

[10]:
```
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    return tokens
```

Now we define our fields. The transformer expects the batch dimension to be first, so we set `batch_first = True`. As we already have the vocabulary for our text, provided by the transformer we set `use_vocab = False` to tell torchtext that we'll be handling the vocabulary side of things. We pass our `tokenize_and_cut` function as the tokenizer. The `preprocessing` argument is a function that takes in the example after it has been tokenized, this is where we will convert the tokens to their indexes. Finally, we define the special tokens - making note that we are defining them to be their index value and not their string value, i.e. `100` instead of `[UNK]` This is because the sequences will already be converted into indexes.

We define the label field as before.

[11]:
```
from torchtext import data

TEXT = data.Field(batch_first = True,
                  use_vocab = False,
                  tokenize = tokenize_and_cut,
                  preprocessing = tokenizer.convert_tokens_to_ids,
                  init_token = init_token_idx,
                  eos_token = eos_token_idx,
                  pad_token = pad_token_idx,
                  unk_token = unk_token_idx)

LABEL = data.LabelField(dtype = torch.float)
```

We load the data and create the validation splits as before.

[12]:
```
from torchtext import datasets

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

[13]:
```
print(f"Number of training examples: {len(train_data)}")
print(f"Number of validation examples: {len(valid_data)}")
print(f"Number of testing examples: {len(test_data)}")
```

```
Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
```

We can check an example and ensure that the text has already been numericalized.

[14]:
```
print(vars(train_data.examples[6]))
```

```
{'text': [5949, 1997, 2026, 2166, 1010, 1012, 1012, 1012, 1012, 1996, 2472,
2323, 2022, 10339, 1012, 2339, 2111, 2514, 2027, 2342, 2000, 2191, 22692, 5691,
2097, 2196, 2191, 3168, 2000, 2033, 1012, 2043, 2016, 2351, 2012, 1996, 2203,
1010, 2009, 2081, 2033, 4756, 1012, 1045, 2018, 2000, 2689, 1996, 3149, 2116,
2335, 2802, 1996, 2143, 2138, 1045, 2001, 2893, 10339, 3666, 2107, 3532, 3772,
1012, 11504, 1996, 3124, 2040, 2209, 9895, 2196, 4152, 2147, 2153, 1012, 2006,
2327, 1997, 2008, 1045, 3246, 1996, 2472, 2196, 4152, 2000, 2191, 2178, 2143,
1010, 1998, 2038, 2010, 3477, 5403, 3600, 2579, 2067, 2005, 2023, 10231, 1012,
1063, 1012, 6185, 2041, 1997, 2184, 1065], 'label': 'neg'}
```

We can use the `convert_ids_to_tokens` to transform these indexes back into readable tokens.

[15]:
```
tokens = tokenizer.convert_ids_to_tokens(vars(train_data.examples[6])['text'])

print(tokens)
```

```
['waste', 'of', 'my', 'life', ',', '.', '.', '.', '.', 'the', 'director',
'should', 'be', 'embarrassed', '.', 'why', 'people', 'feel', 'they', 'need',
'to', 'make', 'worthless', 'movies', 'will', 'never', 'make', 'sense', 'to',
'me', '.', 'when', 'she', 'died', 'at', 'the', 'end', ',', 'it', 'made', 'me',
'laugh', '.', 'i', 'had', 'to', 'change', 'the', 'channel', 'many', 'times',
'throughout', 'the', 'film', 'because', 'i', 'was', 'getting', 'embarrassed',
'watching', 'such', 'poor', 'acting', '.', 'hopefully', 'the', 'guy', 'who',
'played', 'heath', 'never', 'gets', 'work', 'again', '.', 'on', 'top', 'of',
'that', 'i', 'hope', 'the', 'director', 'never', 'gets', 'to', 'make',
'another', 'film', ',', 'and', 'has', 'his', 'pay', '##che', '##ck', 'taken',
'back', 'for', 'this', 'crap', '.', '{', '.', '02', 'out', 'of', '10', '}']
```

Although we've handled the vocabulary for the text, we still need to build the vocabulary for the labels.

[16]:
```
LABEL.build_vocab(train_data)
```

[17]:
```
print(LABEL.vocab.stoi)
```

```
defaultdict(None, {'neg': 0, 'pos': 1})
```

As before, we create the iterators. Ideally we want to use the largest batch size that we can as I've found this gives the best results for transformers.

[18]:
```
BATCH_SIZE = 128

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

1.2 Build the Model

Next, we'll load the pre-trained model, making sure to load the same model as we did for the tokenizer.

[19]:
```
from transformers import BertTokenizer, BertModel

bert = BertModel.from_pretrained('bert-base-uncased')
```

```
I1106 14:57:06.877642 139759243081536 configuration_utils.py:151] loading
configuration file https://s3.amazonaws.com/models.huggingface.co/bert/bert-
base-uncased-config.json from cache at /home/ben/.cache/torch/transformers/4dad0
251492946e18ac39290fcfe91b89d370fee250efe9521476438fe8ca185.bf3b9ea126d8c0001ee8
a1e8b92229871d06d36d8808208cc2449280da87785c
I1106 14:57:06.878792 139759243081536 configuration_utils.py:168] Model config {
  "attention_probs_dropout_prob": 0.1,
```

```
  "finetuning_task": null,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "num_labels": 2,
  "output_attentions": false,
  "output_hidden_states": false,
  "output_past": true,
  "pruned_heads": {},
  "torchscript": false,
  "type_vocab_size": 2,
  "use_bfloat16": false,
  "vocab_size": 30522
}

I1106 14:57:07.421291 139759243081536 modeling_utils.py:337] loading weights
file https://s3.amazonaws.com/models.huggingface.co/bert/bert-base-uncased-
pytorch_model.bin from cache at /home/ben/.cache/torch/transformers/aa1ef1aede44
82d0dbcd4d52baad8ae300e60902e88fcb0bebdec09afd232066.36ca03ab34a1a5d5fa7bc3d03d5
5c4fa650fed07220e2eeebc06ce58d0e9a157
```

Next, we'll define our actual model.

Instead of using an embedding layer to get embeddings for our text, we'll be using the pre-trained transformer model. These embeddings will then be fed into a GRU to produce a prediction for the sentiment of the input sentence. We get the embedding dimension size (called the `hidden_size`) from the transformer via its config attribute. The rest of the initialization is standard.

Within the forward pass, we wrap the transformer in a `no_grad` to ensure no gradients are calculated over this part of the model. The transformer actually returns the embeddings for the whole sequence as well as a *pooled* output. The documentation states that the pooled output is "usually not a good summary of the semantic content of the input, you're often better with averaging or pooling the sequence of hidden-states for the whole input sequence", hence we will not be using it. The rest of the forward pass is the standard implementation of a recurrent model, where we take the hidden state over the final time-step, and pass it through a linear layer to get our predictions.

[20]:
```
import torch.nn as nn

class BERTGRUSentiment(nn.Module):
    def __init__(self,
                 bert,
                 hidden_dim,
                 output_dim,
```

```
                 n_layers,
                 bidirectional,
                 dropout):

        super().__init__()

        self.bert = bert

        embedding_dim = bert.config.to_dict()['hidden_size']

        self.rnn = nn.GRU(embedding_dim,
                          hidden_dim,
                          num_layers = n_layers,
                          bidirectional = bidirectional,
                          batch_first = True,
                          dropout = 0 if n_layers < 2 else dropout)

        self.out = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim,␣
↪output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [batch size, sent len]

        with torch.no_grad():
            embedded = self.bert(text)[0]

        #embedded = [batch size, sent len, emb dim]

        _, hidden = self.rnn(embedded)

        #hidden = [n layers * n directions, batch size, emb dim]

        if self.rnn.bidirectional:
            hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]),␣
↪dim = 1))
        else:
            hidden = self.dropout(hidden[-1,:,:])

        #hidden = [batch size, hid dim]

        output = self.out(hidden)

        #output = [batch size, out dim]
```

```
        return output
```

Next, we create an instance of our model using standard hyperparameters.

[21]:
```
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.25

model = BERTGRUSentiment(bert,
                         HIDDEN_DIM,
                         OUTPUT_DIM,
                         N_LAYERS,
                         BIDIRECTIONAL,
                         DROPOUT)
```

We can check how many parameters the model has. Our standard models have under 5M, but this one has 112M! Luckily, 110M of these parameters are from the transformer and we will not be training those.

[22]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 112,241,409 trainable parameters
```

In order to freeze paramers (not train them) we need to set their `requires_grad` attribute to `False`. To do this, we simply loop through all of the `named_parameters` in our model and if they're a part of the `bert` transformer model, we set `requires_grad = False`.

[23]:
```
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

We can now see that our model has under 3M trainable parameters, making it almost comparable to the `FastText` model. However, the text still has to propagate through the transformer which causes training to take considerably longer.

[24]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 2,759,169 trainable parameters
```

We can double check the names of the trainable parameters, ensuring they make sense. As we can see, they are all the parameters of the GRU (`rnn`) and the linear layer (`out`).

[25]:
```
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name)
```

```
rnn.weight_ih_l0
rnn.weight_hh_l0
rnn.bias_ih_l0
rnn.bias_hh_l0
rnn.weight_ih_l0_reverse
rnn.weight_hh_l0_reverse
rnn.bias_ih_l0_reverse
rnn.bias_hh_l0_reverse
rnn.weight_ih_l1
rnn.weight_hh_l1
rnn.bias_ih_l1
rnn.bias_hh_l1
rnn.weight_ih_l1_reverse
rnn.weight_hh_l1_reverse
rnn.bias_ih_l1_reverse
rnn.bias_hh_l1_reverse
out.weight
out.bias
```

1.3 Train the Model

As is standard, we define our optimizer and criterion (loss function).

[26]:
```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())
```

[27]:
```
criterion = nn.BCEWithLogitsLoss()
```

Place the model and criterion onto the GPU (if available)

[28]:
```
model = model.to(device)
criterion = criterion.to(device)
```

Next, we'll define functions for: calculating accuracy, performing a training epoch, performing an evaluation epoch and calculating how long a training/evaluation epoch takes.

[29]:
```
def binary_accuracy(preds, y):
    """
    Returns accuracy per batch, i.e. if you get 8/10 right, this returns 0.8,␣
 ↪NOT 8
    """
```

```
    #round predictions to the closest integer
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct)
    return acc
```

[30]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[31]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)
```

```
            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[32]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

Finally, we'll train our model. This takes considerably longer than any of the previous models due to the size of the transformer. Even though we are not training any of the transformer's parameters we still need to pass the data through the model which takes a considerable amount of time on a standard GPU.

[34]:
```
N_EPOCHS = 5

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut6-model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 7m 27s
        Train Loss: 0.286 | Train Acc: 88.16%
         Val. Loss: 0.247 |  Val. Acc: 90.26%
Epoch: 02 | Epoch Time: 7m 27s
        Train Loss: 0.234 | Train Acc: 90.77%
```

```
         Val. Loss: 0.229 |  Val. Acc: 91.00%
Epoch: 03 | Epoch Time: 7m 27s
        Train Loss: 0.209 | Train Acc: 91.83%
         Val. Loss: 0.225 |  Val. Acc: 91.10%
Epoch: 04 | Epoch Time: 7m 27s
        Train Loss: 0.182 | Train Acc: 92.97%
         Val. Loss: 0.217 |  Val. Acc: 91.98%
Epoch: 05 | Epoch Time: 7m 27s
        Train Loss: 0.156 | Train Acc: 94.17%
         Val. Loss: 0.230 |  Val. Acc: 91.76%
```

We'll load up the parameters that gave us the best validation loss and try these on the test set - which gives us our best results so far!

[35]:
```
model.load_state_dict(torch.load('tut6-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.198 | Test Acc: 92.31%
```

1.4 Inference

We'll then use the model to test the sentiment of some sequences. We tokenize the input sequence, trim it down to the maximum length, add the special tokens to either side, convert it to a tensor, add a fake batch dimension and then pass it through our model.

[36]:
```
def predict_sentiment(model, tokenizer, sentence):
    model.eval()
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    indexed = [init_token_idx] + tokenizer.convert_tokens_to_ids(tokens) +␣
 ↪[eos_token_idx]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(0)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()
```

[37]:
```
predict_sentiment(model, tokenizer, "This film is terrible")
```

[37]:
```
0.02264496125280857
```

[38]:
```
predict_sentiment(model, tokenizer, "This film is great")
```

[38]:
```
0.9411056041717529
```

A - Using TorchText with Your Own Datasets

September 29, 2020

1 A - Using TorchText with Your Own Datasets

In this series we have used the IMDb dataset included as a dataset in TorchText. TorchText has many canonical datasets included for classification, language modelling, sequence tagging, etc. However, frequently you'll be wanting to use your own datasets. Luckily, TorchText has functions to help you to this.

Recall in the series, we: - defined the `Field`s - loaded the dataset - created the splits

As a reminder, the code is shown below:

```
TEXT = data.Field()
LABEL = data.LabelField()

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split()
```

There are three data formats TorchText can read: `json`, `tsv` (tab separated values) and`csv` (comma separated values).

In my opinion, the best formatting for TorchText is `json`, which I'll explain later on.

1.1 Reading JSON

Starting with `json`, your data must be in the `json lines` format, i.e. it must be something like:

```
{"name": "John", "location": "United Kingdom", "age": 42, "quote": ["i", "love", "the", "unite
{"name": "Mary", "location": "United States", "age": 36, "quote": ["i", "want", "more", "teles
```

That is, each line is a `json` object. See `data/train.json` for an example.

We then define the fields:

[1]:
```
from torchtext import data
from torchtext import datasets

NAME = data.Field()
SAYING = data.Field()
PLACE = data.Field()
```

Next, we must tell TorchText which fields apply to which elements of the `json` object.

For `json` data, we must create a dictionary where: - the key matches the key of the `json` object - the value is a tuple where: - the first element becomes the batch object's attribute name - the second element is the name of the `Field`

What do we mean when we say "becomes the batch object's attribute name"? Recall in the previous exercises where we accessed the `TEXT` and `LABEL` fields in the train/evaluation loop by using `batch.text` and `batch.label`, this is because TorchText sets the batch object to have a `text` and `label` attribute, each being a tensor containing either the text or the label.

A few notes:

- The order of the keys in the `fields` dictionary does not matter, as long as its keys match the `json` data keys.
- The `Field` name does not have to match the key in the `json` object, e.g. we use `PLACE` for the `"location"` field.
- When dealing with `json` data, not all of the keys have to be used, e.g. we did not use the `"age"` field.
- Also, if the values of `json` field are a string then the `Fields` tokenization is applied (default is to split the string on spaces), however if the values are a list then no tokenization is applied. Usually it is a good idea for the data to already be tokenized into a list, this saves time as you don't have to wait for TorchText to do it.
- The value of the `json` fields do not have to be the same type. Some examples can have their `"quote"` as a string, and some as a list. The tokenization will only get applied to the ones with their `"quote"` as a string.
- If you are using a `json` field, every single example must have an instance of that field, e.g. in this example all examples must have a name, location and quote. However, as we are not using the age field, it does not matter if an example does not have it.

[2]:
```
fields = {'name': ('n', NAME), 'location': ('p', PLACE), 'quote': ('s', SAYING)}
```

Now, in a training loop we can iterate over the data iterator and access the name via `batch.n`, the location via `batch.p`, and the quote via `batch.s`.

We then create our datasets (`train_data` and `test_data`) with the `TabularDataset.splits` function.

The `path` argument specifices the top level folder common among both datasets, and the `train` and `test` arguments specify the filename of each dataset, e.g. here the train dataset is located at `data/train.json`.

We tell the function we are using `json` data, and pass in our `fields` dictionary defined previously.

[3]:
```
train_data, test_data = data.TabularDataset.splits(
                            path = 'data',
                            train = 'train.json',
                            test = 'test.json',
                            format = 'json',
```

```
                                fields = fields
)
```

If you already had a validation dataset, the location of this can be passed as the `validation` argument.

[4]:
```
train_data, valid_data, test_data = data.TabularDataset.splits(
                                        path = 'data',
                                        train = 'train.json',
                                        validation = 'valid.json',
                                        test = 'test.json',
                                        format = 'json',
                                        fields = fields
)
```

We can then view an example to make sure it has worked correctly.

Notice how the field names (`n`, `p` and `s`) match up with what was defined in the `fields` dictionary.

Also notice how the word `"United Kingdom"` in `p` has been split by the tokenization, whereas the `"united kingdom"` in `s` has not. This is due to what was mentioned previously, where TorchText assumes that any `json` fields that are lists are already tokenized and no further tokenization is applied.

[5]:
```
print(vars(train_data[0]))
```

```
{'n': ['John'], 'p': ['United', 'Kingdom'], 's': ['i', 'love', 'the', 'united
kingdom']}
```

We can now use `train_data`, `test_data` and `valid_data` to build a vocabulary and create iterators, as in the other notebooks. We can access all attributes by using `batch.n`, `batch.p` and `batch.s` for the names, places and sayings, respectively.

1.2 Reading CSV/TSV

`csv` and `tsv` are very similar, except csv has elements separated by commas and tsv by tabs.

Using the same example above, our `tsv` data will be in the form of:

```
name    location    age quote
John    United Kingdom  42  i love the united kingdom
Mary    United States   36  i want more telescopes
```

That is, on each row the elements are separated by tabs and we have one example per row. The first row is usually a header (i.e. the name of each of the columns), but your data could have no header.

You cannot have lists within `tsv` or `csv` data.

The way the fields are defined is a bit different to `json`. We now use a list of tuples, where each element is also a tuple. The first element of these inner tuples will become the batch object's

attribute name, second element is the `Field` name.

Unlike the `json` data, the tuples have to be in the same order that they are within the `tsv` data. Due to this, when skipping a column of data a tuple of `None`s needs to be used, if not then our `SAYING` field will be applied to the `age` column of the `tsv` data and the `quote` column will not be used.

However, if you only wanted to use the `name` and `age` column, you could just use two tuples as they are the first two columns.

We change our `TabularDataset` to read the correct `.tsv` files, and change the `format` argument to `'tsv'`.

If your data has a header, which ours does, it must be skipped by passing `skip_header = True`. If not, TorchText will think the header is an example. By default, `skip_header` will be `False`.

[6]:
```
fields = [('n', NAME), ('p', PLACE), (None, None), ('s', SAYING)]
```

[7]:
```
train_data, valid_data, test_data = data.TabularDataset.splits(
                                        path = 'data',
                                        train = 'train.tsv',
                                        validation = 'valid.tsv',
                                        test = 'test.tsv',
                                        format = 'tsv',
                                        fields = fields,
                                        skip_header = True
)
```

[8]:
```
print(vars(train_data[0]))
```

```
{'n': ['John'], 'p': ['United', 'Kingdom'], 's': ['i', 'love', 'the', 'united',
'kingdom']}
```

Finally, we'll cover `csv` files.

This is pretty much the exact same as the `tsv` files, expect with the `format` argument set to `'csv'`.

[9]:
```
fields = [('n', NAME), ('p', PLACE), (None, None), ('s', SAYING)]
```

[10]:
```
train_data, valid_data, test_data = data.TabularDataset.splits(
                                        path = 'data',
                                        train = 'train.csv',
                                        validation = 'valid.csv',
                                        test = 'test.csv',
                                        format = 'csv',
                                        fields = fields,
                                        skip_header = True
)
```

[11]:
```
print(vars(train_data[0]))
```

```
{'n': ['John'], 'p': ['United', 'Kingdom'], 's': ['i', 'love', 'the', 'united',
'kingdom']}
```

1.3 Why JSON over CSV/TSV?

1. Your `csv` or `tsv` data cannot be stored lists. This means data cannot be already be tokenized, thus everytime you run your Python script that reads this data via TorchText, it has to be tokenized. Using advanced tokenizers, such as the `spaCy` tokenizer, takes a non-negligible amount of time. Thus, it is better to tokenize your datasets and store them in the `json lines` format.
2. If tabs appear in your `tsv` data, or commas appear in your `csv` data, TorchText will think they are delimiters between columns. This will cause your data to be parsed incorrectly. Worst of all TorchText will not alert you to this as it cannot tell the difference between a tab/comma in a field and a tab/comma as a delimiter. As `json` data is essentially a dictionary, you access the data within the fields via its key, so do not have to worry about "surprise" delimiters.

1.4 Iterators

Using any of the above datasets, we can then build the vocab and create the iterators.

[12]:
```
NAME.build_vocab(train_data)
SAYING.build_vocab(train_data)
PLACE.build_vocab(train_data)
```

Then, we can create the iterators after defining our batch size and device.

By default, the train data is shuffled each epoch, but the validation/test data is sorted. However, TorchText doesn't know what to use to sort our data and it would throw an error if we don't tell it.

There are two ways to handle this, you can either tell the iterator not to sort the validation/test data by passing `sort = False`, or you can tell it how to sort the data by passing a `sort_key`. A sort key is a function that returns a key on which to sort the data on. For example, `lambda x: x.s` will sort the examples by their `s` attribute, i.e their quote. Ideally, you want to use a sort key as the `BucketIterator` will then be able to sort your examples and then minimize the amount of padding within each batch.

We can then iterate over our iterator to get batches of data. Note how by default TorchText has the batch dimension second.

[13]:
```
import torch

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

BATCH_SIZE = 1

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
```

```
    sort = False, #don't sort test/validation data
    batch_size=BATCH_SIZE,
    device=device)

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    sort_key = lambda x: x.s, #sort by s attribute (quote)
    batch_size=BATCH_SIZE,
    device=device)

print('Train:')
for batch in train_iterator:
    print(batch)

print('Valid:')
for batch in valid_iterator:
    print(batch)

print('Test:')
for batch in test_iterator:
    print(batch)
```

```
Train:

[torchtext.data.batch.Batch of size 1]
        [.n]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.p]:[torch.cuda.LongTensor of size 2x1 (GPU 0)]
        [.s]:[torch.cuda.LongTensor of size 5x1 (GPU 0)]

[torchtext.data.batch.Batch of size 1]
        [.n]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.p]:[torch.cuda.LongTensor of size 2x1 (GPU 0)]
        [.s]:[torch.cuda.LongTensor of size 4x1 (GPU 0)]
Valid:

[torchtext.data.batch.Batch of size 1]
        [.n]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.p]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.s]:[torch.cuda.LongTensor of size 2x1 (GPU 0)]

[torchtext.data.batch.Batch of size 1]
        [.n]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.p]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.s]:[torch.cuda.LongTensor of size 4x1 (GPU 0)]
Test:

[torchtext.data.batch.Batch of size 1]
```

```
        [.n]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.p]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.s]:[torch.cuda.LongTensor of size 3x1 (GPU 0)]

[torchtext.data.batch.Batch of size 1]
        [.n]:[torch.cuda.LongTensor of size 1x1 (GPU 0)]
        [.p]:[torch.cuda.LongTensor of size 2x1 (GPU 0)]
        [.s]:[torch.cuda.LongTensor of size 3x1 (GPU 0)]
```

B - A Closer Look at Word Embeddings

September 29, 2020

1 B - A Closer Look at Word Embeddings

We have very briefly covered how word embeddings (also known as word vectors) are used in the tutorials. In this appendix we'll have a closer look at these embeddings and find some (hopefully) interesting results.

Embeddings transform a one-hot encoded vector (a vector that is 0 in elements except one, which is 1) into a much smaller dimension vector of real numbers. The one-hot encoded vector is also known as a *sparse vector*, whilst the real valued vector is known as a *dense vector*.

The key concept in these word embeddings is that words that appear in similar *contexts* appear nearby in the vector space, i.e. the Euclidean distance between these two word vectors is small. By context here, we mean the surrounding words. For example in the sentences "I purchased some items at the shop" and "I purchased some items at the store" the words 'shop' and 'store' appear in the same context and thus should be close together in vector space.

You may have also heard about *word2vec*. *word2vec* is an algorithm (actually a bunch of algorithms) that calculates word vectors from a corpus. In this appendix we use *GloVe* vectors, *GloVe* being another algorithm to calculate word vectors. If you want to know how *word2vec* works, check out a two part series here and here, and if you want to find out more about *GloVe*, check the website here.

In PyTorch, we use word vectors with the `nn.Embedding` layer, which takes a ***[sentence length, batch size]*** tensor and transforms it into a ***[sentence length, batch size, embedding dimensions]*** tensor.

In tutorial 2 onwards, we also used pre-trained word embeddings (specifically the GloVe vectors) provided by TorchText. These embeddings have been trained on a gigantic corpus. We can use these pre-trained vectors within any of our models, with the idea that as they have already learned the context of each word they will give us a better starting point for our word vectors. This usually leads to faster training time and/or improved accuracy.

In this appendix we won't be training any models, instead we'll be looking at the word embeddings and finding a few interesting things about them.

A lot of the code from the first half of this appendix is taken from here. For more information about word embeddings, go here.

1.1 Loading the GloVe vectors

First, we'll load the GloVe vectors. The `name` field specifies what the vectors have been trained on, here the `6B` means a corpus of 6 billion words. The `dim` argument specifies the dimensionality of the word vectors. GloVe vectors are available in 50, 100, 200 and 300 dimensions. There is also a `42B` and `840B` glove vectors, however they are only available at 300 dimensions.

[1]:
```
import torchtext.vocab

glove = torchtext.vocab.GloVe(name = '6B', dim = 100)

print(f'There are {len(glove.itos)} words in the vocabulary')
```

```
There are 400000 words in the vocabulary
```

As shown above, there are 400,000 unique words in the GloVe vocabulary. These are the most common words found in the corpus the vectors were trained on. **In these set of GloVe vectors, every single word is lower-case only.**

`glove.vectors` is the actual tensor containing the values of the embeddings.

[2]:
```
glove.vectors.shape
```

[2]:
```
torch.Size([400000, 100])
```

We can see what word is associated with each row by checking the `itos` (int to string) list.

Below implies that row 0 is the vector associated with the word 'the', row 1 for ',' (comma), row 2 for '.' (period), etc.

[3]:
```
glove.itos[:10]
```

[3]:
```
['the', ',', '.', 'of', 'to', 'and', 'in', 'a', '"', "'s"]
```

We can also use the `stoi` (string to int) dictionary, in which we input a word and receive the associated integer/index. If you try get the index of a word that is not in the vocabulary, you receive an error.

[4]:
```
glove.stoi['the']
```

[4]:
```
0
```

We can get the vector of a word by first getting the integer associated with it and then indexing into the word embedding tensor with that index.

[5]:
```
glove.vectors[glove.stoi['the']].shape
```

[5]:
```
torch.Size([100])
```

We'll be doing this a lot, so we'll create a function that takes in word embeddings and a word then returns the associated vector. It'll also throw an error if the word doesn't exist in the vocabulary.

[6]:
```
def get_vector(embeddings, word):
    assert word in embeddings.stoi, f'*{word}* is not in the vocab!'
    return embeddings.vectors[embeddings.stoi[word]]
```

As before, we use a word to get the associated vector.

[7]:
```
get_vector(glove, 'the').shape
```

[7]:
```
torch.Size([100])
```

1.2 Similar Contexts

Now to start looking at the context of different words.

If we want to find the words similar to a certain input word, we first find the vector of this input word, then we scan through our vocabulary calculating the distance between the vector of each word and our input word vector. We then sort these from closest to furthest away.

The function below returns the closest 10 words to an input word vector:

[8]:
```
import torch

def closest_words(embeddings, vector, n = 10):

    distances = [(word, torch.dist(vector, get_vector(embeddings, word)).item())
                 for word in embeddings.itos]

    return sorted(distances, key = lambda w: w[1])[:n]
```

Let's try it out with 'korea'. The closest word is the word 'korea' itself (not very interesting), however all of the words are related in some way. Pyongyang is the capital of North Korea, DPRK is the official name of North Korea, etc.

Interestingly, we also get 'Japan' and 'China', implies that Korea, Japan and China are frequently talked about together in similar contexts. This makes sense as they are geographically situated near each other.

[9]:
```
word_vector = get_vector(glove, 'korea')

closest_words(glove, word_vector)
```

[9]:
```
[('korea', 0.0),
 ('pyongyang', 3.9039554595947266),
 ('korean', 4.068886756896973),
 ('dprk', 4.2631049156188965),
 ('seoul', 4.340494155883789),
 ('japan', 4.551243782043457),
 ('koreans', 4.615609169006348),
 ('south', 4.65822696685791),
```

```
 ('china', 4.839518070220947),
 ('north', 4.986356735229492)]
```

Looking at another country, India, we also get nearby countries: Thailand, Malaysia and Sri Lanka (as two separate words). Australia is relatively close to India (geographically), but Thailand and Malaysia are closer. So why is Australia closer to India in vector space? This is most probably due to India and Australia appearing in the context of cricket matches together.

[10]:
```
word_vector = get_vector(glove, 'india')

closest_words(glove, word_vector)
```

[10]:
```
[('india', 0.0),
 ('pakistan', 3.6954822540283203),
 ('indian', 4.114313125610352),
 ('delhi', 4.155975818634033),
 ('bangladesh', 4.261017799377441),
 ('lanka', 4.435845851898193),
 ('sri', 4.515716552734375),
 ('australia', 4.806082725524902),
 ('thailand', 4.994781017303467),
 ('malaysia', 5.009334087371826)]
```

We'll also create another function that will nicely print out the tuples returned by our `closest_words` function.

[11]:
```
def print_tuples(tuples):
    for w, d in tuples:
        print(f'({d:02.04f}) {w}')
```

A final word to look at, 'sports'. As we can see, the closest words are most of the sports themselves.

[12]:
```
word_vector = get_vector(glove, 'sports')

print_tuples(closest_words(glove, word_vector))
```

```
(0.0000) sports
(3.5875) sport
(4.4590) soccer
(4.6508) basketball
(4.6561) baseball
(4.8028) sporting
(4.8763) football
(4.9624) professional
(4.9824) entertainment
(5.0975) media
```

1.3 Analogies

Another property of word embeddings is that they can be operated on just as any standard vector and give interesting results.

We'll show an example of this first, and then explain it:

[13]:
```
def analogy(embeddings, word1, word2, word3, n=5):

    #get vectors for each word
    word1_vector = get_vector(embeddings, word1)
    word2_vector = get_vector(embeddings, word2)
    word3_vector = get_vector(embeddings, word3)

    #calculate analogy vector
    analogy_vector = word2_vector - word1_vector + word3_vector

    #find closest words to analogy vector
    candidate_words = closest_words(embeddings, analogy_vector, n+3)

    #filter out words already in analogy
    candidate_words = [(word, dist) for (word, dist) in candidate_words
                       if word not in [word1, word2, word3]][:n]

    print(f'{word1} is to {word2} as {word3} is to...')

    return candidate_words
```

[14]:
```
print_tuples(analogy(glove, 'man', 'king', 'woman'))
```

```
man is to king as woman is to…
(4.0811) queen
(4.6429) monarch
(4.9055) throne
(4.9216) elizabeth
(4.9811) prince
```

This is the canonical example which shows off this property of word embeddings. So why does it work? Why does the vector of 'woman' added to the vector of 'king' minus the vector of 'man' give us 'queen'?

If we think about it, the vector calculated from 'king' minus 'man' gives us a "royalty vector". This is the vector associated with traveling from a man to his royal counterpart, a king. If we add this "royality vector" to 'woman', this should travel to her royal equivalent, which is a queen!

We can do this with other analogies too. For example, this gets an "acting career vector":

[15]:
```
print_tuples(analogy(glove, 'man', 'actor', 'woman'))
```

```
man is to actor as woman is to…
```

```
(2.8133) actress
(5.0039) comedian
(5.1399) actresses
(5.2773) starred
(5.3085) screenwriter
```

For a ”baby animal vector”:

[16]:
```
print_tuples(analogy(glove, 'cat', 'kitten', 'dog'))
```

```
cat is to kitten as dog is to…
(3.8146) puppy
(4.2944) rottweiler
(4.5888) puppies
(4.6086) pooch
(4.6520) pug
```

A ”capital city vector”:

[17]:
```
print_tuples(analogy(glove, 'france', 'paris', 'england'))
```

```
france is to paris as england is to…
(4.1426) london
(4.4938) melbourne
(4.7087) sydney
(4.7630) perth
(4.7952) birmingham
```

A ”musician’s genre vector”:

[18]:
```
print_tuples(analogy(glove, 'elvis', 'rock', 'eminem'))
```

```
elvis is to rock as eminem is to…
(5.6597) rap
(6.2057) rappers
(6.2161) rapper
(6.2444) punk
(6.2690) hop
```

And an ”ingredient vector”:

[19]:
```
print_tuples(analogy(glove, 'beer', 'barley', 'wine'))
```

```
beer is to barley as wine is to…
(5.6021) grape
(5.6760) beans
(5.8174) grapes
(5.9035) lentils
(5.9454) figs
```

1.4 Correcting Spelling Mistakes

Another interesting property of word embeddings is that they can actually be used to correct spelling mistakes!

We'll put their findings into code and briefly explain them, but to read more about this, check out the original thread and the associated write-up.

First, we need to load up the much larger vocabulary GloVe vectors, this is due to the spelling mistakes not appearing in the smaller vocabulary.

Note: these vectors are very large (~2GB), so watch out if you have a limited internet connection.

[20]:
```
glove = torchtext.vocab.GloVe(name = '840B', dim = 300)
```

Checking the vocabulary size of these embeddings, we can see we now have over 2 million unique words in our vocabulary!

[21]:
```
glove.vectors.shape
```

[21]:
```
torch.Size([2196017, 300])
```

As the vectors were trained with a much larger vocabulary on a larger corpus of text, the words that appear are a little different. Notice how the words 'north', 'south', 'pyongyang' and 'dprk' no longer appear in the most closest words to 'korea'.

[22]:
```
word_vector = get_vector(glove, 'korea')

print_tuples(closest_words(glove, word_vector))
```

```
(0.0000) korea
(3.9857) taiwan
(4.4022) korean
(4.9016) asia
(4.9593) japan
(5.0721) seoul
(5.4058) thailand
(5.6025) singapore
(5.7010) russia
(5.7240) hong
```

Our first step to correcting spelling mistakes is looking at the vector for a misspelling of the word 'reliable'.

[23]:
```
word_vector = get_vector(glove, 'relieable')

print_tuples(closest_words(glove, word_vector))
```

```
(0.0000) relieable
(5.0366) relyable
(5.2610) realible
```

```
(5.4719) realiable
(5.5402) relable
(5.5917) relaible
(5.6412) reliabe
(5.8802) relaiable
(5.9593) stabel
(5.9981) consitant
```

Notice how the correct spelling, ”reliable”, does not appear in the top 10 closest words. Surely the misspellings of a word should appear next to the correct spelling of the word as they appear in the same context, right?

The hypothesis is that misspellings of words are all equally shifted away from their correct spelling. This is because articles of text that contain spelling mistakes are usually written in an informal manner where correct spelling doesn't matter as much (such as tweets/blog posts), thus spelling errors will appear together as they appear in context of informal articles.

Similar to how we created analogies before, we can create a ”correct spelling” vector. This time, instead of using a single example to create our vector, we'll use the average of multiple examples. This will hopefully give better accuracy!

We first create a vector for the correct spelling, 'reliable', then calculate the difference between the ”reliable vector” and each of the 8 misspellings of 'reliable'. As we are going to concatenate these 8 misspelling tensors together we need to unsqueeze a ”batch” dimension to them.

[24]:
```
reliable_vector = get_vector(glove, 'reliable')

reliable_misspellings = ['relieable', 'relyable', 'realible', 'realiable',
                         'relable', 'relaible', 'reliabe', 'relaiable']

diff_reliable = [(reliable_vector - get_vector(glove, s)).unsqueeze(0)
                 for s in reliable_misspellings]
```

We take the average of these 8 'difference from reliable' vectors to get our ”misspelling vector”.

[25]:
```
misspelling_vector = torch.cat(diff_reliable, dim = 0).mean(dim = 0)
```

We can now correct other spelling mistakes using this ”misspelling vector” by finding the closest words to the sum of the vector of a misspelled word and the ”misspelling vector”.

For a misspelling of ”because”:

[26]:
```
word_vector = get_vector(glove, 'becuase')

print_tuples(closest_words(glove, word_vector + misspelling_vector))
```

```
(6.1090) because
(6.4250) even
(6.4358) fact
(6.4914) sure
(6.5094) though
```

```
(6.5601) obviously
(6.5682) reason
(6.5856) if
(6.6099) but
(6.6415) why
```

For a misspelling of ”definitely”:

[27]:
```
word_vector = get_vector(glove, 'defintiely')

print_tuples(closest_words(glove, word_vector + misspelling_vector))
```

```
(5.4070) definitely
(5.5643) certainly
(5.7192) sure
(5.8152) well
(5.8588) always
(5.8812) also
(5.9557) simply
(5.9667) consider
(5.9821) probably
(5.9948) definately
```

For a misspelling of ”consistent”:

[28]:
```
word_vector = get_vector(glove, 'consistant')

print_tuples(closest_words(glove, word_vector + misspelling_vector))
```

```
(5.9641) consistent
(6.3674) reliable
(7.0195) consistant
(7.0299) consistently
(7.1605) accurate
(7.2737) fairly
(7.3037) good
(7.3520) reasonable
(7.3801) dependable
(7.4027) ensure
```

For a misspelling of ”package”:

[29]:
```
word_vector = get_vector(glove, 'pakage')

print_tuples(closest_words(glove, word_vector + misspelling_vector))
```

```
(6.6117) package
(6.9315) packages
(7.0195) pakage
(7.0911) comes
```

```
(7.1241) provide
(7.1469) offer
(7.1861) reliable
(7.2431) well
(7.2434) choice
(7.2453) offering
```

For a more in-depth look at this, check out the write-up.

C - Loading, Saving and Freezing Embeddings

September 29, 2020

1 C - Loading, Saving and Freezing Embeddings

This notebook will cover: how to load custom word embeddings in TorchText, how to save all the embeddings we learn during training and how to freeze/unfreeze embeddings during training.

1.1 Loading Custom Embeddings

First, lets look at loading a custom set of embeddings.

Your embeddings need to be formatted so each line starts with the word followed by the values of the embedding vector, all space separated. All vectors need to have the same number of elements.

Let's look at the custom embeddings provided by these tutorials. These are 20-dimensional embeddings for 7 words.

[1]:
```
with open('custom_embeddings/embeddings.txt', 'r') as f:
    print(f.read())
```

```
good 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0
great 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0
awesome 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0
1.0 1.0
bad -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0 -1.0
terrible -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0 -1.0 -1.0
awful -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0
-1.0 -1.0 -1.0 -1.0 -1.0
kwyjibo 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5 0.5 -0.5
0.5 -0.5 0.5 -0.5
```

Now, let's setup the fields.

[2]:
```
import torch
from torchtext import data
```

```
SEED = 1234

torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

TEXT = data.Field(tokenize = 'spacy')
LABEL = data.LabelField(dtype = torch.float)
```

Then, we'll load our dataset and create the validation set.

[3]:
```
from torchtext import datasets
import random

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

We can only load our custom embeddings after they have been turned into a `Vectors` object.

We create a `Vector` object by passing it the location of the embeddings (`name`), a location for the cached embeddings (`cache`) and a function that will later initialize tokens in our embeddings that aren't within our dataset (`unk_init`). As have done in previous notebooks, we have initialized these to $\mathcal{N}(0, 1)$.

[4]:
```
import torchtext.vocab as vocab

custom_embeddings = vocab.Vectors(name = 'custom_embeddings/embeddings.txt',
                                  cache = 'custom_embeddings',
                                  unk_init = torch.Tensor.normal_)
```

```
  0%|          | 0/7 [00:00<?, ?it/s]
```

To check the embeddings have loaded correctly we can print out the words loaded from our custom embedding.

[5]:
```
print(custom_embeddings.stoi)
```

```
{'good': 0, 'great': 1, 'awesome': 2, 'bad': 3, 'terrible': 4, 'awful': 5,
'kwyjibo': 6}
```

We can also directly print out the embedding values.

[6]:
```
print(custom_embeddings.vectors)
```

```
tensor([[ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000],
        [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000],
```

```
        [ 1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,  1.0000,
          1.0000,  1.0000,  1.0000,  1.0000],
        [-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000],
        [-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000],
        [-1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000,
         -1.0000, -1.0000, -1.0000, -1.0000],
        [ 0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,
          0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,  0.5000, -0.5000,
          0.5000, -0.5000,  0.5000, -0.5000]])
```

We then build our vocabulary, passing our `Vectors` object.

Note that the `unk_init` should be declared when creating our `Vectors`, and not here!

[7]:
```
MAX_VOCAB_SIZE = 25_000

TEXT.build_vocab(train_data,
                 max_size = MAX_VOCAB_SIZE,
                 vectors = custom_embeddings)

LABEL.build_vocab(train_data)
```

Now our vocabulary vectors for the words in our custom embeddings should match what we loaded.

[8]:
```
TEXT.vocab.vectors[TEXT.vocab.stoi['good']]
```

[8]:
```
tensor([1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
        1., 1.])
```

[9]:
```
TEXT.vocab.vectors[TEXT.vocab.stoi['bad']]
```

[9]:
```
tensor([-1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1., -1.,
        -1., -1., -1., -1., -1., -1.])
```

Words that were in our custom embeddings but not in our dataset vocabulary are initialized by the `unk_init` function we passed earlier, $\mathcal{N}(0,1)$. They are also the same size as our custom embeddings (20-dimensional).

[10]:
```
TEXT.vocab.vectors[TEXT.vocab.stoi['kwjibo']]
```

[10]:
```
tensor([-0.1117, -0.4966,  0.1631, -0.8817,  0.2891,  0.4899, -0.3853, -0.7120,
         0.6369, -0.7141, -1.0831, -0.5547, -1.3248,  0.6970, -0.6631,  1.2158,
        -2.5273,  1.4778, -0.1696, -0.9919])
```

The rest of the set-up is the same as it is when using the GloVe vectors, with the next step being to set-up the iterators.

[11]:
```
BATCH_SIZE = 64

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

Then, we define our model.

[12]:
```
import torch.nn as nn
import torch.nn.functional as F

class CNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, n_filters, filter_sizes,␣
 ↪output_dim,
                 dropout, pad_idx):
        super().__init__()

        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx =␣
 ↪pad_idx)

        self.convs = nn.ModuleList([
                                    nn.Conv2d(in_channels = 1,
                                              out_channels = n_filters,
                                              kernel_size = (fs, embedding_dim))
                                    for fs in filter_sizes
                                    ])

        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [sent len, batch size]

        text = text.permute(1, 0)

        #text = [batch size, sent len]

        embedded = self.embedding(text)

        #embedded = [batch size, sent len, emb dim]
```

```
        embedded = embedded.unsqueeze(1)

        #embedded = [batch size, 1, sent len, emb dim]

        conved = [F.relu(conv(embedded)).squeeze(3) for conv in self.convs]

        #conv_n = [batch size, n_filters, sent len - filter_sizes[n]]

        pooled = [F.max_pool1d(conv, conv.shape[2]).squeeze(2) for conv in␣
↪conved]

        #pooled_n = [batch size, n_filters]

        cat = self.dropout(torch.cat(pooled, dim = 1))

        #cat = [batch size, n_filters * len(filter_sizes)]

        return self.fc(cat)
```

We then initialize our model, making sure `EMBEDDING_DIM` is the same as our custom embedding dimensionality, i.e. 20.

[13]:
```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 20
N_FILTERS = 100
FILTER_SIZES = [3,4,5]
OUTPUT_DIM = 1
DROPOUT = 0.5
PAD_IDX = TEXT.vocab.stoi[TEXT.pad_token]

model = CNN(INPUT_DIM, EMBEDDING_DIM, N_FILTERS, FILTER_SIZES, OUTPUT_DIM,␣
↪DROPOUT, PAD_IDX)
```

We have a lot less parameters in this model due to the smaller embedding size used.

[14]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 524,641 trainable parameters
```

Next, we initialize our embedding layer to use our vocabulary vectors.

[15]:
```
embeddings = TEXT.vocab.vectors

model.embedding.weight.data.copy_(embeddings)
```

[15]:
```
tensor([[-0.1117, -0.4966,  0.1631,  …,  1.4778, -0.1696, -0.9919],
        [-0.5675, -0.2772, -2.1834,  …,  0.8504,  1.0534,  0.3692],
        [-0.0552, -0.6125,  0.7500,  …, -0.1261, -1.6770,  1.2068],
        …,
        [ 0.5383, -0.1504,  1.6720,  …, -0.3857, -1.0168,  0.1849],
        [ 2.5640, -0.8564, -0.0219,  …, -0.3389,  0.2203, -1.6119],
        [ 0.1203,  1.5286,  0.6824,  …,  0.3330, -0.6704,  0.5883]])
```

Then, we initialize the unknown and padding token embeddings to all zeros.

[16]:
```
UNK_IDX = TEXT.vocab.stoi[TEXT.unk_token]

model.embedding.weight.data[UNK_IDX] = torch.zeros(EMBEDDING_DIM)
model.embedding.weight.data[PAD_IDX] = torch.zeros(EMBEDDING_DIM)
```

Following standard procedure, we create our optimizer.

[17]:
```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())
```

Define our loss function (criterion).

[18]:
```
criterion = nn.BCEWithLogitsLoss()
```

Then place the loss function and the model on the GPU.

[19]:
```
model = model.to(device)
criterion = criterion.to(device)
```

Create the function to calculate accuracy.

[20]:
```
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)
    return acc
```

Then implement our training function...

[21]:
```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:
```

```
        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

...evaluation function...

[22]:
```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

...and our helpful function that tells us how long an epoch takes.

[23]:
```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
```

```
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

We've finally reached training our model!

1.2 Freezing and Unfreezing Embeddings

We're going to train our model for 10 epochs. During the first 5 epochs we are going to freeze the weights (parameters) of our embedding layer. For the last 10 epochs we'll allow our embeddings to be trained.

Why would we ever want to do this? Sometimes the pre-trained word embeddings we use will already be good enough and won't need to be fine-tuned with our model. If we keep the embeddings frozen then we don't have to calculate the gradients and update the weights for these parameters, giving us faster training times. This doesn't really apply for the model used here, but we're mainly covering it to show how it's done. Another reason is that if our model has a large amount of parameters it may make training difficult, so by freezing our pre-trained embeddings we reduce the amount of parameters needing to be learned.

To freeze the embedding weights, we set `model.embedding.weight.requires_grad` to `False`. This will cause no gradients to be calculated for the weights in the embedding layer, and thus no parameters will be updated when `optimizer.step()` is called.

Then, during training we check if `FREEZE_FOR` (which we set to 5) epochs have passed. If they have then we set `model.embedding.weight.requires_grad` to `True`, telling PyTorch that we should calculate gradients in the embedding layer and update them with our optimizer.

[24]:
```
N_EPOCHS = 10
FREEZE_FOR = 5

best_valid_loss = float('inf')

#freeze embeddings
model.embedding.weight.requires_grad = unfrozen = False

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s |␣
↪Frozen? {not unfrozen}')
```

```
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tutC-model.pt')

    if (epoch + 1) >= FREEZE_FOR:
        #unfreeze embeddings
        model.embedding.weight.requires_grad = unfrozen = True
```

```
Epoch: 01 | Epoch Time: 0m 7s | Frozen? True
        Train Loss: 0.724 | Train Acc: 53.68%
         Val. Loss: 0.658 |  Val. Acc: 62.27%
Epoch: 02 | Epoch Time: 0m 6s | Frozen? True
        Train Loss: 0.670 | Train Acc: 59.36%
         Val. Loss: 0.626 |  Val. Acc: 67.51%
Epoch: 03 | Epoch Time: 0m 6s | Frozen? True
        Train Loss: 0.636 | Train Acc: 63.62%
         Val. Loss: 0.592 |  Val. Acc: 70.22%
Epoch: 04 | Epoch Time: 0m 6s | Frozen? True
        Train Loss: 0.613 | Train Acc: 66.22%
         Val. Loss: 0.573 |  Val. Acc: 71.77%
Epoch: 05 | Epoch Time: 0m 6s | Frozen? True
        Train Loss: 0.599 | Train Acc: 67.40%
         Val. Loss: 0.569 |  Val. Acc: 70.86%
Epoch: 06 | Epoch Time: 0m 7s | Frozen? False
        Train Loss: 0.577 | Train Acc: 69.53%
         Val. Loss: 0.520 |  Val. Acc: 76.17%
Epoch: 07 | Epoch Time: 0m 7s | Frozen? False
        Train Loss: 0.544 | Train Acc: 72.21%
         Val. Loss: 0.487 |  Val. Acc: 78.03%
Epoch: 08 | Epoch Time: 0m 7s | Frozen? False
        Train Loss: 0.507 | Train Acc: 74.96%
         Val. Loss: 0.450 |  Val. Acc: 80.02%
Epoch: 09 | Epoch Time: 0m 7s | Frozen? False
        Train Loss: 0.469 | Train Acc: 77.72%
         Val. Loss: 0.420 |  Val. Acc: 81.79%
Epoch: 10 | Epoch Time: 0m 7s | Frozen? False
        Train Loss: 0.426 | Train Acc: 80.28%
         Val. Loss: 0.392 |  Val. Acc: 82.76%
```

Another option would be to unfreeze the embeddings whenever the validation loss stops increasing using the following code snippet instead of the `FREEZE_FOR` condition:

```
if valid_loss < best_valid_loss:
    best_valid_loss = valid_loss
    torch.save(model.state_dict(), 'tutC-model.pt')
```

```
    else:
        #unfreeze embeddings
        model.embedding.weight.requires_grad = unfrozen = True
```

[25]:
```
model.load_state_dict(torch.load('tutC-model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.396 | Test Acc: 82.36%
```

1.3 Saving Embeddings

We might want to re-use the embeddings we have trained here with another model. To do this, we'll write a function that will loop through our vocabulary, getting the word and embedding for each word, writing them to a text file in the same format as our custom embeddings so they can be used with TorchText again.

Currently, TorchText Vectors seem to have issues with loading certain unicode words, so we skip these by only writing words without unicode symbols. **If you know a better solution to this then let me know**

[26]:
```
from tqdm import tqdm

def write_embeddings(path, embeddings, vocab):

    with open(path, 'w') as f:
        for i, embedding in enumerate(tqdm(embeddings)):
            word = vocab.itos[i]
            #skip words with unicode symbols
            if len(word) != len(word.encode()):
                continue
            vector = ' '.join([str(i) for i in embedding.tolist()])
            f.write(f'{word} {vector}\n')
```

We'll write our embeddings to `trained_embeddings.txt`.

[27]:
```
write_embeddings('custom_embeddings/trained_embeddings.txt',
                 model.embedding.weight.data,
                 TEXT.vocab)
```

```
100%|      | 25002/25002 [00:00<00:00, 38085.03it/s]
```

To double check they've written correctly, we can load them as `Vectors`.

[28]:
```
trained_embeddings = vocab.Vectors(name = 'custom_embeddings/trained_embeddings.
↪txt',
                                   cache = 'custom_embeddings',
```

```
                                    unk_init = torch.Tensor.normal_)
```

```
 70%|       | 17550/24946 [00:00<00:00, 87559.48it/s]
```

Finally, let's print out the first 5 rows of our loaded vectors and the same from our model's embeddings weights, checking they are the same values.

[29]:
```
print(trained_embeddings.vectors[:5])
```

```
tensor([[-0.2573, -0.2088,  0.2413, -0.1549,  0.1940, -0.1466, -0.2195, -0.1011,
         -0.1327,  0.1803,  0.2369, -0.2182,  0.1543, -0.2150, -0.0699, -0.0430,
         -0.1958, -0.0506, -0.0059, -0.0024],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
          0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
          0.0000,  0.0000,  0.0000,  0.0000],
        [-0.1427, -0.4414,  0.7181, -0.5751, -0.3183,  0.0552, -1.6764, -0.3177,
          0.6592,  1.6143, -0.1920, -0.1881, -0.4321, -0.8578,  0.5266,  0.5243,
         -0.7083, -0.0048, -1.4680,  1.1425],
        [-0.4700, -0.0363,  0.0560, -0.7394, -0.2412, -0.4197, -1.7096,  0.9444,
          0.9633,  0.3703, -0.2243, -1.5279, -1.9086,  0.5718, -0.5721, -0.6015,
          0.3579, -0.3834,  0.8079,  1.0553],
        [-0.7055,  0.0954,  0.4646, -1.6595,  0.1138,  0.2208, -0.0220,  0.7397,
         -0.1153,  0.3586,  0.3040, -0.6414, -0.1579, -0.2738, -0.6942,  0.0083,
          1.4097,  1.5225,  0.6409,  0.0076]])
```

[30]:
```
print(model.embedding.weight.data[:5])
```

```
tensor([[-0.2573, -0.2088,  0.2413, -0.1549,  0.1940, -0.1466, -0.2195, -0.1011,
         -0.1327,  0.1803,  0.2369, -0.2182,  0.1543, -0.2150, -0.0699, -0.0430,
         -0.1958, -0.0506, -0.0059, -0.0024],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
          0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,
          0.0000,  0.0000,  0.0000,  0.0000],
        [-0.1427, -0.4414,  0.7181, -0.5751, -0.3183,  0.0552, -1.6764, -0.3177,
          0.6592,  1.6143, -0.1920, -0.1881, -0.4321, -0.8578,  0.5266,  0.5243,
         -0.7083, -0.0048, -1.4680,  1.1425],
        [-0.4700, -0.0363,  0.0560, -0.7394, -0.2412, -0.4197, -1.7096,  0.9444,
          0.9633,  0.3703, -0.2243, -1.5279, -1.9086,  0.5718, -0.5721, -0.6015,
          0.3579, -0.3834,  0.8079,  1.0553],
        [-0.7055,  0.0954,  0.4646, -1.6595,  0.1138,  0.2208, -0.0220,  0.7397,
         -0.1153,  0.3586,  0.3040, -0.6414, -0.1579, -0.2738, -0.6942,  0.0083,
          1.4097,  1.5225,  0.6409,  0.0076]], device='cuda:0')
```

All looks good! The only difference between the two is the removal of the ~50 words in the vocabulary that contain unicode symbols.