Text Classification

Minki Kang

MLAI, KAIST

What Are We Going to Learn

The contents of this lecture is as follows:

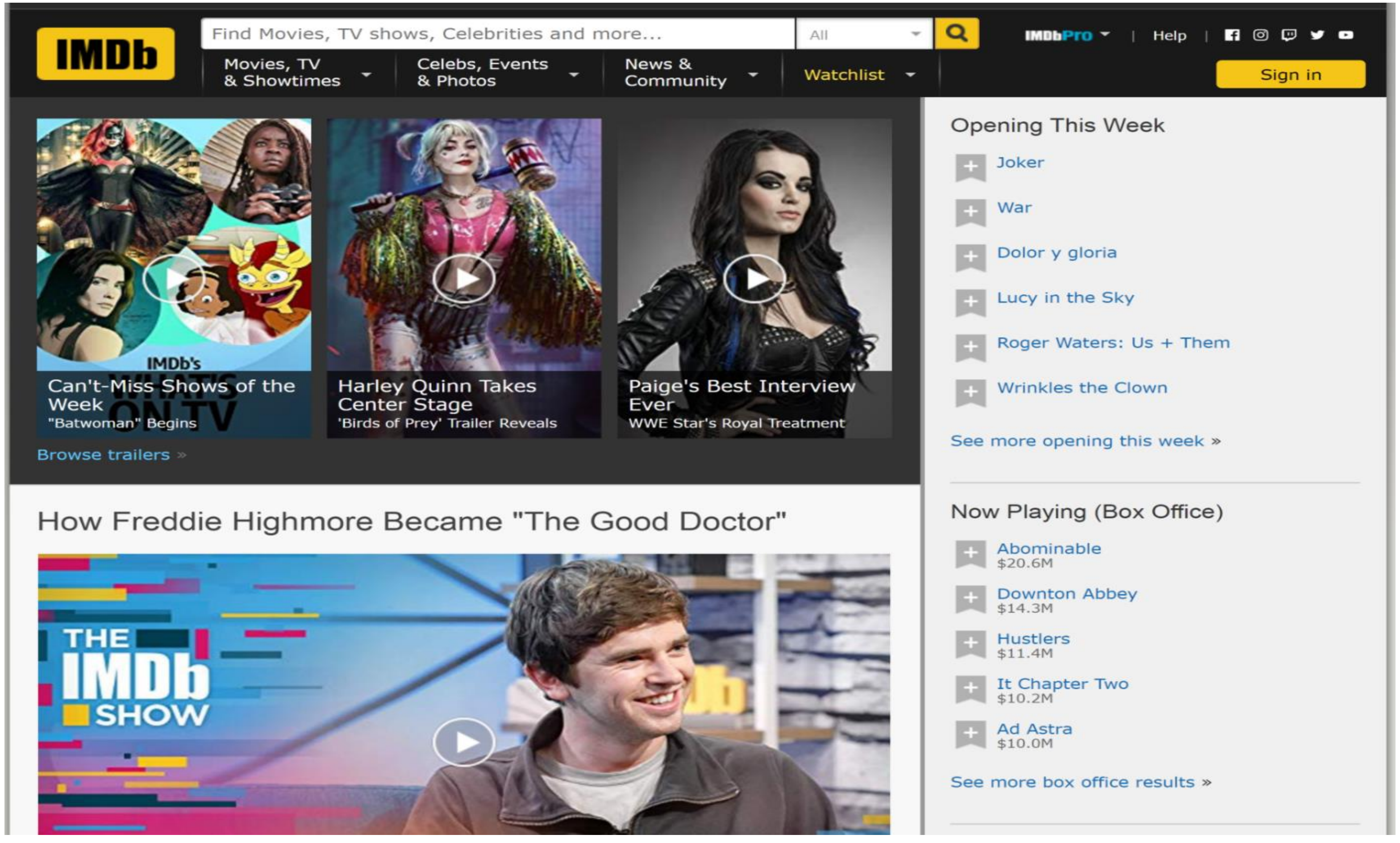
1. Text Classification dataset (IMDb) and model (RNN) overview

2. Introduction for PyTorch, one of the most popular deep learning frameworks

3. Code for IMDb movie review classification using Pytorch
(Code is available at https://github.com/bentrevett/pytorch-sentiment-analysis)

What is the IMDb?

IMDb (Internet Movie Database) is the website (www.imdb.com) which provides ratings and review for Movies and TV shows.



What is the IMDb?

IMDb dataset especially includes the ratings and reviews of movies.

Joker (I) (2019)

User Reviews

+ Review this title

★ 10/10

As a viewer that actually went to TIFF and witnessed this film and didn't want to believe the hype, it is an absolute MASTERPIECE and Phoenix is a certified legend.

JF500 10 September 2019

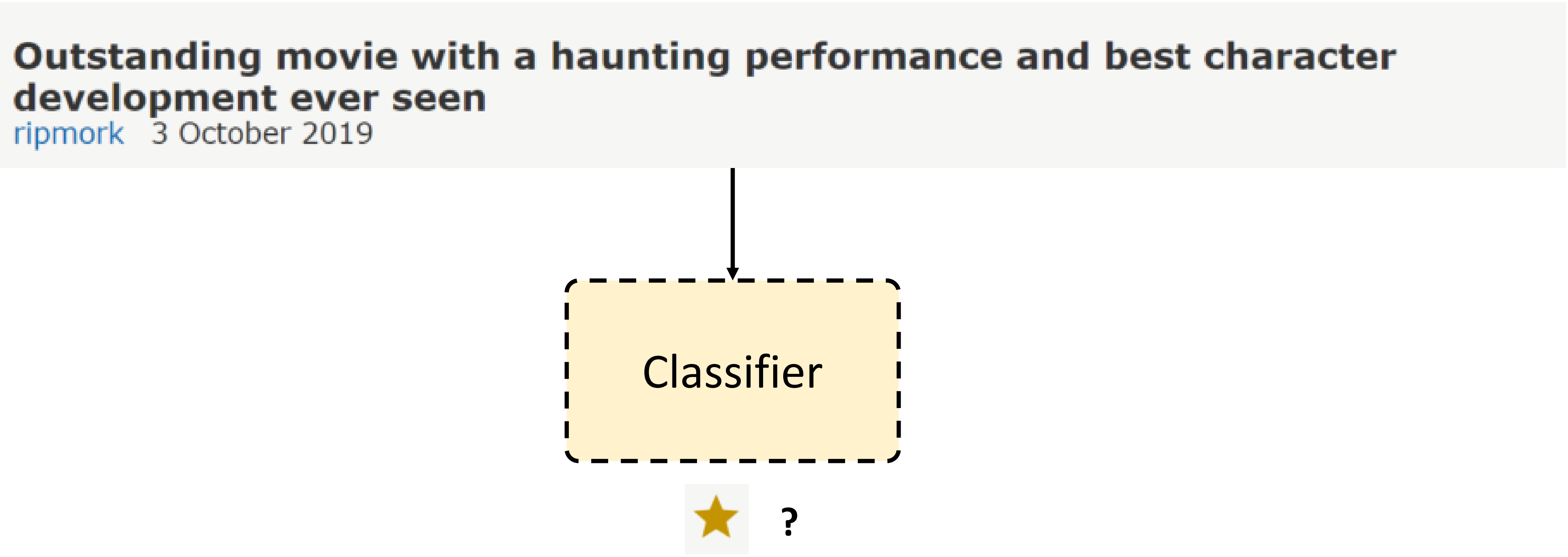
★ 1/10

over rated story

dhdefragx-989-51556 2 October 2019

IMDb Task

The given task for the IMDb dataset is “Classifying” the given reviews to positive rating or negative rating.



***Positive*? Or *Negative*?**

Recurrent Neural Network (RNN)

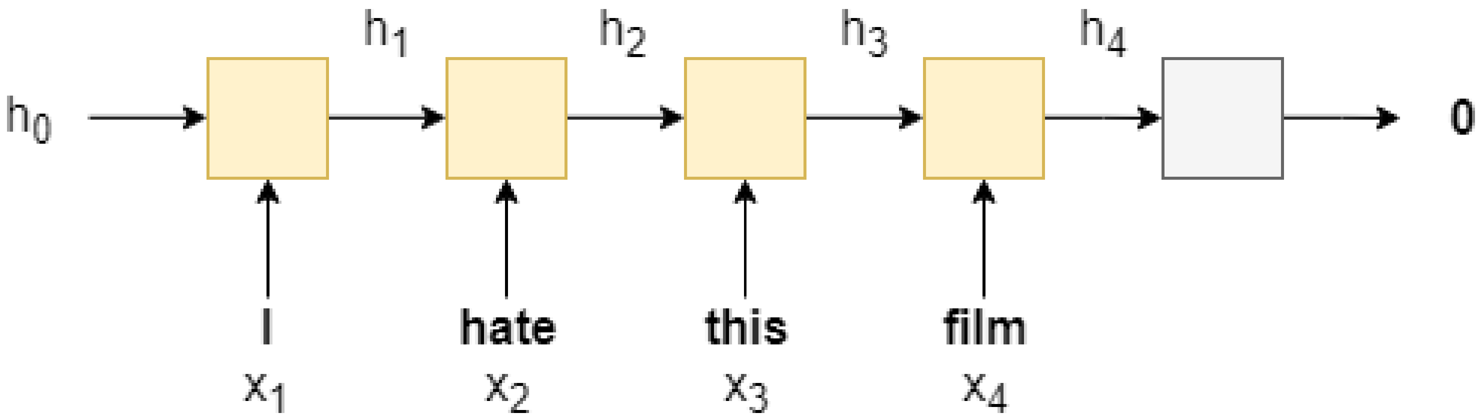
RNN is the simple model for analyzing sequences.

X is the sentence consisting of words $x_1, x_2, \dots, x_T$.

$$X = \{x_1, \dots, x_T\}$$

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

$$\hat{y} = f(h_T)$$



Recurrent Neural Network (RNN)

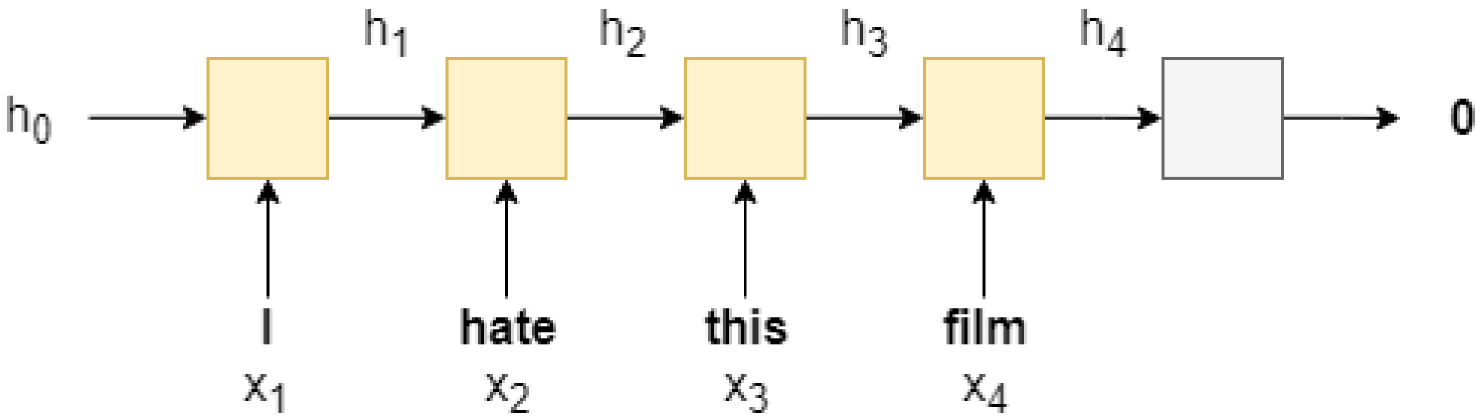
Here, f is the linear layer (also known as a fully connected layer) and $\hat{y}$ is our predicted sentiment.

Predicting "zero" means this sentence is classified to negative sentiment.

$$X = \{x_1, \dots, x_T\}$$

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

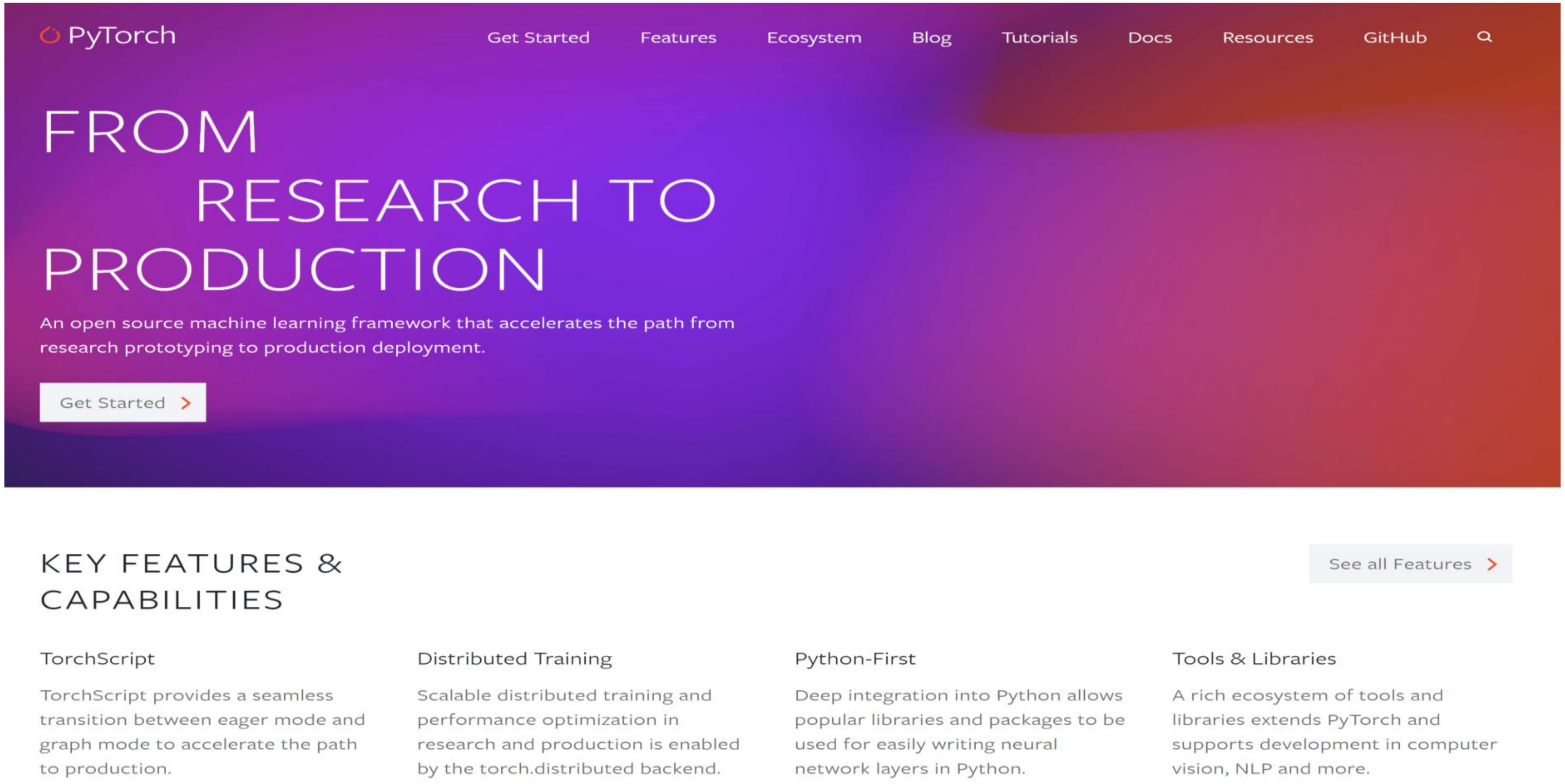
$$\hat{y} = f(h_T)$$



How to implement? PyTorch!

PyTorch is the powerful deep learning framework based on python.

Today, I will explain PyTorch for implementing Movie Review classifier.



How to implement? PyTorch!

Because this is very basic lecture, don't worry if you don't know PyTorch.

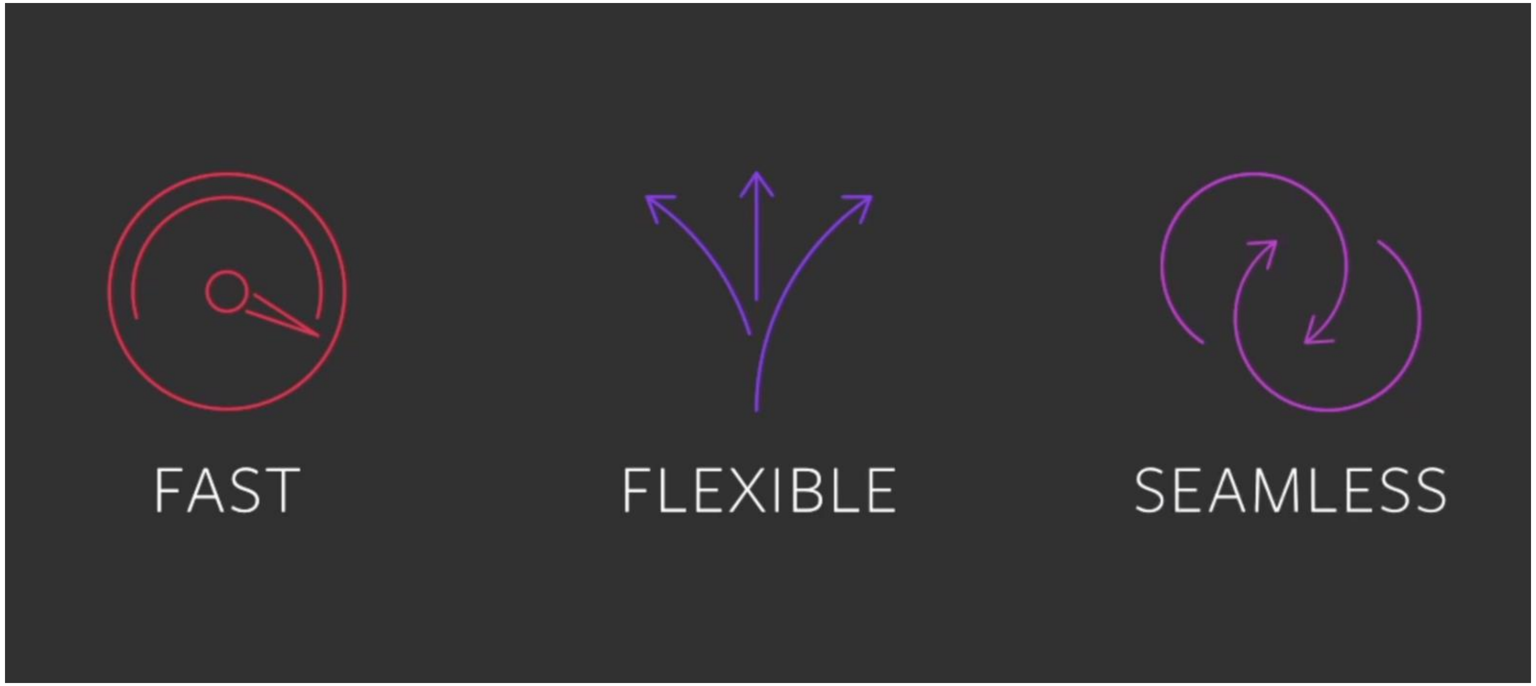
As we know, the detail of PyTorch will be provided in 10.10 or 11.

Week 5	**10.2-10.11**		**자연어 이해 1**	
	10.2	수	자연어처리 입문, 단어 벡터화 (Word embedding)	최재식
	10.7	월	문서 분류 (Text classification)	황성주
	10.8	화	문장 분석 (Sentence parsing)	신진우
	10.10	목	언어 모델 (RNN 기반)	주재걸
	10.11	금	기계 번역 (Seq-to-seq)	주재걸
Week 6	**10.14-18**		**자연어 이해 2**	
	10.14	월	어텐션 기반 모델, 사전 학습 (Transformer, BERT)	주재걸
	10.15	화	자연어 생성 (Text generation)	황성주
	10.16	수	대화 모델 (Conversation agent, Bias & Fairness)	오혜연
	10.17	목	질의 응답 (Attention model, Memory Networks)	주재걸
	10.18	금	자연어 처리 중간평가	황성주

Why PyTorch?

PyTorch provides the fast, flexible implementation, and the seamless transaction for product deployment.

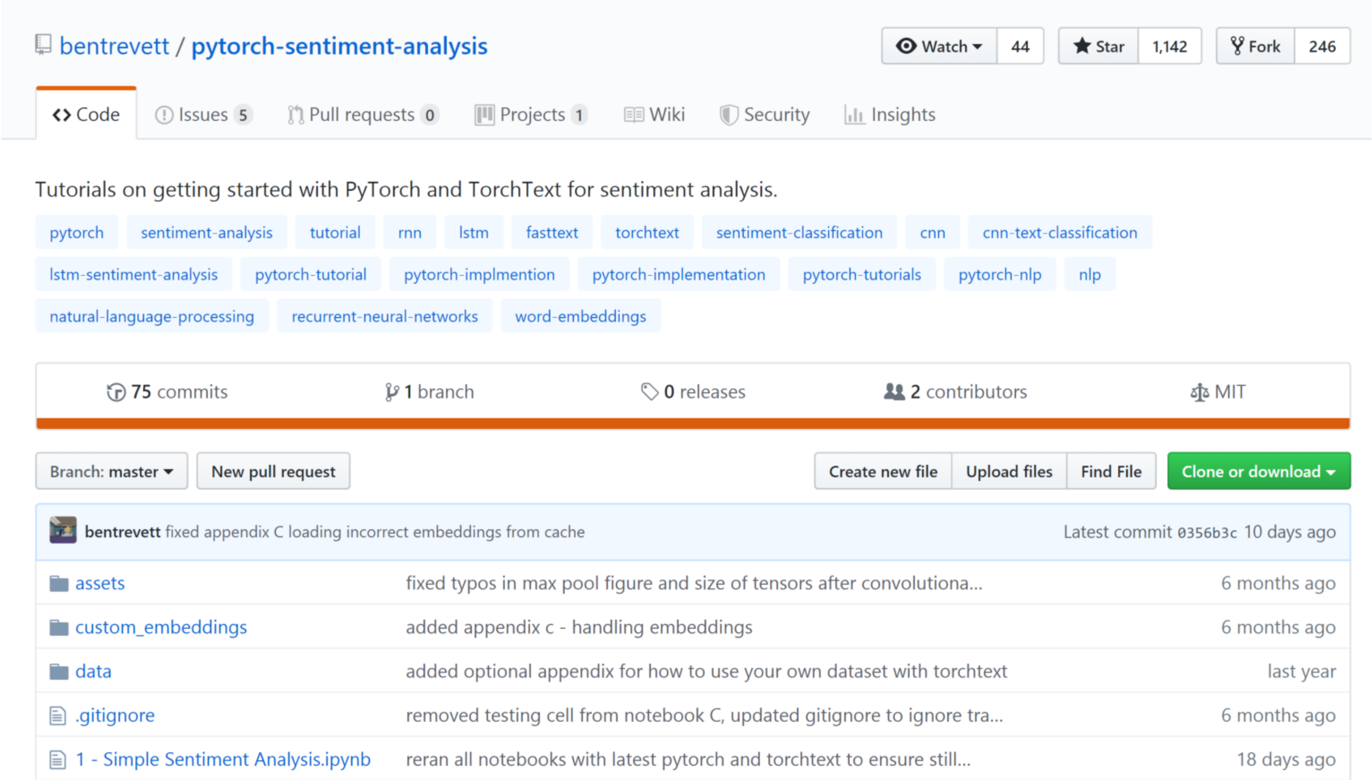
If you want to learn more detail, refer here: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html



Code Review

I will explain about basic of PyTorch and text classifier using codes based on ***Jupyter Notebook***.

This code is based on the code from https://github.com/bentrevett/pytorch-sentiment-analysis



Preparing Data

The ***three*** important components for deep learning are Data, Model, and Optimizer.

First, let's build the data for classification. We use ***torchtext*** for text data.

First of all, import PyTorch and data object from torchtext.

```
[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)
```

Import PyTorch and data object

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

Preparing Data

Then, set seeds for random operation 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)
```

Set Seeds

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

Preparing Data

Then, utilize the data object from torchtext.

“Field” is one of the main concepts of torchtext.

These define how your data should be processed.

In our task, the data consists of both ***the raw string of the review*** and ***the sentiment***.

```
[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)
```

Set Field for text data

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

Preparing Data

Here, tokenize='spacy' means the data field using 'spacy' as ***tokenizer***.

LabelField is the special type of Field that handles ***label*** (sentiment).

```
[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)
```

Set Field for text data

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

Preparing Data - Tokenizer

What is tokenizer?

“Token” means “meaningful words”.

“Tokenizer” makes sentence to tokens.

“I hate this film” $\xrightarrow{\textit{tokenize}}$ ***[“I”, “hate”, “this”, “film”]***

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

Preparing Data

In torchtext, IMDb dataset is provided.

We can simply download using one line of code.

This code automatically downloads both IMDb train and test dataset.

It costs about 1 minute to download the dataset.

```
[2] from torchtext import datasets
    train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
```

Download IMDb dataset

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

Preparing Data

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)}')
```

Count the number of data and print it

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

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

Preparing Data

We can also check an example.

The example includes “tokenized” text and label.

“pos” label means positive, “neg” label means negative.

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

Print 0-th example

```
{'text': ['This', 'is', 'one', 'of', 'the', 'finest', 'films', 'to', 'come', 'out', 'of', 'Hong', 'Kong', "'s", "'", 'New', 'Wave', "'
```

```
ly', 'feel', 'chastened', 'by', 'the', 'experience', '-', 'and', 'somehow', 'better', 'for', 'it', 'over', 'all', '.'], 'label': 'pos'}
```

Label

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

Preparing Data

Then, what we need is validation dataset.

Since the IMDb dataset does not provide validation dataset, we need to make it by splitting training dataset.

Then, we also can see how many examples are for each splits.

Split training dataset into validation dataset

```
[5] import random

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

```
[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
```

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

Preparing Data - Vocabulary

Then, we have to build a ***vocabulary***. This is a effectively a ***look up table*** where every unique word in dataset has a corresponding integer index.

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]

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

Preparing Data - Vocabulary

Because machine learning model ***cannot operate on strings***, each index is used to construct an ***one-hot vector for each word***.

An one-hot vector is a vector where all of the elements are 0, except one, which is 1.

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]

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

Preparing Data - Vocabulary

Problem of one-hot vector is that the dimension of the vector increases as the number of unique words increases.

For instance, the number of unique English words are about 100,000.

In this case, the length of each one-hot vector is 100,000, which is too large!

word	index	one-hot vector
I	0	$\underbrace{[1,\ 0,\ 0,\ \dots\ ,\ 0]}_{\textit{Length}\ =\ 100{,}000}$

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

Preparing Data - Vocabulary

The way to address this problem is ***cutting down*** our vocabulary, ***only taking the top n most common words*** and replace less common words to <unk> token (which means *unknown*).

word	index	one-hot vector
I	0	$\underbrace{[1,\ 0,\ 0,\ \ldots,\ 0]}_{\textit{Length} = 100{,}000}$

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

Preparing Data

This code is for ***building vocabulary using training dataset***.

MAX_VOCAB_SIZE means how many words we want to use.

Therefore, in this example, only ***top 25,000 frequent words*** are included in the vocabulary.

```
[7]  MAX_VOCAB_SIZE = 25_000

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

Build Vocabulary

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

Preparing Data

We can see the number of words included in the vocabulary.

The reason why the number of words included in TEXT vocabulary is 25002 and not 25000 is that “<unk>” token and “<pad>” token are included.

```
[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
```

The number of vocabulary for each field

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

Preparing Data - <pad> token

<pad> token is needed for ***batch*** operation.

To make batch of data, the length of data included in same batch should be the same.

For examples, if we make data batch using sent1 and sent2, <pad> token is added at the end of sent2 to ensure each sentence in the batch is the same size.

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

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

Preparing Data

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

For example, word “the” occurs 200806 times in the training dataset and word “is” occurs 75910 times in the training dataset.

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

[('the', 200806), (',', 190507), ('.', 163859), ('and', 108678), ('a', 108379), ('of', 99904), ('to', 92850), ('is', 75910), (
```

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

Preparing Data

We can also see the vocabulary irectly using either stoi (string to int) or itos (int to string) method.

In code example, 0-th word is ‘<unk>’, 1-st word is ‘<pad>’, 2-nd word is ‘the’, and so on…

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

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

Preparing Data

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 0x7fc739e0aa60>, {'neg': 0, 'pos': 1})
```

Preparing Data

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 at each iteration.

"***BucketIterator***" 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.

```
[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)
```

Build Iterators

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

Preparing Data

Here, the “device” is either CPU or GPU.

If there is detected GPU, iterator is placed on the GPU.

```
[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)
```

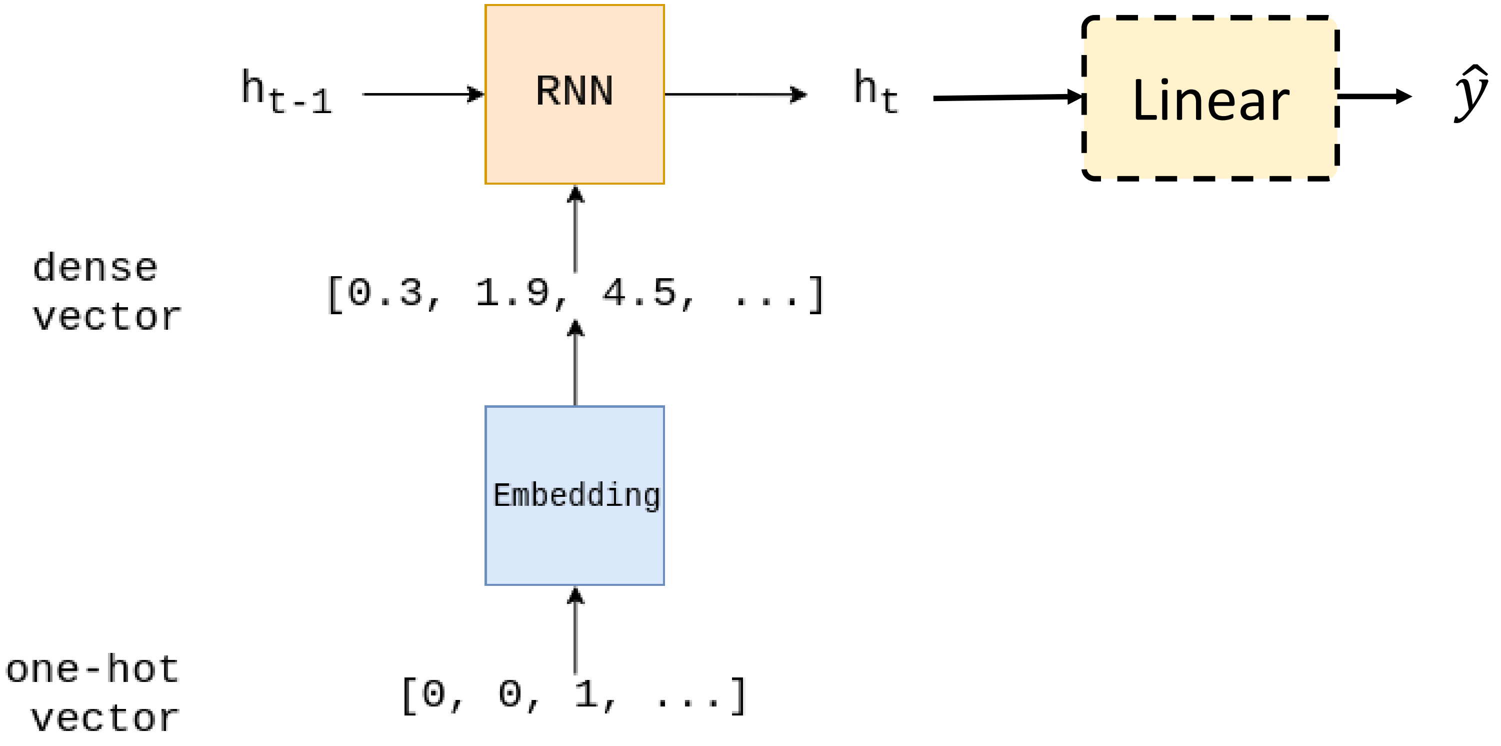
https://github.com/bentrevett/pytorch-sentiment-analysis

Build the Model

From now on, we will build the model for classifying movie reviews.

At the same time, I will introduce how to build model in PyTorch.

What we need for build classifier is three, "***Embedding Layer***", "***RNN***", "***Linear Layer***".

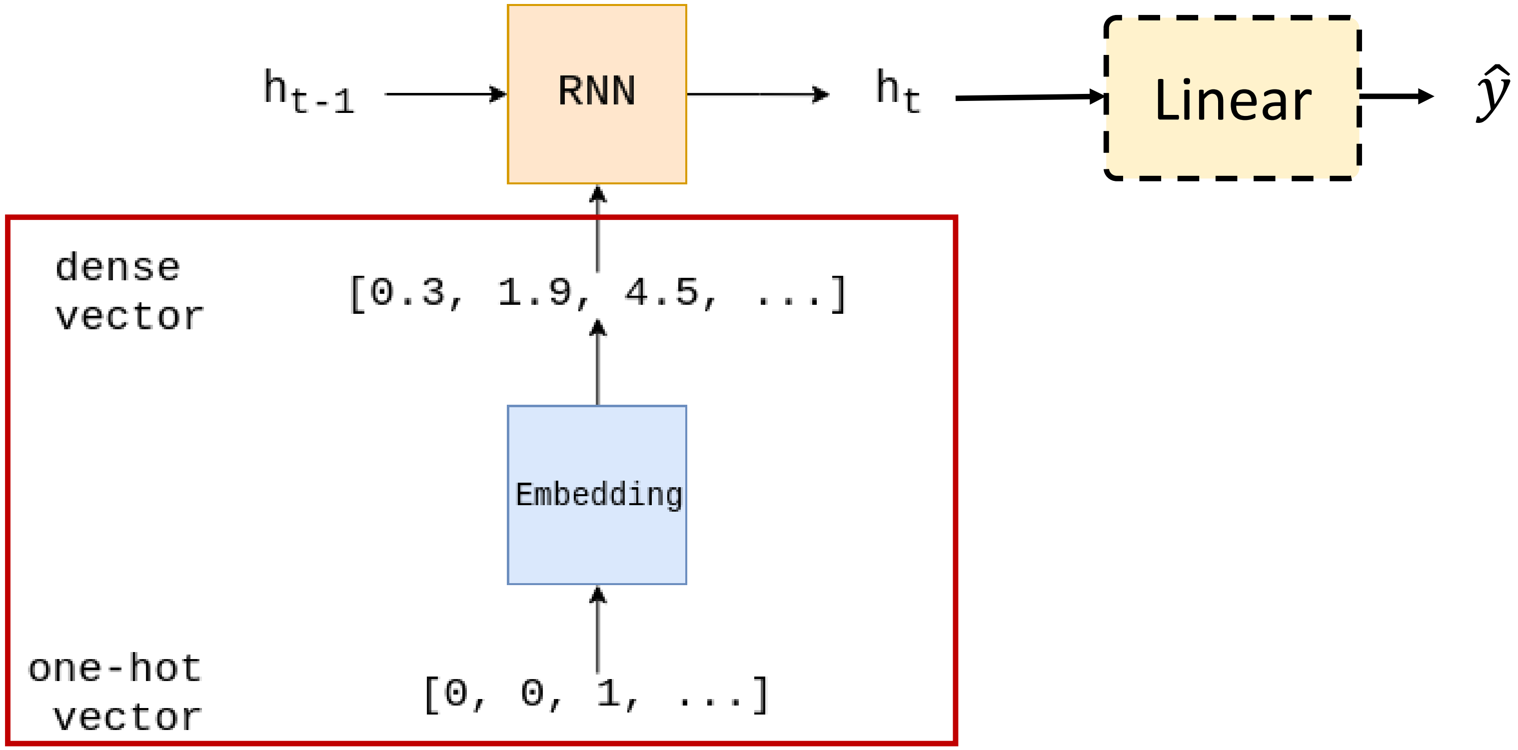


https://github.com/bentrevett/pytorch-sentiment-analysis

Build the Model

Here, ***the embedding layer*** is used to transform our sparse one-hot vector into a ***dense*** embedding vector.

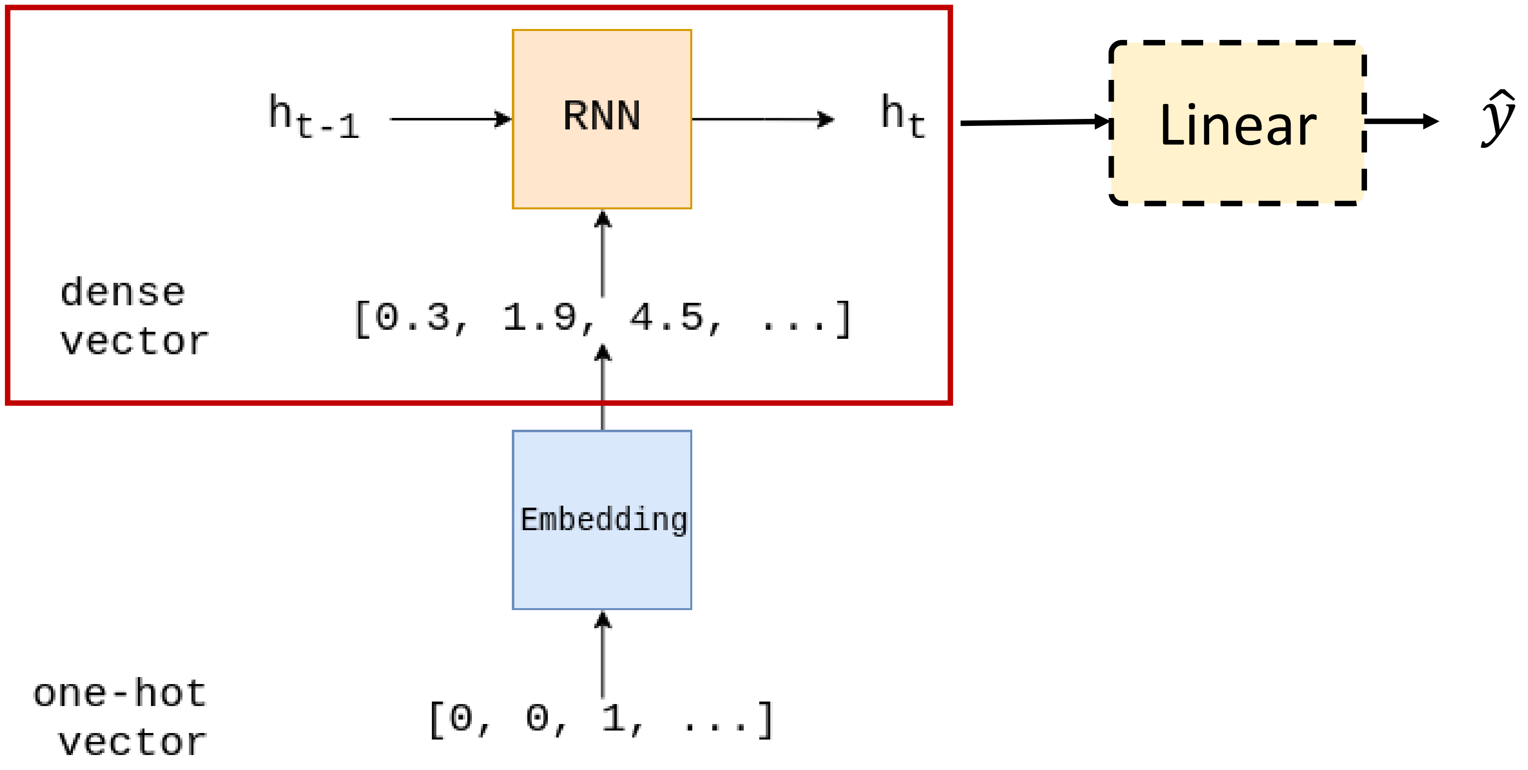
This embedding layer is simply a ***single fully connected layer***.



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

Build the Model

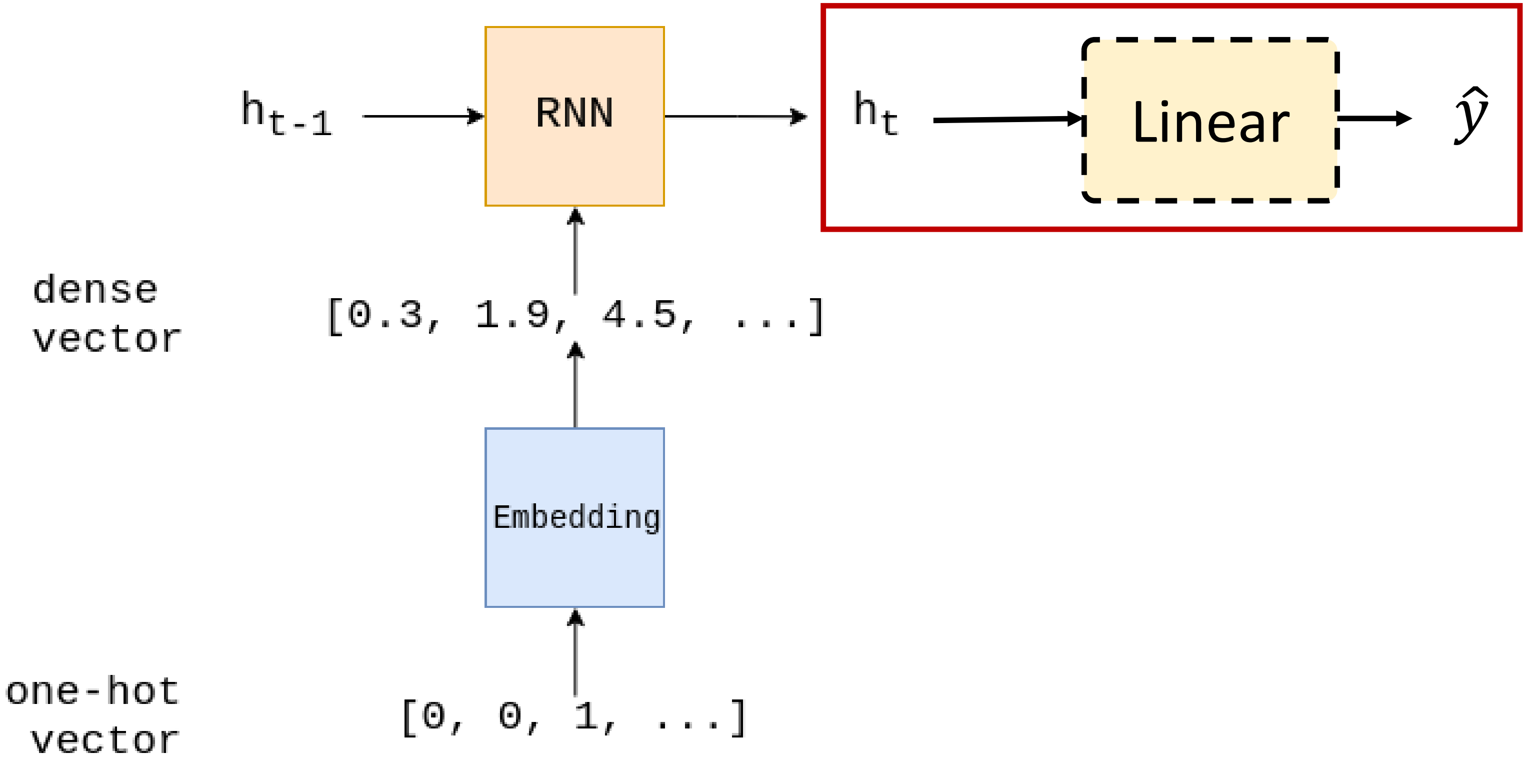
The ***RNN layer*** is our RNN which takes in our dense vector and the previous hidden state h_{t-1}, which it uses to calculate the next hidden state h_t.



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

Build the Model

The ***Linear layer*** takes the final hidden state and feed it through a fully connected layer, $f(h_T)$, transforming it to the correct output dimension.



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

Build the Model

This is code for our classifier model class.

```
[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))
```

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

Build the Model

In ***__init__*** function, we need to define each component – Embedding Layer, RNN Layer, Linear Layer.

```
[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)
```

Embedding Layer

RNN Layer

Linear Layer

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

Build the Model

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

Each batch (text) is a tensor size of ***[sentence length, batch size].***

That is a batch of sentences, each having each word converted into a one-hot vector.

```
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))
```

Input batch

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

Build the Model

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]***.

```
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))
```

Embedding input to "embedded"

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

Build the Model

embedded is then fed into the ***RNN layer***. In this code, the initial hidden state, h_0, is passed as a default zero vector automatically.

The RNN returns 2 tensors, ***output*** of size ***[sentence length, batch size, hidden dim]*** and ***hidden*** of size ***[1, batch size, hidden 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))
```

Take RNN outputs

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

Build the Model

Here, the ***output*** is concatenation of the all hidden state from every time step.

The ***hidden*** is simply the final hidden state, h_T.

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

```
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))
```

Get Prediction

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

Build the Model

We now create an instance of our RNN class.

We define embedding_dim as 100, hidden_dim as 256, and output_dim as 1.

You can change this as you want.

[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)
```

Build the model

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

Build the Model

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.

We can see that our model has ***2,592,105*** trainable parameters.

```
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
```

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

Build the Model

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.

We can see that our model has ***2,592,105*** trainable parameters.

```
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
```

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

Train the Model

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

What we need to create is an ***optimizer***.

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.

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

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

Set the optimizer

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

Train the Model

Then, we need to define loss function which is commonly called a criterion.

We use the ***binary cross entropy with logits***.

The equation for given loss function is as follows:

$$l(x,y) = L = \{l_1, \dots, l_N\}^T, l_n = -w_n\,[y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))]$$

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

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

Place the loss function and criterion on the given device

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

Train the Model

In addition to loss, we need to calculate ***accuracy***, which is the measurement for classifier performance.

Because value of 'preds' is 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 ***rest to 0*** (a negative sentiment).

```
[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
```

Calculate accuracy

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

Train the Model

This is the train function which iterates over all examples, one batch at a time.

```
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)
```

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

Train the Model

Model.train() is sued to put the model in “training mode”, which turns on dropout and batch normalization.

```python
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)
```

Set mode to “training mode”

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

Train the Model

For each batch, we first ***zero the gradients***. Because PyTorch does not automatically remove the gradients calculated from the last, we must manually zero them.

```
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()      Zero the gradients

        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)
```

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

Train the Model

Then, we feed the batch of sentences, batch.text, into the model.

It is executed just calling the model works without model.forward method.

```
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)
```

Feed the input to the model

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

Train the Model

Because the predictions are initially size ***[batch size, 1]***, we need to "***squeeze***" it to the size ***[batch size]***, which is fit size for PyTorch criterion.

```
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)
```

Squeeze the output

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

Train the Model

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.

```
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)
```

Calculate loss and accuracy

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

Train the Model

We calculate the gradient of each parameter with loss.backward().

```
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)
```

Calculate the gradient

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

Train the Model

Then, we update the parameters using the gradients and optimizer algorithm with optimizer.step().

```
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)
```

Update parameters of model

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

Train the Model

Finally, we return the loss and accuracy, averaged across the epoch.

```
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)
```

Return the loss and accuracy

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

Train the Model

Next, we need code for evaluating trained model.

Here is the '***evaluate***' code.

```
[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)
```

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

Train the Model

At first, we need to put the model in "evaluation mode" using mode.eval().

This turns off dropout and batch normalization.

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

         epoch_loss = 0
         epoch_acc = 0

         model.eval()   Set model in "evaluation mode"

         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)
```

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

Train the Model

Since the gradient calculation is not needed for evaluation, we set ***'with no_grad()'*** block.
This causes less memory to be used and speeds up computation.

```
[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)
```

Set 'no_grad()' block

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

Train the Model

The rest of the function is the similar as train.

However, codes such as optimizer.zero_grad(), loss.backward() are deleted.

```
[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)
```

Evaluate

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

Train the Model

This function is needed to calculate the elapsed time.

This is useful because this function can 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
```

Train the Model

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

```
[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}%')
```

Traning and Evaluation

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

Train the Model

We save the model with the least valid loss across all epochs.

```
[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}%')
```

Save the model

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

Train the Model

The desirable output of full pipeline is like this:

```
Epoch: 01 | Epoch Time: 0m 45s
	Train Loss: 0.694 | Train Acc: 50.25%
	 Val. Loss: 0.697 |  Val. Acc: 48.67%
Epoch: 02 | Epoch Time: 0m 44s
	Train Loss: 0.693 | Train Acc: 49.86%
	 Val. Loss: 0.697 |  Val. Acc: 49.21%
Epoch: 03 | Epoch Time: 0m 45s
	Train Loss: 0.693 | Train Acc: 49.85%
	 Val. Loss: 0.697 |  Val. Acc: 49.83%
Epoch: 04 | Epoch Time: 0m 45s
	Train Loss: 0.693 | Train Acc: 50.04%
	 Val. Loss: 0.697 |  Val. Acc: 48.53%
Epoch: 05 | Epoch Time: 0m 45s
	Train Loss: 0.693 | Train Acc: 50.39%
	 Val. Loss: 0.697 |  Val. Acc: 49.98%
```

Accuracy

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

Test the Model

Finally, we need to compute the performance of our classifier with test data.

First, we need to load saved parameters from training.

```
[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}%')
```

Load the parameters

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

Test the Model

Then, we can measure performance with evaluate function.

```
[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

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

Test the Model

Calculated test accuracy is 45.47%.

It seems our classifier doesn't work well…

How to improve this?

```
[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: 0.712 | Test Acc: 45.47%
```

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

Improving the model

There are several ways to improve the model to the advanced model.

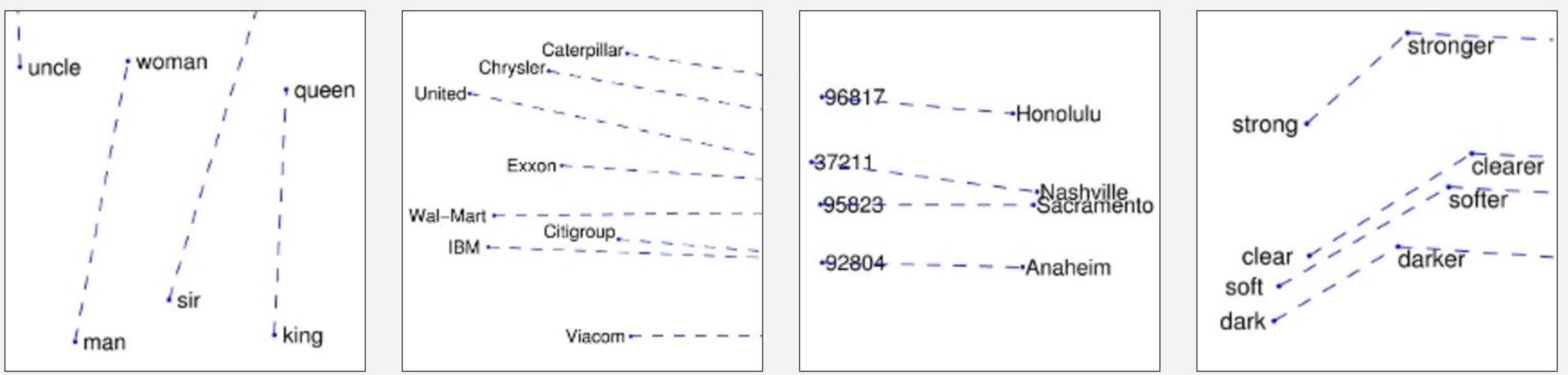
- Pre-trained word embeddings
- Packed Padded sequences
- Different RNN architecture
- Bidirectional RNN
- Multi-Layer RNN
- Regularization
- Different Optimizer

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

Pre-trained word embeddings

GloVe is the one of the most famous word embedding methods.

Using already trained GloVe, we can get ***meaningful embedding*** of words.



man - woman | company - ceo | city - zip code | comparative - superlative

Pennington et al., GloVe: Global Vectors for Word Representation.

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

Pre-trained word embeddings

We can simply use this in code by changing one line.

It costs some time (< 2min) to download GloVe vectors.

```
[7]  MAX_VOCAB_SIZE = 25_000

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

```
[ ]  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)
```

Build vocab using GloVe

Initialize <unk> via Gaussian distribution

Pennington et al., GloVe: Global Vectors for Word Representation.

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

Packed-padded sequence

Using ***packed-padded sequences*** will make our RNN only process the non-padded elements of our sequence, and for any padded element the output will be a zero tensor.

To use this, we need to indicate how long the actual sequences are.

```
[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)
```

```
[7]  import torch
     from torchtext import data

     SEED = 1234

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

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

Add this argument

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

Packed-padded sequence

Another thing for packed padded sequences is that all of the tensors within a batch ***need to be sorted by their lengths***.

This is handled in ***the iterator*** by setting ***sort_within_batch=True***.

```
[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)
```

```
[ ]  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,
         sort_within_batch=True,
         device = device)
```

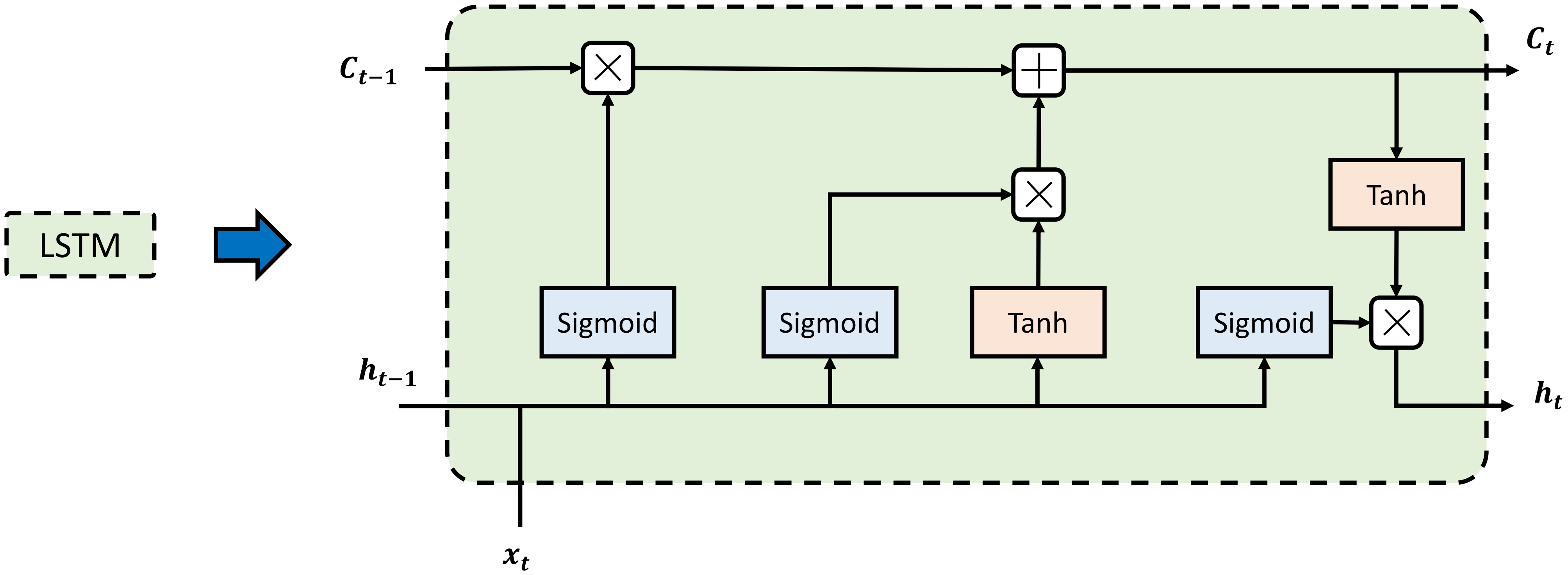
Add this argument

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

Different RNN Architecture

The problem of simple RNN is the ***"vanishing gradient"*** problem.

To address this, Loong Short-Term Memory (LSTM) is proposed.

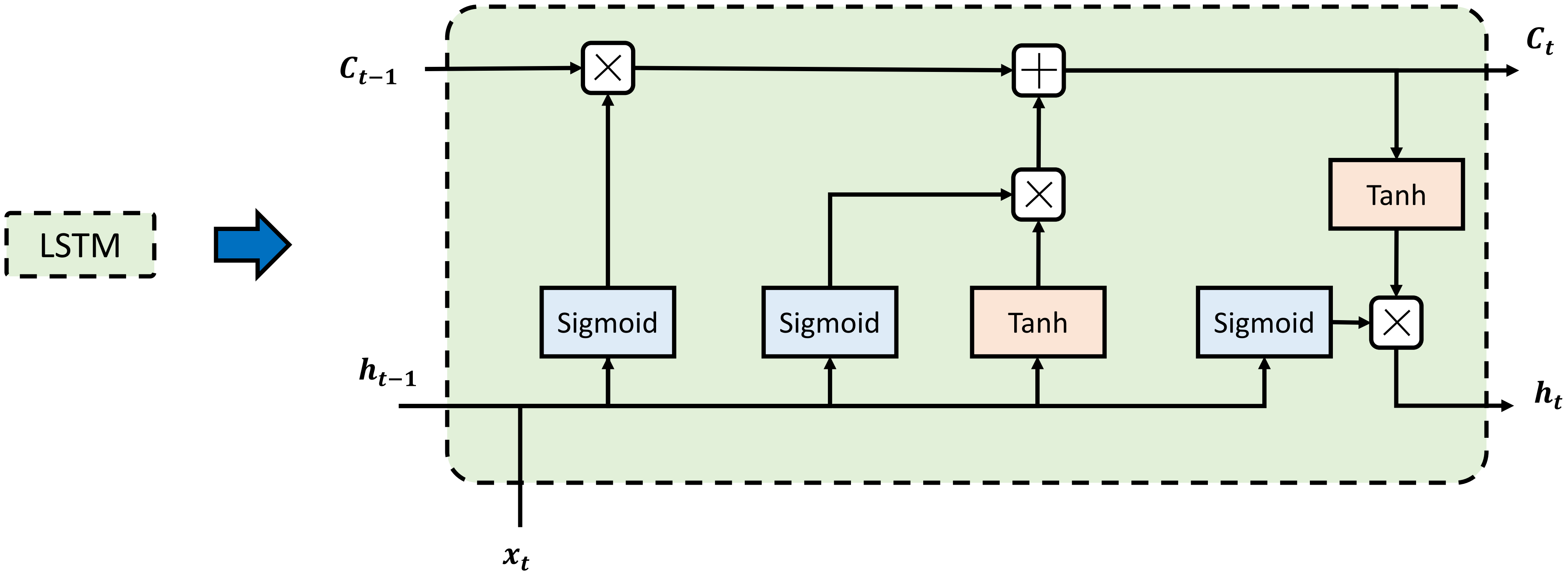


[Hochreiter and Schmidhuber97] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory. Neural Computation 1997

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

LSTM

LSTMs overcome gradient vanishing problem by having ***an extra recurrent state called a cell***, c, - which can be thought of as the “memory” of the LSTM – and the use ***multiple gates*** which control the flow of information into and out of the memory.



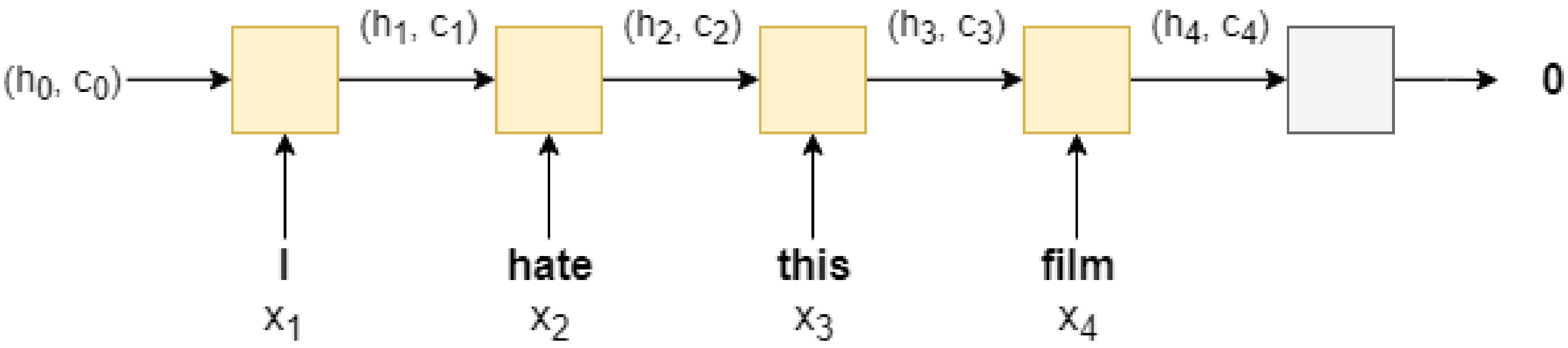
[Hochreiter and Schmidhuber97] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory. Neural Computation 1997

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

LSTM

For simplicity, we can just think LSTM as a function like this:

$$(h_t, c_t) = LSTM(x_t, h_t, c_t)$$

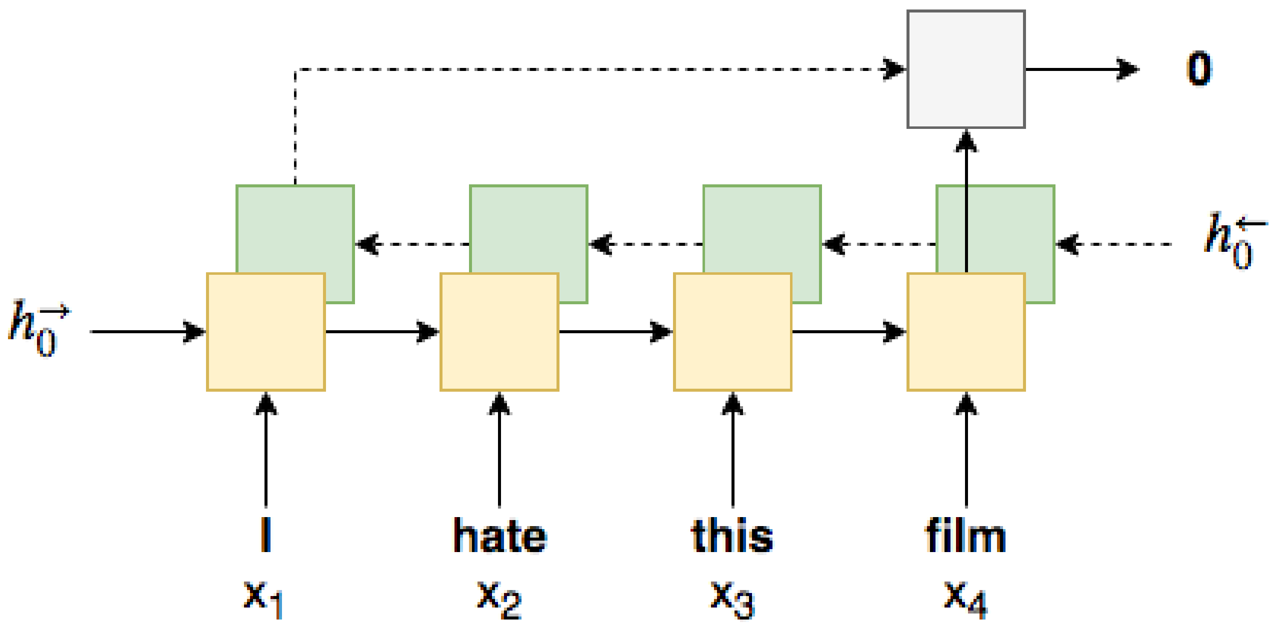


[Hochreiter and Schmidhuber97] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory. Neural Computation 1997

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

Bidirectional RNN

The idea of ***bidirectional RNN*** is simple: just using backward pass as well as forward pass. Therefore, last hidden state can be described as $\hat{y} = f(h_{\overrightarrow{T}}, h_{\overleftarrow{T}})$, which is concatenation of ***last hidden states of forward and backward pass***.

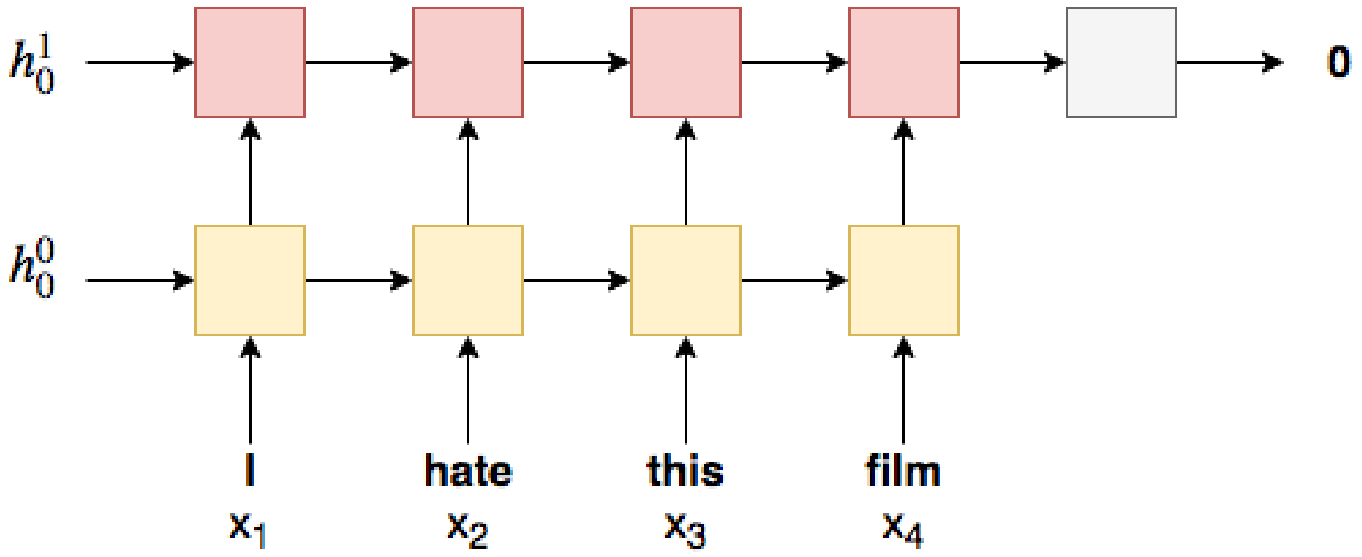


https://github.com/bentrevett/pytorch-sentiment-analysis

Multi-layer RNN

Multi-layer RNNs (also called deep RNNs) are another simple concept.

The idea is that we ***add additional RNNs*** on top of the initial standard RNN, where each RNN added is another layer.



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

Make Advanced Model

Now, let's address reinforcing method to our model.

```
[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)
```

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

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

Make Advanced Model

First, change the name of parameter ‘input_dim’ to ‘vocab_size’.

This is not needed but I change it for better understanding.

```
[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)
```

For using GloVe

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

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

Make Advanced Model

Then, because we pre-trained word embedding, we are ***not going to learn embedding for the <pad>*** token. So, indicate padding_idx to Embedding layer.

```
[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)
```

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

For using GloVe

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

Make Advanced Model

For using LSTM, we just change nn.RNN to nn.LSTM. However, LSTM needs more arguments than RNN.

```
[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)
```

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

Change to LSTM

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

Make Advanced Model

Make LSTM to be bidirectional and multi-layered is simple.

What we just to do is setting bidirectional and num_layers argument.

```
[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)
```

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

Set Arguments for multi-layer and bidirectional

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

Make Advanced Model

If we use bidirectional, output size becomes twice.

So, address this at the final linear layer.

```python
[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)
```

```python
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

Adress twiced hidden

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

Make Advanced Model

If we use bidirectional, output size becomes twice.

So, address this at the final linear layer.

```
[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)
```

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers,
                 bidirectional, dropout, pad_idx):

        super().__init__()

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

        self.rnn = nn.LSTM(embedding_dim,
                           hidden_dim,
                           num_layers=n_layers,
                           bidirectional=bidirectional,
                           dropout=dropout)

        self.fc = nn.Linear(hidden_dim * 2, output_dim)

        self.dropout = nn.Dropout(dropout)
```

Add dropout for regularization

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

Make Advanced Model

We also need to change ***forward*** method.

```python
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))
```

```python
def forward(self, text, text_lengths):

    #text = [sent len, batch size]

    embedded = self.dropout(self.embedding(text))

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

    #pack sequence
    packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)

    packed_output, (hidden, cell) = self.rnn(packed_embedded)

    #unpack sequence
    output, output_lenghts = nn.utils.rnn.pad_packed_sequence(packed_output)

    #output = [sent len, batch size, hid dim, hid dim * num directions]
    #output over padding tokens are zero tensors

    #hidden = [num layers * num directions, batch size, hid dim]
    #cell = [num layers * num directions, batch size, hid dim]

    #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
    #and apply dropout

    hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))

    #hidden = [batch size, hid dim * num directions]

    return self.fc(hidden)
```

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

Make Advanced Model

Add ***dropout layer*** for every linear layer.

```python
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))
```

Address Dropout

```python
def forward(self, text, text_lengths):

    #text = [sent len, batch size]

    embedded = self.dropout(self.embedding(text))

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

    #pack sequence
    packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)

    packed_output, (hidden, cell) = self.rnn(packed_embedded)

    #unpack sequence
    output, output_lenghts = nn.utils.rnn.pad_packed_sequence(packed_output)

    #output = [sent len, batch size, hid dim, hid dim * num directions]
    #output over padding tokens are zero tensors

    #hidden = [num layers * num directions, batch size, hid dim]
    #cell = [num layers * num directions, batch size, hid dim]

    #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
    #and apply dropout

    hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))

    #hidden = [batch size, hid dim * num directions]

    return self.fc(hidden)
```

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

Make Advanced Model

For pack padded sequence, using ***nn.utils.rnn.pack_padded_sequence*** and ***nn.utils.rnn.pad_packed_sequence***.

```
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))
```

```
def forward(self, text, text_lengths):

    #text = [sent len, batch size]

    embedded = self.dropout(self.embedding(text))

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

    #pack sequence
    packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)

    packed_output, (hidden, cell) = self.rnn(packed_embedded)

    #unpack sequence
    output, output_lenghts = nn.utils.rnn.pad_packed_sequence(packed_output)

    #output = [sent len, batch size, hid dim, hid dim * num directions]
    #output over padding tokens are zero tensors

    #hidden = [num layers * num directions, batch size, hid dim]
    #cell = [num layers * num directions, batch size, hid dim]

    #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
    #and apply dropout

    hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))

    #hidden = [batch size, hid dim * num directions]

    return self.fc(hidden)
```

Address pack-padded sequence

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

Make Advanced Model

Get LSTM output, and make it to ***one hidden*** by concat those using ***torch.cat().***

```python
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))
```

```python
def forward(self, text, text_lengths):

    #text = [sent len, batch size]

    embedded = self.dropout(self.embedding(text))

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

    #pack sequence
    packed_embedded = nn.utils.rnn.pack_padded_sequence(embedded, text_lengths)

    packed_output, (hidden, cell) = self.rnn(packed_embedded)

    #unpack sequence
    output, output_lenghts = nn.utils.rnn.pad_packed_sequence(packed_output)

    #output = [sent len, batch size, hid dim * num directions]
    #output over padding tokens are zero tensors

    #hidden = [num layers * num directions, batch size, hid dim]
    #cell = [num layers * num directions, batch size, hid dim]

    #concat the final forward (hidden[-2,:,:]) and backward (hidden[-1,:,:]) hidden layers
    #and apply dropout

    hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))

    #hidden = [batch size, hid dim * num directions]

    return self.fc(hidden)
```

Get hidden and concat two directional hiddens

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

Make Advanced Model

Add additional arguments when building model.

```
[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)
```

```
[ ]  INPUT_DIM = len(TEXT.vocab)
     EMBEDDING_DIM = 100
     HIDDEN_DIM = 256
     OUTPUT_DIM = 1
     N_LAYERS=2
     BIDIRECTIONAL=True
     DROPOUT=0.5
     PAD_IDX=TEXT.vocab.stoi[TEXT.pad_token]

     model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM,
                 N_LAYERS, BIDIRECTIONAL, DROPOUT, PAD_IDX)
```

Add additional arguments

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

Make Advanced Model

Afterwards, add this code after building model.

This code is for ***loading pretrained embeddings*** from GloVe to the embedding layer of our model.

Sure that ***we must initialize embedding of '<unk>' and '<pad>' to zero*** because these are not related to classifying sentiment.

```
pretrained_embeddings = TEXT.vocab.vectors

print(pretrained_embeddings.shape)

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

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)
```

```
torch.Size([25002, 100])
```

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

Make Advanced Model

We can see ***loaded weights for embedding layer***.

The first tow rows of the embedding weights matrix have been set to zeros.

Because we set ‘<pad>’ as padding_idx of the embedding layer, embedding of <pad> will remain zeros throughout training, however the <unk> token embedding will be learned.

```
print(model.embedding.weight.data)
```

```
tensor([[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
        [-0.0382, -0.2449,  0.7281,  ..., -0.1459,  0.8278,  0.2706],
        ...,
        [ 0.2455, -0.0385, -0.4767,  ..., -0.2939, -0.0752,  0.0441],
        [ 0.4327,  0.3958,  0.5878,  ..., -1.1461,  0.2348, -0.2359],
        [-0.3970,  0.4024,  1.0612,  ..., -0.0136, -0.3363,  0.6442]],
       device='cuda:0')
```

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

Make Advanced Model

Then, in training, we can make improvements by changing optimizer.

Let's use Adam optimizer for this time.

```
import torch.optim as optim

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

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

Make Advanced Model

For now, let's train our reinforced model!

Don't forget to change input for model because we add 'text_length' argument for pack padded sequences.

```python
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)
```

```python
def train(model, iterator, optimizer, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        text, text_lengths = batch.text

        predictions = model(text, text_lengths).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)
```

Change input for pack padded sequeces

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

Make Advanced Model

It also needs to be addressed to evaluate function.

```
[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)
```

```
def evaluate(model, iterator, criterion):

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():

        for batch in iterator:

            text, text_lengths = batch.text

            predictions = model(text, text_lengths).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)
```

Change input for pack padded sequeces

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

Advanced Model Result

By addressing all advanced method, we can get much better result during training!

You can see that the advanced model takes ***much longer time*** but results in ***much better accuracy***.

```
Epoch: 01 | Epoch Time: 0m 45s
	Train Loss: 0.694 | Train Acc: 50.25%
	 Val. Loss: 0.697 |  Val. Acc: 48.67%
Epoch: 02 | Epoch Time: 0m 44s
	Train Loss: 0.693 | Train Acc: 49.86%
	 Val. Loss: 0.697 |  Val. Acc: 49.21%
Epoch: 03 | Epoch Time: 0m 45s
	Train Loss: 0.693 | Train Acc: 49.85%
	 Val. Loss: 0.697 |  Val. Acc: 49.83%
Epoch: 04 | Epoch Time: 0m 45s
	Train Loss: 0.693 | Train Acc: 50.04%
	 Val. Loss: 0.697 |  Val. Acc: 48.53%
Epoch: 05 | Epoch Time: 0m 45s
	Train Loss: 0.693 | Train Acc: 50.39%
	 Val. Loss: 0.697 |  Val. Acc: 49.98%
```

<Base Model Result>

```
Epoch: 01 | Epoch Time: 1m 39s
	Train Loss: 0.662 | Train Acc: 59.67%
	 Val. Loss: 0.657 |  Val. Acc: 60.47%
Epoch: 02 | Epoch Time: 1m 40s
	Train Loss: 0.570 | Train Acc: 70.95%
	 Val. Loss: 0.433 |  Val. Acc: 80.95%
Epoch: 03 | Epoch Time: 1m 39s
	Train Loss: 0.431 | Train Acc: 81.24%
	 Val. Loss: 0.340 |  Val. Acc: 85.87%
Epoch: 04 | Epoch Time: 1m 39s
	Train Loss: 0.334 | Train Acc: 86.02%
	 Val. Loss: 0.310 |  Val. Acc: 86.93%
Epoch: 05 | Epoch Time: 1m 39s
	Train Loss: 0.293 | Train Acc: 88.13%
	 Val. Loss: 0.357 |  Val. Acc: 83.99%
```

<Advanced Model Result>

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

Advanced Model Result

Test result using advanced model is also better than the base model.

Let’s try to build classifier with ***at least 80% accuracy*** on the test set ☺

Test Loss: 0.712 | Test Acc: 45.47%

<Base Model Result>

Test Loss: 0.347 | Test Acc: 84.79%

<Advanced Model Result>

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

Assignments

We have 5 assignments for Text Classification – Movie Review Sentiment Analysis.

각 assignment에 대하여 출력 cell을 캡처하여 word 문서로 작성하여 pdf로 하여 보내주시면 됩니다.
(45, 96, 97번 슬라이드 참조)
가능하시다면, Advanced method를 적용한 ipynb 파일을 같이 첨부하여 주시면 감사드리겠습니다.

1. Train with base model which includes ***basic RNN*** with 5 epochs. (2 points)
2. During (1.), report your train and valid accuracy for every epoch and report test accuracy. (2 points)
3. Train using ***advanced method at least with multi-layer bidirectional LSTM*** with 5 epochs (2 points)
4. During (3.), report your train and valid accuracy for every epoch and report test accuracy. (2 points)
5. Report the number of parameters in the two layered bidirectional LSTM. (2 points)

Submission to ***zzxc1133@kaist.ac.kr***