Recurrent Neural Networks

1. IMDB Review Classification Battlefield - Contestants : Feedforward, CNN, RNN, LSTM

In this task, we are going to do sentiment classification on a movie review dataset. We are going to build a feedforward net, a convolutional neural net, a recurrent net and combine one or more of them to understand performance of each of them. A sentence can be thought of as a sequence of words which have semantic connections across time. By semantic connection, we mean that the words that occur earlier in the sentence influence the sentence's structure and meaning in the latter part of the sentence. There are also semantic connections backwards in a sentence, in an ideal case (in which we use RNNs from both directions and combine their outputs). But for the purpose of this tutorial, we are going to restrict ourselves to only uni-directional RNNs.

In [1]:
```
import numpy as np
# fix random seed for reproducibility
np.random.seed(1)
```

In [2]:
```
# We want to have a finite vocabulary to make sure that our word matrices are not arbitrarily small
vocabulary_size = 10000

#We also want to have a finite length of reviews and not have to process really long sentences.
max_review_length = 500
```

TOKENIZATION

For practical data science applications, we need to convert text into tokens since the machine understands only numbers and not really English words like humans can. As a simple example of tokenization, we can see a small example.

Assume we have 5 sentences. This is how we tokenize them into numbers once we create a dictionary.

1. i have books - [1, 4, 7]
2. interesting books are useful [10,2,9,8]
3. i have computers [1,4,6]
4. computers are interesting and useful [6,9,11,10,8]
5. books and computers are both valuable. [2,10,2,9,13,12]
6. Bye Bye [7,7]

Create tokens for vocabulary based on frequency of occurrence. Hence, we assign the following tokens

I-1, books-2, computers-3, have-4, are-5, computers-6,bye-7, useful-8, are-9, and-10,interesting-11, valuable-12, both-13

Thankfully, in our dataset it is internally handled and each sentence is represented in such tokenized form.

Load data

In [3]:
```
from keras.datasets import imdb
from keras.preprocessing import sequence
```

```
Using TensorFlow backend.
/home/daryna/.local/lib/python3.7/site-packages/tensorflow/python
/framework/dtypes.py:516: FutureWarning: Passing (type, 1) or '1t
ype' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/daryna/.local/lib/python3.7/site-packages/tensorflow/python
/framework/dtypes.py:517: FutureWarning: Passing (type, 1) or '1t
ype' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/daryna/.local/lib/python3.7/site-packages/tensorflow/python
/framework/dtypes.py:518: FutureWarning: Passing (type, 1) or '1t
ype' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/daryna/.local/lib/python3.7/site-packages/tensorflow/python
/framework/dtypes.py:519: FutureWarning: Passing (type, 1) or '1t
ype' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/daryna/.local/lib/python3.7/site-packages/tensorflow/python
/framework/dtypes.py:520: FutureWarning: Passing (type, 1) or '1t
ype' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/daryna/.local/lib/python3.7/site-packages/tensorflow/python
/framework/dtypes.py:525: FutureWarning: Passing (type, 1) or '1t
ype' as a synonym of type is deprecated; in a future version of n
umpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/t
ensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning: P
assing (type, 1) or '1type' as a synonym of type is deprecated; i
n a future version of numpy, it will be understood as (type,
(1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/t
ensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning: P
assing (type, 1) or '1type' as a synonym of type is deprecated; i
n a future version of numpy, it will be understood as (type,
(1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/t
ensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning: P
assing (type, 1) or '1type' as a synonym of type is deprecated; i
n a future version of numpy, it will be understood as (type,
(1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/t
ensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning: P
assing (type, 1) or '1type' as a synonym of type is deprecated; i
n a future version of numpy, it will be understood as (type,
(1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/t
ensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning: P
assing (type, 1) or '1type' as a synonym of type is deprecated; i
n a future version of numpy, it will be understood as (type,
```

```
(1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/t
ensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning: P
assing (type, 1) or '1type' as a synonym of type is deprecated; i
n a future version of numpy, it will be understood as (type,
(1,)) / '(1,)type'.
  _np_resource = np.dtype([("resource", np.ubyte, 1)])
```

In [4]:

```
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=v
ocabulary_size)

print('Number of reviews', len(X_train))
print('Length of first and fifth review before padding', len(X_tr
ain[0]) ,len(X_train[4]))
print('First review', X_train[0])
print('First label', y_train[0])
```

```
/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/k
eras/datasets/imdb.py:101: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a list-or-tuple of
lists-or-tuples-or ndarrays with different lengths or shapes) is
deprecated. If you meant to do this, you must specify 'dtype=obje
ct' when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])

Number of reviews 25000
Length of first and fifth review before padding 218 147
First review [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4
468, 66, 3941, 4, 173, 36, 256, 5, 25, 100, 43, 838, 112, 50, 67
0, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2, 336, 385, 39,
4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147,
2025, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43,
530, 38, 76, 15, 13, 1247, 4, 22, 17, 515, 17, 12, 16, 626, 18,
2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2223, 5244, 16, 480, 66,
3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135, 48, 2
5, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2,
8, 4, 107, 117, 5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43,
530, 476, 26, 400, 317, 46, 7, 4, 2, 1029, 13, 104, 88, 4, 381, 1
5, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18, 4, 226, 22,
21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92,
25, 104, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472,
113, 103, 32, 15, 16, 5345, 19, 178, 32]
First label 1

/home/daryna/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/k
eras/datasets/imdb.py:102: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a list-or-tuple of
lists-or-tuples-or ndarrays with different lengths or shapes) is
deprecated. If you meant to do this, you must specify 'dtype=obje
ct' when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

Preprocess data

Pad sequences in order to ensure that all inputs have same sentence length and dimensions.

In [5]:
```
X_train = sequence.pad_sequences(X_train, maxlen=max_review_lengt
h)
X_test = sequence.pad_sequences(X_test, maxlen=max_review_length)
print('Length of first and fifth review after padding', len(X_tra
in[0]) ,len(X_train[4]))
```

```
Length of first and fifth review after padding 500 500
```

In [6]:
```
X_train.shape
```

Out[6]:
```
(25000, 500)
```

Models

In [7]:
```
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from tqdm.notebook import tqdm
from sklearn.model_selection import train_test_split
```

In [8]:
```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "c
pu")
if DEVICE.type == 'cuda':
    torch.set_default_tensor_type('torch.cuda.FloatTensor')
# DEVICE = torch.device("cpu")
print(DEVICE.type)
```

```
cuda
```

MODEL 1(a) : FEEDFORWARD NETWORKS WITHOUT EMBEDDINGS

Let us build a single layer feedforward net with 250 nodes. Each input would be a 500-dim vector of tokens since we padded all our sequences to size 500.

EXERCISE : Calculate the number of parameters involved in this network and implement a feedforward net to do classification without looking at cells below.

In [9]:
```
D_in = X_train.shape[1]
H = 250
D_out = 1

# X, y = torch.from_numpy(X_train).to(DEVICE), torch.from_numpy(y
_train).float().to(DEVICE)
```

In [10]:
```
epochs = 20
verbose = 1
learning_rate = 1e-2
batch_size=64
optimizer = torch.optim.Adam
criteria = nn.BCELoss(reduction='mean')
```

In [11]:
```
def fit_epoch(inputs, labels, model, criteria, optimizer):
    model.train()
    permutation = torch.randperm(inputs.size()[0])
    losses, accs = [], []

    for i in range(0,inputs.size()[0], batch_size):

        indices = permutation[i:i+batch_size]
        batch_x, batch_y = inputs[indices], labels[indices]

        output = model(batch_x)[:,0]
        optimizer.zero_grad()
        loss = criteria(output, batch_y.float())
        loss.backward()
        optimizer.step()

        preds = output > 0.5
        correct = (preds == batch_y).sum()
        acc = correct / float(batch_y.shape[0])

        losses.append(loss.item())
        accs.append(acc.item())
    losses, accs = np.array(losses), np.array(accs)
    return np.mean(losses), np.mean(accs)

def eval_epoch(inputs, labels, model, criteria):
    model.eval()
    ids = [i for i in range(inputs.size()[0])]
    losses, accs = [], []
    for i in range(0,inputs.size()[0], batch_size):

        indices = ids[i:i+batch_size]
        batch_x, batch_y = inputs[indices], labels[indices]

        with torch.set_grad_enabled(False):
            output = model(batch_x)[:,0]
            loss = criteria(output, batch_y.float())

            preds = output > 0.5
            correct = (preds == batch_y).sum()
            acc = correct / float(batch_y.shape[0])

        losses.append(loss.item())
        accs.append(acc.item())

    losses, accs = np.array(losses), np.array(accs)
    return np.mean(losses), np.mean(accs)


def train(X, y, X_val, y_val,
          model, epochs, verbose, learning_rate, criteria, optimi
zer, batch_size=64):

    inputs, labels = torch.from_numpy(X).to(DEVICE), torch.from_n
umpy(y).to(DEVICE)
    inputs_val, labels_val = torch.from_numpy(X_val).to(DEVICE),
torch.from_numpy(y_val).to(DEVICE)

    optimizer = optimizer(model.parameters(), lr=learning_rate)
```

```
    log_template = "\n[{ep:03d}/{epochs:03d}] train_loss: {t_los
s:0.4f} \
    val_loss {v_loss:0.4f} train_acc {t_acc:0.4f} val_acc {v_acc:
0.4f}"
    history = []
    for epoch in range(epochs):
        train_loss, train_acc = fit_epoch(inputs, labels, model,
criteria, optimizer)
        val_loss, val_acc = eval_epoch(inputs_val, labels_val, mo
del, criteria)

        history.append([train_loss, train_acc, val_loss, val_ac
c])
        if (epoch==0) or (epoch%verbose==0) or (epoch==epochs-1):
            print(log_template.format(ep=epoch+1, epochs=epochs,
t_loss=train_loss,
                                           v_loss=val_loss, t_acc
=train_acc, v_acc=val_acc))
    return history
```

In [12]:
```
class SimpleNet(nn.Module):

    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(D_in, H)
        self.out = nn.Linear(H, D_out)
        self.out_act = nn.Sigmoid()

    def forward(self, input_):
        a1 = self.fc1(input_.float())
        a2 = self.out(a1)
        y = self.out_act(a2)
        return y
```

In [13]:
```
simple_model = SimpleNet().to(DEVICE)
```

In [14]:
```
simple_history = train(X_train, y_train, X_test, y_test, simple_m
odel,
      epochs=15, verbose=1, learning_rate=1e-3, criteria=criteri
a, optimizer=optimizer)
```

```
[001/015] train_loss: 13.9541     val_loss 13.8124 train_acc 0.49
60 val_acc 0.5001

[002/015] train_loss: 13.8169     val_loss 13.8244 train_acc 0.50
00 val_acc 0.4999

[003/015] train_loss: 13.8123     val_loss 13.8175 train_acc 0.50
01 val_acc 0.5000

[004/015] train_loss: 13.8131     val_loss 13.8175 train_acc 0.50
01 val_acc 0.5000

[005/015] train_loss: 13.8151     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[006/015] train_loss: 13.8171     val_loss 13.8175 train_acc 0.49
99 val_acc 0.5000

[007/015] train_loss: 13.8151     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[008/015] train_loss: 13.8157     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[009/015] train_loss: 13.8164     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[010/015] train_loss: 13.8144     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[011/015] train_loss: 13.8144     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[012/015] train_loss: 13.8137     val_loss 13.8175 train_acc 0.50
01 val_acc 0.5000

[013/015] train_loss: 13.8190     val_loss 13.8175 train_acc 0.49
99 val_acc 0.5000

[014/015] train_loss: 13.8157     val_loss 13.8175 train_acc 0.50
00 val_acc 0.5000

[015/015] train_loss: 13.8137     val_loss 13.8175 train_acc 0.50
01 val_acc 0.5000
```

Discussion : Why was the performance bad ? What was wrong with tokenization ?

MODEL 1(b) : FEEDFORWARD NETWORKS WITH EMBEDDINGS

What is an embedding layer ?

An embedding is a linear projection from one vector space to another. We usually use embeddings to project the one-hot encodings of words on to a lower-dimensional continuous space so that the input surface is dense and possibly smooth. According to the model, an embedding layer is just a transformation from $\mathbb{R}^{inp}$ to $\mathbb{R}^{emb}$

Do embedding to dim 100 (in keras, tf, PyTorch: with Embedding layer) and after flattening add a dense layer with 250 units. Fit the model.

In [15]:
```
vocabulary_size
```

Out[15]: 10000

In [16]:
```
learning_rate = 1e-3
```

In [17]:
```
H_emb = 100
class EmbeddingNet(nn.Module):

    def __init__(self):
        super().__init__()
        self.emb = nn.Embedding(vocabulary_size, H_emb)
        self.dp1 = torch.nn.Dropout(0.5)
        self.fc1 = nn.Linear(H_emb * D_in, H)
        self.dp2 = torch.nn.Dropout(0.5)
        self.out = nn.Linear(H, D_out)
        self.out_act = nn.Sigmoid()

    def forward(self, input_):
        emb = self.emb(input_.long()).view((input_.size(0), -1))
        dp1 = self.dp1(emb)
        a1 = self.fc1(dp1)
        dp2 = self.dp2(a1)
        a2 = self.out(dp2)
        y = self.out_act(a2)
        return y
```

In [18]:
```
emb_model = EmbeddingNet().to(DEVICE)
emb_history = train(X_train, y_train, X_test, y_test, emb_model,
      epochs=50, verbose=5, learning_rate=1e-3, criteria=criteri
a, optimizer=optimizer)
```

```
[001/050] train_loss: 7.4095     val_loss 0.8028 train_acc 0.5088
val_acc 0.5114

[006/050] train_loss: 0.3679     val_loss 0.4482 train_acc 0.8364
val_acc 0.7992

[011/050] train_loss: 0.2108     val_loss 0.4478 train_acc 0.9165
val_acc 0.8354

[016/050] train_loss: 0.1500     val_loss 0.6077 train_acc 0.9457
val_acc 0.8419

[021/050] train_loss: 0.1260     val_loss 0.6249 train_acc 0.9592
val_acc 0.8430

[026/050] train_loss: 0.1059     val_loss 0.8599 train_acc 0.9688
val_acc 0.8516

[031/050] train_loss: 0.0935     val_loss 1.1010 train_acc 0.9743
val_acc 0.8517

[036/050] train_loss: 0.0971     val_loss 1.1076 train_acc 0.9771
val_acc 0.8550

[041/050] train_loss: 0.0828     val_loss 1.5285 train_acc 0.9829
val_acc 0.8564

[046/050] train_loss: 0.0667     val_loss 1.6448 train_acc 0.9870
val_acc 0.8596

[050/050] train_loss: 0.1006     val_loss 1.9589 train_acc 0.9846
val_acc 0.8570
```

MODEL 2 : CONVOLUTIONAL NEURAL NETWORKS

Text can be thought of as 1-dimensional sequence and we can apply 1-D Convolutions over a set of words. Let us walk through convolutions on text data with this blog.

[http://debajyotidatta.github.io/nlp/deep/learning/word-embeddings/2016/11/27/Understanding-Convolutions-In-Text/ (http://debajyotidatta.github.io/nlp/deep/learning/word-embeddings/2016/11/27/Understanding-Convolutions-In-Text/)](http://debajyotidatta.github.io/nlp/deep/learning/word-embeddings/2016/11/27/Understanding-Convolutions-In-Text/)

Fit a 1D convolution with 200 filters, kernel size 3 followed by a feedforward layer of 250 nodes and ReLU, sigmoid activations as appropriate.

In [19]:
```
D_in
```

Out[19]:
```
500
```

In [20]:
```
H_conv = 200
H=250
C_in = 1
k_size = 3

class ConvNet(nn.Module):

    def __init__(self):
        super().__init__()
        self.c1 =nn.Conv1d(C_in, H_conv, kernel_size=k_size, padd
ing=1)
#         self.p1 = nn.AvgPool1d(H_conv)
        self.fc1 = nn.Linear(H_conv*D_in, H)
        self.fc2 = nn.Linear(H, D_out)
        self.out = nn.ReLU()
        self.out_act = nn.Sigmoid()

    def forward(self, input_):
        conv = self.c1(input_.float().view(input_.size(0), 1, inp
ut_.size(1))) # (N, C_in, L)
        a1 = self.fc1(conv.view(input_.size(0), -1))
        a2 = self.fc2(a1)
        a3 = self.out(a2)
        y = self.out_act(a3)
        return y
```

In [21]:
```
conv_model = ConvNet().to(DEVICE)
conv_history = train(X_train, y_train, X_test, y_test, conv_mode
l,
      epochs=15, verbose=5, learning_rate=1e-3, criteria=criteri
a, optimizer=optimizer)
```

```
[001/015] train_loss: 0.7083     val_loss 0.6931 train_acc 0.4999
val_acc 0.4999

[006/015] train_loss: 0.6931     val_loss 0.6931 train_acc 0.5001
val_acc 0.4999

[011/015] train_loss: 0.6931     val_loss 0.6931 train_acc 0.4999
val_acc 0.4999

[015/015] train_loss: 0.6931     val_loss 0.6931 train_acc 0.5001
val_acc 0.4999
```

MODEL 3 : SIMPLE RNN

Two of the best blogs that help understand the workings of a RNN and LSTM are

1. http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)
2. http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (http://colah.github.io/posts/2015-08-Understanding-LSTMs/)

Mathematically speaking, a simple RNN does the following. It constructs a set of hidden states using the state variable from the previous timestep and the input at current time. Mathematically, a simpleRNN can be defined by the following relation.

$h_t = \sigma(W([h_{t-1},x_{t}])+b)$

If we extend this recurrence relation to the length of sequences we have in hand, we have our RNN network constructed.

Do simple RNN (keras, rf: SimpleRNN layer, pytorch: RNN layer) with 100 units with the input from embedding layer. How are the results different from the previous model?

In [22]:
```
H_emb = 100
class RNNNet(nn.Module):

    def __init__(self):
        super().__init__()
        self.emb = nn.Embedding(vocabulary_size, H_emb)
        self.rnn = nn.RNN(H_emb, H)
        self.dp1 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(D_in * H, H)
        self.dp2 = nn.Dropout(0.5)
        self.out = nn.Linear(H, D_out)
        self.out_act = nn.Sigmoid()

    def forward(self, input_):
        emb = self.emb(input_.long())
        rnn, hid = self.rnn(emb)
        rnn = self.dp1(rnn)
        a1 = self.fc1(rnn.view((input_.size(0), -1)))
        a1 = self.dp2(a1)
        a2 = self.out(a1)
        y = self.out_act(a2)
        return y
```

In [24]:
```
rnn_model = RNNNet().to(DEVICE)
rnn_history = train(X_train, y_train, X_test, y_test, rnn_model,
      epochs=20, verbose=5, learning_rate=1e-4, criteria=criteri
a, optimizer=optimizer)
```

```
[001/020] train_loss: 0.7962     val_loss 0.6883 train_acc 0.5377
val_acc 0.5865

[006/020] train_loss: 0.3621     val_loss 0.6388 train_acc 0.8364
val_acc 0.6993

[011/020] train_loss: 0.2443     val_loss 0.6584 train_acc 0.8953
val_acc 0.7383

[016/020] train_loss: 0.1821     val_loss 0.7017 train_acc 0.9236
val_acc 0.7557

[020/020] train_loss: 0.1467     val_loss 0.7315 train_acc 0.9400
val_acc 0.7673
```

RNNs and vanishing/exploding gradients

Let us use sigmoid activations as example. Derivative of a sigmoid can be written as

$\sigma'(x) = \sigma(x) \cdot \sigma(1-x)$.

Remember RNN is a "really deep" feedforward network (when unrolled in time). Hence, backpropagation happens from h_t all the way to h_1. Also realize that sigmoid gradients are multiplicatively dependent on the value of sigmoid. Hence, if the non-activated output of any layer h_l is < 0, then σ tends to 0, effectively "vanishing" gradient. Any layer that the current layer backprops to $H_{1:L-1}$ do not learn anything useful out of the gradients.

LSTMs and GRU

LSTM and GRU are two sophisticated implementations of RNN which essentially are built on what we call as gates. A gate is a probability number between 0 and 1. For instance, LSTM is built on these state updates

Note : L is just a linear transformation L(x) = W*x + b.

$f_t = \sigma(L([h_{t-1},x_t))$

$i_t = \sigma(L([h_{t-1},x_t))$

$o_t = \sigma(L([h_{t-1},x_t))$

$\hat{C}_t = \tanh(L([h_{t-1},x_t))$

$C_t = f_t * C_{t-1}+i_t*\hat{C}_t$ (Using the forget gate, the neural network can learn to control how much information it has to retain or forget)

$h_t = o_t * \tanh(c_t)$

MODEL 4 : LSTM

In the next step, we will implement a LSTM model to do classification. Use the same architecture as before. Try experimenting with increasing the number of nodes, stacking multiple layers, applyong dropouts etc. Check the number of parameters that this model entails.

In [53]:
```
H_emb = 100
class LSTMNet(nn.Module):


    def __init__(self):
        super().__init__()
        self.emb = nn.Embedding(vocabulary_size, H_emb)
        self.lstm = nn.LSTM(H_emb, H)
        self.dp1 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(D_in * H, H)
        self.dp2 = nn.Dropout(0.5)
        self.out = nn.Linear(H, D_out)
        self.out_act = nn.Sigmoid()

    def forward(self, input_):
        emb = self.emb(input_.long())
#         print(emb.size())
        lstm, hid = self.lstm(emb)
        lstm = self.dp1(lstm)
        a1 = self.fc1(lstm.view((input_.size(0), -1)))
        a1 = self.dp2(a1)
        a2 = self.out(a1)
        y = self.out_act(a2)
        return y
```

In [54]:
```
lstm_model = LSTMNet().to(DEVICE)
lstm_history = train(X_train, y_train, X_test, y_test, lstm_mode
l,
      epochs=20, verbose=5, learning_rate=1e-4, criteria=criteri
a, optimizer=optimizer)
```

```
[001/020] train_loss: 0.7023     val_loss 0.6595 train_acc 0.5483
val_acc 0.6061

---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent
call last)
<ipython-input-54-77bd1d4fa8d2> in <module>
      1 lstm_model = LSTMNet().to(DEVICE)
      2 lstm_history = train(X_train, y_train, X_test, y_test, ls
tm_model,
----> 3       epochs=20, verbose=5, learning_rate=1e-4, criteria=
criteria, optimizer=optimizer)

<ipython-input-11-de9bc5ac7d9e> in train(X, y, X_val, y_val, mode
l, epochs, verbose, learning_rate, criteria, optimizer, batch_siz
e)
     59     history = []
     60     for epoch in range(epochs):
---> 61         train_loss, train_acc = fit_epoch(inputs, labels,
model, criteria, optimizer)
     62         val_loss, val_acc = eval_epoch(inputs_val, labels
_val, model, criteria)
     63

<ipython-input-11-de9bc5ac7d9e> in fit_epoch(inputs, labels, mode
l, criteria, optimizer)
     12         optimizer.zero_grad()
     13         loss = criteria(output, batch_y.float())
---> 14         loss.backward()
     15         optimizer.step()
     16

~/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/torch/tenso
r.py in backward(self, gradient, retain_graph, create_graph)
    193                 products. Defaults to ``False``.
    194         """
--> 195         torch.autograd.backward(self, gradient, retain_gr
aph, create_graph)
    196
    197     def register_hook(self, hook):

~/anaconda3/envs/ml_ukma/lib/python3.7/site-packages/torch/autogr
ad/__init__.py in backward(tensors, grad_tensors, retain_graph, c
reate_graph, grad_variables)
     97     Variable._execution_engine.run_backward(
     98         tensors, grad_tensors, retain_graph, create_grap
h,
---> 99         allow_unreachable=True)  # allow_unreachable flag
    100
    101

KeyboardInterrupt:
```

MODEL 5 : CNN + LSTM

CNNs are good at learning spatial features and sentences can be thought of as 1-D spatial vectors (dimension being connotated by the sequence ordering among the words in the sentence.). We apply a LSTM over the features learned by the CNN (after a maxpooling layer). This leverages the power of CNNs and LSTMs combined. We expect the CNN to be able to pick out invariant features across the 1-D spatial structure(i.e. sentence) that characterize good and bad sentiment. This learned spatial features may then be learned as sequences by an LSTM layer followed by a feedforward for classification.

In [132]:
```
H_conv = 200
H=250
k_size = 3
p_size = 5

class ConvLSTMNet(nn.Module):

    def __init__(self):
        super().__init__()
        self.c1 = nn.Conv1d(C_in, H_conv, k_size, padding=k_size
//2)
        self.p1 = nn.MaxPool1d(p_size)
        self.lstm = nn.LSTM(D_in, H)
        self.fc1 = nn.Linear((H_conv // p_size) * H, D_out)
        self.out = nn.ReLU()
        self.out_act = nn.Sigmoid()

    def forward(self, input_):
        c1 = self.c1(input_.float().view(input_.size(0), 1, input
_.size(1)))
        c1 = self.p1(c1.view(input_.size(0), D_in, H_conv))

        a1, hid = self.lstm(c1.transpose(1, 2).transpose(0, 1))
        a2 = self.fc1(a1.transpose(1,0).reshape((input_.size(0),
-1)))

        a3 = self.out(a2)
        y = self.out_act(a3)
        return y
```

In [133]:
```
convlstm_model = ConvLSTMNet().to(DEVICE)
convlstm_history = train(X_train, y_train, X_test, y_test, convls
tm_model,
      epochs=20, verbose=5, learning_rate=1e-3, criteria=criteri
a, optimizer=optimizer)
```

```
[001/020] train_loss: 0.6932     val_loss 0.6931 train_acc 0.5000
val_acc 0.4999

[006/020] train_loss: 0.6931     val_loss 0.6931 train_acc 0.4999
val_acc 0.4999

[011/020] train_loss: 0.6931     val_loss 0.6931 train_acc 0.5000
val_acc 0.4999

[016/020] train_loss: 0.6931     val_loss 0.6931 train_acc 0.5000
val_acc 0.4999

[020/020] train_loss: 0.6931     val_loss 0.6931 train_acc 0.5001
val_acc 0.4999
```

CONCLUSION

We saw the power of sequence models and how they are useful in text classification. They give a solid performance, low memory footprint (thanks to shared parameters) and are able to understand and leverage the temporally connected information contained in the inputs. There is still an open debate about the performance vs memory benefits of CNNs vs RNNs in the research community.