Part I. Data Processing

a) Import the following libraries:

In [1]:

```
import sys
import os
import json
import pandas as pd
import numpy as np
import optparse

from keras.callbacks import TensorBoard
from keras.models import Sequential, load_model
from keras.layers import LSTM, Dense, Dropout
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
from keras.preprocessing.text import Tokenizer
from collections import OrderedDict
```

```
Using TensorFlow backend.
```

b) We will read the code in slightly differently than before:

In [2]:

```
dataframe = pd.read_csv("dev-access.csv", engine='python', quotechar='|', header=Nor
```

c) We then need to convert to a numpy.ndarray type:

In [3]:

```
dataset = dataframe.values
```

d) Check the shape of the data set - it should be (26773, 2). Spend some time looking at the data.

In [4]:

```
dataset.shape
```

Out[4]:

```
(26773, 2)
```

In [9]:

```
dataset[0:5]
```

Out[9]:


```
array([['{"timestamp":1502738402847,"method":"post","query":{},"pat
h":"/login","statusCode":401,"source":{"remoteAddress":"88.141.113.23
7","referer":"http://localhost:8002/enter"},"route":"/login","header
s":{"host":"localhost:8002","accept-language":"en-us","accept-encodin
g":"gzip, deflate","connection":"keep-alive","accept":"*/*","refere
r":"http://localhost:8002/enter","cache-control":"no-cache","x-request
ed-with":"XMLHttpRequest","content-type":"application/json","content-l
ength":"36"},"requestPayload":{"username":"Carl2","password":"bo"},"re
sponsePayload":{"statusCode":401,"error":"Unauthorized","message":"Inv
alid Login"}}',
        0],
       ['{"timestamp":1502738402849,"method":"post","query":{},"pat
h":"/login","statusCode":401,"source":{"remoteAddress":"88.141.113.23
7"},"route":"/login","headers":{"host":"localhost:8002","connectio
n":"keep-alive","cache-control":"no-cache","accept":"*/*","accept-enco
ding":"gzip, deflate, br","accept-language":"en-US,en;q=0.8,es;q=0.
6","content-type":"application/json","content-length":"47"},"requestPa
yload":{"username":"pafzah","password":"worldburn432"},"responsePayloa
d":{"statusCode":401,"error":"Unauthorized","message":"Invalid Logi
n"}}',
        0],
       ['{"timestamp":1502738402852,"method":"post","query":{},"pat
h":"/login","statusCode":401,"source":{"remoteAddress":"205.49.83.11
8"},"route":"/login","headers":{"host":"localhost:8002","connectio
n":"keep-alive","cache-control":"no-cache","accept":"*/*","accept-enco
ding":"gzip, deflate, br","accept-language":"en-US,en;q=0.8,es;q=0.
6","content-type":"application/json","content-length":"44"},"requestPa
yload":{"username":"Panos1","password":"najrijkom"},"responsePayload":
{"statusCode":401,"error":"Unauthorized","message":"Invalid Login"}}',
        0],
       ['{"timestamp":1502738402852,"method":"post","query":{},"pat
h":"/login","statusCode":401,"source":{"remoteAddress":"205.49.83.11
8","referer":"http://localhost:8002/enter"},"route":"/login","header
s":{"host":"localhost:8002","accept-language":"en-us","accept-encodin
g":"gzip, deflate","connection":"keep-alive","accept":"*/*","refere
r":"http://localhost:8002/enter","cache-control":"no-cache","x-request
ed-with":"XMLHttpRequest","content-type":"application/json","content-l
ength":"47"},"requestPayload":{"username":"vuvpuvehu","password":"pass
word1"},"responsePayload":{"statusCode":401,"error":"Unauthorized","me
ssage":"Invalid Login"}}',
        0],
       ['{"timestamp":1502738402853,"method":"post","query":{},"pat
h":"/login","statusCode":401,"source":{"remoteAddress":"137.196.95.11
6"},"route":"/login","headers":{"host":"localhost:8002","connectio
n":"keep-alive","cache-control":"no-cache","accept":"*/*","accept-enco
ding":"gzip, deflate, br","accept-language":"en-US,en;q=0.8,es;q=0.
6","content-type":"application/json","content-length":"41"},"requestPa
yload":{"username":"Michele","password":"mokgu"},"responsePayload":{"s
tatusCode":401,"error":"Unauthorized","message":"Invalid Login"}}',
        0]], dtype=object)
```


e) Store all rows and the 0th index as the feature data:

In [10]:

```
X = dataset[:,0]
```

f) Store all rows and index 1 as the target variable:

In [11]:

```
Y = dataset[:,1]
```

g) In the next step, we will clean up the predictors. This includes removing features that are not valuable, such as timestamp and source.

In [14]:

```
for index, item in enumerate(X):
    # Quick hack to space out json elements
    reqJson = json.loads(item, object_pairs_hook=OrderedDict)
    del reqJson['timestamp']
    del reqJson['headers']
    del reqJson['source']
    del reqJson['route']
    del reqJson['responsePayload']
    X[index] = json.dumps(reqJson, separators=(',', ':'))
```

In [17]:

```
X[0]
```

Out[17]:

```
'{"method":"post","query":{},"path":"/login","statusCode":401,"request
Payload":{"username":"Carl2","password":"bo"}}'
```

h) We next will tokenize our data, which just means vectorizing our text. Given the data we will tokenize every character (thus char_level = True)

In [18]:

```
tokenizer = Tokenizer(filters='\t\n', char_level=True)
tokenizer.fit_on_texts(X)

# we will need this later
num_words = len(tokenizer.word_index)+1
X = tokenizer.texts_to_sequences(X)
```

In [20]:

```
X[0]
```

```
 17,
 5,
 9,
 12,
 28,
 1,
 10,
 1,
 13,
 5,
 6,
 6,
 32,
 7,
 9,
 11,
 1,
 4,
 1,
 40
```

i) Need to pad our data as each observation has a different length

In [21]:

```
max_log_length = 1024
X_processed = sequence.pad_sequences(X, maxlen=max_log_length)
```

j) Create your train set to be 75% of the data and your test set to be 25%

In [24]:

```
from sklearn.model_selection import train_test_split
SEED = 42
# Split into train and test data, randome_state = SEED for reproducibility
X_train, X_test, y_train, y_test= train_test_split(X_processed, Y, test_size=0.25, r
```

Part II. Model 1 - RNN

The first model will be a pretty minimal RNN with only an embedding layer, simple RNN and Dense layer. The next model we will add a few more layers.

a) Start by creating an instance of a Sequential model

In [25]:

```
model = Sequential()
```

b) From there, add an Embedding layer:

Params:

- input_dim = num_words (the variable we created above)
- output_dim = 32
- input_length = max_log_length (we also created this above)
- Keep all other variables as the defaults (shown below)

In [26]:

```
# Embedding layer
model.add(Embedding(input_dim = num_words, output_dim = 32, input_length = max_log_l
```

c) Add a SimpleRNN layer:

Params:

- units = 32
- activation = 'relu'

In [27]:

```
# SimpleRNN layer
from keras.layers import SimpleRNN
model.add(SimpleRNN(units = 32, activation = "relu"))
```

d) Finally, we will add a Dense layer:

Params:

- units = 1 (this will be our output)
- activation --> you can choose to use either relu or sigmoid.

In [28]:

```
# Dense layer
model.add(Dense(units = 1, activation = "sigmoid"))
```

e) Compile model using the .compile() method:

Params:

- loss = binary_crossentropy
- optimizer = adam
- metrics = accuracy

In [29]:

```
model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy
```

f) Print the model summary

In [30]:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 1024, 32)          2016
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 32)                2080
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 4,129
Trainable params: 4,129
Non-trainable params: 0
_________________________________________________________________
```

g) Use the .fit() method to fit the model on the train data. Use a validation split of 0.25, epochs=3 and batch size = 128.

In [33]:

```
model.fit(X_train, y_train, epochs = 3, validation_split = 0.25, batch_size = 128)
```

```
/Users/arielsmac/anaconda3/lib/python3.7/site-packages/tensorflow/pyth
on/framework/indexed_slices.py:434: UserWarning: Converting sparse Ind
exedSlices to a dense Tensor of unknown shape. This may consume a larg
e amount of memory.
  "Converting sparse IndexedSlices to a dense Tensor of unknown shape.
"

Train on 15059 samples, validate on 5020 samples
Epoch 1/3
15059/15059 [==============================] - 32s 2ms/step - loss: 0.
5743 - accuracy: 0.6749 - val_loss: 0.2578 - val_accuracy: 0.9211
Epoch 2/3
15059/15059 [==============================] - 30s 2ms/step - loss: 0.
2124 - accuracy: 0.9183 - val_loss: 0.1491 - val_accuracy: 0.9452
Epoch 3/3
15059/15059 [==============================] - 36s 2ms/step - loss: 0.
2186 - accuracy: 0.9155 - val_loss: 0.1246 - val_accuracy: 0.9596
```

Out[33]:

```
<keras.callbacks.callbacks.History at 0x15c657f60>
```

h) Use the .evaluate() method to get the loss value & the accuracy value on the test data. Use a batch size of 128 again.

In [34]:

```
model.evaluate(X_test, y_test, batch_size = 128)
```

```
6694/6694 [==============================] - 3s 462us/step
```

Out[34]:

```
[0.1250047843145334, 0.9596653580665588]
```

Part III. Model 2 - LSTM + Dropout Layers

Now we will add a few new layers to our RNN and incorporate the more powerful LSTM. You will be creating a new model here, so make sure to call it something different than the model from Part 2.

a) This RNN needs to have the following layers (add in this order):

- Embedding Layer (use same params as before)
- LSTM Layer (units = 64, recurrent_dropout = 0.5)
- Dropout Layer - use a value of 0.5
- Dense Layer - (use same params as before)

In [35]:

```
model2 = Sequential()
# Embedding layer
model2.add(Embedding(input_dim = num_words, output_dim = 32, input_length = max_log_
# LSTM layer
model2.add(LSTM(units = 64, recurrent_dropout = 0.5))
# Dropout layer for regularization
model2.add(Dropout(rate = 0.5))
# Dense layer
model2.add(Dense(units = 1, activation = "sigmoid"))
```

b) Compile model using the .compile() method:

Params:

- loss = binary_crossentropy
- optimizer = adam
- metrics = accuracy

In [36]:

```
model2.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accurac
```

c) Print the model summary

In [37]:

```
model2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 1024, 32)          2016
_________________________________________________________________
lstm_1 (LSTM)                (None, 64)                24832
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 65
=================================================================
Total params: 26,913
Trainable params: 26,913
Non-trainable params: 0
_________________________________________________________________
```

d) Use the .fit() method to fit the model on the train data. Use a validation split of 0.25, epochs=3 and batch size = 128.

In [39]:

```
model2.fit(X_train, y_train, epochs = 3, validation_split = 0.25, batch_size = 128)
```

```
Train on 15059 samples, validate on 5020 samples
Epoch 1/3
15059/15059 [==============================] - 100s 7ms/step - loss:
0.5791 - accuracy: 0.6838 - val_loss: 0.4239 - val_accuracy: 0.8651
Epoch 2/3
15059/15059 [==============================] - 98s 7ms/step - loss: 0.
3684 - accuracy: 0.8718 - val_loss: 0.2018 - val_accuracy: 0.9510
Epoch 3/3
15059/15059 [==============================] - 98s 6ms/step - loss: 0.
2582 - accuracy: 0.9228 - val_loss: 0.1721 - val_accuracy: 0.9498
```

Out[39]:

```
<keras.callbacks.callbacks.History at 0x16205bf60>
```

e) Use the .evaluate() method to get the loss value & the accuracy value on the test data. Use a batch size of 128 again.

In [40]:

```
model2.evaluate(X_test, y_test, batch_size = 128)
```

```
6694/6694 [==============================] - 11s 2ms/step
```

Out[40]:

```
[0.18120245510850838, 0.9474155902862549]
```

Part IV. Recurrent Neural Net Model 3: Build Your Own

You wil now create your RNN based on what you have learned from Model 1 & Model 2:

a) RNN Requirements:

- Use 5 or more layers
- Add a layer that was not utilized in Model 1 or Model 2 (Note: This could be a new Dense layer or an additional LSTM)

In [49]:

```
model3 = Sequential()

# Embedding layer
model3.add(Embedding(input_dim = num_words, output_dim = 32, input_length = max_log_

# LSTM layer
model3.add(LSTM(units = 64, recurrent_dropout = 0.5))

# Dense layer
model3.add(Dense(units = 64, activation='relu'))

# Dropout layer for regularization
model3.add(Dropout(rate = 0.5))

# Dense layer
model3.add(Dense(units = 1, activation = "sigmoid"))
```

b) Compiler Requirements:

- Try a new optimizer for the compile step
- Keep accuracy as a metric (feel free to add more metrics if desired)

In [54]:

```
model3.compile(optimizer = 'rmsprop', loss = 'binary_crossentropy', metrics = ['accu
```

c) Print the model summary

In [55]:

```
model3.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_5 (Embedding)      (None, 1024, 32)          2016
_________________________________________________________________
lstm_3 (LSTM)                (None, 64)                24832
_________________________________________________________________
dense_5 (Dense)              (None, 64)                4160
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 65
=================================================================
Total params: 31,073
Trainable params: 31,073
Non-trainable params: 0
_________________________________________________________________
```

d) Use the .fit() method to fit the model on the train data. Use a validation split of 0.25, epochs=3 and batch size = 128.

In [56]:

```
model3.fit(X_train, y_train, epochs = 3, validation_split = 0.25, batch_size = 128)
```

```
Train on 15059 samples, validate on 5020 samples
Epoch 1/3
15059/15059 [==============================] - 111s 7ms/step - loss:
0.5562 - accuracy: 0.7039 - val_loss: 0.2370 - val_accuracy: 0.9568
Epoch 2/3
15059/15059 [==============================] - 107s 7ms/step - loss:
0.3324 - accuracy: 0.8847 - val_loss: 0.1531 - val_accuracy: 0.9627
Epoch 3/3
15059/15059 [==============================] - 106s 7ms/step - loss:
0.2597 - accuracy: 0.9150 - val_loss: 0.1537 - val_accuracy: 0.9576
```

Out[56]:

```
<keras.callbacks.callbacks.History at 0x15d047f98>
```

e) Use the .evaluate() method to get the loss value & the accuracy value on the test data. Use a batch size of 128 again.

In [57]:

```
model3.evaluate(X_test, y_test, batch_size = 128)
```

```
6694/6694 [==============================] - 11s 2ms/step
```

Out[57]:

```
[0.14574184229757167, 0.9589184522628784]
```

Part V. Conceptual Questions

5) Explain the difference between the relu activation function and the sigmoid activation function.

An activation function in neural network decides whether a neuron gets activated, and relu and sigmoid represent 2 different activation functions that we can choose from. The plot of the sigmoid function is shown below. The sigmoid function will transform an input value into an output between 0 and 1. When used in a neural network, this function would cause the inputs that ranged between -2 and 2 to be sensitive to change, while inputs outside of this range would become saturated at output values of 1 and 0. Because neural network uses backpropagation to compute the error gradients and update the weights, if the output of the activation function is very small, it can result in the vanishing gradient problem where the lower layer connection weights remain unchanged and the training never converges to a good solution.

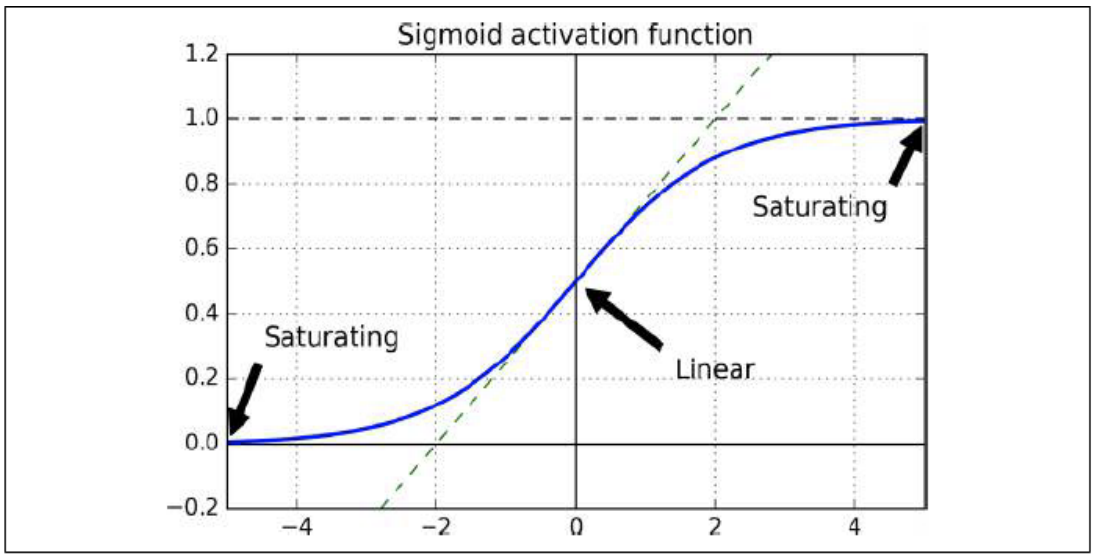
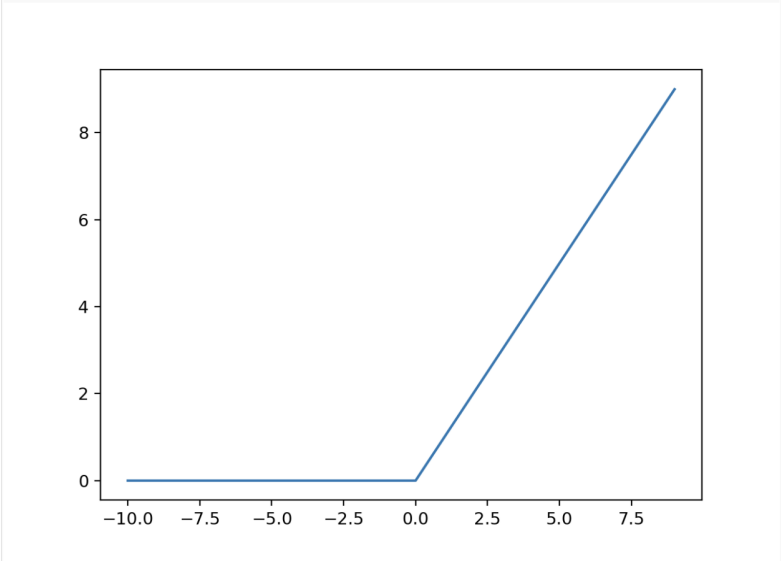


Figure 11-1. Logistic activation function saturation

The ReLu function gives an positive output if the input is positive, and 0 otherwise. Unlike sigmoid which is continuous throughout, ReLu is a piecewise function, because half of the output is linear (the positive output) while the other half is nonlinear. The linear part helps preserve properties that make linear models easy to optimize with gradient-based methods (no vanishing gradient) while allowing for complex relationships to be modeled. Unlike the sigmoid function that learns to approximate a zero output (i.e. a value very close to zero but not a true zero), the ReLu function is capable of outputting a true zero value. Such allowance of true zero values means some neurons will not be activated - this is called a sparse representation and is a desirable property as it can accelerate learning, reduce computation, and simplify the model. This makes ReLu less computationally taxing than sigmoid. However, a drawback of ReLu is the horizontal part where gradient is 0, as this cause some neurons to stop responding to variations in the input/error and is known as the dying ReLu problem. This issue can solved by using a variant of the ReLu function such as the LeaklyReLu.



Line Plot of Rectified Linear Activation for Negative and Positive Inputs

6) Describe what one epoch actually is (epoch was a parameter used in the .fit() method).

An epoch is a measure of the number of times all of the training vectors are used once to update the weights. In neural network, an epoch is completed when an entire dataset is passed both forward and backward through the network once.

Since one epoch is too big to feed to the model at once we divide it in several smaller batches, hence the batch_size is another parameter. One epoch is complete when all the batches in the dataset have been trained. Let's say we have a dataset of 2000 observations, and we devide them into batches of 500. 500 is the batchsize and it will take 4 iterations to complete 1 epoch. And during each iteration, the weights of the neurons are updated.

Passing the entire dataset through a neural network once (i.e. 1 epoch) is not enough. We need to pass the full dataset multiple times to the same neural network so the model's performance can improve. But it is also important to not go through too many training epochs as that would lead to the model learning all the noise in the training data and overfit.

7) Explain how dropout works (you can look at the keras code and/or documentation) for (a) training, and (b) test data sets.

Dropout happens only during training. At every training step, every neuron has a probability p (this is known as the dropout rate) of being temporarily dropped out, i.e. it is entirely ignored during this training step, but it may be active during the next step (see visualization below). After training, the neurons don't get dropped anymore. As such, a unique neural network is generated at each training step. Due to the different neural network at each step, the resulting neural network is an averaging ensemble of all these smaller neural networks. This makes dropout a popular and effective regularization technique.

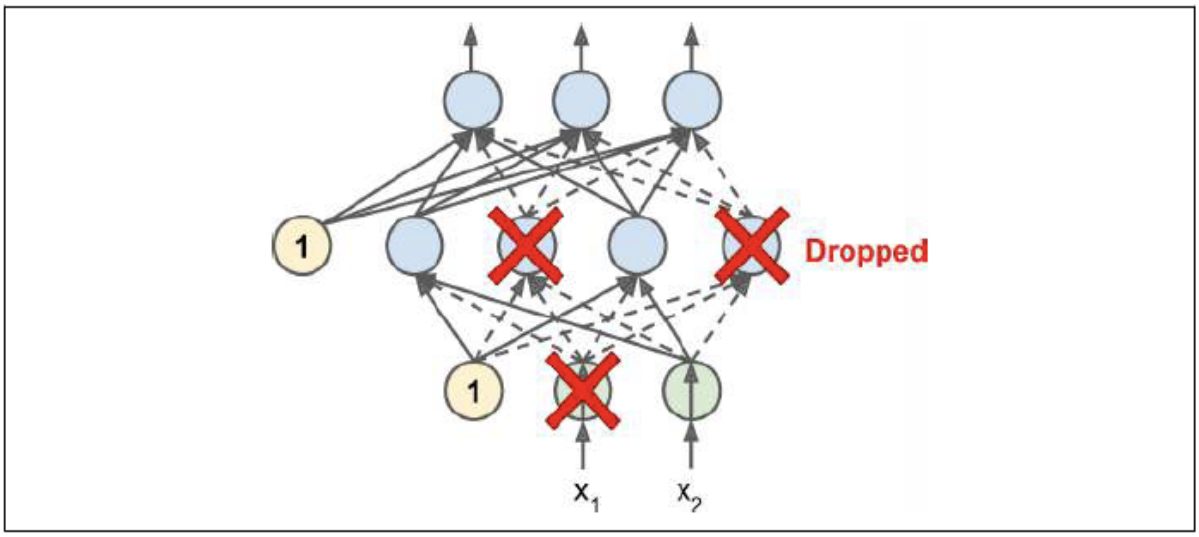


Figure 11-9. Dropout regularization

8) Explain why problems such as this homework assignment are better modeled with RNNs than CNNs. What type of problem will CNNs outperform RNNs on?

This dataset involves sequence modeling, and it is many-to-one sequence prediction that involved a sequence of multiple steps as input mapped to a class prediction. Traditional feed-forward neural networks like CNNs do not share features across different positions of the network. In other words, these models assume that all inputs (and outputs) are independent of each other, thus this model is not suitable for sequence prediction since the previous inputs are inherently important in predicting the next output.

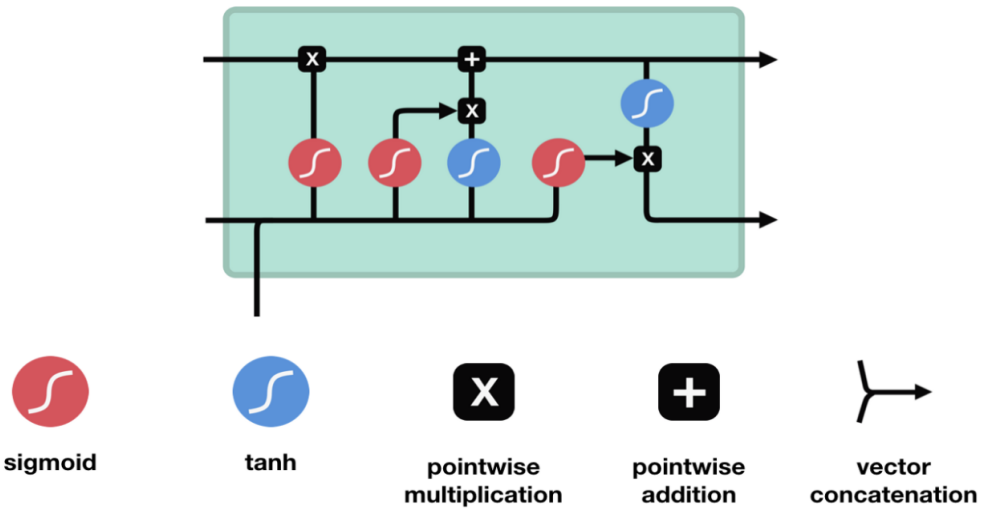
RNN is suitable for sequence modeling because it has a looping mechanism that acts as a highway to allow information to flow from one step to the next. This information is the hidden state, which is a representation of previous inputs. In addition, RNNs are able to maintain state between sequence elements, mapping input sequence to fixed-sized output vectors; this makes it preferrable for sequence modeling. The types of sequence-modeling RNN can handle are one-to-many, many-to-one and many-to-many sequence modeling, including sequences of text and spoken language represented a time series.

CNNs will outperform RNNs on image processing problems. Due to CNN's feature extraction property, it is able to develop an internal representation of an image. Therefore, CNN's pattern recognition is location invariant and can scale in variant strctures in the data - once it learns to recognize a pattern in one location, it can recognize it in any other location. This makes CNN particularly suitable for images and more generally, data with spatial relaationship.

9) Explain what RNN problem is solved using LSTM and briefly describe how.

LSTM solved the vanishing gradient problem in RNN, which is the culprit of RNN's short term memory. As the RNN processes more steps, it has trouble retaining information from previous steps. This is caused by the nature of back-propagation, an algorithm used to train and optimize neural networks. Back-propagation leverages gradients to update weights of a neural network. The gradient values will exponentially shrink as it propagates through each time step, resulting in small adjustment to the neural network weights and little contribution to learning, especially for the early layers. Because these layers don’t learn, the neural network can forget what it's seen in longer sequences, thus having a short-term memory.

LSTM is capable of learning long-term dependencies using the "gate" mechanism to regulate the flow of information. These gates are different tensor operations that can learn what information to add or remove to the hidden state. Because of this ability, short-term memory is less of an issue for them. Below is an illustration of the architecture of LSTM cell.



LSTM Cell and It's Operations

The core concept of LSTM's are the cell state and its various gates. The cell state act as a transport highway that transfers relative information down the sequence chain. Since the cell state can carry relevant information throughout the processing of the sequence, information from the earlier time steps can make its way to later time steps, reducing the effects of short-term memory. During this process, information gets added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state and they can learn what information is relevant to keep or forget during training via the sigmoid activation function.

In [ ]: