UNIT-III

RECURRENT NEURAL NETWORKS

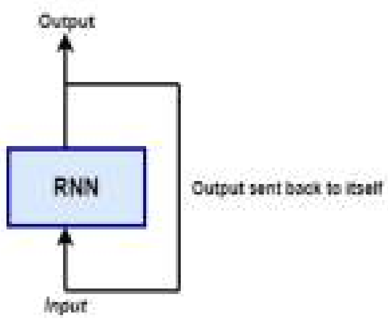
INTRODUCTION:

NEED FOR RNN:

- Convolutional neural network uses the local correlation of data and the idea of weight sharing.
- It is very suitable for pictures with spatial and local correlation.
- Text we are reading, speech signal while we speak, change of stock market over the time etc. are some signals which have spatial as well as temporal dimension
- This type of data does not necessarily have local relevance, and the length of the data in the time dimension is also variable.
- Convolutional neural networks are not good at processing such data.

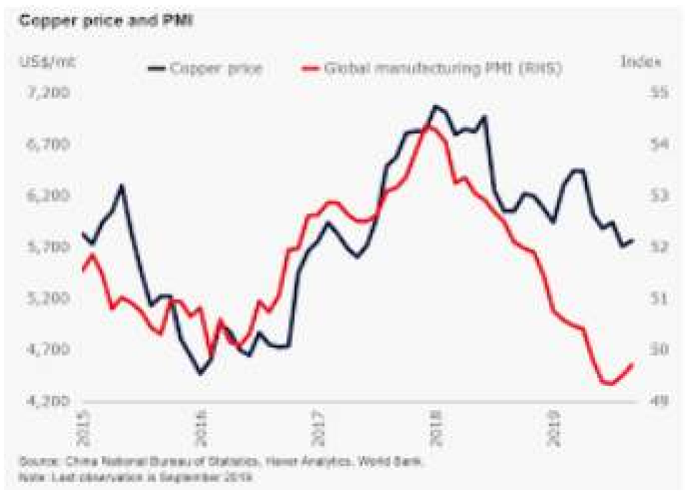
INTRODUCTION TO RNN:

- A recurrent neural network (**RNN**) is a kind of artificial neural network mainly used in **speech recognition** and **natural language processing** (NLP).
- RNN is used in deep learning and in the development of models that imitate the activity of neurons in the human **brain**.
- Recurrent Networks are designed to **recognize patterns** in sequences of data, such as **text, genomes, handwriting, the spoken word,** and **numerical** time series data emanating from sensors, stock markets, and government agencies.
- A recurrent neural network looks similar to a traditional neural network except that a memory-state is added to the neurons.
- The computation is to include a simple memory.
- The recurrent neural network is a type of deep learning-oriented algorithm, which follows a sequential approach.
- In neural networks, we always assume that each input and output is dependent on all other layers.
- These types of neural networks are called recurrent because they sequentially perform mathematical computations.



SEQUENCE REPRESENTATION METHOD:

- Data with order is generally called a sequence.
- **For example**, commodity price data that changes over time is a very typical sequence.
- Assume, price trend of a commodity between January to June as a 1D vector [*x*1, *x*2, *x*3, *x*4, *x*5, *x*6], and its shape is [6].



- **Example 2:**
- price change trend of ***b*** **goods** from January to June, you can record it as a 2-dimensional tensor:

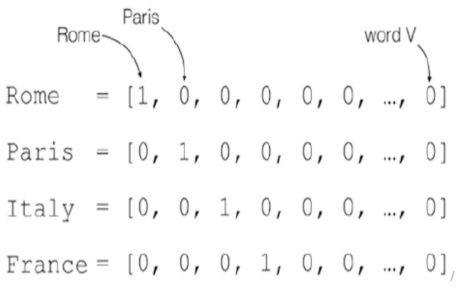
$$\left[\left[x_1^{(1)},x_2^{(1)},\cdots,x_6^{(1)}\right],\left[x_1^{(2)},x_2^{(2)},\cdots,x_6^{(2)}\right],\cdots,\left[x_1^{(b)},x_2^{(b)},\cdots,x_6^{(b)}\right]\right]$$

- where *b* represents the number of commodities, and the tensor shape is [*b*, 6].
- SO WHAT'S THE PROBLEM?? For the previous case, A tensor with shape [b, s] is needed, where **b is the number of sequences and s is the length of the sequence.**
- However, many signals cannot be directly represented by a scalar value.

- For example, to represent feature vectors of length n generated by each timestamp, a tensor of shape **[b, s, n]** is required.
- Consider **more complex text data: sentences.**
- The word generated on each timestamp is a character, not a numerical value.
- Neural networks cannot directly process string data.
- Conversion of words or characters into numerical values becomes particularly critical, in neural network applications like NLP.

REPRESENTATION METHOD OF TEXT SEQUENCES:

- Consider a sentence having 'n' words.
- The process of encoding text into numbers as **Word Embedding.**Lets use **one – hot encoding** location names is as shown below:



```
Rome   = [1, 0, 0, 0, 0, 0, …, 0]
Paris  = [0, 1, 0, 0, 0, 0, …, 0]
Italy  = [0, 0, 1, 0, 0, 0, …, 0]
France = [0, 0, 0, 1, 0, 0, …, 0]
```

- **Disadvantage of One Hot Encoding:** The one-hot encoding vector is high-dimensional and extremely sparse, with a large number of positions as 0s.
- Therefore, it is computationally expensive and also not conducive to the neural network training.
- It ignores the **semantic relevance inherent in words. Example:** "like," "dislike," "Rome," "Paris," "like," and "dislike".
- For a group of such words, if one-hot encoding is used, there is no correlation between the obtained vectors, and the semantic relevance of the original text cannot be well reflected.

SEMANTIC LEVEL OF RELEVANCE:

- In NLP, the semantic level of relevance can be well reflected through the **word vector.**
- One way to measure the correlation between word vectors is the cosine similarity:

$$similarity(a,b) \triangleq \cos\cos(\theta) = \frac{a \cdot b}{|a| \cdot |b|}$$

where *a* and *b* represent two word vectors.

EMBEDDING LAYER:

- In a neural network, the representation vector of a word can be obtained directly through training.
- We call the representation layer of the word **Embedding layer.**
- The Embedding layer is responsible for encoding the word into a word vector *v*.
- It accepts the word number *i* using digital encoding, such as 2 for “I” and 3 for “me”.
- The total number of words in the system is recorded as *Nvocab*, and the output is vector *v* with length *n*:

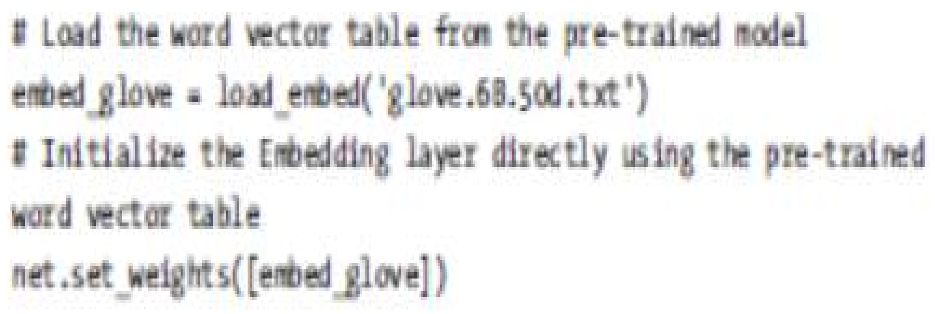
$$v = f_\theta(i|N_{vocab}, n)$$

- The Embedding layer is very simple to implement. Build a lookup table with shape [*Nvocab*, *n*].
- For any word number *i*, you only need to query the vector at the corresponding position and return: *v* = *table[i]*
- The Embedding layer is trainable. It can be placed in front of the neural network to complete the conversion of words to vectors.
- The resulting representation vector can continue to pass through the neural network to complete subsequent tasks, and calculate the error *L*.
- Sample Code:

```
x = tf.range(10) # Generate a digital code of 10 words
x = tf.random.shuffle(x) # Shuffle
# Create a layer with a total of 10 words, each word is
represented by a vector of length 4
net = layers.Embedding(10, 4)
out = net(x) # Get word vector
```

PRETRAINED WORD VECTORS:

- The lookup table of the Embedding layer is initialized randomly and needs to be trained from scratch.
- In fact, we can use pre-trained Word Embedding models to get the word representation.
- The word vector based on pre-trained models is equivalent to transferring the knowledge of the entire semantic space, which can often get better performance.
- Currently, the widely used pre-trained models include Word2Vec and GloVe.
- They have been trained on a massive corpus to obtain a better word vector representation and can directly export the learned word vector table to facilitate migration to other tasks.
- For example, the GloVe model GloVe.6B.50d has a vocabulary of 400,000, and each word is represented by a vector of length 50.
- Users only need to download the corresponding model file in order to use it. The “glove6b50dtxt.zip” model file is about 69MB.
- For the Embedding layer, random initialization is no longer used. Instead, we use the pre-trained model parameters to initialize the query table of the Embedding layer.



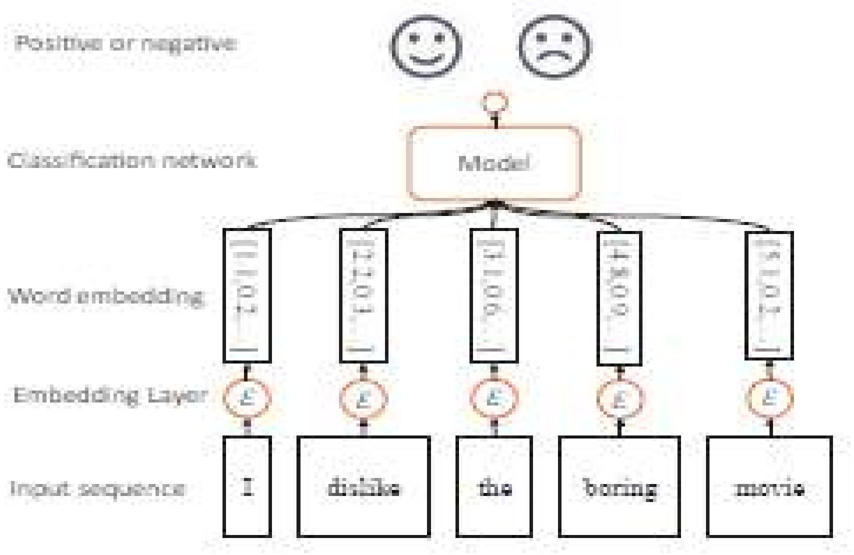
```
# Load the word vector table from the pre-trained model
embed_glove = load_embed('glove.6B.50d.txt')
# Initialize the Embedding layer directly using the pre-trained
word vector table
net.set_weights([embed_glove])
```

<u>**DEALING WITH SEQUENCE SIGNAL:**</u>

Consider a sentence:

“I hate this boring movie”

Through the Embedding layer, it can be converted into a tensor with shape [*b*, *s*, *n*], where b is the number of sentences, s is the sentence length, and n is the length of the word vector. The preceding sentence can be expressed as a tensor with shape [1,5,10], where 5 represents the length of the sentence word, and 10 represents the length of the word vector.The sentiment classification task extracts the overall semantic features expressed by the text data and thereby predict the sentiment type of the input text: **positive or negative.**

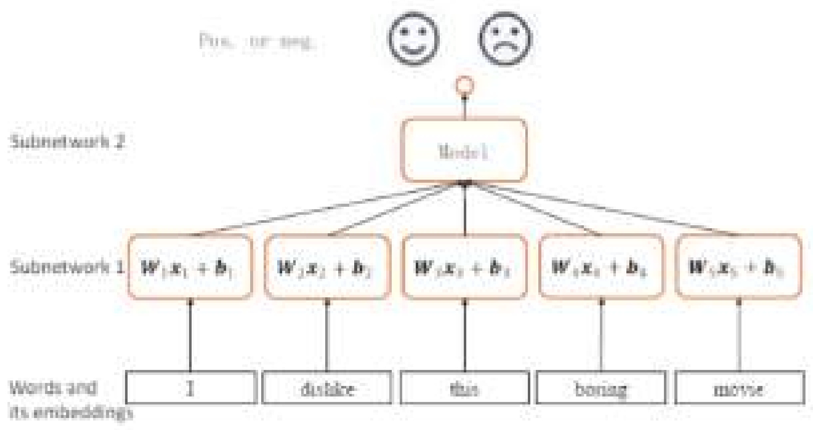


NETWORK ARCHITECTURE 1 -FULLY CONNECTED ARCHITECTURE:

- The first thing we think of is that for each word vector, a fully connected layer network can be used.

  $o = \sigma(W_t x_t + b_t)$

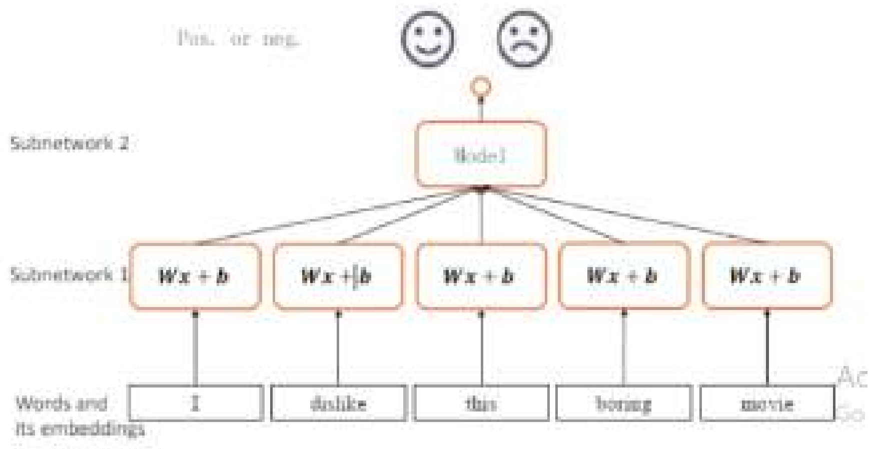
- Extract semantic features, as shown in Figure, using a fully connected layer.



- Disadvantage of network scheme 1: The amount of network parameters is considerable, and the memory usage and calculation cost are high. At the same time, since the length s of each sequence is not the same, the network structure changesdynamically.
- Each fully connected layer sub-network *Wi* and *bi* can only sense the input of the current word vector and cannot perceive the context information before and after, resulting in the lack of overall sentence semantics. Each sub-network can only extract high-level features based on its own input.

NETWORK ARCHITECTURE 2 – SHARED WEIGHTS:

- We already know that convolutional neural networks is better than fully connected networks in processing locally related data.
- This is because it makes full use of the idea of weight sharing and greatly reduces the amount of network parameters, which makes the network training more efficient.
- can we learn from the idea of weight sharing when dealing with sequence signals?



PROS:

- Amount of parameters is greatly reduced,

- Network training becomes more stable and efficient

CONS:

- This network structure does not consider the order of sequences
- The same output can still be obtained by shuffling the order of the word vectors.
- Therefore, it cannot obtain effective global semantic information.

NETWORK ARCHITECTURE 3 – GLOBAL SEMANTICS:

- Using Memory Mechanism the network extract the semantic information of word vectors in order and accumulate it into the global semantic information of the entire sentence.
- If the network can provide a separate memory variable, each time the feature of the word vector is extracted then the memory variable is refreshed.
- This is done until the last input is completed and thus the memory variable at this time stores the semantic features of all sequence

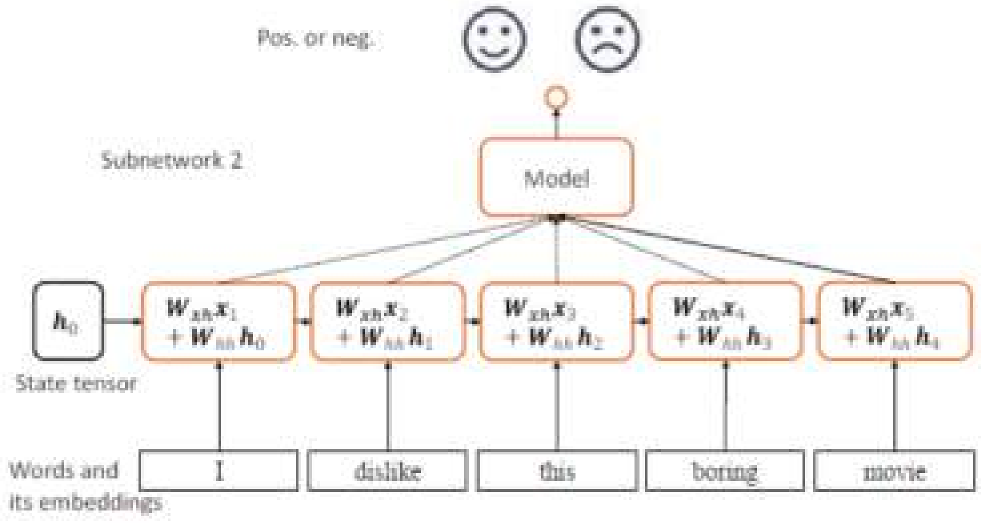
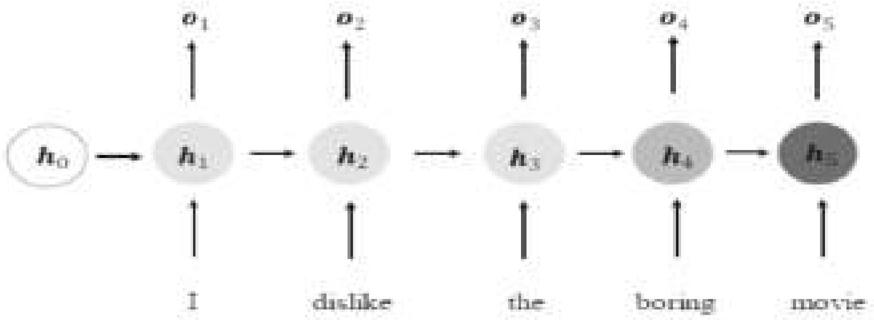


***Figure 11-6.** Recurrent neural network (no bias added)*

The state tensor h refresh mechanism for each timestamp t is:

$$h_t = \sigma\left(W_{xh}x_t + W_{hh}h_{t-1} + b\right)$$

EXPANDED RNN:



- At each time stamp t, the network layer accepts the input *xt* of the current time stamp and the network state vector of the previous time stamp *ht* − 1:

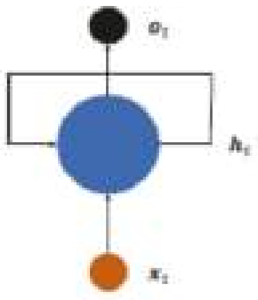
$$h_t = f_\theta(h_{t-1}, x_t)$$

- After transformation, the new state vector *ht* of the current time stamp is obtained and written into the memory state.
- Here *fθ* represents the operation logic of the network, and *θ* is the network parameter set.
- At each time stamp, the network layer has an output to produce

$$o_t, o_t = g_\phi(h_t),$$

FOLDED RNN:

- The preceding network structure is folded on the time stamp, as shown in Figure.
- The network cyclically accepts each feature vector *xt* of the sequence, refreshes the internal state vector *ht*, and forms the output *ot* at the same time.
- For this kind of network structure, we call it the recurrent neural network (RNN).



Folded RNN model

USING RNN LAYERS:

- let's learn how to implement the RNN layer in TensorFlow.
- In TensorFlow, the $\sigma(Wxhxt + Whhht - 1 + b)$ calculation can be completed by layers.SimpleRNNCell() function.
- It should be noted that in TensorFlow, RNN stands for recurrent neural network in a general sense. For the basic recurrent neural network we are currently introducing, it is generally called SimpleRNN.
- The difference between SimpleRNN and SimpleRNNCell is that the layer with cell only completes the forward operation of one timestamp, while the layer without cell is generally implemented based on the cell layer, which has already completed multiple timestamp cycles internally. Therefore, it is more convenient and faster to use.

SIMPLE RNN:

```
import tensorflow as tf

# Define the input feature length
n = 4

# Create a SimpleRNNCell with a cell state vector feature length of 3
cell = tf.keras.layers.SimpleRNNCell(3)

# Build the cell with the input feature length
cell.build(input_shape=(None, n))

# Get the trainable variables (wxh, whh, b)
trainable_variables = cell.trainable_variables

# Print the trainable variables
for var in trainable_variables:
    print(var)
```

In this code:

- We define the input feature length 'n' as 4.
- We create a SimpleRNNCell with a cell state vector feature length of 3.
- We use the cell.build(input_shape=(None, n)) method to build the cell with the specified input feature length.
- We then access the trainable variables of the cell using cell.trainable_variables and print them.
- This code will give you the weights and bias tensors (wxh, whh, b) associated with the SimpleRNNCell.

It can be seen that SimpleRNNCell maintains three tensors internally, the kernel variable is the tensor Wxh, the recurrent_kernel variable is the tensor Whh, and the bias variable is the bias vector b. However, the memory vector h of RNN is not maintained by SimpleRNNCell, and the user needs to initialize the vector $h0$ and record the ht on each time stamp. The forward operation can be completed by calling the cell instance:

$$o_t, [h_t] = Cell(x_t, [h_{t-1}])$$

For SimpleRNNCell, *ot* = *ht*, is the same object. There’s no additional linear layer conversion. [*ht*] is wrapped in a list. This setting is for uniformity with RNN variants such as LSTM and GRU. In the initialization phase of the recurrent neural network, the state vector *h*0 is generally initialized to an all-zero vector.

```
import tensorflow as tf

# Initialize state vector. Wrap with list, unified format
h0 = [tf.zeros([4, 64])]
x = tf.random.normal([4, 80, 100])  # Generate input tensor, 4 sentences of 80 words
xt = x[:, 0, :]  # The first word of all sentences

# Construct a Cell with input feature n=100, sequence length s=80, state length=64
cell = tf.keras.layers.SimpleRNNCell(64)

# Reshape xt to match the input requirements of the SimpleRNNCell
xt = tf.reshape(xt, [4, 100])

# Forward calculation
out, h1 = cell(xt, h0)

print(out.shape, h1[0].shape)
```

In this code:

- You correctly initialize the state vector h0.
- You create random input data x for 4 sentences, each with 80 words, and you extract the first word using xt.
- You construct a SimpleRNNCell with the specified input feature size (n=100), sequence length (s=80), and state length (64).
- You reshape xt to match the input requirements of the cell, ensuring it has the shape [batch_size, input_feature].
- You perform the forward calculation by passing xt and h0 to the cell and print the output shapes.

It can be seen that after one timestamp calculation, the shape of the output and the state tensor are both [b, h], and the ids of the two are printed as follows:

```
print(id(out), id(h1[0]))
```

Post execution we find that The two ids are the same, that is, the state vector is directly used as the
output vector. For the training of length s, it is necessary to loop through the cell class s times to complete one forward operation of the network layer.

Lets attempt to now to iterate over the sequence length and apply the SimpleRNNCell to each time step to get both the output at each time step and the final state.

```
import tensorflow as tf

# Initialize state vector. Wrap with list, unified format
h0 = [tf.zeros([4, 64])]
x = tf.random.normal([4, 80, 100])  # Generate input tensor, 4 sentences of 80 words

# Construct a Cell with input feature n=100, sequence length s=80, state length=64
cell = tf.keras.layers.SimpleRNNCell(64)

# Save a list of state vectors on each time step
h = h0
state_history = []

# Unpack the input in the dimension of the sequence length to get xt: [batch_size,
input_feature]
for xt in tf.unstack(x, axis=1):
    out, h = cell(xt, h)
    state_history.append(out)

# The final output can aggregate the output on each time step or take the output of the last
time step
final_output = out  # Output of the last time step
# final_output = tf.concat(state_history, axis=1)  # Aggregate outputs from all time steps

print(final_output.shape)
```

In this code:

- state_history is used to collect the output at each time step.
- The tf.unstack operation is used to unpack the input tensor along the time dimension (axis=1) to get each xt for each time step.
- The SimpleRNNCell is applied to each xt, updating the state h at each time step.
- You can choose to aggregate the outputs from all time steps or take only the output from the last time step, as shown in the comments.

The output variable out of the last time stamp will be the final output of the network. In fact, you can also save the output on each timestamp, and then sum or average it as the final output of the network.

<u>**MULTI LAYER SIMPLERNNCELL NETWORK:**</u>

Like the convolutional neural network, although the recurrent neural network has been expanded many times on the time axis, it can only be counted as one network layer. By stacking multiple cell classes in the depth direction, the network can achieve the same effect as a deep convolutional
neural network, which greatly improves the expressive ability of the network. However, compared with the number of deep layers of tens or hundreds of convolutional neural networks, recurrent neural networks are prone to gradient diffusion and gradient explosion. Deep recurrent neural networks are very difficult to train.

Lets apply two SimpleRNN cells sequentially to the input data, with each cell having its own initial state vector.

```
import tensorflow as tf

# Generate input data: 4 sentences of 80 words, each with 100 features
x = tf.random.normal([4, 80, 100])

# Extract the input at the first time step
xt = x[:, 0, :]

# Define the first SimpleRNNCell (cell0) with a memory state vector length of 64
cell0 = tf.keras.layers.SimpleRNNCell(64)

# Define the second SimpleRNNCell (cell1) with a memory state vector length of 64
cell1 = tf.keras.layers.SimpleRNNCell(64)

# Initialize the state vectors for cell0 and cell1
h0 = [tf.zeros([4, 64])]  # Initial state vector for cell0
h1 = [tf.zeros([4, 64])]  # Initial state vector for cell1

# Apply cell0 to the input
out0, h0 = cell0(xt, h0)

# Apply cell1 to the output of cell0
out1, h1 = cell1(out0, h1)

print("Output of cell0:", out0.shape)
print("Output of cell1:", out1.shape)
```

In this code:

We generate random input data 'x' representing 4 sentences, each with 80 words, and 100 features.
We extract the input at the first time step 'xt'.
We define two SimpleRNN cells ('cell0' and 'cell1'), each with a memory state vector length of 64.
We initialize the state vectors 'h0' and 'h1' for cell0 and cell1.
We apply 'cell0' to 'xt' and update 'h0' with the new state.
We then apply 'cell1' to the output of 'cell0' and update 'h1' with the new state.
Finally, we print the shapes of the outputs of both cells.
This code demonstrates applying two SimpleRNN cells sequentially to the input data.

Now lets Calculate multiple times on the time axis to realize the forward operation of the entire network. The input xt on each time stamp first passes through the first layer to get the output out0, and then passes through the second layer to get the output out1.

```
 import tensorflow as tf

# Generate input data: 4 sentences of 80 words, each with 100 features
x = tf.random.normal([4, 80, 100])
```

```
# Define the first SimpleRNNCell (cell0) with a memory state vector length of 64
cell0 = tf.keras.layers.SimpleRNNCell(64)

# Define the second SimpleRNNCell (cell1) with a memory state vector length of 64
cell1 = tf.keras.layers.SimpleRNNCell(64)

# Initialize the state vectors for cell0 and cell1
h0 = [tf.zeros([4, 64])]  # Initial state vector for cell0
h1 = [tf.zeros([4, 64])]  # Initial state vector for cell1

# Initialize an empty list to store the outputs at each time step
output_sequence = []

# Iterate through the input sequences
for xt in tf.unstack(x, axis=1):
    # Apply cell0 to the input 'xt'
    out0, h0 = cell0(xt, h0)

    # Apply cell1 to the output of cell0
    out1, h1 = cell1(out0, h1)

    # Append the output of cell1 to the output_sequence
    output_sequence.append(out1)

# Stack the output_sequence to get the final output tensor
final_output = tf.stack(output_sequence, axis=1)

print("Final Output Shape:", final_output.shape)
```

The preceding method first completes the propagation of the input on one time stamp on all layers and then calculates the input on all time stamps in a loop.
In fact, it is also possible to first complete the calculation of all time stamps input on the first layer, and save the output list of the first layer on all time stamps, and then calculate the propagation of the second layer, the third layer, etc. as in the following:

```
import tensorflow as tf

# Generate input data: 4 sentences of 80 words, each with 100 features
x = tf.random.normal([4, 80, 100])

# Define the first SimpleRNNCell (cell0) with a memory state vector length of 64
cell0 = tf.keras.layers.SimpleRNNCell(64)

# Define the second SimpleRNNCell (cell1) with a memory state vector length of 64
cell1 = tf.keras.layers.SimpleRNNCell(64)

# Initialize the state vectors for cell0 and cell1
h0 = [tf.zeros([4, 64])]  # Initial state vector for cell0
h1 = [tf.zeros([4, 64])]  # Initial state vector for cell1
```

```
# Initialize lists to store the outputs at each time step for both layers
output_sequence_cell0 = []
output_sequence_cell1 = []

# Iterate through the input sequences for the first layer (cell0)
for xt in tf.unstack(x, axis=1):
   # Apply cell0 to the input 'xt' and save the output
   out0, h0 = cell0(xt, h0)
   output_sequence_cell0.append(out0)

# Iterate through the output sequences of the first layer (cell0) for the second layer (cell1)
for xt in output_sequence_cell0:
   # Apply cell1 to the input 'xt' and save the output
   out1, h1 = cell1(xt, h1)
   output_sequence_cell1.append(out1)

# Stack the output sequences for both layers to get the final output tensors
final_output_cell0 = tf.stack(output_sequence_cell0, axis=1)
final_output_cell1 = tf.stack(output_sequence_cell1, axis=1)

print("Final Output for cell0 Shape:", final_output_cell0.shape)
print("Final Output for cell1 Shape:", final_output_cell1.shape)
```

SIMPLE RNN LAYER:

Through the use of the SimpleRNNCell layer, we can understand every detail of the forward operation of the recurrent neural network. In actual use, for simplicity, we do not want to manually implement the internal calculation process of the recurrent neural network, such as the initialization of the state vector at each layer and the operation of each layer on the time axis. Using the SimpleRNN high-level interface can help us achieve this goal very conveniently.

```
import tensorflow as tf

# Create a SimpleRNN layer with a state vector length of 64
layer = tf.keras.layers.SimpleRNN(64)

# Generate input data: 4 sentences of 80 words, each with 100 features
x = tf.random.normal([4, 80, 100])

# Apply the SimpleRNN layer to the input data
out = layer(x)

# Print the shape of the output
print("Output Shape:", out.shape)
```

When creating a SimpleRNN layer in TensorFlow and setting return_sequences=True, the layer will return outputs for all timestamps. Here's how you can do it:

```
import tensorflow as tf
```

```
# Create a SimpleRNN layer with a state vector length of 64 and return sequences
layer = tf.keras.layers.SimpleRNN(64, return_sequences=True)

# Generate input data: 4 sentences of 80 words, each with 100 features
x = tf.random.normal([4, 80, 100])

# Apply the SimpleRNN layer to the input data
out = layer(x)

# The 'out' variable now contains the outputs for all timestamps
# You can access the output at each timestamp as needed
print(out.shape)
```

To create MULTI LAYER RNN i.e. a sequence of SimpleRNN layers where all layers except the last one return sequences that can be used as input for the next layer, you can use a Sequential model in TensorFlow. Here's how you can do it:

```
import tensorflow as tf

# Create a Sequential model with multiple SimpleRNN layers
net = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(64, return_sequences=True),
    tf.keras.layers.SimpleRNN(64, return_sequences=True),
    tf.keras.layers.SimpleRNN(64),
])

# Generate input data: 4 sentences of 80 words, each with 100 features
x = tf.random.normal([4, 80, 100])

# Forward calculation
out = net(x)

# 'out' now contains the final output
print(out.shape)
```

HANDS ON SENTIMENT ANALYSIS USING RNN AND IMDB DATASET:

The classic IMDB movie review dataset is used here to complete the sentiment classification task. The IMDB movie review dataset contain 50,000 user reviews. The evaluation tags are divided into negative and positive. User reviews with IMDB rating <5 are marked as 0, which means negative; user reviews with IMDB rating ≥ 7 are marked as 1, which means positive. Twenty-five thousand film reviews were used for the training set and 25,000 were used for the test set. The IMDB dataset can be loaded by datasets tool provided by Keras as follows:

First specify the batch size, vocabulary size, maximum sentence length, and word vector feature length.You load the IMDB dataset using keras.datasets.imdb.load_data(). You print the shapes of the training and test data, along with the length of the first training and test samples, and the shapes of the corresponding labels.

```
import tensorflow as tf
from tensorflow import keras

# Define dataset parameters
batch_size = 128        # Batch size
total_words = 10000     # Vocabulary size (N_vocab)
max_review_len = 80     # Maximum length of sentences
embedding_len = 100     # Word vector feature length (n)

# Load the IMDB dataset using Keras
(x_train, y_train), (x_test, y_test) = keras.datasets.imdb.load_data(num_words=total_words)

# Print the shapes of the input data and labels
print("Training Data Shape:", x_train.shape)     # (number of samples, maximum sequence
length)
print("Length of First Training Sample:", len(x_train[0]))  # Length of the first training
sample
print("Training Labels Shape:", y_train.shape)     # (number of samples,)
print("Test Data Shape:", x_test.shape)           # (number of samples, maximum sequence
length)
print("Length of First Test Sample:", len(x_test[0]))      # Length of the first test sample
print("Test Labels Shape:", y_test.shape)          # (number of samples)

#lets fetch the word-to-index mapping for the IMDb dataset.
# Get the word-to-index mapping for the IMDb dataset
word_index = keras.datasets.imdb.get_word_index()

# Print out the first few words and their corresponding numbers in the coding table
for k, v in list(word_index.items())[:10]:
    print(f"Word: {k}, Index: {v}")

“””you will start making modifications to the word-to-index mapping to account for special
tokens such as padding, start, unknown, and unused tokens. You're also creating a reversed
mapping to convert numerical indices back to words.”””

# Original word-to-index mapping (word_index)

# Add 3 to all values in the word_index dictionary
word_index = {k: (v + 3) for k, v in word_index.items()}

# Define special tokens and their corresponding indices
word_index["<PAD>"] = 0  # Padding token
word_index["<START>"] = 1  # Start token
word_index["<UNK>"] = 2  # Unknown word token
word_index["<UNUSED>"] = 3  # Unused token

# Create the reversed index mapping (numerical index to word)
reverse_word_index = dict((value, key) for (key, value) in word_index.items())
```

‘’’ The decode_review function you provided takes a list of numerical indices (representing words) and converts them into a human-readable text by looking up the corresponding words in the reverse_word_index mapping. If a word index is not found in the mapping, it is replaced with a question mark "?".’’’

```
def decode_review(text):
    return ' '.join([reverse_word_index.get(i, '?') for i in text])
decoded_review = decode_review(x_train[0])
print(decoded_review)

decoded_review = decode_review(x_train[0])
print(decoded_review)
```

‘’’ lets truncate and pad the sequences in the training and test data to make them of equal length. Long sentences are truncated to the specified max_review_len, retaining the beginning of the sentence, and short sentences are padded in front with zeros to reach the desired length.’’’

```
x_train = keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_review_len)
x_test = keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_review_len)
```

‘’’ After truncating or filling to the same length, wrap it into a dataset
object through the Dataset class, and add the commonly used dataset
processing flow, the code is as follows:’’’

```
# Create a TensorFlow Dataset from the training data and labels
db_train = tf.data.Dataset.from_tensor_slices((x_train, y_train))

# Shuffle the training data with a buffer size of 1000 and batch it, dropping incomplete
batches
db_train = db_train.shuffle(1000).batch(batchsz, drop_remainder=True)

# Create a TensorFlow Dataset from the test data and labels and batch it, dropping incomplete
batches
db_test = tf.data.Dataset.from_tensor_slices((x_test, y_test))
db_test = db_test.batch(batchsz, drop_remainder=True)

# Print dataset attributes
print('x_train shape:', x_train.shape, tf.reduce_max(y_train), tf.reduce_min(y_train))
print('x_test shape:', x_test.shape)
```

lets create a network model now. To create a custom model class named MyRNN that inherits from the base Model class and contains an Embedding layer, two RNN layers, and one classification layer, you can use TensorFlow and Keras.

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

```
class MyRNN(keras.Model):
    def __init__(self, units, total_words, embedding_len, max_review_len, batchsz):
        super(MyRNN, self).__init__()

        # Initialize RNN cell states
        self.state0 = [tf.zeros([batchsz, units])]
        self.state1 = [tf.zeros([batchsz, units])]

        # Word vector encoding
        self.embedding = layers.Embedding(total_words, embedding_len,
input_length=max_review_len)

        # Construct 2 RNN cells with dropout to prevent overfitting
        self.rnn_cell0 = layers.SimpleRNNCell(units, dropout=0.5)
        self.rnn_cell1 = layers.SimpleRNNCell(units, dropout=0.5)

        # Classification network to classify the output features of cells
        self.outlayer = layers.Dense(1)

    def call(self, inputs):
        x = self.embedding(inputs)  # Word embedding
        state0 = self.state0
        state1 = self.state1

        # Iterate through the sequence with RNN cells
        for word in tf.unstack(x, axis=1):
            out0, state0 = self.rnn_cell0(word, state0)
            out1, state1 = self.rnn_cell1(out0, state1)

        # Apply the classification layer to the final RNN output
        x = self.outlayer(out1)
        return x

‘’’ The final output is obtained from the last layer's last timestamp, and it is passed through a
sigmoid activation function to obtain the probability that a given input belongs to the positive
class (p(y is positive | x)).’’’

def call(self, inputs, training=None):
    x = inputs  # [b, 80]

    # Word vector embedding: [b, 80] => [b, 80, 100]
    x = self.embedding(x)

    # Pass through 2 RNN cells: [b, 80, 100] => [b, 64]
    state0 = self.state0
    state1 = self.state1
    for word in tf.unstack(x, axis=1):  # word: [b, 100]
        out0, state0 = self.rnn_cell0(word, state0, training)
        out1, state1 = self.rnn_cell1(out0, state1, training)
    # Last layer's last timestamp as the network output: [b, 64] => [b, 1]
    x = self.outlayer(out1, training)

    # Pass through the activation function, p(y is positive | x)
    prob = tf.sigmoid(x)

    return prob

‘’’LETS TRAIN & TEST THE MODEL’’’

‘’’For simplicity, here we use Keras’ Compile&Fit method to train the
network. Set the optimizer to Adam optimizer, the learning rate is
0.001, the error function uses the two-class cross-entropy loss function
BinaryCrossentropy, and the test metric uses the accuracy rate. The code is
as follows:’’’’

def main():
    units = 64  # RNN state vector length n
    epochs = 20  # Training epochs

    # Create an instance of your custom RNN model
    model = MyRNN(units)

    # Compile the model
    model.compile(
        optimizer=tf.optimizers.Adam(0.001),
        loss=tf.losses.BinaryCrossentropy(),
        metrics=['accuracy']
    )

    # Fit the model on the training dataset and validate on the test dataset
    model.fit(db_train, epochs=epochs, validation_data=db_test)

    # Evaluate the model on the test dataset
    model.evaluate(db_test)
```

NOTE: After 20 Epoch trainings, the network achieves 80.1% accuracy rate at testing dataset.

GRADIENT VANISHING & GRADIENT EXPLODING:

Gradient Vanishing:

Problem Explanation:

Gradient vanishing occurs when the gradients computed during backpropagation through time (BPTT) become very small, approaching zero.
It often happens when using activation functions like sigmoid or tanh, which have derivatives that are close to zero in certain regions.

Causes:

Repeated multiplications of small gradients across many time steps in the network.
The nature of RNN architectures that involve long sequences.
Solutions:

Use Activation Functions: Replace sigmoid or tanh activations with ReLU (Rectified Linear Unit) or variants. ReLU has a simple derivative that is either 0 or 1, which mitigates the vanishing gradient problem.
Advanced Architectures: Switch to more advanced RNN architectures like Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU), which are designed to capture long-range dependencies without suffering from vanishing gradients.
Gradient Clipping: Clip gradients during training to prevent them from growing too large or too small.

Gradient Exploding:

Problem Explanation:

Gradient exploding occurs when gradients become extremely large during BPTT.
It can lead to unstable training and numerical overflow issues.
Causes:

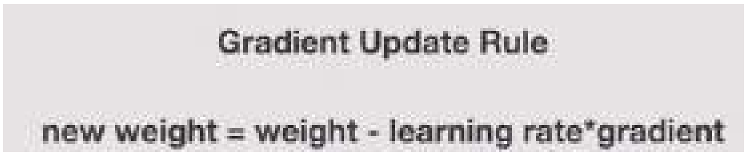
Poorly initialized model weights, especially when weights are initialized too large.
Inadequate choice of the learning rate.
Unstable recurrent computations.
Solutions:

Weight Initialization: Properly initialize model weights using techniques like Xavier/Glorot initialization, which scales weights appropriately.
Reduce Learning Rate: Use learning rate schedules that gradually reduce the learning rate during training to stabilize updates.
Gradient Clipping: Clip gradients during training to prevent them from becoming too large and destabilizing the optimization process.
Batch Normalization: Apply batch normalization to stabilize activations within the network, which can help mitigate exploding gradients.
It's essential to experiment with these techniques to find the right combination that works best for your specific RNN architecture and dataset. In practice, using more advanced RNN architectures like LSTMs or GRUs is a common strategy, as these architectures are designed to handle long sequences and mitigate both gradient vanishing and exploding issues to some extent.

LSTM & GRU

LIMITATION OF RNN:

- LSTM & GRU are more evolved versions of the vanilla RNN.
- During back propagation RNN undergoes vanishing gradient problem.
- A **gradient** is the value used to upgrade the weight of a Neural network.


Gradient Update Rule

new weight = weight - learning rate*gradient

- **Vanishing Gradient problem:** During back propagation through time, if the gradient shrinks then it leads to vanishing gradient descent i.e gradient value becomes extremely small , which doesn't contribute much to learning.
- Generally, in RNN, the earlier layers of the network don't learn much due to vanishing gradient descent problem.
- This leads RNN to **FORGET** and thus have a short term memory in "long sequences".
- RNN cannot retain long term sequences in the memory!
- Reason of failure: Long term sequences & Inherent relationship between each sentence.
- Solution to this short term memory are LSTM AND GRU
- LSTM and GRU: these have the mechanism with **gates** which can regulate the flow of information, Recognise which data is important to keep / throw away!! They Use relevant information to make predictions

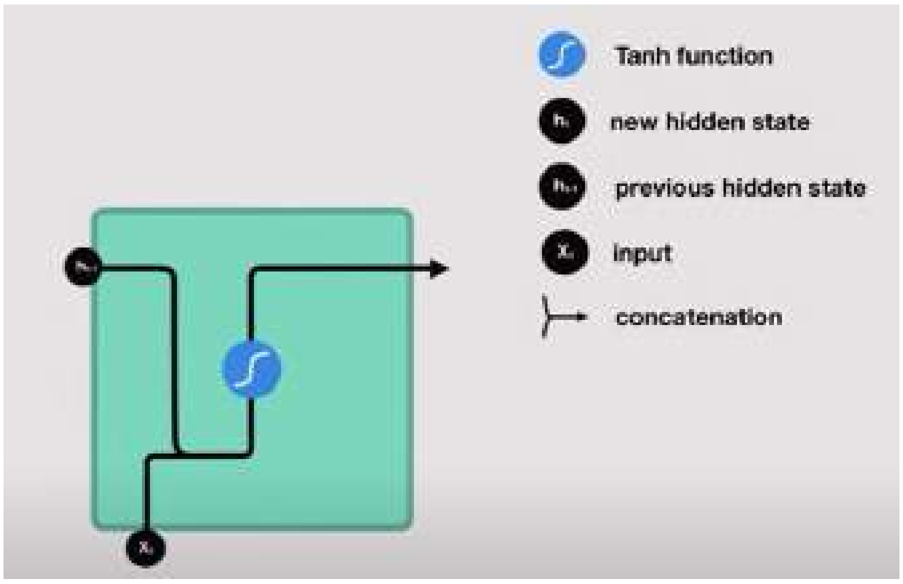
APPLICATION OF LSTM & GRU:

- Speech recognition models
- Speech synthesis
- Text generation
- Generation of captions for videos!!

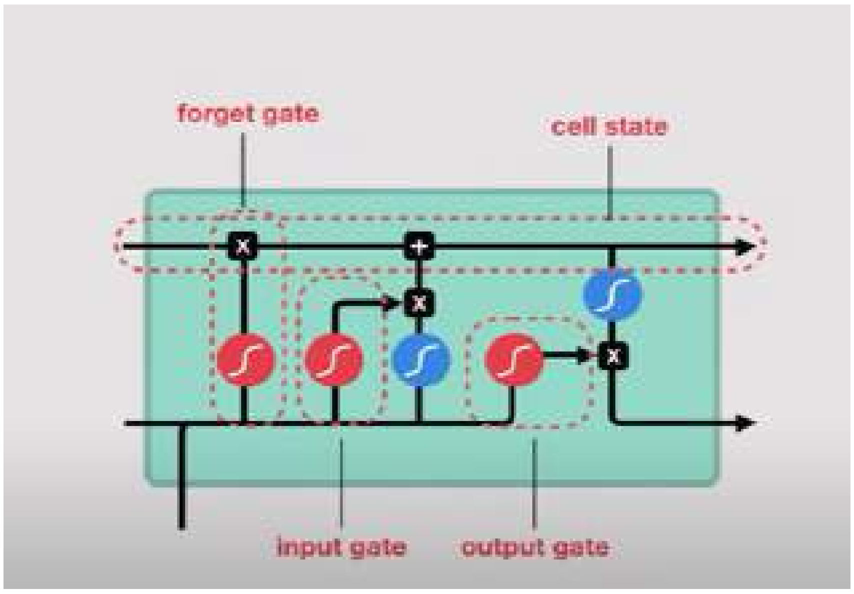
LONG SHORT TERM MEMORY – LSTM:

- Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture used in the field of deep learning.
- It was proposed in 1997 by **Sepp Hochreiter** and **Jurgen schmidhuber**.
- Unlike standard feed-forward neural networks, LSTM has feedback connections.
- It can process not only single data points (such as images) but also entire sequences of data (such as speech or video).

- LSTM is an application to tasks such as unsegmented, **connected handwriting recognition,** or **speech recognition**.
- *Long Short- Term Memory (LSTM) networks are a modified version of recurrent neural networks, which makes it easier to remember past data in memory.*
- Before going to actual architecture lets understand simple working of cells of RNN & LSTM individually.
- Lets consider RNN: first words get transformed into machine readable vectors
- These sequence of words are processed one by one where in previous hidden state is passed to next step of the sequence
- Lets see how a hidden state is calculated?
- A vector with current input & previous input is created and its passed to a Tanh activation function.
- Output of this will be a new hidden state.

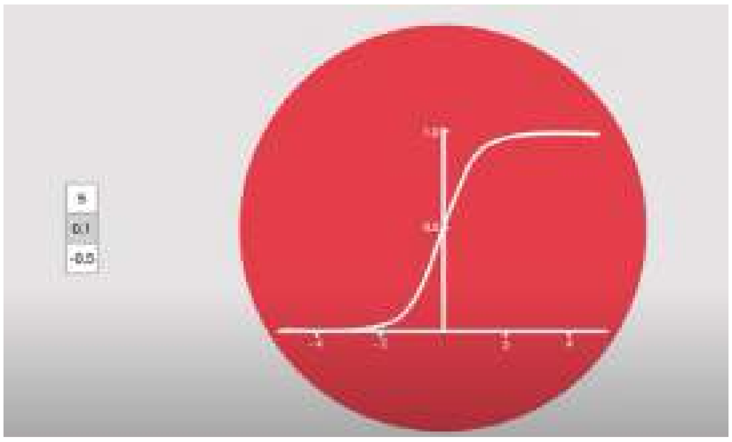


- Tanh activation function is a squishing function used to regulate the value of a neural network
- Information vectors undergo various mathematical transformations in this process.
- Some values can explode in this process, causing other values to be insignificant. Tanh function ensures the value remains between -1 and +1
- LSTM has similar control flow as that of RNN
- It processes data sequentially passing on the information as it propagates forward .
- Major difference lies in the operations within the cells of LSTM!!

- It can achieve long term dependency.
- It knows which information to discard or retain.
- How does LSTM achieve it????
- Answer: using GATE and CELL STATES
- THE LSTM CELL :
- 

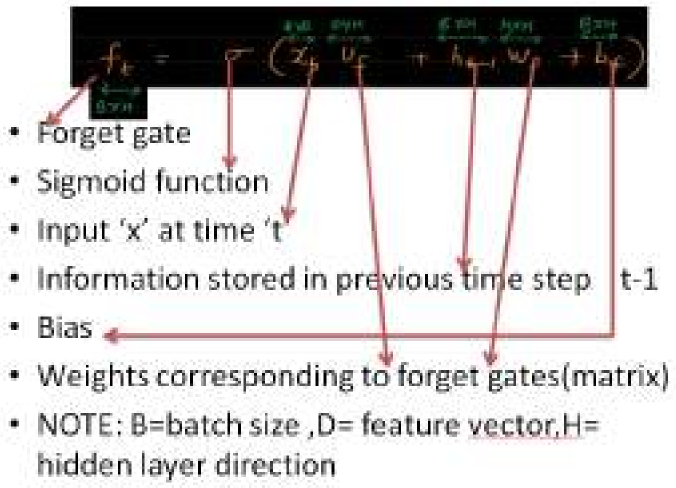
- The **cell state** acts like a sequence highway which transfers the relative information down to the sequence chain. **Its like a memory of the network!!**
- **Cell state can transfer information from previous time steps** as well thus it reduces the effect of short term memory!
- Information can get added or removed via the **Gates** present in the cell.
- Gates are a kind of neural networks which decides which information to retain or discard on the cell state!!
- **Gates are trained to learn which information to retain or discard.**
- **Hidden state** in LSTM is broken into two states:
- CELL STATE: called internal memory where all the information is stored
- HIDDEN STATE: used for computing the output
- NOTE: cell state and hidden state are shared across every time step.

- GATES IN LSTM:
  - Gates contain sigmoid Activation.
  - Sigmoid function squishes Values between 0 & 1.
  - This can help to update or forget the data.
  - Thus a network learns which data to remember and which to forget!!!!
  - There are 3 different gates which regulate the information flow in an LSTM cell.



- **FORGET GATE:**
  - Decides which information is kept or thrown away!
  - Information from previous hidden state and current input is passed to sigmoid function, output expected is 0 or 1 out of it.
  - Closer to 0 means FORGET
  - Closer to 1 means RETAIN
  - EXAMPLE SEQUENCE: MARK is a good singer. He lives in California. JACOB is also good singer.

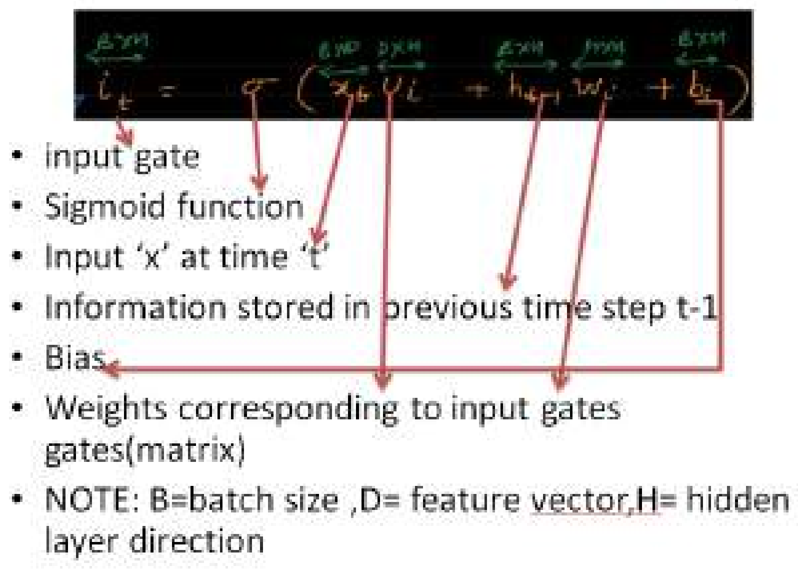
    MARK---→JACOB



$$f_t = \sigma(x_t U_f + h_{t-1} W_f + b_f)$$

- Forget gate
- Sigmoid function
- Input 'x' at time 't'
- Information stored in previous time step t-1
- Bias
- Weights corresponding to forget gates(matrix)
- NOTE: B=batch size ,D= feature vector,H= hidden layer direction

- **INPUT GATE:**
  - Updates the cell state
  - Previous hidden state and current input is passed to sigmoid function which decides which values will be updated by transforming values between 0 & 1
  - 0 means NOT IMPORTANT
  - 1 means IMPORTANT
  - NOTE: to regulate the network we also Pass hidden state and current input to tanh Activation function.
  - We multiply tanh activation function output with sigmoid output.
  - This is done majorly to decide which information from tanh output must be retained or discarded.
  - **After this we have enough information to calculate cell state!!!!**
  - Cell state is multiplied by forget vector which helps which information to be retained or dropped.
  - The output of input gate goes to a polarised addition which updates the cell state to new values.

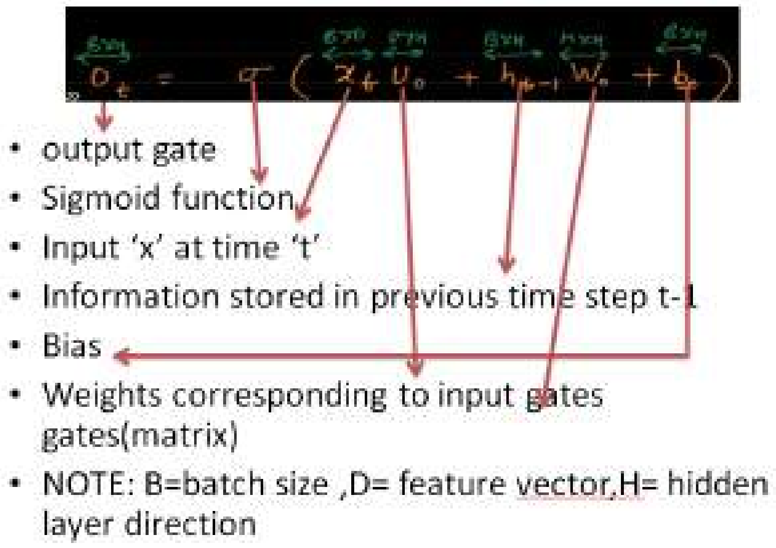


$$i_t = \sigma(x_t U_i + h_{t-1} W_i + b_i)$$

- input gate
- Sigmoid function
- Input 'x' at time 't'
- Information stored in previous time step t-1
- Bias
- Weights corresponding to input gates gates(matrix)
- NOTE: B=batch size ,D= feature vector,H= hidden layer direction

- **OUTPUT GATE:**
  - Decides what the next hidden state should be? i.e it decides which information should be taken from the cell to give as an output.
  - Hidden states contains information belonging to previous inputs.
  - Hidden states are also used for predictions.
  - For this we pass on current input andhidden state to a sigmoid function and the newly modified cell state to a tanh function.
  - Tanh output & sigmoid output are multiplied to decide which information is to be carried by the hidden state!!!!
  - The new hidden state and new cell state are then passed on to the next time step.
  - Example sequence : Jacob made a debut album which was super hit. Congratulations!---------

    Answer: Jacob.



$$\overset{B\times H}{O_t} = \sigma(\overset{B\times D}{x_t}\,\overset{D\times H}{U_o} + \overset{B\times H}{h_{t-1}}\,\overset{H\times H}{W_o} + \overset{B\times H}{b_o})$$

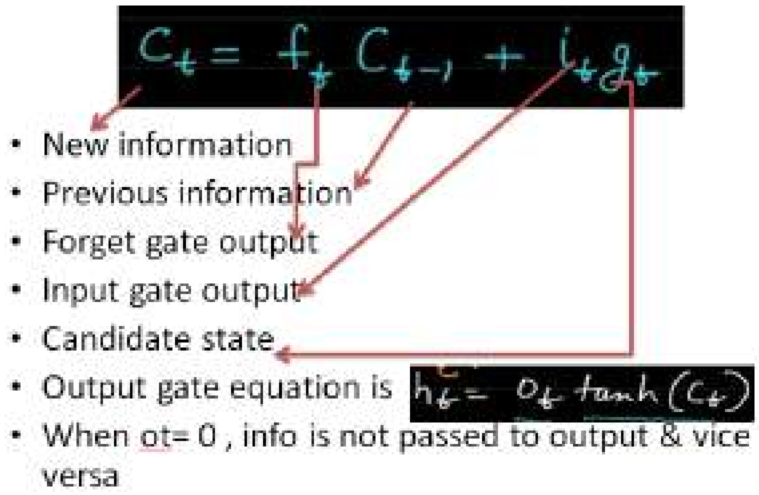
- output gate
- Sigmoid function
- Input 'x' at time 't'
- Information stored in previous time step t-1
- Bias
- Weights corresponding to input gates gates(matrix)
- NOTE: B=batch size ,D= feature vector,H= hidden layer direction

CANDIDATE STATE:

- The main function of candidate state is to update the cell state
- This means adding new information and removing information from previous state

$$\overset{B\times H}{g_t} = \tanh(\overset{B\times D}{x_t}\,\overset{D\times H}{U_g} + \overset{B\times H}{h_{t-1}}\,\overset{H\times H}{W_g} + \overset{B\times H}{b_g})$$

FINAL EQUATION:



$$C_t = f_t\, C_{t-1} + i_t\, g_t$$

- New information
- Previous information
- Forget gate output
- Input gate output
- Candidate state
- Output gate equation is $h_t = O_t \tanh(C_t)$
- When ot= 0 , info is not passed to output & vice versa

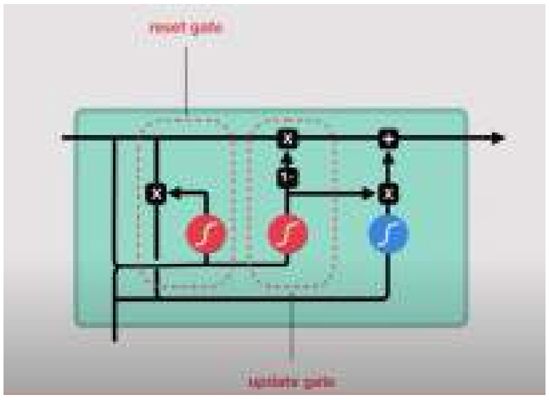
GATED RECURRENT UNIT – GRU:

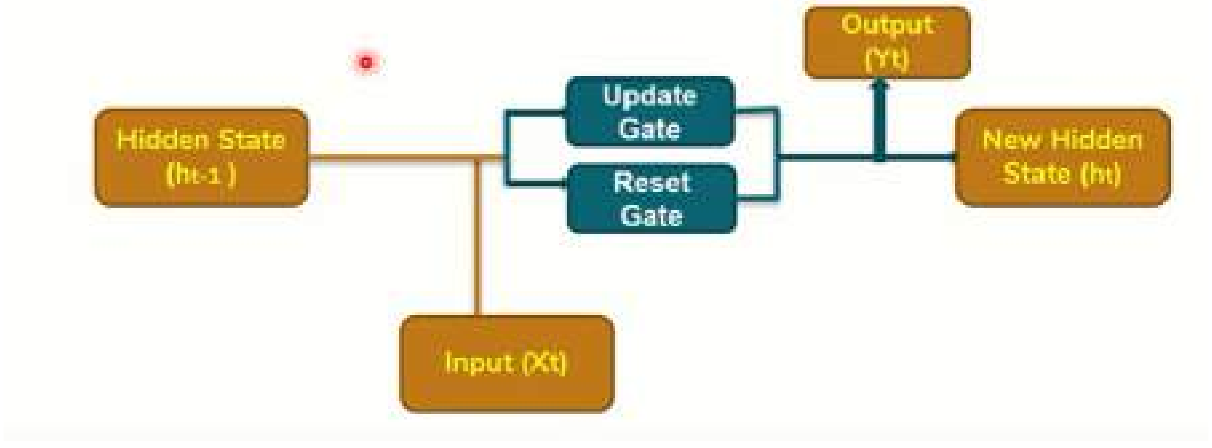
- New generation of RNN
- Similar to LSTM in functionality i.e. solves vanishing gradient problem but much simpler than LSTM

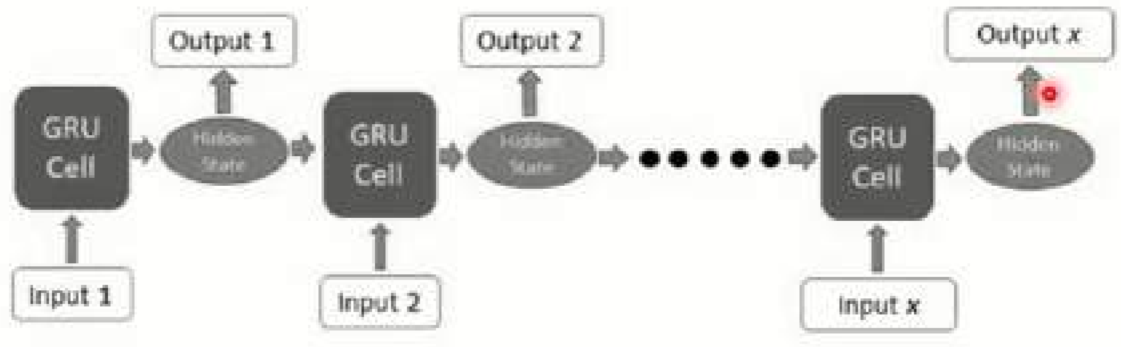
- Invented in 2014 by K.Cho.
- Just uses hidden state in place of cell state to transfer information
- It has 2 gates: RESET gate and UPDATE gate

UPDATE GATE AND RESET GATE:

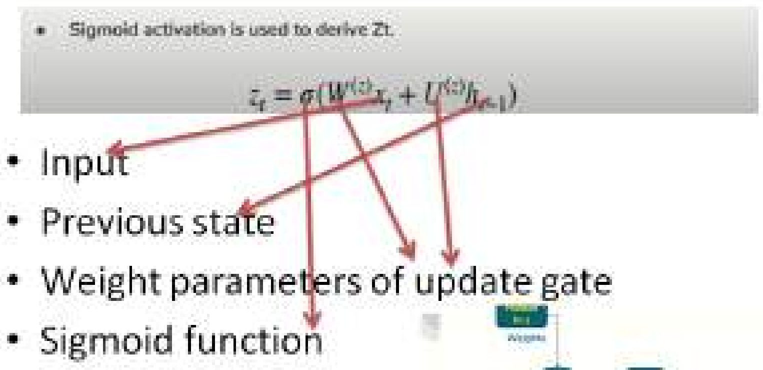
- Similar to Forget and input gate of LSTM
- Decides which information to discard/ retain
- Reset gate decides which past information to forget.







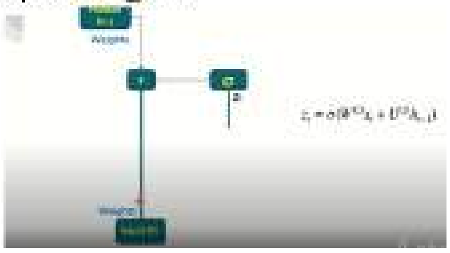
UPDATE GATE EQUATION:



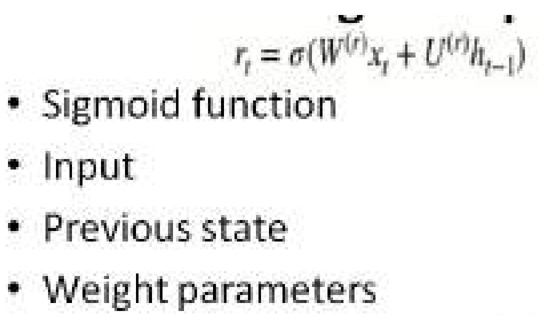
- Sigmoid activation is used to derive Zt.

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

- Input
- Previous state
- Weight parameters of update gate
- Sigmoid function

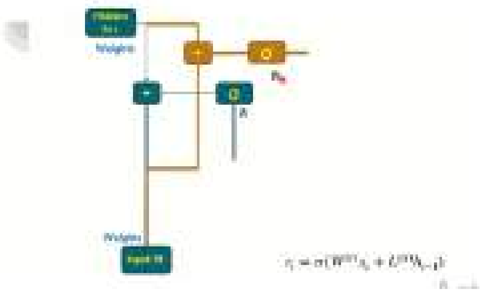


RESET GATE EQUATION:



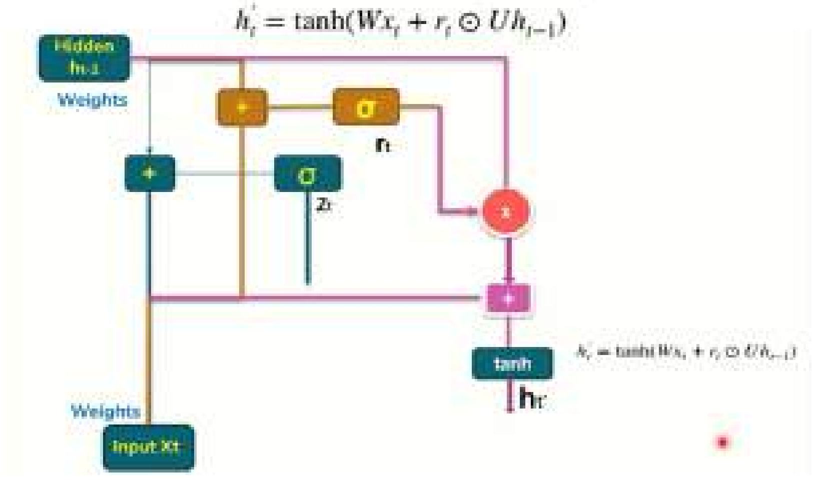
$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

- Sigmoid function
- Input
- Previous state
- Weight parameters



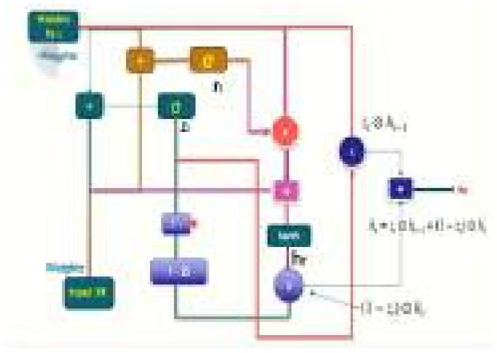
- Next we will use reset gate output to move ahead i.e to create a memory component.

$$h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$$



- Now we will calculate a vector to hold the current information of the unit and passes it down the network!!
- For this we use update gate..

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t'$$



KEY DIFFERENCES BETWEEN LSTM AND GRU:

- The GRU has two gates, LSTM has three gates
- GRU does not possess any internal memory, they don't have an output gate that is present in LSTM
- In LSTM the input gate and target gate are coupled by an update gate
- In GRU reset gate is applied directly to the previous hidden state.
- In LSTM the responsibility of reset gate is taken by the two gates i.e., input and target.

GRU ADVANTAGES:

- They have fewer tensor operations than LSTM hence they are faster to train.

- Which one is best to use LSTM or GRU??
- TRY BOTH BASED ON USED CASE & DECIDE!!

SENTIMENT ANALYSIS USING LSTM/GRU :

The IMDb dataset is a popular dataset for sentiment analysis, specifically binary sentiment classification (positive or negative) of movie reviews. You can perform sentiment analysis on the IMDb dataset using LSTM and GRU neural networks. Here's a step-by-step guide on how to do it using Python and TensorFlow/Keras:

```
import tensorflow as tf
from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, GRU, Dense
# Load IMDb dataset, limit the vocabulary size to the most frequent 10,000 words
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=10000)

# Pad sequences to a fixed length (e.g., 250)
max_sequence_length = 250
X_train = pad_sequences(X_train, maxlen=max_sequence_length)
X_test = pad_sequences(X_test, maxlen=max_sequence_length)
model_lstm = Sequential()
model_lstm.add(Embedding(input_dim=10000, output_dim=128, input_length=max_sequence_length))
model_lstm.add(LSTM(64))  # You can replace LSTM with GRU
model_lstm.add(Dense(1, activation='sigmoid'))

model_lstm.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model_lstm.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=3, batch_size=64)
loss, accuracy = model_lstm.evaluate(X_test, y_test)
```

```
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

-

OVERFITTING

INTRODUCTION:

GENERALIZATION ABILITY:

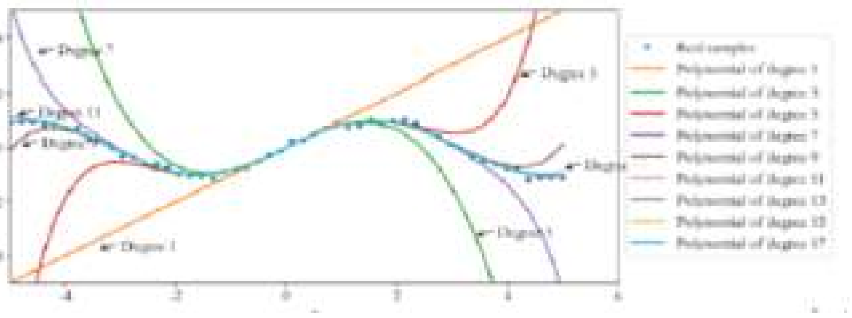
- The ability of machine learning to learn the real model of the data from the training set, so that it can perform well on the unseen test set.

CAPACITY OF THE MODEL:

- The expressive power of the model.
- When the model's expressive power is weak, such as a single linear layer, it can only learn a linear model and not perform well on nonlinear model.
- When the model's expressive power is too strong, it may be possible to reduce the noise modalities of the training set, but leads to poor performance on the test set (generalization ability is weak).
- Thus, the model's ability to fit complex functions is called Model capacity.
- Its major indicator is the size of its hypothesis space.
- Consider the following examples , to understand the concept of model capacity in a better way:

  EXAMPLE:

- $P_{data} = \{(x, y)|y = \sin(x)\ ,\ x \in [-5{,}5]\}$
- A small number of points are sampled from the real distribution to form the training set, which contains the observation error ϵ, as shown by the small dots in Figure.



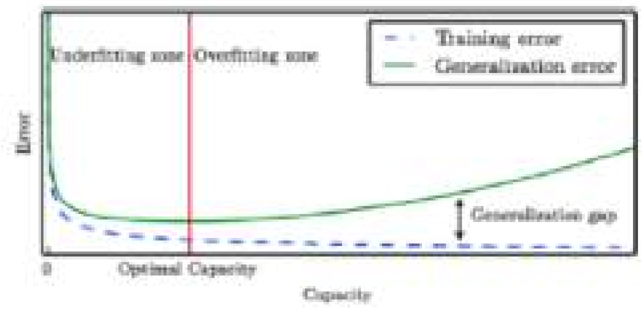
- Initially, If we only search the model space of all first-degree polynomials and set the bias to 0, that is, *y* = *ax*, as shown by the straight line of the first-degree polynomial.
- After increasing the hypothesis space again, as shown in the polynomial curves of 7, 9, 11, 13, 15, and 17 in Figure, the larger the hypothesis space of the function, the more likely it is to find a function model that better approximates the real distribution.

CONS OF USING EXCESSIVELY LARGE HYPOTHESIS SPACE:

- will undoubtedly increase the search difficulty
- Increase in computational cost.
- Doesn't guarantee a better model
- Presence of Observation errors in training hurts the generalization ability of the model.

OVERFITTING AND UNDERFITTING:

- Because the distribution of real data is often unknown and complicated, it is impossible to deduce the type of distribution function and related parameters.
- Therefore, when choosing the capacity of the learning model, people often choose a slightly larger model capacity based on empirical values.
- When the capacity of the model is too large, it may appear to perform better on the training set, but perform worse on the test set.
- When the capacity of the model is too small, it may have poor performance in both the training set and the testing set as shown in the area to the left of the red vertical line in Figure.



REASON FOR OVER FITTING:

- When the capacity of the model is too large, in addition to learning the modalities of the training set data, the network model also learns additional observation errors, resulting in the learned model performing better on the training set, but poor in unseen samples.
- Thus, the generalization ability of the model is weak.
- We call this phenomenon as overfitting

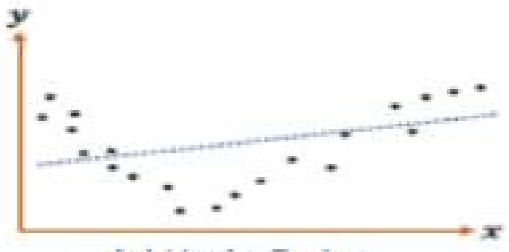
REASON FOR UNDERFITTING:

- When the capacity of the model is too small, the model cannot learn the modalities of the training set data well, resulting in poor performance on both the training set and the unseen samples.

- We call this phenomenon as under fitting.

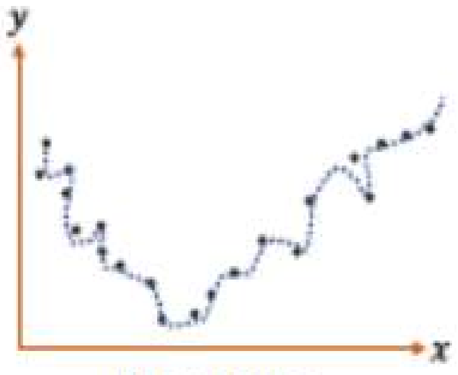
EXAMPLE:

- Consider a degree 2 polynomial data distribution.
- If we use a simple linear function to learn, we will find it difficult to learn a better function, resulting in the underfitting phenomenon that the training set and the test set do not perform well, as shown in Figure.



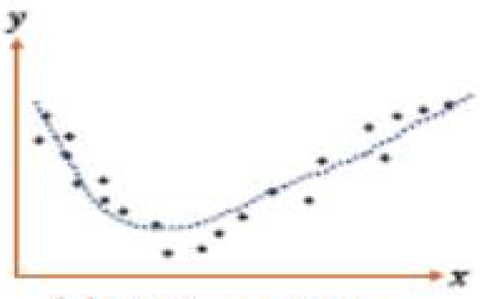
(a) Underfitting

- If you use a more complex function model to learn, it is possible that the learned function will excessively "fit" the training set samples, but resulting in poor performance on the test set, that is, overfitting, as shown in Figure



(c) Overfitting

- Only when the capacity of the learned model and the real model roughly match, the model can have a good generalization ability, as shown in Figure



(b) Matching capacity

SOLUTION TO UNDERFITTING:

- The problem of under fitting can be solved by increasing the number of layers of the neural network.
- It can also be solved by increasing the size of the intermediate dimension.
- However, because modern deep neural network models can easily reach deeper layers, the capacity of the model used for learning is generally sufficient.
- In real applications, more overfitting phenomena occur.

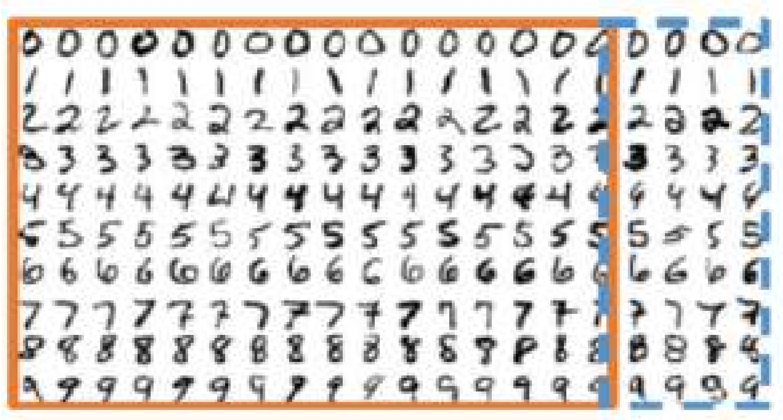
SOLUTION TO OVERFITTING:

1. Data set Division
2. Model Design
3. Regularization
4. Drop Out
5. Data Augmentation

DATASET DIVISION:

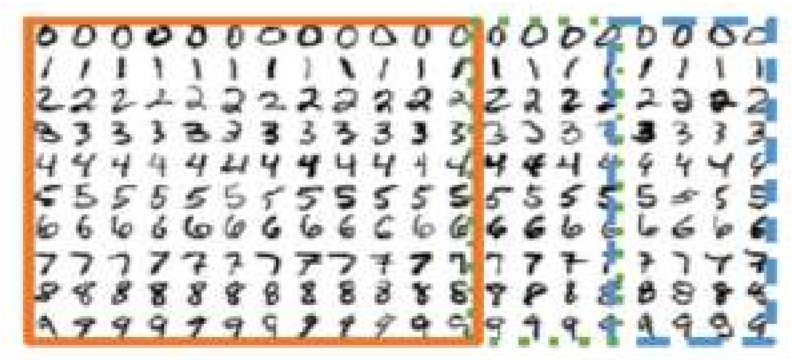
- Earlier we used to divide the data set into a training set and a test set.
- In order to select model hyper parameters and detect over fitting, it is generally necessary to split the original training set into three subsets:

  Training set, validation set, and test set.
- We know that training set *Dtrain* is used to train model parameters,
- The test set *Dtest* is used to test the generalization ability of the model.
- Example, Training set = 80% of MNIST dataset and Test set = 20% of MNIST data set.



Training and testing dataset division

- the performance of the test set cannot be used as feedback for model training.
- we need to be able to pick out more suitable model hyperparameters during model training to determine whether the model is overfitting.
- Therefore, we need to divide the training set into training set and validation set.



. *Training, validation, and test dataset*

- The divided training set has the same function as the original training set and is used to train the parameters of the model, **while the validation set is used to select the hyperparameters of the model.**

FUNCTIONS OF VALIDATION DATASET:

- Adjust the learning rate, weight decay coefficient, training times, etc. according to the performance of the validation set.
- Readjust the network topology according to the performance of the validation set.
- According to the performance of the validation set, determine whether it is overfitting or underfitting.
- the training set, validation set, and test set can be divided according to a custom ratio, such as the common 60%-20%-20% division.

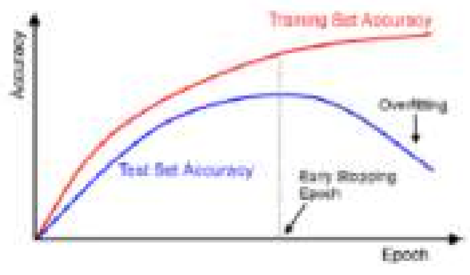
DIFFERENCE BETWEEN TEST & VALIDATION SETS:

- The algorithm designer can adjust the settings of various hyperparameters of the model according to the performance of the validation set to improve the generalization ability of the model.
- But the performance of the test set cannot be used to adjust the model.

EARLY STOPPING:

EPOCH:

- one batch updating in the training set one Step, and iterating through all the samples in the training set once is called an Epoch.
- It is generally recommended to perform a validation operation after several Epochs else it introduces additional computation costs.
- If the training error of the model is low and the training accuracy is high, but the validation error is high and the validation accuracy rate is low, overfitting may occur.
- If the errors on both the training set and the validation set are high and the accuracy is low, then underfitting may occur.



Training process diagram

EXAMPLE: A TYPICAL CLASSIFICATION

NOTE 1: In the laterstage of training, even with the same network structure, due to the change in the actual capacity of the model, we observed the phenomenon of overfitting

NOTE2:

- This means that for neural networks, even if the network hyperparameters amount remains unchanged (i.e., the maximum capacity of the network is fixed), the model may still appear to be overfitting.
- It is because the effective capacity of the neural network is closely related to the state of the network parameters
- As the number of training Epochs increased, the overfitting became more andmore serious.
- We can observe early stopping epoch as the vertical dotted line is in the best state of the network, there is no obvious overfitting phenomenon, and the generalization ability of the network is the best.

  When it is found that the validation accuracy has not decreased for successive Epochs, we can predict that the most suitable Epoch may have been reached, so we can stop training.

REGULARIZATION:

- By designing network models with different layers and sizes, the initial function hypothesis space can be provided for the optimization algorithm, but the actual capacity of the model can change as the network parameters are optimized and updated.

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon$$

- The capacity of the preceding model can be simply measuredthrough *n*.
- By limiting the sparsity of network parameters, the actual capacity of the network can be constrained.
- This constraint is generally achieved by adding additional parameter sparsity penalties to the loss function.
- Optimization goal before adding constraint:

$$min\ L\left(f_\theta(x), y\right), (x, y) \in D^{train}$$

- Optimisation goal before adding constraint:

$$min\ L\left(f_\theta(x), y\right) + \lambda \cdot \Omega(\theta), (x, y) \in D^{train}$$

- where *Ω*(*ϑ*) represents the sparsity constraint function on the network parameters *ϑ*.
- The sparsity constraint of the parameter *ϑ* is achieved by constraining the *L* norm of the parameter, that is:

$$\Omega(\theta)=\sum_{\theta_i} \|\theta_i\|_l$$

-
- The goal of an optimization algorithm is to minimize the original loss function L(x,y) and also to reduce network sparisty *Ω*(*ϑ*)
- Here *λ* is the weight parameter to balance the importance of *L*(*x*, *y*) and *Ω*(*ϑ*).
- Larger *λ* means that the sparsity of the network is more important; smaller *λ* means that the training error of the network is more important.
- By selecting the appropriate *λ*, you can get better training performance, while ensuring the sparsity of the network, which lead to a good generalization ability.
- Commonly used regularization methods are L0, L1, and L2 regularization.

L0 regularization:

- L0 regularization refers to the regularization calculation method using the L0 norm as the sparsity penalty term *Ω*(*ϑ*).

$$\Omega(\theta)=\sum_{\theta_i} \|\theta_i\|_0$$

-

- The L0 norm ||*ϑi*||o is defined as the number of non-zero elements in *ϑi*.

- This constraint can force the connection weights in the network to be mostly 0, thereby reducing the actual amount of network parameters and network capacity.

- DISADVANTAGE: However, because the L0 norm is not derivable, gradient descent algorithm cannot be used for optimization. L0 norm is not often used in neural networks

L1 Regularization

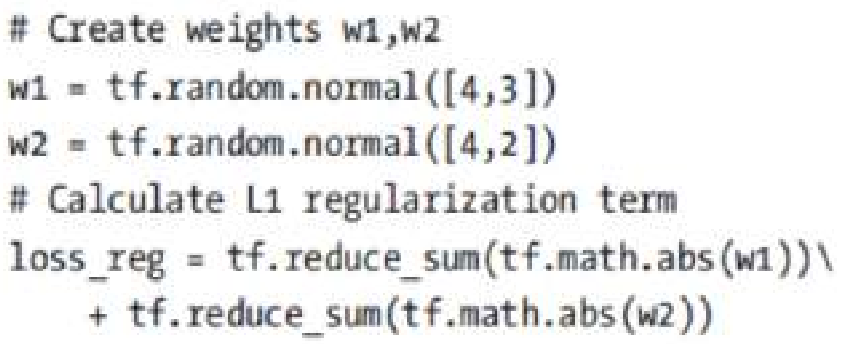
- The L1 regularization refers to the regularization calculation method using the L1 norm as the sparsity penalty term *Ω*(*ϑ*).

$$\Omega(\theta)=\sum_{\theta_i} \|\theta_i\|_1$$

- The L1 norm ||*ϑi*||1 is defined as the sum of the absolute values of all elements in the tensor *ϑi*.

- L1 regularization is also called **Lasso** regularization, which is continuously derivable and widely used in neural networks.

- IMPLEMENTATION:



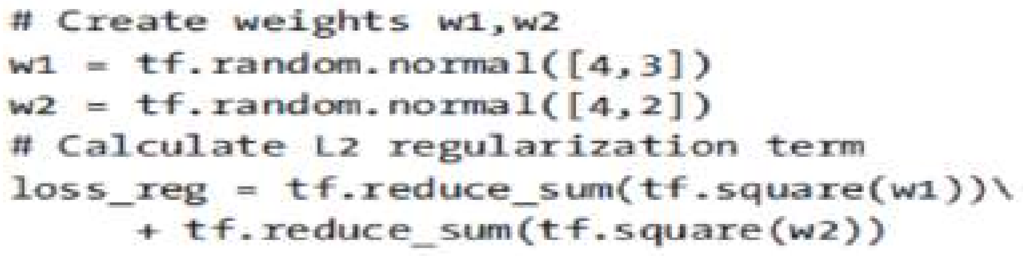
```
# Create weights w1,w2
w1 = tf.random.normal([4,3])
w2 = tf.random.normal([4,2])
# Calculate L1 regularization term
loss_reg = tf.reduce_sum(tf.math.abs(w1))\
    + tf.reduce_sum(tf.math.abs(w2))
```

L2 regularization:

- The L2 regularization refers to the regularization calculation method using the L2 norm as the sparsity penalty term *Ω*(*ϑ*).

$$\Omega(\theta)=\sum_{\theta_i} \|\theta_i\|_2$$

- The L2 norm ||*ϑi*||2 is defined as the sum of squares of the absolute values of all elements in the tensor *ϑi*.
- L1 regularization is also called **Ridge** regularization, which is continuously derivable and widely used in neural networks.
- IMPLEMENTATION:



```
# Create weights w1,w2
w1 = tf.random.normal([4,3])
w2 = tf.random.normal([4,2])
# Calculate L2 regularization term
loss_reg = tf.reduce_sum(tf.square(w1))\
    + tf.reduce_sum(tf.square(w2))
```

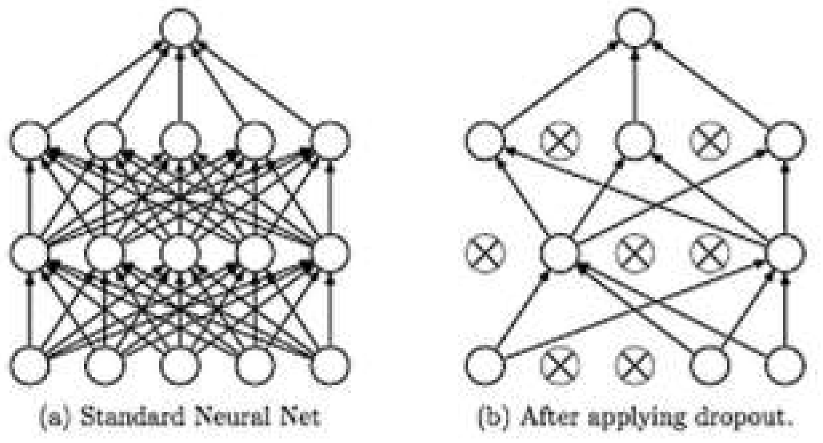
What does Regularization achieve?

- A standard least squares model tends to have some variance in it. Such model won't generalize well for a data set different than its training data.
- ***Regularization, significantly reduces the variance of the model, without substantial increase in its bias***.
- So the tuning parameter λ, used in the regularization, controls the impact on bias and variance.
- As the value of λ rises, it reduces the value of coefficients and thus reducing the variance.

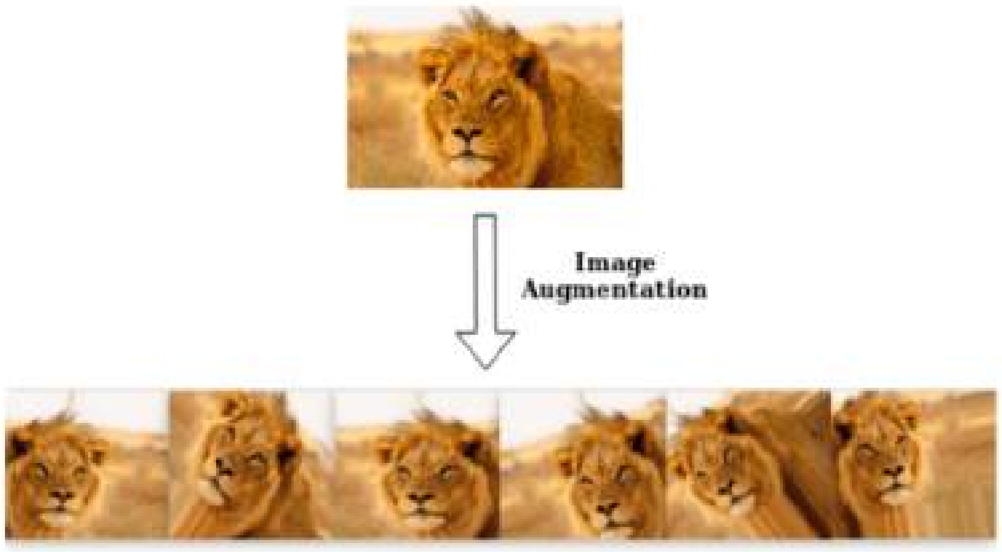
- ***Till a point, this increase in λ is beneficial as it is only reducing the variance(hence avoiding overfitting), without loosing any important properties in the data.***
- But after certain value, the model starts loosing important properties, giving rise to bias in the model and thus underfitting.
- Therefore, the value of λ should be carefully selected.

DROPOUT:

- Dropout works by essentially "dropping" a neuron from the input or hidden layers. Multiple neurons are removed from the network, meaning they practically do not exist — their incoming and outcoming connections are also destroyed.
- This artificially creates a multitude of smaller, less complex networks. This forces the model to not become solely dependent on one neuron, meaning it has to diversify its approach and develop a multitude of methods to achieve the same result.
- Dropout is applied to a neural network by randomly dropping neurons in every layer (including the input layer). A pre-defined dropout rate determines the chance of each neuron being dropped. For example, a dropout rate of 0.25 means that there is a 25% chance of a neuron being dropped. Dropout is applied during every epoch during the training of the model.



DATA AUGMENTATION:
One of the best techniques for reducing overfitting is to increase the size of the training dataset. As discussed in the previous technique, when the size of the training data is small, then the network tends to have greater control over the training data.
So, to increase the size of the training data i.e, increasing the number of images present in the dataset, we can use data augmentation, which is the easiest way to diversify our data and make the training data larger.
Some of the popular image augmentation techniques are flipping, translation, rotation, scaling,cropping, changing brightness, adding noise etc



Here are some commonly used data augmentation techniques for various types of data:

Image Data Augmentation:
a. Rotation: Rotate images by various degrees to simulate different angles.
b. Flip: Horizontally and/or vertically flip images.
c. Zoom: Randomly zoom in or out of images.
d. Crop: Randomly crop and resize images to different dimensions.
e. Translation: Shift images horizontally and/or vertically.
f. Brightness and Contrast Adjustments: Randomly adjust brightness and contrast.
g. Noise: Add random noise to the images.
h. Color Jitter: Randomly change hue, saturation, and brightness.

Text Data Augmentation:
a. Synonym Replacement: Replace words with their synonyms.
b. Random Deletion: Randomly delete words from the text.
c. Random Swap: Randomly swap the positions of two words in a sentence.
d. Back-Translation: Translate text to another language and then back to the original language.
e. Insertion: Insert random words into the text.

Time Series Data Augmentation:
a. Time Warping: Slightly warp the time series by stretching or compressing it.
b. Jittering: Add small random noise to the data points.
c. Rolling Window: Apply rolling window transformations to create new data points.

Audio Data Augmentation:
a. Pitch Shifting: Change the pitch of the audio.
b. Time Stretching: Stretch or compress the audio in time.
c. Noise Injection: Add background noise to the audio.
d. Speed Perturbation: Vary the playback speed of the audio.

Tabular Data Augmentation:
a. Feature Scaling: Scale features within a certain range.
b. Feature Selection: Randomly select a subset of features.
c. Data Perturbation: Add random noise to the data.

Synthetic Data Generation:
a. Generative Adversarial Networks (GANs): Generate synthetic data samples that mimic the real data distribution.
b. Variational Autoencoders (VAEs): Create new data points from the latent space of the VAE.
When applying data augmentation, it's essential to strike a balance. Too much augmentation can lead to model training on noisy or unrealistic data, while too little may not effectively combat overfitting. Experiment with different augmentation techniques and parameters to find the right balance for your specific problem and dataset. Cross-validation can help in evaluating the impact of data augmentation on your model's performance.