Kalinga University

Faculty of CS & IT

Course- BCAAIML
Subject:- Advance Neural Network & Deep Learning
Course Code – BCAAIML505

Sem- 5th

UNIT-4

Bidirectional RNNs – Concepts and use: -

A Bidirectional Recurrent Neural Network (BRNN) is an extension of the traditional RNN that processes sequential data in both forward and backward directions. This allows the network to utilize both past and future context when making predictions providing a more comprehensive understanding of the sequence.

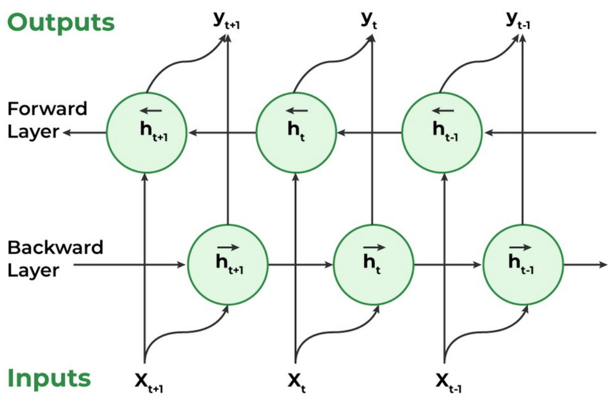
Like a traditional RNN, a BRNN moves forward through the sequence, updating the hidden state based on the current input and the prior hidden state at each time step. The key difference is that a BRNN also has a backward hidden layer which processes the sequence in reverse, updating the hidden state based on the current input and the hidden state of the next time step.

Compared to unidirectional RNNs BRNNs improve accuracy by considering both the past and future context. This is because the two hidden layers i.e forward and backward complement each other and predictions are made using the combined outputs of both layers.

Example:

Consider the sentence: "I like apple. It is very healthy."

In a traditional unidirectional RNN the network might struggle to understand whether "apple" refers to the fruit or the company based on the first sentence. However a BRNN would have no such issue. By processing the sentence in both directions, it can easily understand that "apple" refers to the fruit, thanks to the future context provided by the second sentence ("It is very healthy.").



Bi-directional Recurrent Neural Network

Working of Bidirectional Recurrent Neural Networks (BRNNs)

1. Inputting a Sequence: A sequence of data points each represented as a vector with the same dimensionality is fed into the BRNN. The sequence may have varying lengths.

2. Dual Processing: BRNNs process data in two directions:

- **Forward direction**: The hidden state at each time step is determined by the current input and the previous hidden state.
- **Backward direction**: The hidden state at each time step is influenced by the current input and the next hidden state.

3. Computing the Hidden State: A non-linear activation function is applied to the weighted sum of the input and the previous hidden state creating a memory mechanism that allows the network to retain information from earlier steps.

4. Determining the Output: A non-linear activation function is applied to the weighted sum of the hidden state and output weights to compute the output at each step. This output can either be:

- The final output of the network.
- An input to another layer for further processing.

Implementation of Bi-directional Recurrent Neural Network

Here's a simple implementation of a Bidirectional RNN using Keras and TensorFlow for sentiment analysis on the **IMDb dataset** available in keras:

1. Loading and Preprocessing Data

We first load the IMDb dataset and preprocess it by padding the sequences to ensure uniform length.

- **warnings.filterwarnings('ignore')** suppresses any warnings during execution.
- **imdb.load_data(num_words=features)** loads the IMDb dataset, considering only the top 2000 most frequent words.
- **pad_sequences(X_train,maxlen=max_len)** and **pad_sequences(X_test, maxlen=max_len)** pad the training and test sequences to a maximum length of 50 words ensuring consistent input size.

```
import warnings

warnings.filterwarnings('ignore')

from keras.datasets import imdb

from keras.preprocessing.sequence import pad_sequences


features = 2000  # Number of most frequent words to consider

max_len = 50     # Maximum length of each sequence


(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=features)


X_train = pad_sequences(X_train, maxlen=max_len)

X_test = pad_sequences(X_test, maxlen=max_len)
```

2. Defining the Model Architecture

We define a Bidirectional Recurrent Neural Network model using Keras. The model uses an embedding layer with 128 dimensions, a Bidirectional SimpleRNN layer with 64 hidden units and a dense output layer with a sigmoid activation for binary classification.

- **Embedding()** layer maps input features to dense vectors of size embedding (128), with an input length of len.
- **Bidirectional(SimpleRNN(hidden))** adds a bidirectional RNN layer with hidden (64) units.
- **Dense(1, activation='sigmoid')** adds a dense output layer with 1 unit and a sigmoid activation for binary classification.
- **model.compile()** configures the model with Adam optimizer, binary cross-entropy loss and accuracy as the evaluation metric.

```
from keras.models import Sequential

from keras.layers import Embedding, Bidirectional, SimpleRNN, Dense



embedding_dim = 128

hidden_units = 64



model = Sequential()



model.add(Embedding(features, embedding_dim, input_length=max_len))



model.add(Bidirectional(SimpleRNN(hidden_units)))



model.add(Dense(1, activation='sigmoid'))



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

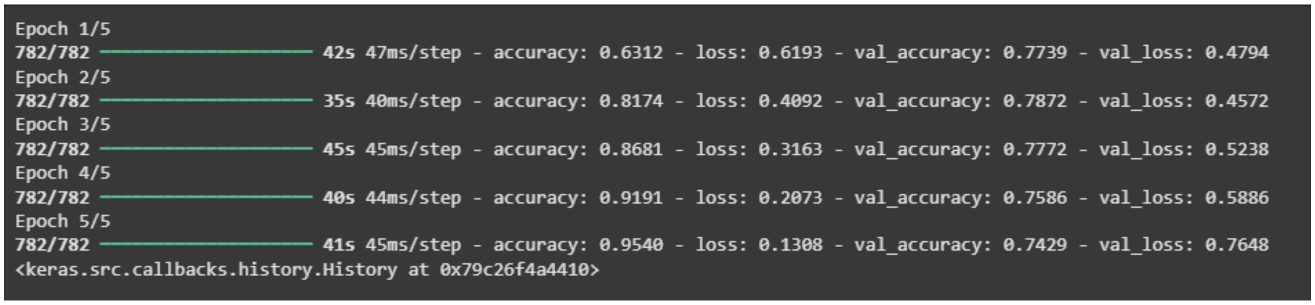
3. Training the Model

As we have compiled our model successfully and the data pipeline is also ready so, we can move forward toward the process of training our BRNN.

- **batch_size=32** defines how many samples are processed together in one iteration.
- **epochs=5** sets the number of times the model will train on the entire dataset.
- **model.fit()** trains the model on the training data and evaluates it using the provided validation data.

```
batch_size = 32

epochs = 5



model.fit(X_train, y_train,

        batch_size=batch_size,

        epochs=epochs,

        validation_data=(X_test, y_test))
```

Output:



```
Epoch 1/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 42s 47ms/step - accuracy: 0.6312 - loss: 0.6193 - val_accuracy: 0.7739 - val_loss: 0.4794
Epoch 2/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 35s 40ms/step - accuracy: 0.8174 - loss: 0.4092 - val_accuracy: 0.7872 - val_loss: 0.4572
Epoch 3/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 45s 45ms/step - accuracy: 0.8681 - loss: 0.3163 - val_accuracy: 0.7772 - val_loss: 0.5238
Epoch 4/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 40s 44ms/step - accuracy: 0.9191 - loss: 0.2073 - val_accuracy: 0.7586 - val_loss: 0.5886
Epoch 5/5
782/782 ━━━━━━━━━━━━━━━━━━━━ 41s 45ms/step - accuracy: 0.9540 - loss: 0.1308 - val_accuracy: 0.7429 - val_loss: 0.7648
<keras.src.callbacks.history.History at 0x79c26f4a4410>
```

Training the Model

4. Evaluating the Model

Now as we have our model ready let's evaluate its performance on the validation data using different evaluation metrics. For this purpose we will first predict the class for the validation data using this model and then compare the output with the true labels.

- **model.evaluate(X_test, y_test)** evaluates the model's performance on the test data (X_test, y_test), returning the loss and accuracy.

```
loss, accuracy = model.evaluate(X_test, y_test)



print('Test accuracy:', accuracy)
```

Output :

```
Test accuracy: 0.7429199814796448
```

Here we achieved a accuracy of 74% and we can increase it accuracy by more fine tuning.

5. Predict on Test Data

We will use the model to predict on the test data and compare the predictions with the true labels.

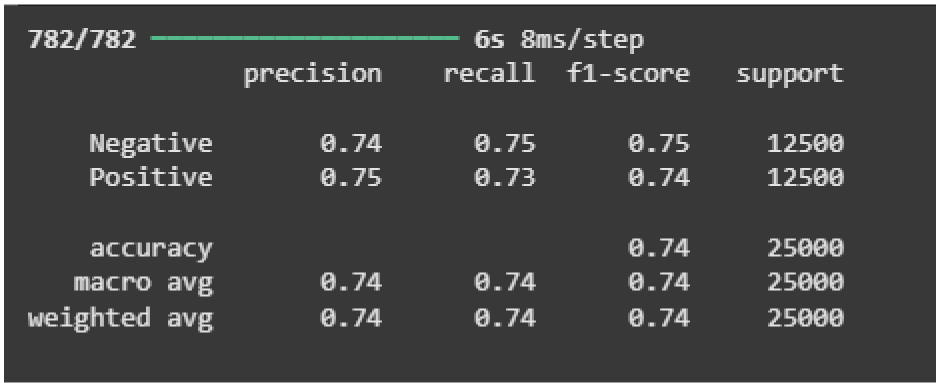
- **model.predict(X_test)** generates predictions for the test data.
- **y_pred = (y_pred > 0.5)** converts the predicted probabilities into binary values (0 or 1) based on a threshold of 0.5.
- **classification_report(y_test, y_pred, target_names=['Negative', 'Positive'])** generates and prints a classification report including precision, recall, f1-score and support for the negative and positive classes.

```
from sklearn.metrics import classification_report



y_pred = model.predict(X_test)
```

y_pred = (y_pred > 0.5)

print(classification_report(y_test, y_pred, target_names=['Negative', 'Positive']))

Output:



```
782/782 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step
              precision    recall  f1-score   support

    Negative       0.74      0.75      0.75     12500
    Positive       0.75      0.73      0.74     12500

    accuracy                           0.74     25000
   macro avg       0.74      0.74      0.74     25000
weighted avg       0.74      0.74      0.74     25000
```

Predict on Test Data

Advantages of BRNNs

- **Enhanced Context Understanding**: Considers both past and future data for improved predictions.
- **Improved Accuracy**: Particularly effective for NLP and speech processing tasks.
- **Better Handling of Variable-Length Sequences**: More flexible than traditional RNNs making it suitable for varying sequence lengths.
- **Increased Robustness**: Forward and backward processing help filter out noise and irrelevant information, improving robustness.

Challenges of BRNNs

- **High Computational Cost**: Requires twice the processing time compared to unidirectional RNNs.
- **Longer Training Time**: More parameters to optimize result in slower convergence.
- **Limited Real-Time Applicability**: Since predictions depend on the entire sequence hence they are not ideal for real-time applications like live speech recognition.
- **Less Interpretability**: The bidirectional nature of BRNNs makes it more difficult to interpret predictions compared to standard RNNs.

Applications of Bidirectional Recurrent Neural Networks (BRNNs)

BRNNs are widely used in various natural language processing (NLP) tasks, including:

- **Sentiment Analysis**: By considering both past and future context they can better classify the sentiment of a sentence.

- **Named Entity Recognition (NER)**: It helps to identify entities in sentences by analyzing the context in both directions.

- **Machine Translation**: In encoder-decoder models, BRNNs allow the encoder to capture the full context of the source sentence in both directions hence improving translation accuracy.

- **Speech Recognition**: By considering both previous and future speech elements it enhance the accuracy of transcribing audio.

In deep learning the encoder-decoder model is a type of neural network that is mainly used for tasks where both the input and output are sequences. This architecture is used when the input and output sequences are not the same length for example translating a sentence from one language to another, summarizing a paragraph, describing an image with a caption or convert speech into text. It works in two stages:

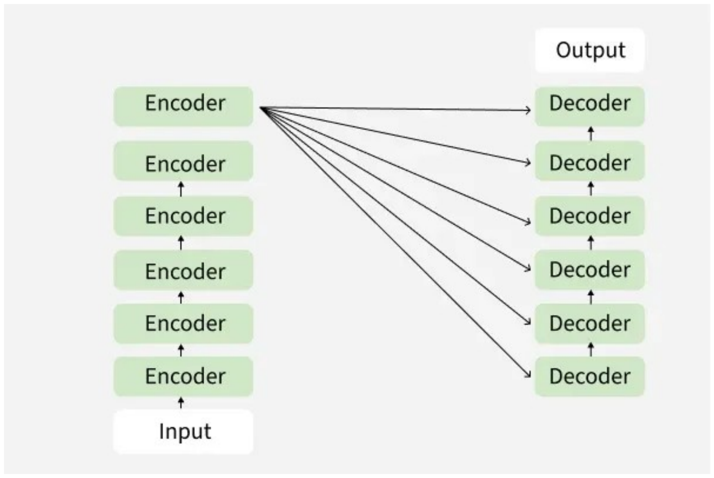
- **Encoder**: The **encoder** takes the input data like a sentence and processes each word one by one then creates a single, fixed-size summary of the entire input called a **context vector** or **latent space**.

- **Decoder**: The **decoder** takes the context vector and begins to produce the output one step at a time.

For example, in **machine translation** an encoder-decoder model might take an English sentence as input (like "I am learning AI") and translate it into French ("Je suis en train d'apprendre l'IA").

Encoder-Decoder Model Architecture:-

In an encoder-decoder model both the encoder and decoder are separate networks each one has its own specific task. These networks can be different types such as Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), Gated Recurrent Units (GRUs), Convolutional Neural Networks (CNNs) or even more advanced models like Transformers.



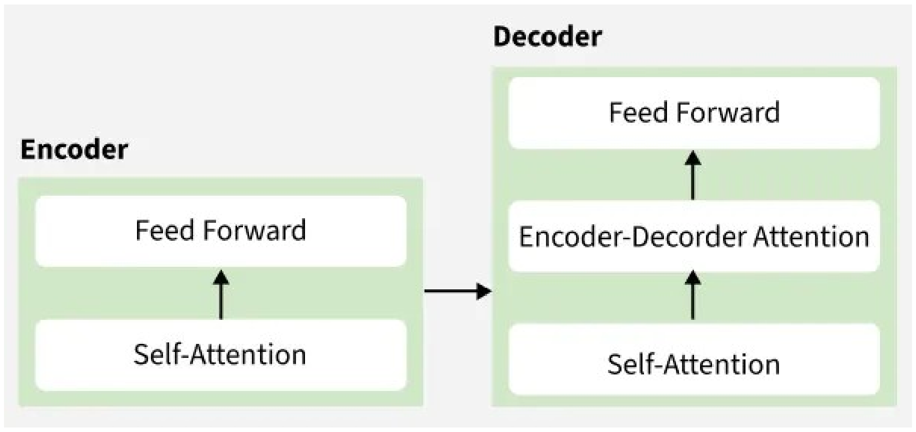
Encoder Decoder Architecture

Encoder

The encoder's job is to process the input data and convert it into a form that the model can understand. It does this using two main steps:

1. **Self-Attention Layer**: This layer helps the encoder focus on different parts of the input data that are important for understanding the context. For example in a sentence it allows the model to consider how each word relates to the others.

2. **Feed-Forward Neural Network**: After the self-attention layer this network processes the information further to capture complex patterns and relationships in the data.



Decoder

The decoder takes the processed information from the encoder and generates the output. It also has three main components:

1. **Self-Attention Layer**: Similar to the encoder this layer allows the decoder to focus on different parts of the output it has generated.

2. **Encoder-Decoder Attention Layer**: This unique layer enables the decoder to focus on relevant parts of the input data help to generate more accurate outputs.

3. **Feed-Forward Neural Network**: Like the encoder the decoder uses this network to process the information and generate the final output.

Working of Encoder Decoder Model

The actual working of the encoder decoder model is shown in below diagram. Now we will understand it stepwise:

Step 1: Tokenizing the Input Sentence

- The sentence "I am learning AI" is first broken into tokens: **["I", "am", "learning", "AI"]**.

- Each word (token) is converted into a **vector** that a machine can understand. This process is called **embedding**.

Step 2: Encoding the Input

- The **Encoder** processes these embeddings using **self-attention**.

- Self-attention helps the encoder to focus on important words. For example while encoding "learning", it understands its relation with "I" and "AI."

- After processing the encoder generates a **Context Vector** which captures the meaning of the entire sentence. For example in the image The arrows show how each word relates to the others during encoding. The final output from the encoder is the **context representation**

Step 3: Passing the Context to the Decoder

- The **Context Vector** is passed to the Decoder as shown in image.

- It acts like a summary of the full input sentence.

Step 4: Decoder Generates Output Step-by-Step

- The **Decoder** uses the context and starts creating the output one word at a time.
- First it predicts the first word then uses that to predict the second word and so on

Step 5: Decoder Attention

- While generating each word the decoder **attends** to different parts of the input sentence to make better predictions.
- For example when translating "learning," it might pay more attention to the word "learning" in the input.

Step 6: Producing the Final Output

- The decoder continues generating until the full translated sentence is produced.
- Each output token depends on the previous ones and the input context. You finally see the output tokens generated on the right side of the diagram completing the translation.

Implementation of Encoder and Decoder

Step 1: Import Libraries and Load dataset

In this step we import all the necessary libraries like numpy , pandas , string and Tokenizer , pad_sequence for preprocessing the text into model-friendly format and load the dataset. You can download the dataset from here

```
import numpy as np, pandas as pd, string

from string import digits

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, LSTM, Embedding, Dense
from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences



lines = pd.read_csv("/content/Hindi_English_Truncated_Corpus.csv", encoding='utf-8')

lines = lines[lines['source'] == 'ted'][['english_sentence', 'hindi_sentence']].dropna().drop_duplicates()

lines = lines.sample(n=25000, random_state=42)
```

Step 2: Text Cleaning

```
def clean_text(text):

    exclude = set(string.punctuation)

    text = ''.join(ch for ch in text if ch not in exclude)

    text = text.translate(str.maketrans('', '', digits))

    return text.strip().lower()
```

The above code remove punctuation and digits and converts text to lowercase and strips whitespace.

```
lines['english_sentence'] = lines['english_sentence'].apply(clean_text)

lines['hindi_sentence'] = lines['hindi_sentence'].apply(clean_text)

lines['hindi_sentence'] = lines['hindi_sentence'].apply(lambda x: 'start_ ' + x + ' _end')
```

It applies Applies cleaning and adds special tokens to Hindi sentences to mark start and end (start_, _end).

Step 4: Tokenization

```
eng_tokenizer = Tokenizer()

eng_tokenizer.fit_on_texts(lines['english_sentence'])

eng_seq = eng_tokenizer.texts_to_sequences(lines['english_sentence'])



hin_tokenizer = Tokenizer(filters='')

hin_tokenizer.fit_on_texts(lines['hindi_sentence'])

hin_seq = hin_tokenizer.texts_to_sequences(lines['hindi_sentence'])
```

Converts text to sequences of integers using word indices. Hindi tokenizer keeps_because of special tokens.

Step 5: Padding

```
max_eng_len = max(len(seq) for seq in eng_seq)

max_hin_len = max(len(seq) for seq in hin_seq)



encoder_input = pad_sequences(eng_seq, maxlen=max_eng_len, padding='post')

decoder_input = pad_sequences(hin_seq, maxlen=max_hin_len, padding='post')
```

Pads sequences to uniform length

decoder_target = np.zeros((decoder_input.shape[0], decoder_input.shape[1], 1))

decoder_target[:, 0:-1, 0] = decoder_input[:, 1:]

decoder_target is shifted version of decoder_input used for teacher forcing. Like if decoder input id "start_maine dekha" the target is "maine dekha_end".

Step 6: Define Model Architecture

Encoder:

encoder_inputs = Input(shape=(None,))

enc_emb = Embedding(eng_vocab_size, latent_dim)(encoder_inputs)

enc_outputs, state_h, state_c = LSTM(latent_dim, return_state=True)(enc_emb)

encoder_states = [state_h, state_c]

It embeds English input and Passes through LSTM. Keeps hidden (state_h) and cell state (state_c) to pass to decoder.

Decoder:

decoder_inputs = Input(shape=(None,))

```
dec_emb_layer = Embedding(hin_vocab_size, latent_dim)

dec_emb = dec_emb_layer(decoder_inputs)

decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)

decoder_outputs, _, _ = decoder_lstm(dec_emb, initial_state=encoder_states)

decoder_dense = Dense(hin_vocab_size, activation='softmax')

decoder_outputs = decoder_dense(decoder_outputs)
```

It embeds Hindi input. Uses initial states from encoder and Outputs probability distribution over Hindi vocabulary at each time step.

Step 7. Compile and Train

```
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy')

model.fit([encoder_input, decoder_input], decoder_target, batch_size=64, epochs=20, validation_split=0.2)
```

Trains on source (encoder_input) and target (decoder_input) with shifted targets and uses RMSProp optimizer and cross-entropy loss.

Step 8: Inference Models

To translate new sentences after training:
Encoder Inference

```
encoder_model_inf = Model(encoder_inputs, encoder_states)
```

Returns hidden/cell states given an English sentence.

Decoder Inference

```
decoder_state_input_h = Input(shape=(latent_dim,))

decoder_state_input_c = Input(shape=(latent_dim,))

decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]

dec_inf_emb = dec_emb_layer(decoder_inputs)

dec_outputs_inf, state_h_inf, state_c_inf = decoder_lstm(dec_inf_emb, initial_state=decoder_states_inputs)

decoder_outputs_inf = decoder_dense(dec_outputs_inf)

decoder_model_inf = Model([decoder_inputs] + decoder_states_inputs, [decoder_outputs_inf, state_h_inf, state_c_inf])
```

Step 9: Reverse Lookup

```
reverse_eng = {v: k for k, v in eng_tokenizer.word_index.items()}

reverse_hin = {v: k for k, v in hin_tokenizer.word_index.items()}
```

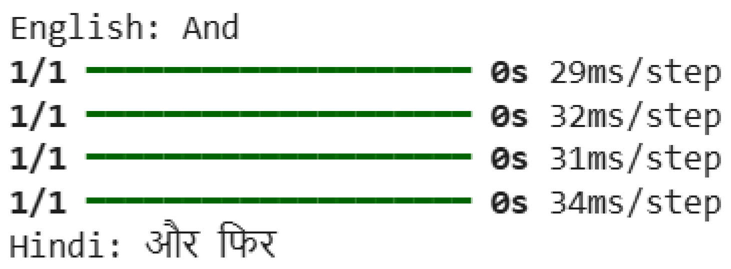
Used to convert indices back to words during decoding.

Step 10: Translate Function

```
def translate(sentence):

    sentence = clean_text(sentence)

    seq = eng_tokenizer.texts_to_sequences([sentence])

    padded = pad_sequences(seq, maxlen=max_eng_len, padding='post')

    states = encoder_model_inf.predict(padded)



    target_seq = np.zeros((1, 1))

    target_seq[0, 0] = hin_tokenizer.word_index['start_']



    decoded = []

    while True:

        output, h, c = decoder_model_inf.predict([target_seq] + states)

        token_index = np.argmax(output[0, -1, :])

        word = reverse_hin.get(token_index, '')



        if word == '_end' or len(decoded) >= max_hin_len:

            break



        decoded.append(word)

        target_seq = np.zeros((1, 1))
        target_seq[0, 0] = token_index

        states = [h, c]



    return ' '.join(decoded)



print("English: And")

print("Hindi:", translate("And"))
```

It prepares input sentence. Starts decoding with <start> token and Iteratively predicts next word and feeds it back until <end> is predicted. and the test the model with example

Output:



```
English: And
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 34ms/step
Hindi: और फिर
```

English to Hindi Translation

Sequence-to-sequence learning with RNNs:-

Sequence-to-Sequence learning models take in one sequence of words, numbers, or any other data as input, and give back another sequence as output that makes sense based on the input. These models are used when we perform tasks that require understanding input data in sequence and then generating another sequence. It is similar to how we understand we don't interpret words one by one but as a whole sentence.

Purpose of Sequence-to-Sequence Learning

There are different tasks where we have to deal with an order or sequence such as language, where word order matters a lot. Below are a few examples where sequence-to-sequence learning is very important:

- **Language Translation**: This model is used for converting sentences from one language to another.
- **Text Summarization**: It is used for creating a shorter version of a longer text while retaining its main ideas.
- **Speech Recognition**: It is used for changing spoken language into written text.
- **Chatbots**: It is used for generating responses based on input given by us.

Training Sequence-to-Sequence Models

Training Seq2Seq models mainly involves Supervised Learning, where we have both input sequences and the correct output sequences.

Below is example to understand, suppose:

- **Input:** "Hello" (in English)
- **Output:** "Hola" (in Spanish)

The model learns to map English words to their Spanish counterparts by looking at many examples.

Common Algorithms Used

- **Recurrent Neural Networks (RNNs)**: RNNs are used because they work well with sequential data. However, standard RNNs struggle with long sequences.
- **Long Short-Term Memory (LSTMs)**: LSTMs are a type of RNN that solves the problem of remembering long sequences. They are widely used in Seq2Seq models because they handle dependencies between words better.
- **Gated Recurrent Units (GRUs)**: Similar to LSTMs but slightly simpler. GRUs are also used in Seq2Seq models.
- **Transformers**: Recently, **transformers** have become the most popular model for Seq2Seq tasks because they use attention mechanisms to handle long sequences more efficiently. Models like **BERT** and **GPT** are based on transformers and perform incredibly well on NLP tasks.

Applications of Sequence-to-Sequence Learning

There are different applications of sequence-to-sequence learning. Some of very common applications are mentioned below:

1. **Machine Translation:** The most popular application of Seq2Seq is it is used in language translation. Services such as google translate which we have seen in google uses Seq2Seq models to translate text from one language to another.

2. **Chatbots:** When we ask a chatbot any question, it responds by generating a sequence of answer in text format based on our input sequence i.e. question. Seq2Seq models allow chatbots to generate context-aware responses.

3. **Text Summarization:** This model is used in text summarization which takes in a long document and generates a shorter and precise summary. This is useful for news articles, research papers, and other lengthy documents.

4. **Speech Recognition:** Seq2Seq models are used in converting the speech-to-text They convert spoken audio sequences input into text sequences output. Virtual assistants such as Siri and Google Assistant uses the Seq2Seq models for accurate transcription of speech.

5. **Caption Generation:** Seq2Seq models can also be used to generate captions for images, GIFS and videos. The input sequence is the image or video, and the output sequence will be the image or video with caption which describe the meaning of that image or video.

Code Example: Using Seq2Seq in TensorFlow

Below is the simple example using **TensorFlow** to build a Seq2Seq model for language translation:

```
english_sentences = ['hello', 'world']

spanish_sentences = ['hola', 'mundo']



import numpy as np

import tensorflow as tf
```

```
from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, LSTM, Dense


# Configuration

num_encoder_tokens = 256  # Number of unique input tokens (for simplicity, assume ASCII range)

num_decoder_tokens = 256  # Number of unique output tokens (same assumption)

latent_dim = 64           # Latent dimensionality of the encoding space


# Encoder

encoder_inputs = Input(shape=(None, num_encoder_tokens))

encoder = LSTM(latent_dim, return_state=True)

encoder_outputs, state_h, state_c = encoder(encoder_inputs)

encoder_states = [state_h, state_c]


# Decoder

decoder_inputs = Input(shape=(None, num_decoder_tokens))

decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)

decoder_outputs, _, _ = decoder_lstm(decoder_inputs, initial_state=encoder_states)

decoder_dense = Dense(num_decoder_tokens, activation='softmax')

decoder_outputs = decoder_dense(decoder_outputs)


# Model

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

# Dummy data for demonstration; in practice, use actual tokenized and one-hot encoded data
batch_size = 2
seq_length = 5  # Max length of the sequence
encoder_input_data = np.zeros((batch_size, seq_length, num_encoder_tokens), dtype='float32')
decoder_input_data = np.zeros((batch_size, seq_length, num_decoder_tokens), dtype='float32')
decoder_target_data = np.zeros((batch_size, seq_length, num_decoder_tokens), dtype='float32')

# Simple one-hot encoding for demonstration
for i, (input_text, target_text) in enumerate(zip(english_sentences, spanish_sentences)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, ord(char)] = 1.0
    for t, char in enumerate(target_text):
        decoder_input_data[i, t, ord(char)] = 1.0
        if t > 0:
            decoder_target_data[i, t - 1, ord(char)] = 1.0

# Train the model
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size=batch_size,
          epochs=50)  # Increase epochs for real training data
```

```
# Inference models for testing

encoder_model = Model(encoder_inputs, encoder_states)



decoder_state_input_h = Input(shape=(latent_dim,))

decoder_state_input_c = Input(shape=(latent_dim,))

decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]



decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)

decoder_states = [state_h, state_c]

decoder_outputs = decoder_dense(decoder_outputs)

decoder_model = Model([decoder_inputs] + decoder_states_inputs, [decoder_outputs] + decoder_states)



def decode_sequence(input_seq):

    states_value = encoder_model.predict(input_seq)



    target_seq = np.zeros((1, 1, num_decoder_tokens))

    target_seq[0, 0, ord(' ')] = 1.  # Assuming ' ' (space) as the 'start' character for decoding



    stop_condition = False

    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict([target_seq] + states_value)
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = chr(sampled_token_index)
        decoded_sentence += sampled_char

        # Corrected condition: Check if sampled_char is ' ' or length exceeds seq_length
        if sampled_char == ' ' or len(decoded_sentence) > seq_length:
            stop_condition = True

        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        states_value = [h, c]

    return decoded_sentence

# Example of how to use this function:
test_input = np.zeros((1, seq_length, num_encoder_tokens), dtype='float32')
for t, char in enumerate('hello'):
    test_input[0, t, ord(char)] = 1.0

translated_sentence = decode_sequence(test_input)
```

print("Translated sentence:", translated_sentence)

Output:

```
Epoch 1/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step - accuracy: 0.0000e+00 - loss: 3.8820
Epoch 2/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 50ms/step - accuracy: 0.1000 - loss: 3.8677
Epoch 3/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 56ms/step - accuracy: 0.2000 - loss: 3.8564
Epoch 4/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.2000 - loss: 3.8459
.
.
.
Epoch 49/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - accuracy: 0.2000 - loss: 1.3696
Epoch 50/50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 57ms/step - accuracy: 0.2000 - loss: 1.3609
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 88ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
Translated sentence: oooooo
```

BPTT – Backpropagation Through Time:-

Recurrent Neural Networks (RNNs) are designed to process sequential data. Unlike traditional neural networks, RNN outputs depend not only on the current input but also on previous inputs through a memory element. This memory allows RNNs to capture temporal dependencies in data such as time series or language.

Training RNNs involves backpropagation where instead of updating weights based only on the current timestep tt, we also consider all previous timesteps : t−1,t−2,t−3,…t−1,t−2,t−3,… .

This method is called Backpropagation Through Time (BPTT) which extends traditional backpropagation to sequential data by unfolding the network over time and summing gradients across all relevant time steps. This method enables RNNs to learn complex temporal patterns.

RNN Architecture

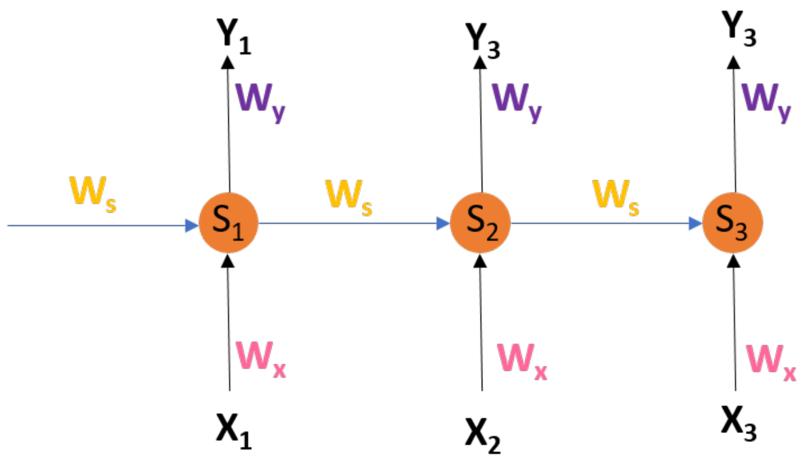
At each timestep tt, the RNN maintains a hidden state StSt, which acts as the network's memory summarizing information from previous inputs. The hidden state StSt updates by combining the current input XtXt and the previous hidden stateSt−1St−1, applying an activation function to introduce non-linearity. Then the output YtYtis generated by transforming this hidden state.

St=g1(WxXt+WsSt−1)St=g1(WxXt+WsSt−1)

- StSt represents the hidden state (memory) at time tt.
- XtXt is the input at time t.t.
- YtYt is the output at time t.t.
- Ws,Wx,WyWs,Wx,Wy are weight matrices for hidden states, inputs and outputs, respectively.

Yt=g2(WySt)Yt=g2(WySt)

where g1g1 and g2g2 are activation functions.



RNN Architecture

Error Function at Time t=3t=3

To train the network, we measure how far the predicted output YtYt is from the desired output dtdt using an error function. We use the squared error to measure the difference between the desired output dtdt and actual output YtYt:

Et=(dt−Yt)2Et=(dt−Yt)2

At t=3t=3:

E3=(d3−Y3)2E3=(d3−Y3)2

This error quantifies the difference between the predicted output and the actual output at time 3.

Updating Weights Using BPTT

BPTT updates the weights Wy,Ws,WxWy,Ws,Wx to minimize the error by computing gradients. Unlike standard backpropagation, BPTT unfolds the network across time steps, considering how errors at time tt depend on all previous states.

We want to adjust the weights WyWy, WsWs and WxWx to minimize the error E3E3.

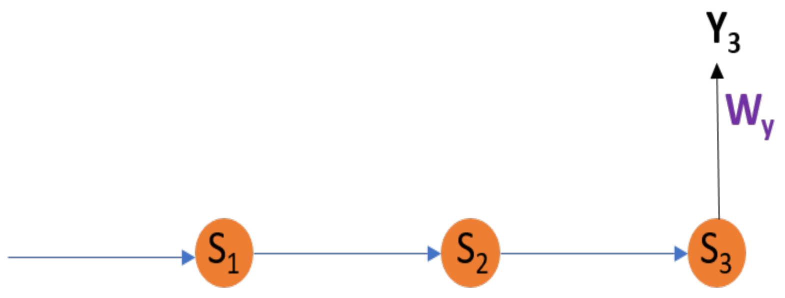
1. Adjusting Output Weight WyWy

The output weight WyWy affects the output directly at time 3. This means we calculate how the error changes as Y3Y3 changes, then how Y3Y3 changes with respect to WyWy. Updating WyWy is straightforward because it only influences the current output.

Using the chain rule:

∂E3∂Wy=∂E3∂Y3×∂Y3∂Wy∂Wy∂E3=∂Y3∂E3×∂Wy∂Y3

- E3E3 depends on Y3Y3, so we differentiate E3E3 w.r.t. Y3Y3.
- Y3Y3 depends on WyWy, so we differentiate Y3Y3 w.r.t. WyWy.



Adjusting Wy

2. Adjusting Hidden State Weight WsWs

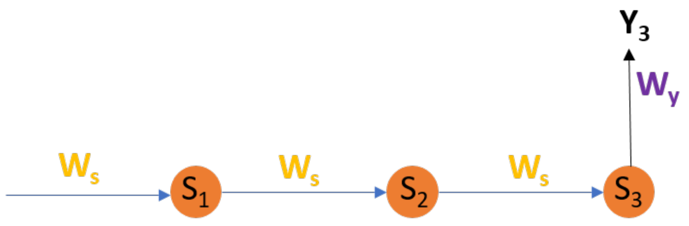
The hidden state weight WsWs influences not just the current hidden state but all previous ones because each hidden state depends on the previous one. To update WsWs, we must consider how changes to WsWs affect all hidden statesS1,S2,S3S1,S2,S3 and consequently the output at time 3.

The gradient for WsWs considers all previous hidden states because each hidden state depends on the previous one:

∂E3∂Ws=∑i=13∂E3∂Y3×∂Y3∂Si×∂Si∂Ws∂Ws∂E3=∑i=13∂Y3∂E3×∂Si∂Y3×∂Ws∂Si

Breaking down:

- Start with the error gradient at output Y3Y3.
- Propagate gradients back through all hidden states S3,S2,S1S3,S2,S1 since they affect Y3Y3.
- Each SiSi depends on WsWs, so we differentiate accordingly.



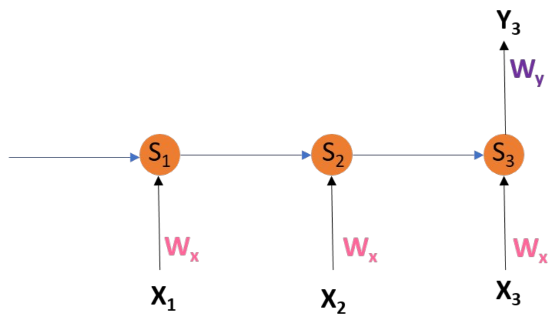
Adjusting Ws

3. Adjusting Input Weight WxWx

Similar to WsWs, the input weight WxWx affects all hidden states because the input at each timestep shapes the hidden state. The process considers how every input in the sequence impacts the hidden states leading to the output at time 3.

∂E3∂Wx=∑i=13∂E3∂Y3×∂Y3∂Si×∂Si∂Wx∂Wx∂E3=∑i=13∂Y3∂E3×∂Si∂Y3×∂Wx∂Si

The process is similar to WsWs, accounting for all previous hidden states because inputs at each timestep affect the hidden states.



Adjusting Wx

Advantages of Backpropagation Through Time (BPTT)

- Captures Temporal Dependencies: BPTT allows RNNs to learn relationships across time steps, crucial for sequential data like speech, text and time series.
- Unfolding over Time: By considering all previous states during training, BPTT helps the model understand how past inputs influence future outputs.
- Foundation for Modern RNNs: BPTT forms the basis for training advanced architectures such as LSTMs and GRUs, enabling effective learning of long sequences.
- Flexible for Variable Length Sequences: It can handle input sequences of varying lengths, adapting gradient calculations accordingly.

Limitations of BPTT

- Vanishing Gradient Problem: When backpropagating over many time steps, gradients tend to shrink exponentially, making early time steps contribute very little to weight updates. This causes the network to “forget” long-term dependencies.
- Exploding Gradient Problem: Gradients may also grow uncontrollably large, causing unstable updates and making training difficult.

Solutions

- Long Short-Term Memory (LSTM): Special RNN cells designed to maintain information over longer sequences and mitigate vanishing gradients.
- Gradient Clipping: Limits the magnitude of gradients during backpropagation to prevent explosion by normalizing them when exceeding a threshold.

Vanishing gradient problem in RNNs:-

The vanishing gradient problem is a challenge that emerges during backpropagation when the derivatives or slopes of the activation functions become progressively smaller as we move backward through the layers of a neural network. This phenomenon is particularly prominent in deep networks with many layers, hindering the effective training of the model. The weight updates becomes extremely tiny, or even exponentially small, it can significantly prolong the training time, and in the worst-case scenario, it can halt the training process altogether.

Why the Problem Occurs?

During backpropagation, the gradients propagate back through the layers of the network, they decrease significantly. This means that as they leave the output layer and return to the input layer, the gradients become progressively smaller. As a result, the weights associated with the initial levels, which accommodate these small gradients, are updated little or not at each iteration of the optimization process.

The vanishing gradient problem is particularly associated with the sigmoid and hyperbolic tangent (tanh) activation functions because their derivatives fall within the range of 0 to 0.25 and 0 to 1, respectively. Consequently, extreme weights becomes very small, causing the updated weights to closely resemble the original ones. This persistence of small updates contributes to the vanishing gradient issue.

The sigmoid and tanh functions limit the input values to the ranges [0,1] and [-1,1], so that they saturate at 0 or 1 for sigmoid and -1 or 1 for Tanh. The derivatives at points becomes zero as they are moving. In these regions, especially when inputs are very small or large, the gradients are very close to zero. While this may not be a major concern in shallow networks with a few layers, it is a more pronounced issue in deep networks. When the inputs fall in saturated regions, the gradients approach zero, resulting in little update to the weights of the previous layer. In simple networks this does not pose much of a problem, but as more layers are added, these small gradients, which multiply between layers, decay significantly and consequently the first layer tears very slowly , and hinders overall model performance and can lead to convergence failure.

How can we identify?

Identifying the vanishing gradient problem typically involves monitoring the training dynamics of a deep neural network.

- One key indicator is observing model weights converging to 0 or stagnation in the improvement of the model's performance metrics over training epochs.
- During training, if the loss function fails to decrease significantly, or if there is erratic behavior in the learning curves, it suggests that the gradients may be vanishing.
- Additionally, examining the gradients themselves during backpropagation can provide insights. Visualization techniques, such as gradient histograms or norms, can aid in assessing the distribution of gradients throughout the network.

How can we solve the issue?

- Batch Normalization : Batch normalization normalizes the inputs of each layer, reducing internal covariate shift. This can help stabilize and accelerate the training process, allowing for more consistent gradient flow.

- Activation function: Activation function like Rectified Linear Unit (ReLU) can be used. With ReLU, the gradient is 0 for negative and zero input, and it is 1 for positive input, which helps alleviate the vanishing gradient issue. Therefore, ReLU operates by replacing poor enter values with 0, and 1 for fine enter values, it preserves the input unchanged.

- Skip Connections and Residual Networks (ResNets): Skip connections, as seen in ResNets, allow the gradient to bypass certain layers during backpropagation. This facilitates the flow of information through the network, preventing gradients from vanishing.

- Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs): In the context of recurrent neural networks (RNNs), architectures like LSTMs and GRUs are designed to address the vanishing gradient problem in sequences by incorporating gating mechanisms .

- Gradient Clipping: Gradient clipping involves imposing a threshold on the gradients during backpropagation. Limit the magnitude of gradients during backpropagation, this can prevent them from becoming too small or exploding, which can also hinder learning.

Build and train a model for Vanishing Gradient Problem

let's see how the problems occur , and way to handle them.

Step 1: Import Libraries

First, import the necessary libraries for model

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

```
import tensorflow as tf
import keras
from sklearn.model_selection import train_test_split
from keras.layers import Dense
from keras.models import Sequential
import seaborn as sns
from imblearn.over_sampling import RandomOverSampler
from keras.layers import Dense, Dropout
from keras.optimizers import Adam
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from keras.initializers import random_normal
from keras.constraints import max_norm
from keras.optimizers import SGD
```

Step 2: Loading dataset

The code loads two CSV files (Credit_card.csv and Credit_card_label.csv) into Pandas DataFrames, df and labels.

<u>Link to dataset: Credit card details Binary classification.</u>

```
df = pd.read_csv('/content/Credit_card.csv')
labels = pd.read_csv('/content/Credit_card_label.csv')
```

Step 3: Data Preprocessing

We create a new column 'Approved' in the DataFrame by converting the 'label' column from the 'labels' DataFrame to integers.

```
dep = 'Approved'



df[dep] = labels.label.astype(int)



df.loc[df[dep] == 1, 'Status'] = 'Approved'

df.loc[df[dep] == 0, 'Status'] = 'Declined'
```

Step 4: Feature Engineering

We perform some feature engineering on the data, creating new columns 'Age', 'EmployedDaysOnly', and 'UnemployedDaysOnly' based on existing columns.

It converts categorical variables in the 'cats' list to numerical codes using pd.Categorical and fills missing values with the mode of each column.

```
cats = [

    'GENDER', 'Car_Owner', 'Propert_Owner', 'Type_Income',

    'EDUCATION', 'Marital_status', 'Housing_type', 'Mobile_phone',

    'Work_Phone', 'Phone', 'Type_Occupation', 'EMAIL_ID'

]
```

```
conts = [
    'CHILDREN', 'Family_Members', 'Annual_income',
    'Age', 'EmployedDaysOnly', 'UnemployedDaysOnly'
]
def proc_data():
    df['Age'] = -df.Birthday_count // 365
    df['EmployedDaysOnly'] = df.Employed_days.apply(lambda x: x if x > 0 else 0)
    df['UnemployedDaysOnly'] = df.Employed_days.apply(lambda x: abs(x) if x < 0 else 0)

    for cat in cats:
        df[cat] = pd.Categorical(df[cat])

    modes = df.mode().iloc[0]
    df.fillna(modes, inplace=True)

proc_data()
```

Step 5: Oversampling due to heavily skewed data and Data Splitting

```
X = df[cats + conts]
y = df[dep]
```

X_over, y_over = RandomOverSampler().fit_resample(X, y)

X_train, X_val, y_train, y_val = train_test_split(X_over, y_over, test_size=0.25)

Step 6: Encoding

The code applies the cat.codes method to each column specified in the cats list. The method is applicable to Pandas categorical data types and assigns a unique numerical code to each unique category in the categorical variable. The result is that the categorical variables are replaced with their corresponding numerical codes.

X_train[cats] = X_train[cats].apply(lambda x: x.cat.codes)

X_val[cats] = X_val[cats].apply(lambda x: x.cat.codes)

Step 7: Model Creation

Create a Sequential model using Keras. A Sequential model allows you to build a neural network by stacking layers one after another.

model = Sequential()

Step 8: Adding layers

Adding 10 dense layers to the model. Each dense layer has 10 units (neurons) and uses the sigmoid activation function. The first layer specifies input_dim=18, indicating that the input data has 18 features. This is the input layer. The last layer has a single neuron and uses sigmoid activation, making it suitable for binary classification tasks.

```
model.add(Dense(10, activation='sigmoid', input_dim=18))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(10, activation='sigmoid'))

model.add(Dense(1, activation='sigmoid'))
```

Step 9: Model Compilation

This step specifies the loss function, optimizer, and evaluation metrics.

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

Step 10: Model training

Train the model using the training data (X_train and y_train) for 100 epochs.The training history is stored in the history object, which contains information about the training process, including loss and accuracy at each epoch.

history = model.fit(X_train, y_train, epochs=100)

Output:

```
Epoch 1/100
65/65 [==============================] - 3s 3ms/step - loss: 0.7027 - accuracy: 0.5119
Epoch 2/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6936 - accuracy: 0.5119
Epoch 3/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6933 - accuracy: 0.5119
.
.
Epoch 97/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6930 - accuracy: 0.5119
Epoch 98/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6930 - accuracy: 0.5119
Epoch 99/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6932 - accuracy: 0.5119
Epoch 100/100
65/65 [==============================] - 0s 3ms/step - loss: 0.6929 - accuracy: 0.5119
```

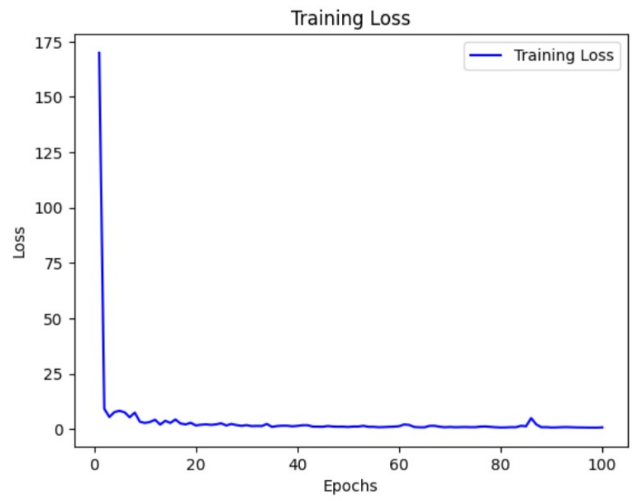
Step 11: Plotting the training loss

```
loss = history.history['loss']

epochs = range(1, len(loss) + 1)

plt.plot(epochs, loss, 'b', label='Training Loss')

plt.title('Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')
```

plt.legend()

plt.show()

Output:



Loss does not change much as gradient becomes too small

Solution for Vanishing Gradient Problem

Step 1: Scaling

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_val_scaled = scaler.transform(X_val)

Step 2: Modify the Model

1. Deeper Architecture: Augment model with more layers with increased numbers of neurons in each layer. Deeper architectures can capture more complex relationships in the data.

2. Early Stopping: Early stopping is implemented to monitor the validation loss. Training will stop if the validation loss does not improve for a certain number of epochs (defined by patience).

3. Increased Dropout: Dropout layers are added after each dense layer to help prevent overfitting.

4. Adjusting Learning Rate: The learning rate is set to 0.001. You can experiment with different learning rates.

```
model2 = Sequential()



model2.add(Dense(128, activation='relu', input_dim=18))

model2.add(Dropout(0.5))

model2.add(Dense(256, activation='relu'))

model2.add(Dropout(0.5))

model2.add(Dense(128, activation='relu'))

model2.add(Dropout(0.5))

model2.add(Dense(64, activation='relu'))

model2.add(Dropout(0.5))

model2.add(Dense(1, activation='sigmoid'))



early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)



model2.compile(loss='binary_crossentropy',                optimizer=Adam(learning_rate=0.001),
metrics=['accuracy'])
```

history2 = model2.fit(X_train_scaled, y_train, epochs=100, validation_data=(X_val_scaled, y_val), batch_size=32, callbacks=[early_stopping])

Output:

```
Epoch 1/100
65/65 [==============================] - 3s 8ms/step - loss: 0.7167 - accuracy: 0.5308
- val_loss: 0.6851 - val_accuracy: 0.5590
Epoch 2/100
65/65 [==============================] - 0s 5ms/step - loss: 0.6967 - accuracy: 0.5367
- val_loss: 0.6771 - val_accuracy: 0.6259
Epoch 3/100
65/65 [==============================] - 0s 5ms/step - loss: 0.6879 - accuracy: 0.5488
- val_loss: 0.6767 - val_accuracy: 0.5721
Epoch 4/100
65/65 [==============================] - 0s 5ms/step - loss: 0.6840 - accuracy: 0.5673
- val_loss: 0.6628 - val_accuracy: 0.6114
.
.
Epoch 96/100
65/65 [==============================] - 0s 7ms/step - loss: 0.1763 - accuracy: 0.9349
- val_loss: 0.1909 - val_accuracy: 0.9301
Epoch 97/100
65/65 [==============================] - 0s 7ms/step - loss: 0.1653 - accuracy: 0.9325
- val_loss: 0.1909 - val_accuracy: 0.9345
Epoch 98/100
65/65 [==============================] - 1s 8ms/step - loss: 0.1929 - accuracy: 0.9237
- val_loss: 0.1975 - val_accuracy: 0.9229
Epoch 99/100
65/65 [==============================] - 1s 9ms/step - loss: 0.1846 - accuracy: 0.9281
- val_loss: 0.1904 - val_accuracy: 0.9330
Epoch 100/100
65/65 [==============================] - 0s 7ms/step - loss: 0.1885 - accuracy: 0.9228
- val_loss: 0.1981 - val_accuracy: 0.9330
```

Evaluation metrics

predictions = model2.predict(X_val_scaled)

```
rounded_predictions = np.round(predictions)

report = classification_report(y_val, rounded_predictions)

print(f'Classification Report:\n{report}')
```

Output:

```
22/22 [==============================] - 0s 2ms/step
Classification Report:
              precision    recall  f1-score   support
           0       1.00      0.87      0.93       352
           1       0.88      1.00      0.94       335
    accuracy                           0.93       687
   macro avg       0.94      0.93      0.93       687
weighted avg       0.94      0.93      0.93       687
```

What is Exploding Gradient?

The exploding gradient problem is a challenge encountered during training deep neural networks. It occurs when the gradients of the network's loss function with respect to the weights (parameters) become excessively large.

Why Exploding Gradient Occurs?

The issue of exploding gradients arises when, during backpropagation, the derivatives or slopes of the neural network's layers grow progressively larger as we move backward. This is essentially the opposite of the vanishing gradient problem.

The root cause of this problem lies in the weights of the network, rather than the choice of activation function. High weight values lead to correspondingly high derivatives, causing significant deviations in new weight values from the previous ones. As a result, the gradient fails to converge and can lead to the network oscillating around local minima, making it challenging to reach the global minimum point.

In summary, exploding gradients occur when weight values lead to excessively large derivatives, making convergence difficult and potentially preventing the neural network from effectively learning and optimizing its parameters.

As we discussed earlier, the update for the weights during backpropagation in a neural network is given by:

ΔWi=−α⋅∂L∂WiΔWi=−α⋅∂Wi∂L

where,

- ΔWiΔWi : The change in the weight WiWi
- α: The learning rate, a hyperparameter that controls the step size of the update.
- L: The loss function that measures the error of the model.
- ∂L∂Wi∂Wi∂L: The partial derivative of the loss function with respect to the weight WiWi , which indicates the gradient of the loss function with respect to that weight.

The exploding gradient problem occurs when the gradients become very large during backpropagation. This is often the result of gradients greater than 1, leading to a rapid increase in values as you propagate them backward through the layers.

Mathematically, the update rule becomes problematic when∣∇Wi∣>1∣∇Wi∣>1, causing the weights to increase exponentially during training.

How can we identify the problem?

Identifying the presence of exploding gradients in deep neural network requires careful observation and analysis during training. Here are some key indicators:

- The loss function exhibits erratic behavior, oscillating wildly instead of steadily decreasing suggesting that the network weights are being updated excessively by large gradients, preventing smooth convergence.
- The training process encounters "NaN" (Not a Number) values in the loss function or other intermediate calculations..
- If network weights, during training exhibit significant and rapid increases in their values, it suggests the presence of exploding gradients.
- Tools like TensorBoard can be used to visualize the gradients flowing through the network.

How can we solve the issue?

- Gradient Clipping: It sets a maximum threshold for the magnitude of gradients during backpropagation. Any gradient exceeding the threshold is clipped to the threshold value, preventing it from growing unbounded.
- Batch Normalization: This technique normalizes the activations within each mini-batch, effectively scaling the gradients and reducing their variance. This helps prevent both vanishing and exploding gradients, improving stability and efficiency.

Build and train a model for Exploding Gradient Problem

We work on the same preprocessed data from the Vanishing gradient example but define a different neural network.

Step 1: Model creation and adding layers

```
model = Sequential()



model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0), input_dim=18))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(10, activation='tanh', kernel_initializer=random_normal(mean=0.0,
stddev=1.0)))

model.add(Dense(1, activation='sigmoid'))
```

Step 2: Model compiling

optimizer = SGD(learning_rate=1.0)

model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])

Step 3: Model training

history = model.fit(X_train, y_train, epochs=100)

Output:

```
Epoch                                                                    1/100
65/65 [==============================] - 2s 5ms/step - loss: 0.7919 - accuracy: 0.5032
Epoch                                                                    2/100
65/65 [==============================] - 0s 4ms/step - loss: 0.7440 - accuracy: 0.5017
.
.
Epoch                                                                   99/100
65/65 [==============================] - 0s 4ms/step - loss: 0.7022 - accuracy: 0.5085
Epoch                                                                  100/100
65/65 [==============================] - 0s 5ms/step - loss: 0.7037 - accuracy: 0.5061
```

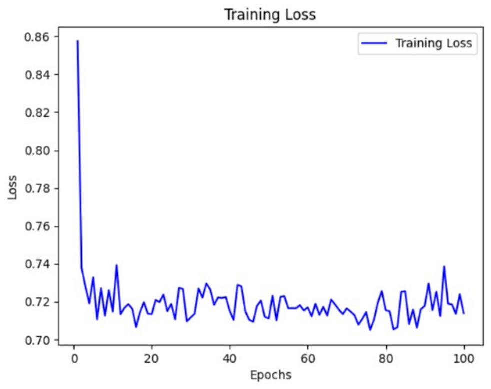
Step 4: Plotting training loss

loss = history.history['loss']

epochs = range(1, len(loss) + 1)

```
plt.plot(epochs, loss, 'b', label='Training Loss')

plt.title('Training Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()
```

Output:



It is observed that the loss does not converge and keeps fluctuating which shows we have encountered an exploding gradient problem.

Solution for Exploding Gradient Problem

Below methods can be used to modify the model:

1. Weight Initialization: The weight initialization is changed to 'glorot_uniform,' which is a commonly used initialization for neural networks.

2. Gradient Clipping: The clipnorm parameter in the Adam optimizer is set to 1.0, which performs gradient clipping. This helps prevent exploding gradients.

3. Kernel Constraint: The max_norm constraint is applied to the kernel weights of each layer with a maximum norm of 2.0. This further helps in preventing exploding gradients.

LSTM – Introduction and architecture:-

For college students studying deep learning, understanding **Long Short-Term Memory (LSTM)** networks is essential. LSTMs are a special and very powerful type of Recurrent Neural Network (RNN) specifically designed to overcome a major limitation of traditional RNNs: the **vanishing gradient problem**. They are the workhorse behind many modern AI applications that deal with sequential data, like language translation and speech recognition.

1. Introduction to LSTMs

Imagine trying to understand a very long sentence where the key information needed to interpret the end of the sentence was at the very beginning. Standard RNNs often "forget" information that occurred many steps ago. This inability to connect information over long sequences is due to the **vanishing gradient problem**.

Long Short-Term Memory (LSTM) networks were introduced to solve this. They are a special kind of RNN that can:

- **Learn long-term dependencies:** They can remember information for extended periods.
- **Mitigate vanishing gradients:** Their unique structure helps gradients flow more easily through many time steps.

LSTMs achieve this by incorporating a more complex internal mechanism than simple RNNs, primarily through what's called a **"cell state"** and several **"gates."**

2. Recap: Why Standard RNNs Struggle (The Vanishing Gradient Problem)

In a basic RNN, information is passed from one time step to the next via a single hidden state. When training RNNs with Backpropagation Through Time (BPTT), gradients are calculated by multiplying many derivatives together as they propagate backward through time.

- If these derivatives are small (which they often are with activation functions like tanh or sigmoid), repeatedly multiplying them makes the overall gradient for early time steps extremely tiny – it "vanishes."
- This means the weights associated with earlier parts of the sequence receive very small updates, and the network effectively "forgets" information that appeared long ago.

LSTMs are specifically engineered to prevent this forgetting.

3. The Core Idea of LSTMs: The Cell State (or Memory Cell)

The most important addition in an LSTM is the **Cell State (C_t)**. Think of it as a **"conveyor belt"** that runs straight through the entire chain of LSTM cells, carrying information across many time steps with minimal degradation.

- **Role:** The cell state acts as the "long-term memory" of the LSTM. It can store information for extended periods without it being lost or overwhelmed by new inputs.
- **Information Flow:** Unlike the hidden state in a vanilla RNN which is constantly being overwritten, the cell state allows information to flow mostly unchanged through the chain, only being modified by gates. This direct pathway is key to preventing vanishing gradients (often called the **Constant Error Carousel**).

4. LSTM Architecture: The Gates

The flow of information into and out of the cell state is controlled by three special structures called **"gates."** Each gate is essentially a small neural network (usually a sigmoid layer) that outputs numbers between 0 and 1. These outputs are then used to multiply (pointwise) information, effectively deciding how much of that information to "let through":

- A value close to **0** means "let nothing through" (block information).
- A value close to **1** means "let everything through" (allow information to pass).

Let's break down each gate:

a. Forget Gate (f_t)

- **Purpose:** This gate decides what information should be **thrown away** from the previous cell state (C_{t-1}). It determines which old memories are no longer relevant.
- **Inputs:** It takes the current input (x_t) and the hidden state from the previous time step (h_{t-1}).
- **Mechanism:** It's a sigmoid layer. $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- **Output:** A vector of numbers between 0 and 1.
- **Analogy:** Imagine a selective filter. If you're reading a sentence like "The **boy**, who was playing loudly outside, **ran** home.", once you process "playing loudly outside," the forget gate might decide to keep the "boy" (singular subject) information in memory, but perhaps deem the details of "playing loudly outside" less crucial for the main verb "ran," allowing them to be 'forgotten' from the long-term cell state.

b. Input Gate (i_t) and Candidate Cell State (C_tilde_t)

- **Purpose:** This gate decides what new information from the current input (x_t) should be **stored** in the cell state.
- **Mechanism (two parts):**
    1. **Input Gate Layer (i_t):** A sigmoid layer that decides which values will be updated (i.e., which new pieces of information are important enough to consider adding). $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
    2. **Candidate Cell State (C_tilde_t):** A tanh layer that creates a vector of new candidate values that could be added to the cell state. $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
- **Analogy:** The input gate acts like a switch deciding if new info gets added, and the tanh layer acts like a "measuring tape" determining what that new info actually is.

c. Update Cell State (C_t)

- **Purpose:** This is where the old memory and new information are combined to form the **new cell state** for the current time step (C_t).
- **Mechanism:**
  - The previous cell state (C_{t-1}) is multiplied by the forget gate's output (f_t). This "forgets" the irrelevant old information.
  - The candidate cell state (C_tilde_t) is multiplied by the input gate's output (i_t). This "adds" the relevant new information.
  - These two results are then summed together to get the new cell state C_t.
  - **Formula:** C_t = f_t * C_{t-1} + i_t * C_tilde_t
- **Analogy:** This is the "mixing bowl" where the relevant parts of the old memory are blended with the relevant parts of the new input to create the updated, long-term memory.

d. Output Gate (o_t)

- **Purpose:** This gate decides what information from the new cell state (C_t) should be **outputted** as the current hidden state (h_t). This h_t is also what is often passed on as the output of the LSTM cell at this time step, or fed into the next layer/task.
- **Inputs:** Similar to other gates, it takes h_{t-1} and x_t.
- **Mechanism:**
  1. A sigmoid layer decides which parts of the cell state are relevant for the current output. o_t = σ(W_o · [h_{t-1}, x_t] + b_o)
  2. The new cell state (C_t) is put through a tanh activation function (to scale its values between -1 and 1).
  3. These two results are then multiplied together to produce the new hidden state h_t.

  - **Formula:** h_t = o_t * tanh(C_t)
- **Analogy:** The "presenter" that looks at the updated long-term memory (C_t) and decides what specific bits of information are important to share as the current short-term summary (h_t).

5. Summary of LSTM Information Flow

At each time step t:

1. The LSTM receives the current input x_t, the previous hidden state h_{t-1}, and the previous cell state C_{t-1}.
2. The **Forget Gate** (f_t) decides what to discard from C_{t-1}.
3. The **Input Gate** (i_t) and **Candidate Cell State** (C_tilde_t) decide what new information to add.
4. The **New Cell State (C_t)** is calculated by combining the "forgotten" old information with the "new" relevant information.

5. The **Output Gate** (o_t) decides what part of the C_t should be revealed as the new hidden state h_t (which serves as the cell's output for this step and its short-term memory).

6. Why LSTMs Solve Vanishing Gradients

The magic of LSTMs in combating vanishing gradients lies primarily in the **Cell State (C_t)**:

- The cell state acts as a direct pathway (C_{t-1} to C_t) that allows gradients to flow across many time steps with minimal modification (via addition and element-wise multiplication by values that are not necessarily less than 1 consistently).
- The gates (especially the forget and input gates) learn to regulate the flow of information onto this cell state. They can learn to keep gradients around for a long time by setting forget gate values close to 1, effectively "protecting" the information and allowing the gradient signal to persist.
- This direct gradient path avoids the repeated multiplications of small derivatives that plague vanilla RNNs, ensuring that weights corresponding to even very early parts of the sequence can receive meaningful updates.

7. Applications of LSTMs

LSTMs have been incredibly successful in a wide range of sequential data tasks:

- **Machine Translation:** Powering the encoder-decoder models for language translation.
- **Speech Recognition:** Converting spoken words into text.
- **Text Summarization:** Generating concise summaries of longer documents.
- **Sentiment Analysis:** Determining the emotional tone of text.
- **Time Series Prediction:** Forecasting stock prices, weather patterns, etc.
- **Handwriting Recognition:** Converting handwritten text into digital form.

LSTM gates and operations:-

1. Introduction: The Power of Gates in LSTMs

As you know, LSTMs are a sophisticated type of Recurrent Neural Network designed to learn and remember information over long sequences, effectively solving the vanishing gradient problem. The secret to their success lies in a unique internal structure that includes a **"cell state"** (or memory cell) and three crucial **"gates."**

- **Cell State (C_t):** This is the core memory of the LSTM, acting like a horizontal "conveyor belt" that runs through the entire sequence. It's designed to carry information across many time steps with minimal degradation, preventing information loss and enabling gradients to flow more stably.

- **Gates:** These are like intelligent filters or switches that control how much information flows into, out of, and within the cell state. Each gate is essentially composed of two main parts:
  1. **A Sigmoid Neural Network Layer (σ):** This layer takes some inputs and outputs numbers between 0 and 1. This output determines how much of a particular piece of information should be "let through" the gate. (0 means "block completely," 1 means "allow completely," and values in between allow partial flow).
  2. **A Pointwise Multiplication Operation (*):** The output from the sigmoid layer is then multiplied element-wise with the data being regulated, effectively applying the gate's decision.

Let's delve into each of the three main gates and their operations within one LSTM cell at a specific time step t. Each gate takes the current input x_t and the previous hidden state h_{t-1} as its primary inputs.

2. The Three Gates and Their Operations

a. The Forget Gate (f_t)

- **Purpose:** This gate decides what information from the previous cell state (C_{t-1}) should be **thrown away** or "forgotten." It filters out irrelevant old memories that are no longer useful for the current context.
- **Inputs:**
  - x_t: The current input to the LSTM cell.
  - h_{t-1}: The hidden state (short-term memory/output) from the previous time step.
- **Operation (Formula):** $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
  - Here, W_f and b_f are learned weight matrices and bias vector for the forget gate.
  - [h_{t-1}, x_t] represents the concatenation of the previous hidden state and the current input.
  - σ is the sigmoid activation function.
- **Analogy:** Imagine you're writing a long story. The forget gate helps you decide which minor details from earlier chapters are no longer relevant to the current plot development and can be ignored.
  - **Example:** In a sentence like "The **boy**, who wore a blue hat, **was** happy.", when the LSTM processes "was", the forget gate might determine that the exact color of the hat is no longer critical for understanding the subject "boy" and can be partially forgotten from the long-term cell state, but the "boy" (singular) information must be retained.

b. The Input Gate (i_t) and Candidate Cell State (C_tilde_t)

- **Purpose:** This gate decides what **new information** from the current input (x_t) should be **stored** in the cell state. This is a two-step process:
  1. Deciding which values to update.
  2. Creating a candidate for the new values.

- **Inputs:**
  - x_t: The current input.
  - h_{t-1}: The previous hidden state.
- **Operation (Formulas):**
  1. **Input Gate Layer (i_t):** $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
     - This sigmoid layer decides which specific features of the new input (x_t) are important enough to consider adding to the cell state.
  2. **Candidate Cell State (C_tilde_t):** $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
     - This tanh layer creates a vector of new candidate values that could potentially be added to the cell state. tanh squashes values between -1 and 1, helping to normalize the candidate values.
- **Analogy:** If the forget gate is about "taking out the trash," the input gate is about "bringing in new groceries." The input gate (i_t) decides what slots in your pantry are available for new items, and the candidate cell state (C_tilde_t) represents the actual new items you're considering putting in.
  - **Example:** In "The man was **very happy**.", when processing "very", the input gate might decide to strongly incorporate the intensifying effect of "very" into the cell state, because it modifies the "happy" state.

c. Updating the Cell State (C_t)

- **Purpose:** This is the central operation of the LSTM. It combines the filtered old information (from the forget gate) with the relevant new information (from the input gate and candidate cell state) to create the **new, updated cell state** for the current time step. This new C_t then becomes C_{t-1} for the next time step.
- **Inputs:**
  - f_t: Output of the forget gate.
  - C_{t-1}: Previous cell state.
  - i_t: Output of the input gate.
  - C_tilde_t: Candidate cell state.
- **Operation (Formula):** $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
  - $f_t * C_{t-1}$: This part applies the forget gate's decision. Elements of C_{t-1} multiplied by a small f_t value are "forgotten," while those multiplied by a value close to 1 are "kept."
  - $i_t * \tilde{C}_t$: This part applies the input gate's decision. Elements of C_tilde_t multiplied by a small i_t value are mostly ignored, while those multiplied by a value close to 1 are strongly "added" to the cell state.
- **Analogy:** This is like managing your personal diary. You might cross out (forget) some old, irrelevant notes ($f_t * C_{t-1}$) and then write down (add) some important new experiences ($i_t * \tilde{C}_t$) to form your updated diary entry (C_t).

d. The Output Gate (o_t)

- **Purpose:** This gate decides what information from the newly updated cell state (C_t) should be **outputted** as the current hidden state (h_t). This h_t is the short-term memory passed to the next LSTM cell and also typically serves as the actual output of the LSTM cell at this time step (e.g., used for prediction).
- **Inputs:**
  - x_t: The current input.
  - h_{t-1}: The previous hidden state.
- **Operation (Formulas):**

  1. **Output Gate Layer (o_t):** o_t = σ(W_o · [h_{t-1}, x_t] + b_o)
     - This sigmoid layer decides which parts of the current cell state are relevant to output as the current hidden state.
  2. **Filtered Cell State:** tanh(C_t)
     - The cell state C_t is passed through a tanh activation function. This scales the values of the cell state to be between -1 and 1, making them suitable for the hidden state.
  3. **New Hidden State (h_t):** h_t = o_t * tanh(C_t)
     - The output gate's decision (o_t) is multiplied element-wise with the tanh(C_t) to produce the final h_t.
- **Analogy:** From your updated diary (C_t), you decide what information is currently important enough to share with others (h_t). You don't necessarily share everything in your diary, just the relevant bits.

  - **Example:** In a machine translation task, at a certain word, the output gate will extract only the necessary information from the C_t to help predict the next word in the target language.

3. Summary of LSTM Information Flow within one cell:

1. **Forget Gate (f_t):** Checks x_t and h_{t-1} to decide what to discard from C_{t-1}.
2. **Input Gate (i_t) & Candidate (C_tilde_t):** Check x_t and h_{t-1} to decide what new information to add to C_t.
3. **Update Cell State (C_t):** C_t is formed by combining the "kept" parts of C_{t-1} and the "added" parts of C_tilde_t. This is the core memory update.
4. **Output Gate (o_t):** Checks x_t and h_{t-1} to decide what parts of the updated C_t should be revealed as the new hidden state h_t.

4. Why This Gating Mechanism Works So Well

The intricate dance of these gates allows LSTMs to:

- **Selectively Remember/Forget:** They learn to dynamically decide what information is important to keep in the long-term memory (cell state) and what to discard.
- **Control Gradient Flow:** The additive nature of the cell state update (C_t = f_t * C_{t-1} + ...) and the gating mechanisms prevent gradients from vanishing or exploding by offering direct pathways for error signals to flow through many time steps without

repeated multiplications of small values. The cell state acts like a "highway" for gradients.

Comparison: LSTM vs. RNN:-

Differences: LSTM vs. RNN

Feature	Basic/Vanilla RNN	LSTM (Long Short-Term Memory)
1. Architecture / Internal Structure	Simpler. A single recurrent hidden layer/unit.	More complex. Features a dedicated **Cell State** and three controlling **Gates**.
2. Memory Mechanism	Information is stored solely in the **hidden state (h_t)**, which is constantly overwritten at each time step.	Has two forms of memory:
		- **Cell State (C_t):** Acts as a "long-term memory" or "conveyor belt," designed to carry information across many time steps with minimal degradation.
		- **Hidden State (h_t):** Serves as "short-term memory" and the output of the cell, derived from the cell state.
3. Handling Long-Term Dependencies	**Struggles significantly.** Prone to the **vanishing gradient problem**, causing it to "forget" information from earlier time steps in long sequences.	**Excels.** Specifically designed to overcome vanishing gradients. The gates and cell state enable it to **learn and remember information over very long durations.**
4. Gradient Flow	Gradients often **vanish** (become very small) as they propagate backward through many time steps, making it hard to update weights for early inputs.	Provides a more **stable gradient flow** due to the direct path of the cell state and the gating mechanisms, mitigating vanishing gradients (Constant Error Carousel).
5. Training Stability	Can be **unstable** to train for long sequences due to vanishing/exploding gradients.	Generally **more stable** and easier to train for tasks involving long sequences.
6. Computational Complexity	**Less complex.** Fewer parameters, making each time step computation faster.	**More complex.** More parameters (due to the gates and multiple linear transformations), making each time step computation slower.
7. Use Cases / Performance	Primarily for shorter sequences or simpler sequential tasks. Rarely used alone for complex real-world problems today.	**Widely used** and highly effective for demanding tasks requiring deep understanding of context over long sequences. Dominates in areas like

Feature	Basic/Vanilla RNN	LSTM (Long Short-Term Memory)
		machine translation, speech recognition, and complex text analysis.

Introduction to Autoencoders:=

Imagine distilling the essence of complex information into its most fundamental form, so efficiently that you can rebuild a close approximation of the original from this compact summary. This concept lies at the heart of **Autoencoders**, a compelling type of neural network primarily used for unsupervised learning. They operate on a simple yet profound principle: learning to reconstruct their own input.

What Exactly Are Autoencoders?

At its core, an Autoencoder is a neural network designed to achieve data compression and feature learning without explicit labels. Unlike supervised learning models that predict an output Y from an input X, an Autoencoder's goal is to produce an output X' that is as identical as possible to its input X. This seemingly simple task forces the network to discover and internalize the most significant underlying patterns and features within the data.

The "auto-" prefix is key here, signifying that the network learns to encode its own input. The objective isn't to classify or predict, but to create a valuable, condensed representation of the data.

The Purpose Behind Autoencoders

Autoencoders are versatile tools employed for several critical tasks in machine learning:

- **Dimensionality Reduction:** They excel at reducing the number of features in a dataset while retaining most of the important information. By compressing high-dimensional data into a low-dimensional "code" and then reconstructing it, an Autoencoder learns to discard redundant information while preserving the most vital aspects. This can be invaluable for data visualization or as a preparatory step for other models.
    - Consider: Taking a 28x28 pixel image (784 dimensions) and compressing it into a 10-dimensional vector, from which the original image can still be faithfully recreated.
- **Feature Learning (or Representation Learning):** Without needing human-labeled data, Autoencoders can automatically extract meaningful, abstract features from raw, unstructured data without explicit labels.

- **Data Denoising:** An Autoencoder can be trained to receive a corrupted (noisy) version of data and learn to output its clean, uncorrupted counterpart.
- **Anomaly Detection:** By learning to perfectly reconstruct "normal" data, an Autoencoder will struggle to reconstruct "anomalous" data points, resulting in a high reconstruction error that signals an outlier.

The Inner Workings: Autoencoder Architecture

An Autoencoder's design is elegantly symmetrical, typically comprising two interconnected parts: an **Encoder** and a **Decoder**, with a crucial **Bottleneck Layer** in between.

The Encoder

- **Role:** The encoder is the first half of the Autoencoder. Its mission is to ingest the raw input data and transform it into a compact, lower-dimensional representation.
- **Structure:** It's typically a series of interconnected neural network layers. These could be fully connected layers for generic data, or convolutional layers when dealing with image inputs.
- **Output:** The encoder's final output is known as the **Latent Space Representation** (also called the "code," "bottleneck features," or "encoded representation"). This is the compressed, essence-capturing summary of the input.

The Bottleneck Layer (Latent Space)

- **Role:** This is the nexus of the Autoencoder – the narrowest point in the network. The number of neurons in this layer is deliberately set to be significantly smaller than the input or output layers.
- **Significance:** This enforced constraint is what compels the encoder to learn a truly efficient representation. If this layer were too wide, the network might simply memorize the input without learning to extract meaningful features, bypassing the core objective of compression.

The Decoder

- **Role:** The decoder is the second half of the Autoencoder. Its task is to take the compressed latent space representation and faithfully reconstruct the original input data from it.
- **Structure:** It generally mirrors the encoder's structure but in reverse order. For instance, if the encoder has layers shrinking from 1000 to 500 to 100 neurons, the decoder would expand from 100 to 500 to 1000 neurons.
- **Output:** The decoder's output is the **reconstructed data** (let's call it X'), which should ideally be a near-perfect replica of the initial input X.

How Autoencoders Learn: The Training Process

Autoencoders are trained using an unsupervised learning paradigm, which means they don't require external labels:

- **Self-Supervision:** The input data X serves as its own target output X'. The network learns by trying to accurately reproduce what it sees.
- **Reconstruction Loss:** During training, the core objective is to minimize the **reconstruction loss** (or reconstruction error). This metric quantifies the dissimilarity between the original input X and its reconstructed counterpart X'.
    - **Common Loss Functions:**
        - **Mean Squared Error (MSE):** Frequently used for continuous data, such as pixel values in images. Loss = $\Sigma(X - X')^2$
        - **Binary Cross-Entropy:** Often applied when dealing with binary data or pixel values normalized between 0 and 1.
- **Optimization:** Standard optimization techniques, like gradient descent and backpropagation, are employed to adjust the weights and biases of both the encoder and decoder. The goal is always to reduce the reconstruction loss.
- **Feature Extraction:** By continuously forcing the network to recreate the input from a constrained, smaller representation and adjusting its weights based on the reconstruction error, the Autoencoder intrinsically learns to capture only the most crucial and discriminative features within that compressed latent space.

A Simple Analogy: Data Zipping

Think of an Autoencoder like an intelligent "zip" and "unzip" utility for your data:

- You feed it a large file (your input data).
- The **encoder** acts as the "zipping" component, compressing that large file into a much smaller archive (your latent space representation).
- The **decoder** functions as the "unzipping" component, attempting to reconstruct the original large file from that smaller archive.
- If the unzipped file is nearly identical to the original, it implies that the zipping process (the encoding) was highly effective and preserved all essential information. The Autoencoder, through iterative training, becomes proficient at both compressing and decompressing to minimize any discrepancies.

Common Variants and Applications

Beyond the basic Autoencoder, several specialized types exist for diverse applications:

- **Denoising Autoencoders (DAE):** Trained to recover a clean input from a deliberately corrupted or noisy version.
- **Variational Autoencoders (VAEs):** A more advanced type that learns the probability distribution of the latent space, enabling the generation of entirely new data that resembles the training set.

- **Sparse Autoencoders:** Introduce a penalty during training to encourage only a small subset of neurons in the latent layer to be active, promoting more focused feature extraction.
- **Deep Autoencoders:** Simply autoencoders with multiple hidden layers within both their encoder and decoder sections.

Autoencoders are invaluable tools across various AI domains:

- **Dimensionality Reduction:** For tasks like visualizing high-dimensional data or reducing computational burden for subsequent models.
- **Feature Learning:** Automatically discovering rich, abstract features from raw data for improved model performance.
- **Anomaly Detection:** Identifying outliers in datasets, as anomalous inputs result in high reconstruction errors.
- **Data Denoising:** Cleaning up noisy images, audio, or other data streams.
- **Generative Models (with VAEs):** Creating novel data instances, such as new images or text sequences.

Undercomplete Autoencoders:-

An undercomplete autoencoder is a type of autoencoder of aims to learn a compressed representation of its input data. It is termed "undercomplete" because it forces the representation to have a lower dimensionality than the input itself, thereby learning to capture only the most essential features.

How Undercomplete Autoencoders Work

The operation of an undercomplete autoencoder involves several key steps:

1. **Compression**: The encoder processes the input data to form a condensed representation, focusing on the most significant attributes of the data.

2. **Reconstruction**: The decoder then attempts to reconstruct the original data from this compressed form, aiming to minimize discrepancies between the original and reconstructed data.

3. **Optimization**: Through iterative training and backpropagation, the network optimizes the weights and biases to reduce the reconstruction error, refining the model's ability to compress and reconstruct data accurately.

Difference between Vanilla Autoencoders and Undercomplete Autoencoders

1. **Dimensionality Reduction**: An undercomplete autoencoder specifically focuses on reducing the dimensionality of the input data, while a vanilla autoencoder does not necessarily do so.

2. **Bottleneck Layer Size**: In an undercomplete autoencoder, the bottleneck layer has fewer neurons than the input layer, enforcing a compressed representation. In contrast, a vanilla autoencoder might have a bottleneck layer with an equal or greater number of neurons compared to the input layer.

3. **Generalization**: Undercomplete autoencoders, by reducing dimensionality, often capture the most important features of the data, leading to better generalization. Vanilla autoencoders, without a reduced bottleneck, might overfit the data, capturing noise as well as signal.

Example:

- **Vanilla Autoencoder**: Input size = 784, Hidden layer 1 size = 128, Bottleneck size = 64, Output size = 784.

- **Undercomplete Autoencoder**: Input size = 784, Hidden layer 1 size = 128, Bottleneck size = 32, Output size = 784.

Why the "Undercomplete" Constraint Matters

The intentional design choice to make the latent space smaller than the input is crucial for the Autoencoder's effectiveness:

- **Forced Compression:** The network is explicitly forced to compress the input information. It cannot simply copy the input features directly through the bottleneck. To reconstruct the input accurately from fewer dimensions, the encoder must learn to identify and retain only the most critical information.
- **Meaningful Feature Extraction:** This compression constraint compels the Autoencoder to extract salient, non-redundant features. It learns to capture the underlying structure and patterns within the data that are essential for its reconstruction. If the latent space were too large, the network might just learn an identity function, essentially memorizing the input without truly understanding or compressing it.
- **Preventing Trivial Solutions:** Without the undercomplete bottleneck, an Autoencoder could trivially learn an "identity function" – mapping inputs directly to outputs without any meaningful compression or feature learning. The undercomplete nature ensures the network performs actual dimensionality reduction.

Regularized Autoencoders:-

Autoencoders, at their core, learn to compress data into a latent representation and then reconstruct it. While **Undercomplete Autoencoders** achieve this by simply making the latent space smaller than the input, this isn't always sufficient to guarantee the learning of truly valuable, robust, or non-trivial representations. This is where **regularization** comes into play.

Regularization in Autoencoders refers to the technique of adding extra constraints or penalties to the learning objective, beyond just minimizing the reconstruction error. These added terms or architectural modifications force the Autoencoder to learn more meaningful, useful, or robust representations, especially when faced with complex data or the risk of overfitting.

Primary Purposes of Regularization:

- **Prevent Overfitting:** Encourage the model to generalize well to unseen data, rather than just memorizing the training examples.
- **Encourage Better Feature Learning:** Guide the network to discover more robust, disentangled, or sparse features within the data.
- **Achieve Specific Goals:** Tailor the Autoencoder for tasks like denoising or creating interpretable representations.

Let's explore some of the most common types of regularized autoencoders.

Common Types of Regularized Autoencoders

a. Sparse Autoencoders

- **Problem Addressed:** Even with an undercomplete bottleneck, all neurons in the latent layer might be active for most inputs. Or, if the latent space is not undercomplete (i.e., it's equal to or larger than the input dimension), the Autoencoder could easily learn a trivial identity function. Sparse Autoencoders aim to make the learned representations more efficient and interpretable.
- **Core Idea:** Encourage **sparsity** in the latent (bottleneck) layer. This means ensuring that for any given input, only a small subset of neurons in the latent layer are active (i.e., have non-zero or significantly non-zero activations), while the majority remain inactive or close to zero.
- **How it's Achieved:** A **sparsity penalty** term is added to the standard reconstruction loss function.
    - **Kullback-Leibler (KL) Divergence:** A common method is to penalize the difference between the desired low activation probability (e.g., 0.05) and the actual average activation of each neuron in the latent layer. This penalty pushes the average activation of latent neurons towards the desired low value.
    - **L1 Regularization:** Less common for direct sparsity of activations, but applying an L1 penalty to the weights can encourage some weights to become zero, leading to a simpler model. For activations, it directly penalizes the sum of absolute values of activations.
- **Benefit:**

- ◦ **Specialized Features:** Each active neuron in the sparse latent code tends to become specialized in detecting a very specific feature in the input.
    - **Reduced Overfitting:** By limiting the network's effective capacity (only a few neurons are active), it helps prevent overfitting.
    - **Interpretability:** The sparse codes can sometimes offer more interpretable representations of the data.
- **Use Case:** Learning meaningful features from unlabeled data, particularly useful when the latent dimension might be large or even overcomplete, as it still forces the learning of compact representations.

b. Denoising Autoencoders (DAE)

- **Problem Addressed:** A basic Autoencoder might learn to simply map noisy input directly to noisy output if the input is slightly corrupted, or it might become overly sensitive to minor variations. It could also learn an identity function too easily if the data is too clean.
- **Core Idea:** Make the Autoencoder robust to input corruption by training it to reconstruct the original, clean, uncorrupted input from a deliberately corrupted (noisy) version of that input.
- **How it's Achieved:**
    - During training, the original input X is first intentionally corrupted to create a noisy version X_noisy. Common corruption methods include:
        - **Adding Gaussian noise:** Random values drawn from a Gaussian distribution are added to the input.
        - **Masking noise (Dropout):** Randomly setting a percentage of input features (e.g., pixels) to zero.
        - **Salt-and-pepper noise:** Randomly setting some features to their minimum or maximum value.
    - The **corrupted input X_noisy is fed to the encoder.**
    - The **decoder is trained to reconstruct the original, uncorrupted input X.**
- **Benefit:**
    - **Robust Feature Learning:** The network is forced to learn more robust and essential features that capture the true underlying structure of the data, rather than just memorizing noise or minor variations.
    - **Improved Generalization:** By learning to "undo" the corruption, the DAE becomes better at extracting meaningful representations from real-world, imperfect data.
    - **Effective Denoising:** The trained decoder can then be used specifically to clean up noisy data.
- **Use Case:** Denoising images, audio, or other data, and learning robust features for downstream tasks.

c. Contractive Autoencoders (CAE)

- **Core Idea:** Encourage the Autoencoder to learn representations that are **robust to small perturbations** in the input space. It's about ensuring that a small change in the input doesn't lead to a large change in the latent representation.
- **How it's Achieved:** An additional penalty term is added to the reconstruction loss. This penalty is proportional to the Frobenius norm of the Jacobian matrix of the encoder's hidden layer activations with respect to the input. In simpler terms, it measures how sensitive the encoded representation is to tiny variations in the input.
- **Benefit:** Promotes the learning of highly stable and robust features, useful for situations where input variations are common.
- **Use Case:** Learning robust features, particularly in domains sensitive to minor input fluctuations.

Why Regularization is Crucial

Regularization techniques are vital for Autoencoders for several reasons:

- **Enhanced Generalization:** By preventing overfitting, regularization ensures that the Autoencoder performs well not just on the data it was trained on, but also on new, unseen data.
- **Improved Robustness:** Regularized Autoencoders learn representations that are less sensitive to noise, missing data, or minor variations in the input.
- **Superior Feature Learning:** They compel the network to extract more meaningful, fundamental, and sometimes more interpretable features from the data, going beyond simple compression.
- **Preventing Trivial Learning:** Especially when the latent space is not strictly undercomplete, regularization is essential to prevent the Autoencoder from learning a simple identity mapping that offers no real insight or compression.

Stochastic and Contractive Autoencoders:-

Autoencoders are neural networks that learn to compress data and then reconstruct it, effectively discovering latent representations. While **Undercomplete Autoencoders** achieve compression by forcing data through a narrow bottleneck, the learned representations can sometimes be overly sensitive to minor input variations or might not generalize well. This is where **Regularized Autoencoders** come in, adding extra constraints to encourage more robust, meaningful, or even generative representations. Among these, **Stochastic Autoencoders** (with Variational Autoencoders as the prime example) and **Contractive Autoencoders** offer distinct approaches to enhancing learning.

Regularization: The Guiding Hand

Regularization in Autoencoders involves adding specific penalties to the standard reconstruction loss or introducing architectural modifications that guide the learning process. The aim is to:

- Prevent the network from overfitting to the training data.
- Force the discovery of more useful, stable, or interpretable features.
- Enable new capabilities, such as data generation.

Let's delve into Stochastic and Contractive Autoencoders.

1. Stochastic Autoencoders (Focusing on Variational Autoencoders - VAEs)

While the term "Stochastic Autoencoder" can encompass various models that introduce randomness, the most prominent and impactful example is the **Variational Autoencoder (VAE)**. VAEs fundamentally differ from traditional autoencoders in their approach to the latent space.

- **Core Idea:** Instead of mapping an input to a single, fixed point in the latent space, a VAE's encoder maps the input to a **probability distribution** over the latent space. This means the latent code for a given input is not a single vector, but a distribution from which a vector can be sampled.
- **How it Works:**
    1. **Encoder (Q(z|X)):** The encoder takes an input X and, instead of outputting a single latent vector z, it outputs the parameters (typically the mean μ and log-variance $\log \sigma^2$) of a probability distribution (e.g., a Gaussian distribution) in the latent space.
    2. **Sampling (z from distribution):** During the forward pass, a latent vector z is **sampled** from this learned distribution. This sampling process introduces stochasticity. To allow backpropagation, a "reparameterization trick" is often used (e.g., $z = \mu + \exp(0.5 * \log \sigma^2) * \varepsilon$, where ε is sampled from a standard Gaussian).
    3. **Decoder (P(X|z)):** The decoder then takes this sampled latent vector z and attempts to reconstruct the original input X.
- **Loss Function:** A VAE's objective function has two components:

    1. **Reconstruction Loss:** Measures how well the decoder reconstructs the input from the sampled latent code. (e.g., Mean Squared Error or Binary Cross-Entropy).
    2. **KL Divergence Loss:** A regularization term that measures the difference between the learned latent distribution (from the encoder) and a simple, predefined prior distribution (e.g., a standard normal Gaussian). This term encourages the latent space to be well-structured, continuous, and "smooth," preventing isolated clusters and making interpolation possible.
- **Why "Stochastic"?** The random sampling of z from the learned latent distribution introduces randomness into the encoding-decoding process, making it stochastic.
- **Benefits:**

    - **Generative Capabilities:** Because the latent space is a continuous probability distribution, you can sample new z vectors from the prior distribution and feed them to the decoder to generate entirely new data instances that resemble the training data.

    - **Smoother Latent Space:** The KL divergence regularization ensures that similar inputs map to similar distributions in the latent space, leading to a more organized and interpretable latent representation.
    - **Robustness:** The stochastic nature helps the model learn more robust features by requiring it to reconstruct inputs from slightly varied latent codes.
- **Use Cases:** Data generation (images, text, audio), anomaly detection (as outliers will likely fall into low-probability regions of the latent space), and learning disentangled representations.

2. Contractive Autoencoders (CAE)

- **Problem Addressed:** Standard Autoencoders, even undercomplete ones, can sometimes be overly sensitive to minor, irrelevant variations in the input data. This means a tiny change in the input could lead to a drastically different latent representation, making the features less stable or robust.
- **Core Idea:** Encourage the Autoencoder to learn a representation that is **robust and locally insensitive to small perturbations** in the input data. In simpler terms, if the input changes only slightly, the learned latent code should remain largely unchanged.
- **How it's Achieved:** A **contractive penalty** term is added to the standard reconstruction loss.
    - This penalty is based on the **Jacobian matrix** of the encoder's hidden layer activations with respect to the input. The Jacobian essentially measures how much the output of the encoder changes for a small change in the input.
    - The **Frobenius norm** of this Jacobian matrix is minimized. By minimizing this norm, the CAE forces the encoder to have a "contractive" mapping – meaning it squashes small variations in the input, leading to a more stable latent representation.
- **Loss Function:** Total Loss = Reconstruction Loss + $\lambda * ||J||\_F^2$
    - Where J is the Jacobian matrix of the encoder's hidden layer outputs with respect to the input, and λ is a regularization strength parameter.
- **Benefits:**
    - **Highly Robust Features:** The learned features are more stable and invariant to small input variations or noise.
    - **Improved Generalization:** By preventing over-sensitivity to minor details in the input, the CAE generalizes better to unseen data.
    - **Denser Representations:** Unlike sparse autoencoders, CAEs tend to produce dense (non-sparse) representations where all latent neurons contribute to the representation.
- **Use Cases:** Learning highly robust feature representations, particularly in domains where input data might have slight variations or noise that shouldn't alter the core features.

Comparison and Importance

Both Stochastic (like VAEs) and Contractive Autoencoders are types of regularized Autoencoders aimed at learning more valuable and robust representations, but they achieve this through different mechanisms:

- **Stochastic (VAEs):** Introduces randomness into the encoding process by learning a distribution over the latent space. Its primary strength lies in its generative capabilities and creating a smooth, continuous latent space.
- **Contractive:** Focuses on making the mapping from input to latent space locally insensitive by penalizing the encoder's derivatives. It aims for robustness against small input perturbations, rather than introducing stochasticity itself.