DL-Cheat-Codes (/github/nikitaprasad21/DL-Cheat-Codes/tree/main)
/ RNN-Models (/github/nikitaprasad21/DL-Cheat-Codes/tree/main/RNN-Models)

What is RNN?

A recurrent neural network (RNN) is a deep learning model that is trained to process and convert a sequential data input into a specific sequential data output.

What is Seqential Data?

Sequential data is data—such as words, sentences, audio (Speech) or time-series data—where sequential components are interrelate based on complex semantics and syntax rules.

What are the types of recurrent neural networks?

The following are several common RNN types.

- **One-to-Many RNNs**: In this architecture, the RNN receives a single input at the beginning (the "one" part), and then it generates a sequence of outputs (the "many" part).
    - This can be useful for tasks such as image captioning, where the input is an image, and the output is a sequence of words describing the content of the image.
    - **Example**: *In image captioning, the RNN receives an image as input and generates a sequence of words to describe the contents of the image.*
- **Many-to-One RNNs**: Conversely, in many-to-one RNNs, the network receives a sequence of inputs (the "many" part) and produces a single output (the "one" part).
    - This setup is commonly used for tasks like sentiment analysis, where the input is a sequence of words, and the output is a sentiment label (positive or negative).
    - **Example**: *In sentiment analysis, the RNN receives a sequence of words representing a movie review and predicts whether the sentiment is positive, negative, and neutral from input reviews*
- **Many-to-Many RNNs (Sequence-to-Sequence)**: In this architecture, the RNN takes a sequence of inputs and produces a sequence of outputs, can be of variable lengths.
    - This setup is used for tasks such as machine translation, where the input is a sequence of words in one language, and the output is a sequence of words in another language.
    - **Example**: *In machine translation, the RNN receives a sequence of words in English and generates a sequence of words in French.*

- **Many-to-Many (Synced) RNNs**: In this variant of many-to-many RNNs, the input and output sequences have the same length.
    - This architecture is used for tasks like named entity recognition (NER), where the input is a sequence of words, and the output is a sequence of labels indicating named entities.
    - **Example**: *In named entity recognition, the RNN receives a sequence of words and generates a sequence of labels indicating whether each word is part of a named entity (e.g., person, organization, location).*

What about One-to-One?

While the "one-to-one" architecture is straightforward (each input is processed independently, and there are no recurrent connections or feedback loops.) and suitable for certain types of tasks, it doesn't leverage the sequential nature of data or capture temporal dependencies, which are essential for tasks like sequence prediction, time series forecasting, or natural language processing.

- This architecture is used for tasks like simple classification tasks, regression tasks, or tasks where each input represents a standalone instance.
- **Example**: *Suppose you have a dataset of images, and each image is associated with a single label indicating the object in the image. You can train a "one-to-one" neural network to classify each image into its respective category without considering any sequential or temporal information.*

How RNN Model Works?

Unlike feedforward neural networks, which process each input independently,

RNNs maintain an **internal state (memory)** that allows them to *capture temporal dependencies and patterns in sequential data*.

- **Sequential Input**: RNNs take sequential input data, such as text, time series, or audio, where the order of the elements matters. Each element in the sequence is typically represented as a vector or a sequence of feature vectors.
- **Recurrent Connections**: RNNs have recurrent connections that allow information to persist over time steps. At each time step, the RNN processes an input vector and updates its internal state based on both the current input and the previous state.
- **Hidden State Update**: At each time step t, the RNN computes a new hidden state

$h\ t$ using the current input $x\ t$ and the previous hidden state $h\ t\ -\ 1$. This hidden state serves as a summary of the information processed so far and is passed to the next time step.

$$h\,t = f(W\,h \cdot h\,t{-}1 + W\,x \cdot x\,t + b)$$

where:

- h

t is the hidden state at time step t

- f is the activation function (e.g., tanh or ReLU)
- W

h is the weight matrix for the recurrent connections

- W

x is the weight matrix for the input connections

- b is the bias vector
- **Output**: Depending on the task, the RNN may produce an output at each time step (sequence-to-sequence) or only at the final time step (sequence-to-vector). The output at each time step t is computed based on the current hidden state

$h\,t$.

- **Training**: RNNs are typically trained using backpropagation through time (BPTT), which is an extension of the standard backpropagation algorithm. BPTT computes the gradients of the loss function with respect to the model parameters (weights and biases) by unrolling the network through time and applying the chain rule.

How to convert words into vectors?

In Recurrent Neural Networks (RNNs), integer encoding and word embedding are two common techniques used to represent words as vectors before feeding them into the model.

In [ ]:
```
from keras.datasets import imdb
from keras import Sequential
from keras.layers import Dense,SimpleRNN
```

In [ ]:
```
(train_input,train_target),(test_input,test_target) = imdb.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/imd
b.npz
17464789/17464789 [==============================] - 0s 0us/step
```

In [ ]:
```
train_input[0]
```

In [ ]:
```
train_input = pad_sequences(train_input,padding='post',maxlen=100)
test_input = pad_sequences(test_input,padding='post',maxlen=100)
```

Integer Encoding:

Integer encoding involves assigning a unique integer index to each word in the vocabulary.

Each word in the input text is replaced by its corresponding integer index.

Integer encoding is a simple and efficient way to represent words as numerical values. However, it does not capture the semantic relationships between words.

Integer-encoded sequences are typically one-hot encoded before being fed into the RNN.

In [ ]:
```
model = Sequential()

model.add(SimpleRNN(32, input_shape=(100,1), return_sequences=False))
model.add(Dense(1,activation='sigmoid'))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 32)                1088

 dense (Dense)               (None, 1)                 33

=================================================================
Total params: 1121 (4.38 KB)
Trainable params: 1121 (4.38 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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

model.fit(train_input,train_target,epochs=5,validation_data=(test_input,test_target))
```

```
Epoch 1/5
782/782 [==============================] - 52s 64ms/step - loss: 0.6946 - accuracy: 0.
5040 - val_loss: 0.6925 - val_accuracy: 0.5109
Epoch 2/5
782/782 [==============================] - 48s 61ms/step - loss: 0.6933 - accuracy: 0.
5101 - val_loss: 0.6924 - val_accuracy: 0.5137
Epoch 3/5
782/782 [==============================] - 48s 62ms/step - loss: 0.6930 - accuracy: 0.
5113 - val_loss: 0.6943 - val_accuracy: 0.5051
Epoch 4/5
782/782 [==============================] - 48s 61ms/step - loss: 0.6928 - accuracy: 0.
5116 - val_loss: 0.6946 - val_accuracy: 0.5109
Epoch 5/5
782/782 [==============================] - 48s 62ms/step - loss: 0.6927 - accuracy: 0.
5111 - val_loss: 0.6939 - val_accuracy: 0.4997
```

Out[ ]: <keras.src.callbacks.History at 0x7f0bcd35dd50>

Word Embedding:

Word embedding is a dense vector representation of words in a high-dimensional space, during the training process to capture semantic relationships between words. It is capable of capturing context and meaning from the surrounding words in a sequence.

Note: Words with similar meanings are represented by similar vectors in the embedding space.

In [ ]:
```
from keras.layers import Embedding
```

`embeddings_initializer='uniform'` is an argument passed to the Embedding layer in the Keras model. It specifies the method used to initialize the embedding weights, which are the vectors that represent each word in the vocabulary.

The 'uniform' initializer randomly initializes the weights within a uniform distribution. This means that each weight is sampled from a uniform distribution between a lower and upper bound. The default lower and upper bounds are -1 and 1, respectively.

-- It is often used when there is no prior knowledge about the relationships between the words in the vocabulary.

In [ ]:
```
model_1 = Sequential()

model_1.add(Embedding(10000, 2, input_length=100, embeddings_initializer='uniform'))
model_1.add(SimpleRNN(32,return_sequences=False))
model_1.add(Dense(1, activation='sigmoid'))

model_1.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_1 (Embedding)     (None, 100, 2)            20000

 simple_rnn_1 (SimpleRNN)    (None, 32)                1120

 dense_1 (Dense)             (None, 1)                 33

=================================================================
Total params: 21153 (82.63 KB)
Trainable params: 21153 (82.63 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In [ ]:
```
model_1.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
history = model_1.fit(train_input,train_target,epochs=5,validation_data=(test_input,tes
```

```
Epoch 1/5
782/782 [==============================] - 101s 125ms/step - loss: 0.6826 - acc: 0.564
8 - val_loss: 0.6364 - val_acc: 0.6627
Epoch 2/5
782/782 [==============================] - 72s 93ms/step - loss: 0.5210 - acc: 0.7473
- val_loss: 0.4240 - val_acc: 0.8092
Epoch 3/5
782/782 [==============================] - 75s 96ms/step - loss: 0.3712 - acc: 0.8450
- val_loss: 0.4197 - val_acc: 0.8237
Epoch 4/5
782/782 [==============================] - 79s 102ms/step - loss: 0.2892 - acc: 0.8853
- val_loss: 0.4265 - val_acc: 0.8312
Epoch 5/5
782/782 [==============================] - 76s 97ms/step - loss: 0.2481 - acc: 0.9067
- val_loss: 0.4215 - val_acc: 0.8344
```

Little overfitting but better accuracy than Interger Encoding Vectors.

Word embeddings are more expressive than integer encoding and are widely used in natural language processing, NLP tasks.

Applications of RNN

RNNs are well-suited for tasks involving sequential data, such as natural language processing (e.g., language modeling, machine translation, sentiment analysis), time series analysis (e.g., stock price prediction, weather forecasting), and speech recognition.

What are the limitations of recurrent neural networks?

- **Exploding gradient**

Gradient exploding refers to the phenomenon where gradients become extremely large during the backpropagation process, causing the weights of the network to update by large amounts.

In RNNs, gradient exploding often occurs when the gradients are backpropagated through many time steps, causing them to accumulate and grow exponentially.

This can lead to unstable training behavior and make it difficult for the model to converge to a good solution.

- **Vanishing gradient**

The vanishing gradient problem is a condition where the model's gradient approaches zero in training. When the gradient vanishes, the RNN fails to learn effectively from the training data, resulting in underfitting.

An underfit model can't perform well in real-life applications because its weights weren't adjusted appropriately. RNNs are at risk of vanishing and exploding gradient issues when they process long data sequences.

- **Slow training time**

In RNNs, slow training can be caused by various factors, including gradient exploding or vanishing gradients, which can make it difficult for the model to learn effectively from the training data.

Additionally, the computational complexity of training RNNs, especially when processing long sequences or large datasets, can contribute to slow training times.

For example, an RNN model can analyze a user's sentiment from a couple of sentences. However, it requires massive computing power, memory space, and time to summarize a page of an essay.

What are some variants of recurrent neural network architecture?

1. **Bidirectional Recurrent Neural Networks (BRNN)**

This architecture processes data sequences with forward and backward layers of hidden nodes.

The forward layer works similarly to the RNN, which stores the previous input in the hidden state and uses it to predict the subsequent output.

Meanwhile, the backward layer works in the opposite direction by taking both the current input and the future hidden state to update the present hidden state.

Combining both layers enables the BRNN to improve prediction accuracy by considering past and future contexts.

Advantage: Improves prediction accuracy by considering past and future contexts.

Example: Predicting the word "enjoy" in the sentence "You enjoy data science."

2. **Long Short-Term Memory (LSTM) Networks**

This RNN variant enables the model to expand its memory capacity to accommodate a longer timeline.

Whereas, RNN can only remember the immediate past input. It can't use inputs from several previous sequences to improve its prediction.

Advantage: Enables the model to remember dependencies over longer sequences.

Example: "You love data science. Your favorite topic is RNN.", the LSTM might recognize the relationship between "data science" and "RNN" as related topics.

3. **Gated Recurrent Units (GRU)**

It is an RNN that enables selective memory retention. The model adds an update and forgets the gate to its hidden layer, which can store or remove information in the memory.

Advantage: Provides a balance between memory retention and computational efficiency. Selectively remembering important information while discarding irrelevant details.

How do transformers overcome the limitations of recurrent neural networks?

Transformers are deep learning models that use self-attention mechanisms in an encoder-decoder feed-forward neural network. They can process sequential data the same way that RNNs do.

- **Self-attention**

Transformers don't use hidden states to capture the interdependencies of data sequences. Instead, they use a self-attention head to process data sequences in parallel.

This enables transformers to train and process longer sequences in less time than an RNN does. With the self-attention mechanism, transformers overcome the memory limitations and sequence interdependencies that RNNs face.

Transformers can process data sequences in parallel and use positional encoding to remember how each input relates to others.

- **Parallelism**

Transformers solve the gradient issues that RNNs face by enabling parallelism during training. By processing all input sequences simultaneously, a transformer isn't subjected to backpropagation restrictions because gradients can flow freely to all weights.

They are also optimized for parallel computing, which graphic processing units (GPUs) offer for generative AI developments. Parallelism enables transformers to scale massively and handle complex NLP tasks by building larger models.

In [ ]: