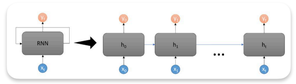
RNN (Recurrent Neural Network)



Recurrent Neural Network (RNN)

Definition: A deep learning architecture that improves the performance of neural networks on current and future inputs by using past information. It is specialized for processing sequence data.

RNNs have a characteristic where the output from the hidden layer nodes, after passing through an activation function, is **sent towards the output layer while also being used as input for the next calculation of the hidden layer nodes themselves.**

Key Features:

- **Hidden State** :
  - RNNs have a vector called the hidden state, which acts as memory. The hidden state stores information from the previous time step and is used to calculate the hidden state of the next time step along with the current time step's input.
- **Recurrent Structure:**
  - RNNs have a loop structure. Through this loop structure, the neural network stores past information in the hidden state and performs operations sequentially on each element of the sequence data.
- **Sequence Data Processing**
  - RNNs can handle the variable length of sequence data. They receive input at each time step, update the hidden state, and generate output as needed.

How it Works

1. **Input** :The current time step's input x_t *is* input to the hidden layer nodes.
2. **Hidden layer calculation** : The hidden layer nodes use the current time step's input x_t *a*nd the previous time step's hidden state h_{t-1} to calculate through an activation function.

- $h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$

3. Output: The result ht of the hidden layer is passed to the output layer and used to generate the final output y_t

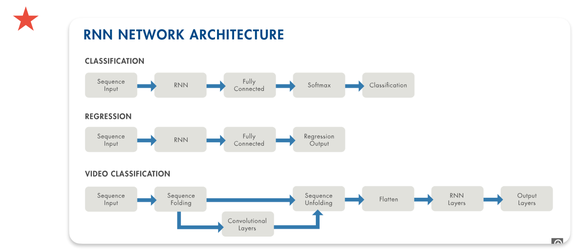
$$y_t = g(W_{hy}h_t + b_y)$$

y_t

This computation typically involves an activation function (e.g., ReLU, tanh). The core formula is something like: *ht = activation(Wxh * xt + Whh * ht-1 + b)*, where *Wxh* is the weight matrix for the input, *Whh* is the weight matrix for the previous hidden state, and *b* is the bias.

4. Recurrence

The key to RNNs! The newly calculated hidden state, h_t, is *also* fed back as input to the hidden layer for the *next* time step. This allows the network to maintain information about the past inputs in the sequence and use it to influence the processing of future inputs.



Advantages:

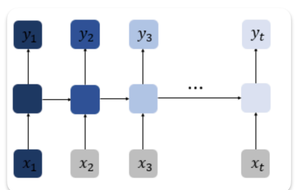
- Sequence Data Processing: Can effectively process variable-length sequence data.
- Context Information Utilization
  - Can improve the prediction accuracy of the current time step by utilizing past information.

Disadvantages:

Long-Term Dependency Problem:The problem of past information being lost as the length of the sequence increases.

Vanishing/Exploding Gradient Problem: The problem that the gradient becomes too small or too large during the learning process, and learning does not proceed properly.

Applications:
Natural Language Processing (NLP): Text generation, machine translation, sentiment analysis, part-of-speech tagging, etc.
Speech Recognition: Converting speech signals into text.
Time Series Data Analysis: Stock price prediction, weather prediction, etc.
Video Analysis: Video classification, action recognition, etc.



Long-Term Dependency issue

As the time steps of a vanilla RNN become longer, the information from the beginning is **not adequately conveyed to the end.** The image above represents the process of **information loss** as the time step progresses, with the information quantity of the first input value, x1, represented in a dark navy blue, and the color gradually fading. As we move further along the sequence, the information quantity of x1 is lost, and in situations where the time steps are long enough, the influence of the overall information of x1 may be almost meaningless."

GRU/Gated Recurrent Unit

- **Applications:** GRUs are widely used in various applications, including:
  - Natural Language Processing (NLP) for tasks like language modeling, machine translation, and sentiment analysis.
  - Speech recognition systems.
  - Time series prediction, including financial time series analysis.
  - Healthcare applications such as patient monitoring and disease prediction.
  - Gesture recognition and video analysis.
  - Infectious disease prediction.

- **Advantages:**
- GRUs have a simpler architecture with fewer parameters than LSTMs, making them computationally more efficient.
- They address the vanishing gradient problem, which can occur when training traditional RNNs on long sequences of data.
- GRUs can capture long-term dependencies in data.
-
- **Comparison with LSTMs:**
- GRUs often achieve comparable performance to LSTMs.
- The choice between GRUs and LSTMs depends on the specific task, dataset, and computational resources available.
  In LSTMs, there are three gates: output, input, and forget gates. In contrast, GRUs have only two gates: update and reset gates.
  GRUs are known to have faster training speeds than LSTMs, but various evaluations have shown that GRUs exhibit similar performance to LSTMs.
-

GRUs with fewer parameters tend to perform slightly better when the amount of data is small, while LSTMs tend to perform better with larger datasets. There is more research and usage of LSTMs than GRUs, likely because LSTM is an earlier architecture."

GRU initiation with Kera

```
model.add(GRU(hidden_size, input_shape=(timesteps, input_dim)))
```

Code

```
import numpy as np

# Define the RNN cell
def rnn_cell(input, hidden_state, weight_x, weight_h, bias):

  combined_input = np.dot(weight_x, input) + np.dot(weight_h, hidden_state) + bias
  new_hidden_state = np.tanh(combined_input)
  return new_hidden_state

# Example usage
# 1. Define hyperparameters
input_dim = 1  # Dimensionality of the input
hidden_units = 2  # Dimensionality of the hidden state
timesteps = 3  # Length of the input sequence
batch_size = 1

# 2. Initialize weights and biases (randomly)
weight_x = np.random.randn(hidden_units, input_dim)
weight_h = np.random.randn(hidden_units, hidden_units)
bias = np.zeros((hidden_units, 1))

# 3. Create an input sequence (example)
input_sequence = np.array([[0.1], [0.5], [0.9]]) # Shape: (timesteps, input_dim)

# 4. Initialize the hidden state (usually to zeros)
hidden_state = np.zeros((hidden_units, 1))

# 5. Iterate through the input sequence
for t in range(timesteps):
  input_t = input_sequence[t].reshape(input_dim, 1) # Reshape for matrix multiplication
  hidden_state = rnn_cell(input_t, hidden_state, weight_x, weight_h, bias)
  print(f"Hidden state at timestep {t+1}:\n{hidden_state}")

print("\nFinal hidden state:")
print(hidden_state)
```

```
"""
  Performs the RNN cell computation.

  Args:
    input: Input at the current timestep (scalar or vector).
    hidden_state: Previous hidden state (vector).
    weight_x: Weight for the input.
    weight_h: Weight for the hidden state.
    bias: Bias term.

  Returns:
    The new hidden state.
  """
```

rnn_cell Function
Represent a single RNN cell . It takes the current input, the previous hidden state, the weights, and the bias as input, and returns the new hidden state.