Recurrent Neural Network (RNN) Architecture Explained

Sushmita Poudel · Follow
6 min read · Aug 28, 2023

Listen Share

This article will provide insights into RNNs and the concept of backpropagation through time in RNN, as well as delve into the problem of vanishing and exploding gradient descent in RNNs.

Recurrent Neural Networks (RNNs) were introduced to address the limitations of traditional neural networks, such as FeedForward Neural Networks (FNNs), when it comes to processing sequential data. FNN takes inputs and process each input independently through a number of hidden layers without considering the order and context of other inputs. Due to which it is unable to handle sequential data effectively and capture the dependencies between inputs. As a result, FNNs are not well-suited for sequential processing tasks such as, language modeling, machine translation, speech recognition, time series analysis, and many other applications that requires sequential processing. To address the limitations posed by traditional neural networks, RNN comes into the picture.

RNN overcome these limitations by introducing a recurrent connection that allow information to flow from one time-step to the next. This recurrent connection enables RNNs to maintain internal memory, where the output of each step is fed back as an input to the next step, allowing the network to capture the information from previous steps and utilize it in the current step, enabling model to learn temporal dependencies and handle input of variable length.

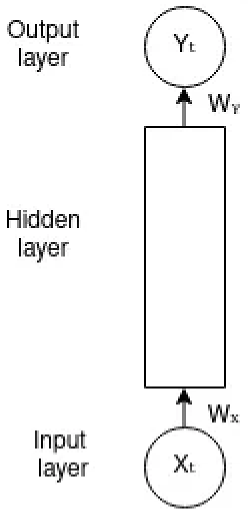


fig 1: FeedForward Neural Network (FNN). Image by Author

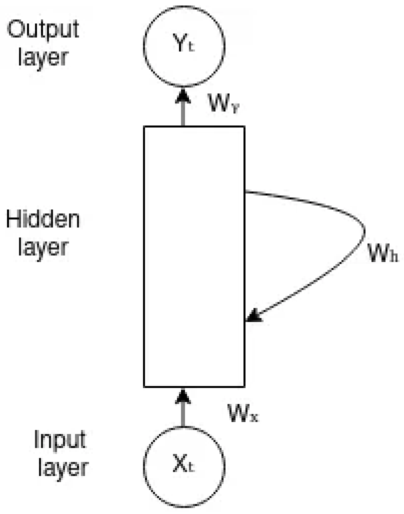


fig 2: Recurrent Neural Network (RNN). Image by Author

Architecture Of RNN

For more clear understanding of the concept of RNN, let's look at the unfolded RNN diagram.

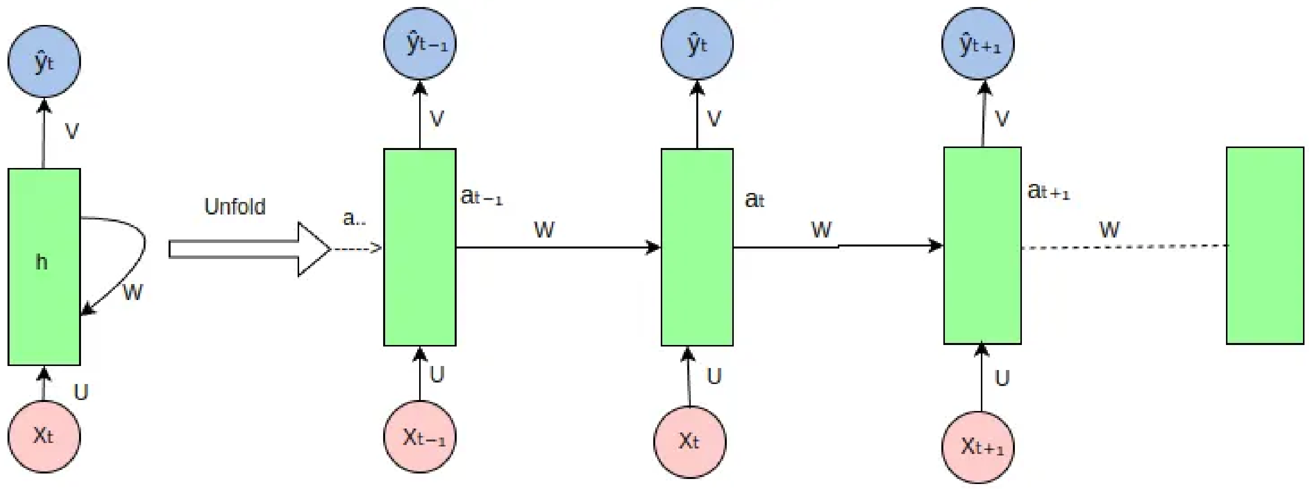


fig 3: RNN Unfolded. Image by Author.

The RNN takes an input vector **X** and the network generates an output vector **y** by scanning the data sequentially from left to right, with each time step updating the hidden state and producing an output. It shares the same parameters across all time steps. This means that, the same set of parameters, represented by U, V, W is used consistently throughout the network. `U` represents the weight parameter governing the connection from input layer **X** to the hidden layer **h** , `W` represents the weight associated with the connection between hidden layers, and `V` for the connection from hidden layer **h** to output layer **y**. This sharing of parameters allows the RNN to effectively capture temporal dependencies and process sequential data more efficiently by retaining the information from previous input in its current hidden state.

At each time step t, the hidden state a_t is computed based on the current input x_t , previous hidden state a_{t-1} and model parameters as illustrated by the following formula:

$a_t = f(a_{t-1}, x_t; \theta)$ — — — — — (1)

It can also be written as,

$a_t = f(U * X_t + W * a_{t-1} + b)$
where,

- a_t represents the output generated from the hidden layer at time step `t` .
- x_t is the input at time step `t` .
- θ represents a set of learnable parameters(weights and biases).
- `U` is the weight matrix governing the connections from the input to the hidden layer; $U \in \theta$
- `W` is the weight matrix governing the connections from the hidden layer to itself (recurrent connections); $W \in \theta$
- `V` represents the weight associated with connection between hidden layer and output layer; $V \in \theta$
- a_{t-1} is the output from hidden layer at time `t-1` .
- `b` is the bias vector for the hidden layer; $b \in \theta$

- f is the activation function.

For a finite number of time steps T=4, we can expand the computation graph of a Recurrent Neural Network, illustrated in Figure 3, by applying the equation (1) T-1 times.

$a_4 = f(a_3, x_4; \theta)$ — — — — — (2)

Equation (2) can be expanded as,

$a_4 = f(U * X_4 + W* a_3 + b)$

$a_3 = f(U * X_3 + W* a_2 + b)$

$a_2 = f(U * X_2 + W* a_1 + b)$

The output at each time step t, denoted as y_t is computed based on the hidden state output a_t using the following formula,

$\hat{y}_t = f(a_t; \theta)$ — — — — — (3)

Equation (3) can be written as,

$\hat{y}_t = f(V * a_t + c)$

when t=4, $\hat{y}_4 = f(V * a_4 + c)$

where,

- $\hat{y}_t$ is the output predicted at time step t.
- V is the weight matrix governing the connections from the hidden layer to the output layer.
- c is the bias vector for the output layer.

Backpropagation Through Time (BPTT)

Backpropagation involves adjusting the model's parameters (weights and biases) based on the error between predicted output and the actual target value. The goal of backpropagation is to improve the model's performance by minimize the loss function. Backpropagation Through Time is a special variant of backpropagation used to train RNNs, where the error is propagated backward through time until the initial time step t=1. Backpropagation involves two key steps: forward pass and backward pass.

1. **Forward Pass:** During forward pass, the RNN processes the input sequence through time, from `t=1` to `t=n`, where `n` is the length of input sequence. In each forward propagation, the following calculation takes place

$$a_t = U * X_t + W * a_{t-1} + b$$
$$a_t = \tanh(a_t)$$
$$\hat{y}_t = \mathrm{softmax}(V * a_t + c)$$

After processing the entire sequence, RNN generates a sequence of predicted outputs *$\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_t]$*. Loss is then computed by comparing predicted output ŷ at each time step with actual target output y. Loss function given by,

$$L(y, \hat{y}) = (1/t) * \Sigma(y_t — \hat{y}_t)^2 \text{ — — —— — -> MSE Loss}$$

2. Backward Pass: The backward pass in BPTT involves computing the gradients of the loss function with respect to the network's parameters (`U`, `W`, `V` and biases) over each time step.

Let's explore the concept of backpropagation through time by computing the gradients of loss at time step t=4. The figure below also serves as an illustration of backpropagation for time step 4.

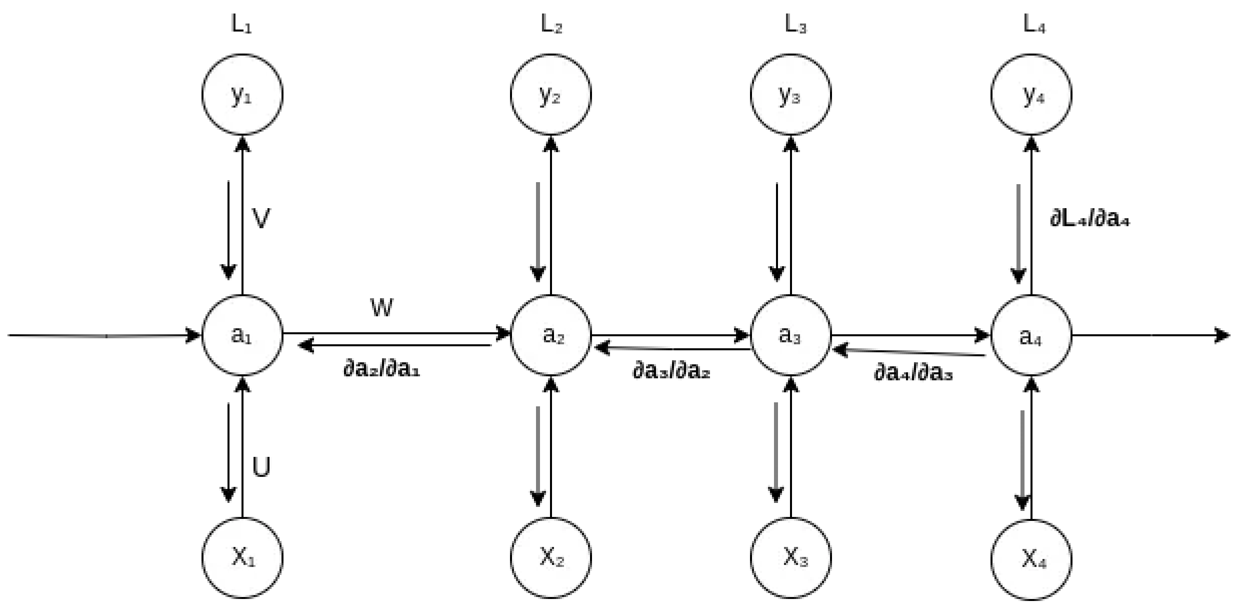


fig 4: Back Propagation Through Time (BPTT). Image by Author

Derivative of loss L w.r.t V

Loss `L` is a function of predicted value `ŷ`, so using the chain rule ∂L/∂V can be written as,

$\partial L/\partial V = (\partial L/\partial \hat{y}) * (\partial \hat{y}/\partial V)$

Derivative of loss L w.r.t W

Applying the chain rule of derivatives ∂L/∂W can be written as follows: The loss at the 4th time step is dependent upon `ŷ` due to the fact that the loss is calculated as a function of `ŷ`, which is in turn dependent on the current time step's hidden state `a₄`, `a₄` is influenced by both `W` and `a₃`, and again `a₃` is connected to both `a₂` and `W`,and `a₂` depends on `a₁` and also on `W`.

$\partial L_4/\partial W = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial W)$

Derivative of loss L w.r.t U

Similarly, ∂L/∂U can be written as,

$\partial L_4/\partial U = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial U)$

Here we're summing up the gradients of loss across all time steps which represents the key difference between BPTT and regular backpropagation approach.

Limitations of RNN

During backpropagation, gradients can become too small, leading to the vanishing gradient problem, or too large, resulting in the exploding gradient problem as they propagate backward through time. In the case of vanishing gradients, the issue is that the gradient may become too small where the network struggles to capture long-term dependencies effectively. It can still converge during training but it may take a very very long time. In contrast, in exploding gradient problem, large gradient can lead to numerical instability during training, causing the model to deviate from the optimal solution and making it difficult for the network to converge to global minima.

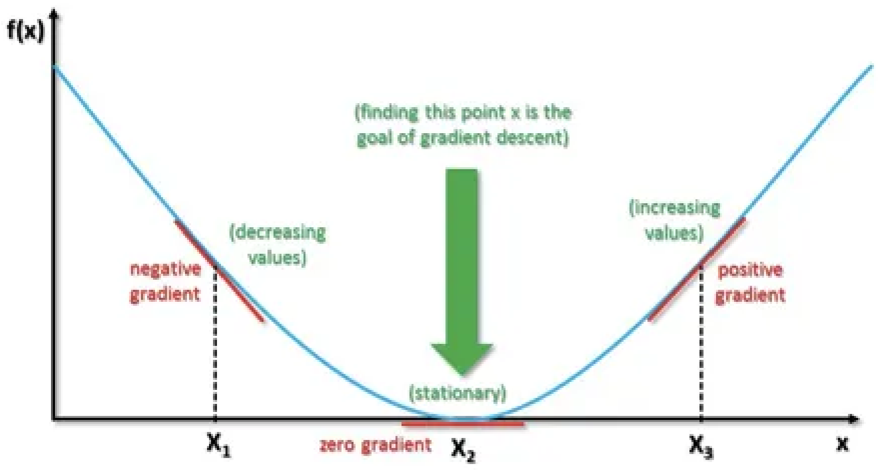


fig 5: Gradient Descent. Image Source

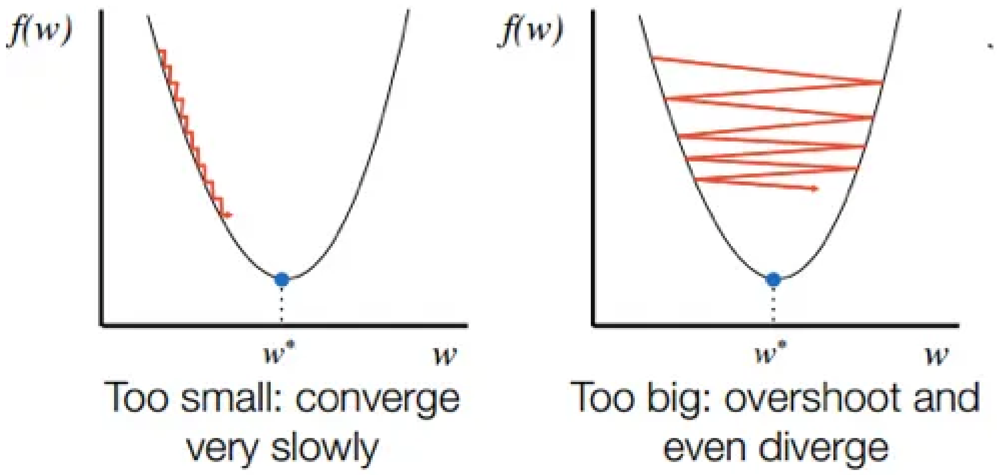

$f(w)$
w^*
w
Too small: converge very slowly
$f(w)$
w^*
w
Too big: overshoot and even diverge

fig 6: Vanishing and Exploding Gradient. Image Source

To address these problems, variations of RNN like Long-short term memory (LSTM) and Gated Recurrent Unit (GRU) networks have been introduced.

Rnn Architecture   Backpropagation   Bptt   Neural Networks   Gradient Descent

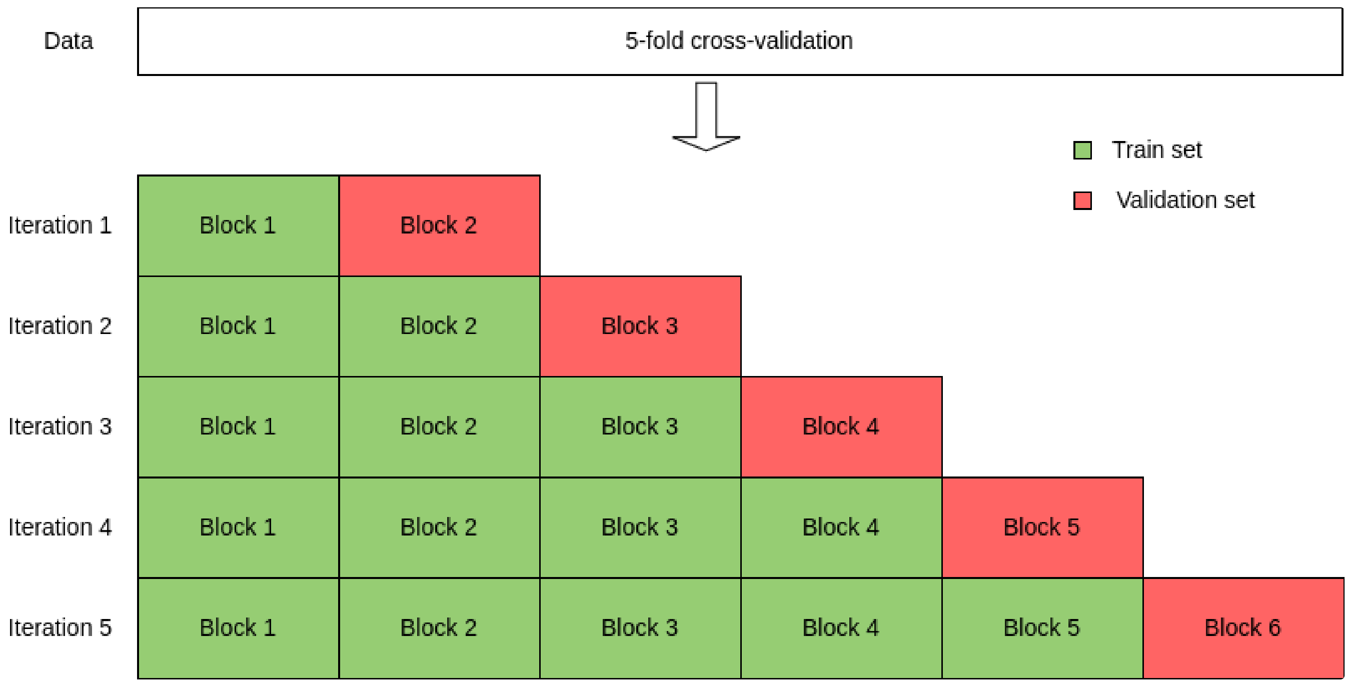
Follow

Written by Sushmita Poudel

16 Followers

Machine Learning Engineer

More from Sushmita Poudel



Sushmita Poudel

Cross Validation in Time Series Forecasting

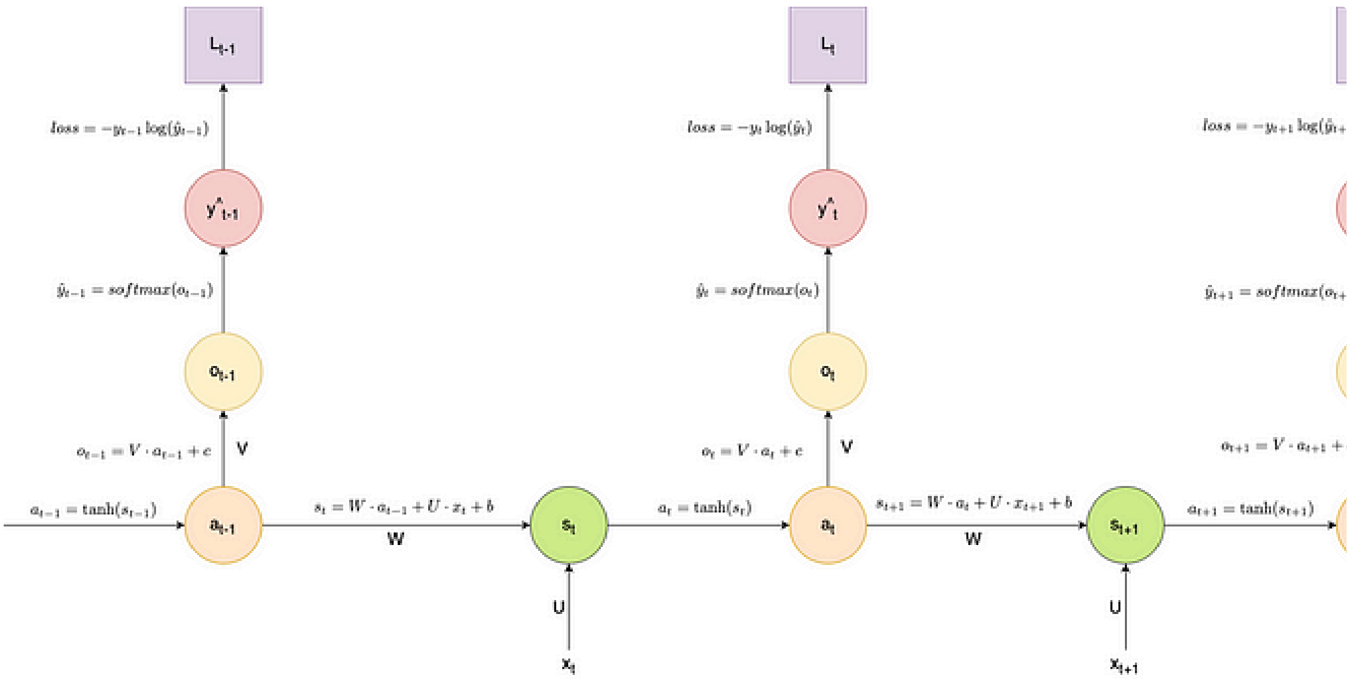
Cross-validation is a statistical method used to evaluate the performance of machine learning models. It involves dividing a dataset into...

5 min read · Mar 29, 2023

2

See all from Sushmita Poudel

Recommended from Medium



L_{t-1}
L_t
$loss = -y_{t-1}\log(\hat{y}_{t-1})$
$loss = -y_t\log(\hat{y}_t)$
$loss = -y_{t+1}\log(\hat{y}_{t+}$
$\hat{y}_{t-1}$
$\hat{y}_t$
$\hat{y}_{t-1} = softmax(o_{t-1})$
$\hat{y}_t = softmax(o_t)$
$\hat{y}_{t+1} = softmax(o_{t+}$
o_{t-1}
o_t
$o_{t-1} = V \cdot a_{t-1} + c$
V
$o_t = V \cdot a_t + c$
V
$o_{t+1} = V \cdot a_{t+1} +$
$a_{t-1} = \tanh(s_{t-1})$
a_{t-1}
$s_t = W \cdot a_{t-1} + U \cdot x_t + b$
W
s_t
$a_t = \tanh(s_t)$
a_t
$s_{t+1} = W \cdot a_t + U \cdot x_{t+1} + b$
W
s_{t+1}
$a_{t+1} = \tanh(s_{t+1})$
U
x_t
U
x_{t+1}

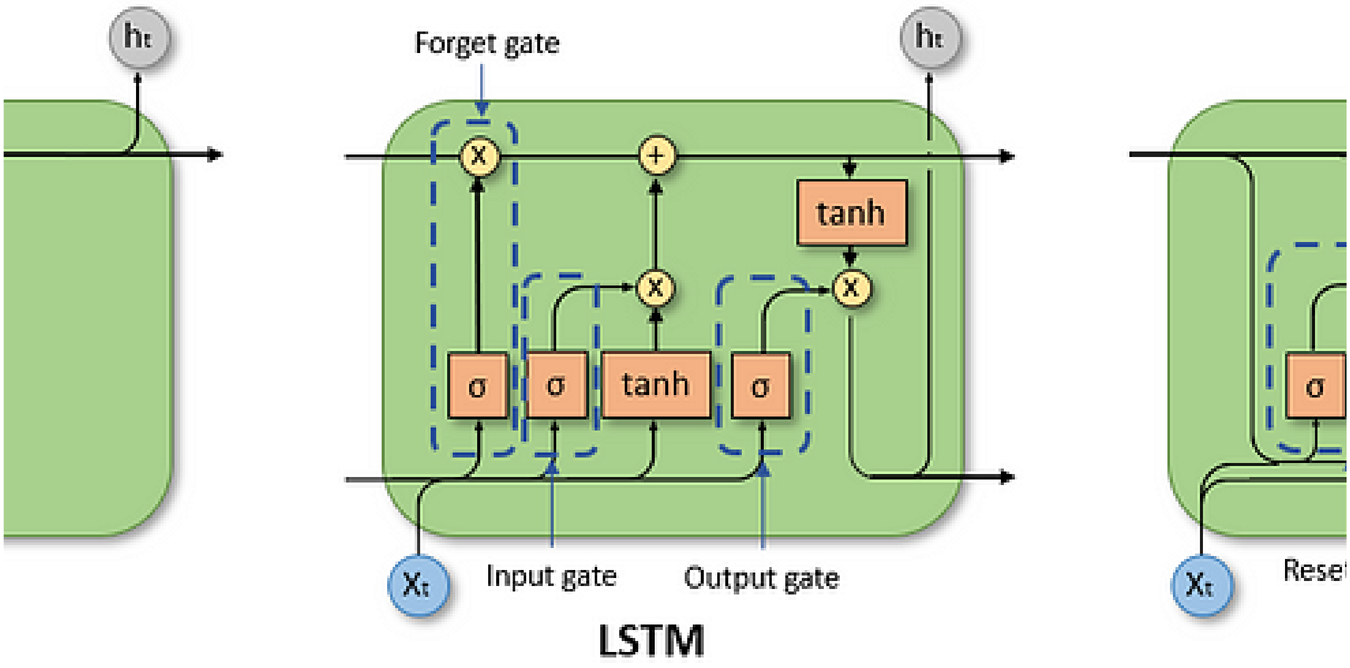
Long Nguyen

Building a Recurrent Neural Network From Scratch

In this blog post, we will explore Recurrent Neural Networks (RNNs) and the mathematics behind their forward and backward passes

14 min read · Jan 28, 2024

108



Jonte Dancker in Towards Data Science

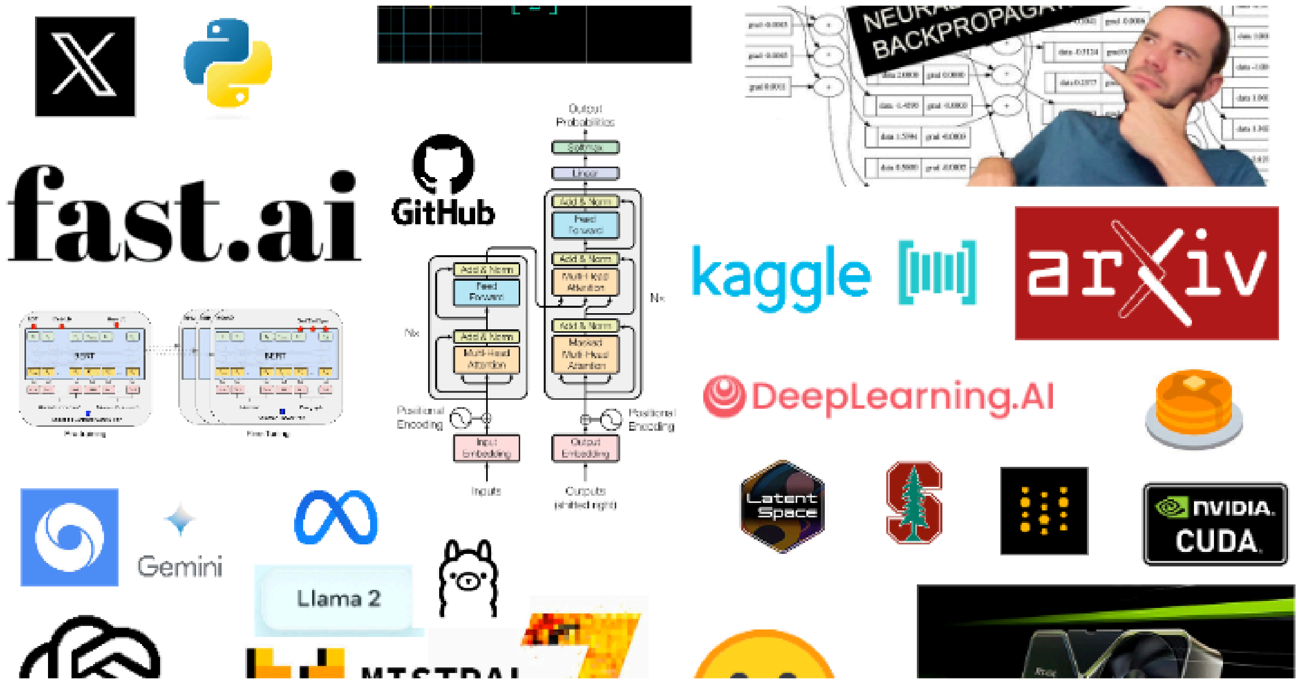
A Brief Introduction to Recurrent Neural Networks

An introduction to RNN, LSTM, and GRU and their implementation

12 min read · Dec 26, 2022

578 7

Lists



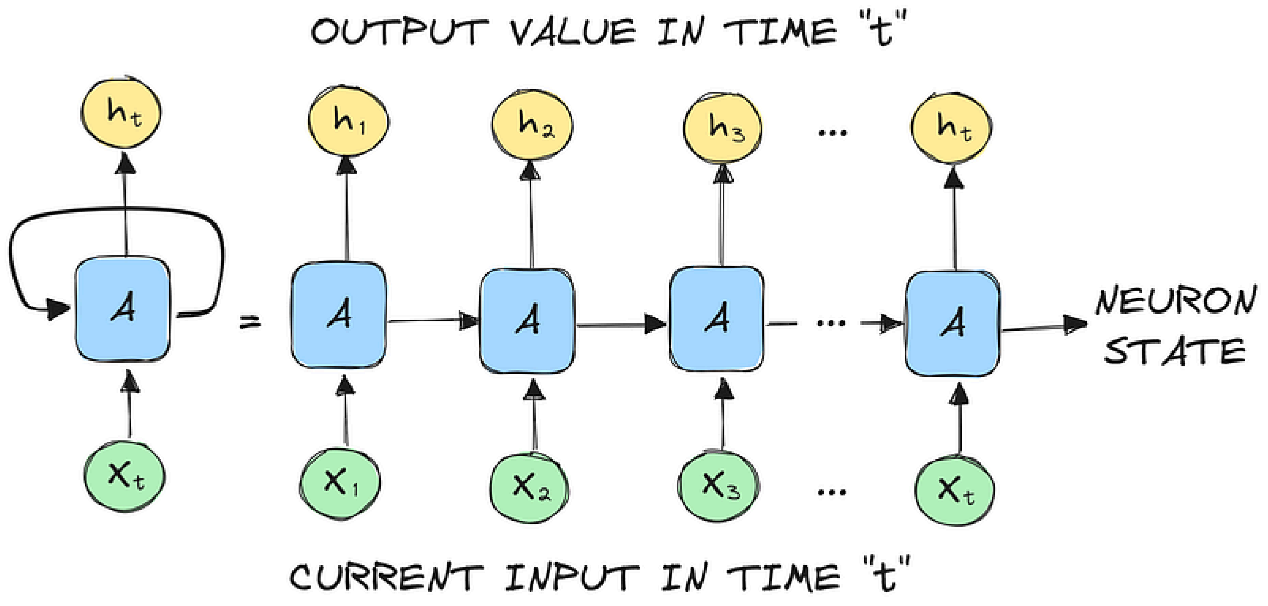
Benedict Neo in bitgrit Data Science Publication

Roadmap to Learn AI in 2024

A free curriculum for hackers and programmers to learn AI

11 min read · Feb 21, 2024

4.2K 51



Rafał Buczyński in Python in Plain English

Understanding Recurrent Neural Networks

1. Introduction to Neural Networks

10 min read · Sep 10, 2023

66 3

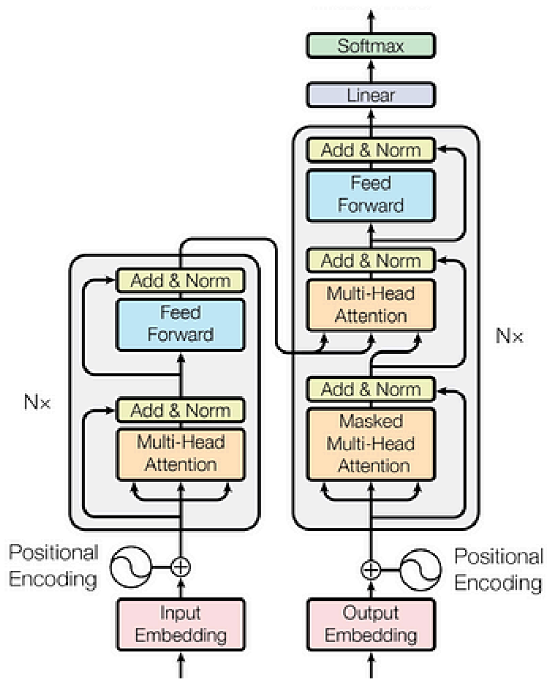
Merve Bayram Durna

Neural Networks in NLP: RNN, LSTM, and GRU

The Complete NLP Guide: Text to Context #5

10 min read · Jan 12, 2024

255 1



Hrithick Sen

The Math Behind the Machine: A Deep Dive into the Transformer Architecture

The transformer architecture was introduced in the paper "Attention is All You Need" by Vaswani and a team of researchers at Google. In the...

12 min read · Jan 16, 2024

192 5

See more recommendations