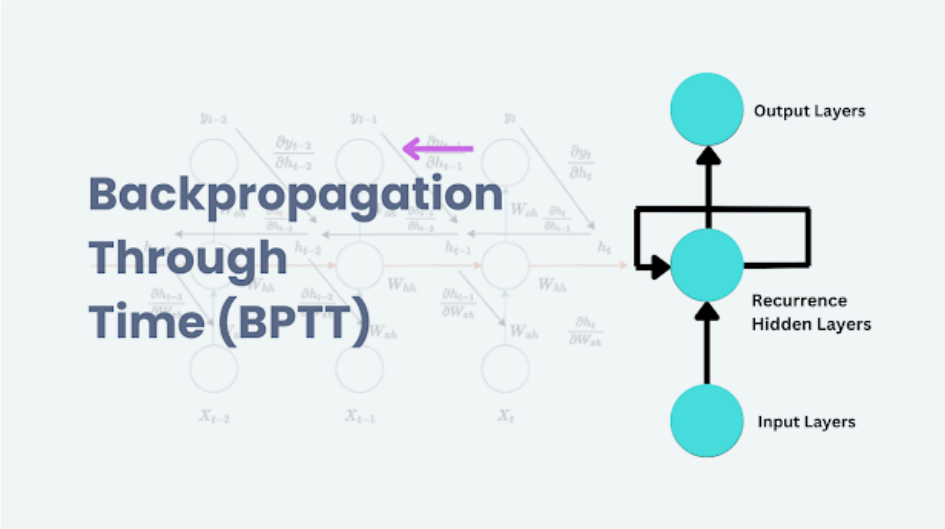
Backpropagation Through Time (BPTT): Explained With Derivations

In this article, we will delve into the intricate details of the BPTT algorithm and how it is used for training RNNs.

Sidharth GN Updated: June 2, 2025 · 10 min read



Introduction

Backpropagation is one of the most successful algorithms in Machine Learning laid the foundation of deep learning where it enables training Deep Neural Networks for optimizing weights. However, its implementation can vary based on the architecture of the Neural Network.

For instance, in the case of Recurrent Neural Networks such as **LSTMs (Long Short Term Memory)**, a distinct version of the backpropagation algorithm is needed, even though the underlying concept remains similar.

Recurrent Neural Networks specialize in processing sequences of data, introducing the concept of **time steps**, where each step corresponds to a specific moment in the sequence.

This temporal aspect enables RNNs to excel in tasks involving sequential data like text, speech, and time series.

For RNNs to learn sequential data, a variant of the backpropagation algorithm known as "**Backpropagation Through Time" (BPTT)** is used. In this article, we will delve into the intricate details of the BPTT algorithm and how it is used for training RNNs. We will cover the intuition and derivation of BPTT for training RNNs using Gradient Descent.

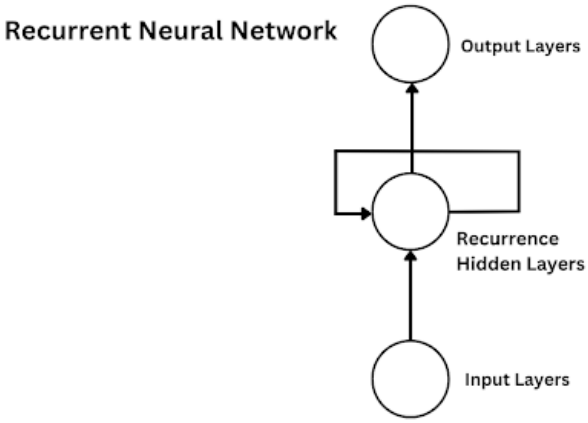
The Basic Intuition of BPTT

In the previous article, we discussed the architecture and forward pass in vanilla RNN, you can check it here. In the case of traditional Neural Network architecture where you have an input layer, hidden layers, and an output layer, backpropagation can be simply applied by finding the gradient of loss through chain rule directly from the output layer to the input layer through single or multiple hidden layers, but when comes to RNN, we need to also consider flowing the gradients through multiple time steps.

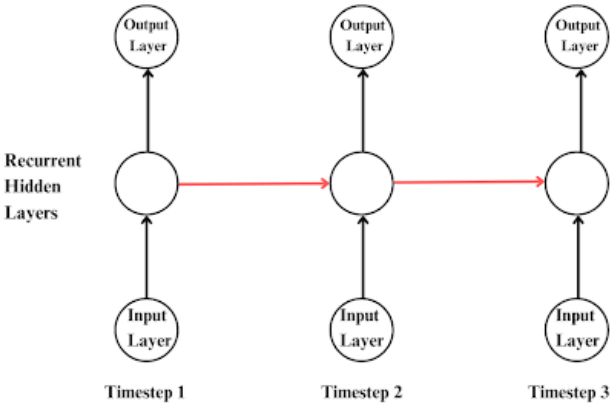
So why is the word "time" used in BPTT? In RNNs, the data is processed step-by-step (one at a time). The term "time" is used in BPTT to highlight the fact that the network's computations are performed step by step, simulating the passage of time as the network processes each element of the sequence.

A time step in a Recurrent Neural Network (RNN) refers to a specific moment or instance in a sequence of data being processed by the network. In the context of sequential data, such as text, speech, or time series, the RNN processes one element of the sequence at each time step. This element could be a word in a sentence, a single data point in a time series, or a phoneme in speech.

You can visualize each time step as a collection of Feed Forward Neural Networks where the hidden layers of each network are connected together,



Now if you flatten it you'll get something like this,



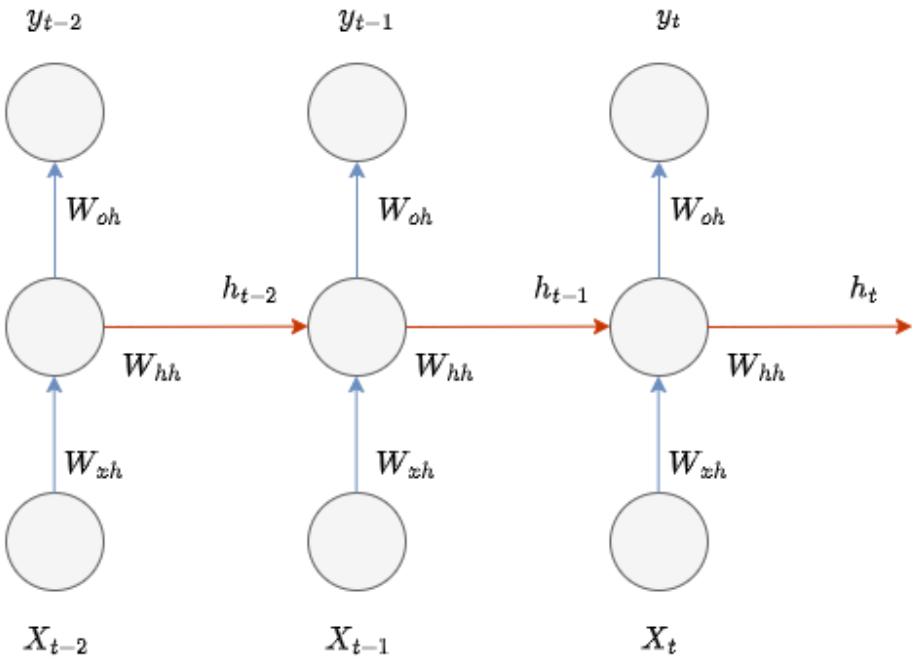
Backpropagation is applied to each time step, starting from the last one and moving backward to the first. This process is known as Backpropagation Through Time. Alright, let's see how this is done,

Derivation of BPTT

We're getting into the derivations of BPTT but before that, we need to take a look at how forward pass or forward propagation is done in RNNs,

Forward Pass

Here is the diagram that shows the inputs, weights, outputs, and hidden states, of an RNN,



The W_{xh} are the weights connecting from the input layer to the hidden layer of each time step. The W_{hh} are the weights in the hidden layer, and finally, the W_{oh} are the weights corresponding to the output layer.

The X_t, X_{t-1}, and, X_{t-2} are the inputs to each corresponding time step, and y_t, y_{t-1}, y_{t-2} are the outputs from each timestep. However, we need to represent the information passing from one time step to another. Let's call this h_t as for the **hidden state** in a particular time step. So automatically, the hidden states in the previous layers become h_{t-1}, h_{t-2}, and so on.

The basic idea of the forward pass in RNN is to calculate the values of hidden states in each time step, ie, h_t.

Now intuitively, what do you think the formula to calculate h_t when considering the last time step?, is it simply $W_{hh}.X_t$? Partially yes, but we need to also consider h_{t-1} coming from the hidden layer of the previous time step which is simply $W_{hh}.h_{t-1}$.

So considering this idea, the general formula for performing forward propagation through the hidden states will be,

$$h_t = \phi(W_{xh}.X_t + W_{hh}.h_{t-1} + b_t)$$

In RNNs and their variations, we typically use a **Hyperbolic tangent (Tanh)** activation function, so when applying the tanh activation the formula can be changed to,

$$h_t = tanh(W_{xh}.\, X_t + W_{hh}.\, h_{t-1} + b_t)$$

We now have the formula for passing inputs forward through hidden states. However, to achieve the intended outcome, we also need to obtain outputs in each time step. How can we compute these outputs? Here is the formula to do so.

$$y_t = \phi(W_{oh}.\, h_t + b_o)$$

Simple, just computing the dot product between W_{oh} and h_t plus a bias b_o. The ϕ is basically the activation function which can be any function according to the problem you are trying to solve. For instance, you can use sigmoid or softmax if you are dealing with probability problems.

Backward pass

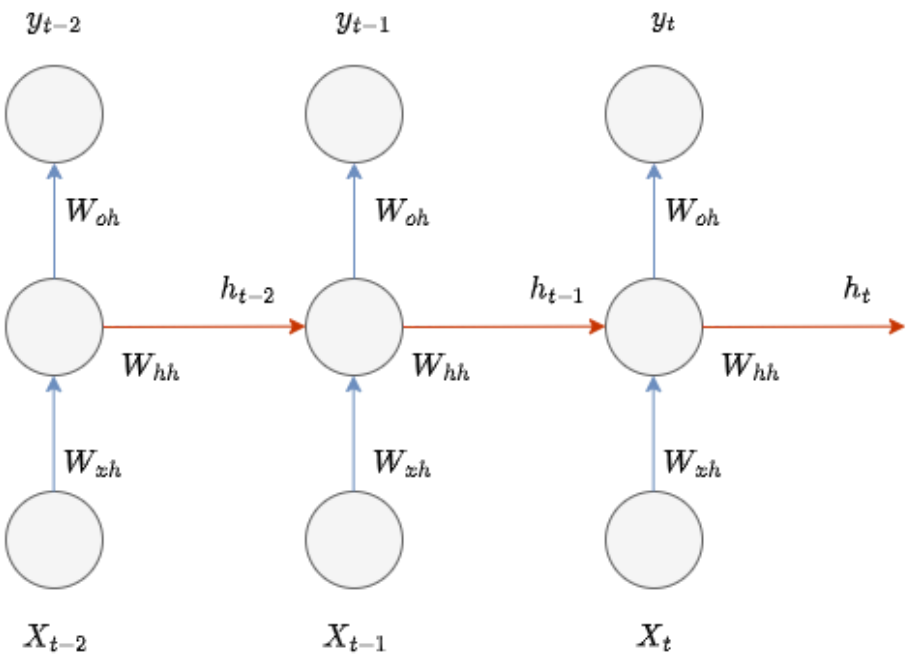
Let's figure out how to derive BPTT in RNN. The first thing we need to do is to choose a Loss function, Here I'm choosing the **Mean Squared Error (MSE)** Loss, below is the formula for calculating the MSE loss function,

$$\mathrm{L} = \frac{1}{N}\sum_{t=1}^{N}(y - y_t)^2$$

For propagating the loss backward, we need to find the derivative of loss, the derivative of MSE is simply,

$$\frac{\partial L}{\partial y_t} = \frac{2}{n}\sum_{t=t}^{n}(y - y_t)$$

Now comes the most important parts of the derivation. First, what is our objective with BPTT, to find the optimal weights and biases in each update in our RNN, ie, we need to update weights using gradient descent, there are three categories of weights as we discussed, The weights connecting inputs W_{xh}, the weights through hidden states W_{hh}, and the output weights W_{oh}.



Derivative of Loss with respect to Weights W_{xh}, W_{hh}, and, W_{oh}

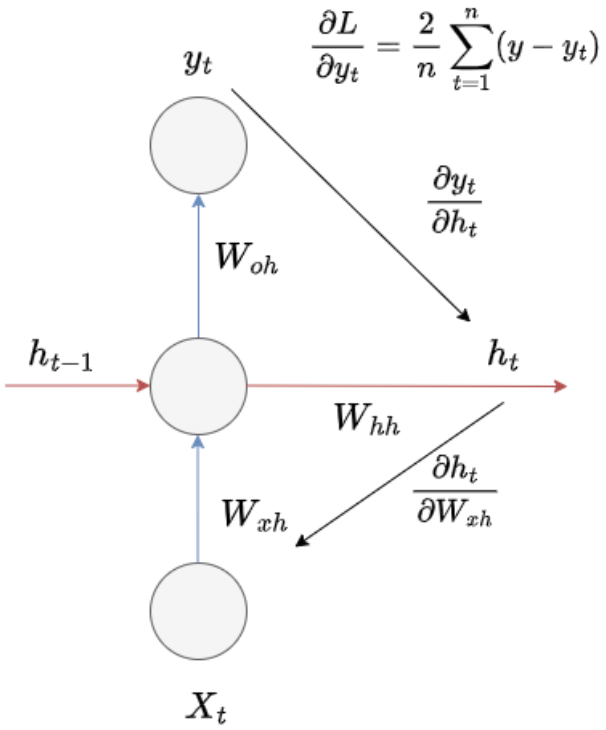
Think about it, what are the terms that affect the change in weights W when considering a single time step?

- Obviously, we can say the change in loss with respect to the change in output $\frac{\partial L}{\partial y_t}$ is influencing weights since weights are a part of the output produced, Similarly,
- The change in the output with respect to the change in hidden states $\frac{\partial y_t}{\partial h_t}$ is influencing weights,
- Finally, the change hidden states with respect to the change in Weights $\frac{\partial h_t}{\partial W}$.

This is the elegant **chain rule** in calculus written in words, this tells how much the weights W changes with respect to values related to it, the basic idea of BPTT goes something like we described above,

Let's consider W_{xh}, which is the weights connecting the input layer and hidden layer, and see how the gradients are passed through the network.

Here is a way to understand the flow of gradients in a single time step,



BPTT Single Timestep.

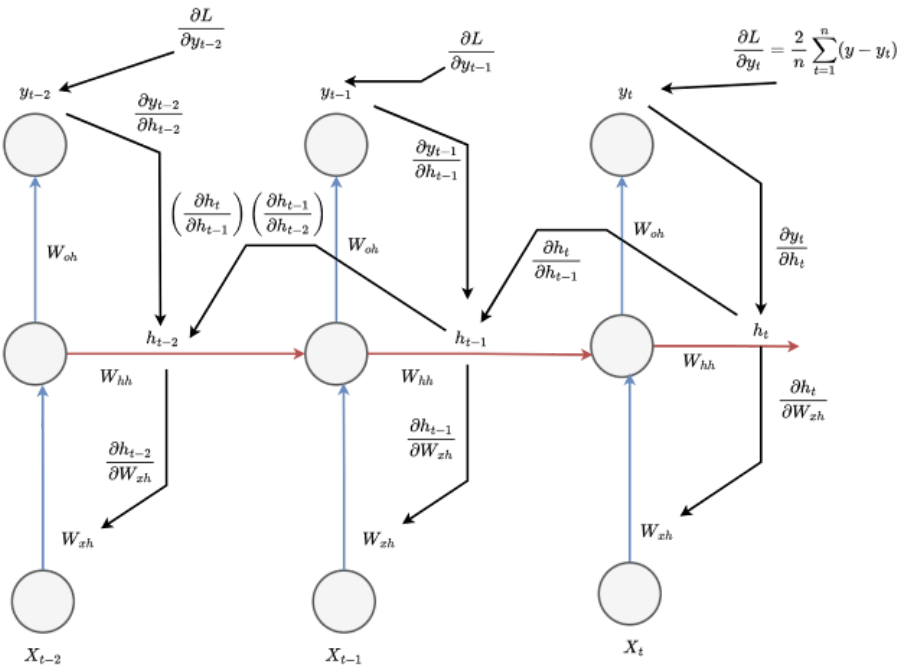
It is better to look at the diagram above and visualize how it works in your mind. Here is how you write them in formulas,

$$\frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W_{xh}}$$

Here's how to calculate the gradient of loss in relation to input weights for a single time step. However, what adjustments should we make to the formula if we have multiple time steps?

When dealing with multiple time steps, the gradients of each time step should be summed up. But beyond this, due to the interconnected nature of hidden states across time, it's

crucial to accumulate the gradients of these hidden states as well. For this, you need to multiply the gradients of hidden states together in each time step.



BPTT Multiple Timestep.

It seems like a lot of gibberish out there above, but this is how gradients flow through a Recurrent Neural Network. Let's see how we can compute one by one,

Considering single time step,

$$\frac{\partial L}{\partial W_{xh}} = \frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W_{xh}}$$

Considering 2 time steps,

$$\frac{\partial L}{\partial W_{xh}} = \underbrace{\frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W_{xh}}}_{i=0} + \underbrace{\frac{\partial L}{\partial y_{t-1}}\frac{\partial y_{t-1}}{\partial h_{t-1}}\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_{t-1}}{\partial W_{xh}}}_{i=1}$$

And three time steps,

$$\frac{\partial L}{\partial W_{xh}} = \underbrace{\frac{\partial L}{\partial y_t}\frac{\partial y_t}{\partial h_t}\frac{\partial h_t}{\partial W_{xh}}}_{i=0} + \underbrace{\frac{\partial L}{\partial y_{t-1}}\frac{\partial y_{t-1}}{\partial h_{t-1}}\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_{t-1}}{\partial W_{xh}}}_{i=1} + \underbrace{\frac{\partial L}{\partial y_{t-2}}\frac{\partial y_{t-2}}{\partial h_{t-2}}\left(\frac{\partial h_t}{\partial h_{t-1}}\right)\left(\frac{\partial h_{t-1}}{\partial h_{t-2}}\right)\frac{\partial h_{t-2}}{\partial W_{xh}}}_{i=2}$$

What are you observing from the equations? As time steps increase the hidden state gradient in accumulating. Now consider thousands of time steps, in that case, the gradients of hidden states (values in the center of the equation) grow over and over again in each time step, unfortunately, we cannot write it, so lets us represent this in the general formula, here is how to generalize it,

$$\frac{\partial L}{\partial W_{xh}} = \sum_{i=0}^{t}\Bigg[\underbrace{\frac{\partial L}{\partial y_{t-i}}\frac{\partial y_{t-i}}{\partial h_{t-i}}}_{R_{t-i}}\left(\prod_{j=(t-i+1)}^{t}\frac{\partial h_j}{\partial h_{j-1}}\right)\underbrace{\frac{\partial h_{t-i}}{\partial W_{xh}}}_{S_{t-i}}\Bigg]$$

Where, y_{t-i} represents the output at time step $t-i$

h_{t-i} represents the hidden state at time step $t-i$

Now this symbol $\prod$ is the product that is used to multiply a sequence of values, in our case, it is used to accumulate the gradients of hidden states through time.

Let's find the derivative of loss with respect to weights in the hidden units, It is so simple, All we need to do is to substitute W_{hh} in the place of W_{xh} in the equation,

$$\frac{\partial L}{\partial W_{hh}} = \sum_{i=0}^{t}\Bigg[\underbrace{\frac{\partial L}{\partial y_{t-i}}\frac{\partial y_{t-i}}{\partial h_{t-i}}}_{R_{t-i}}\left(\prod_{j=(t-i+1)}^{t}\frac{\partial h_j}{\partial h_{j-1}}\right)\underbrace{\frac{\partial h_{t-i}}{\partial W_{hh}}}_{T_{t-i}}\Bigg]$$

The idea is the same except you find the derivatives with respect to W_{hh}.

This can be a little different when comes to the output layer, for finding W_{oh}, you just only need to consider the output y_t and Weights W_{oh}, so it can be written as,

$$\frac{\partial L}{\partial W_{oh}} = \sum_{i=0}^{t} \frac{\partial L}{\partial y_{t-i}} \frac{\partial y_{t-i}}{\partial W_{oh}}$$

Alright, now let's see how we can calculate bias gradients,

Derivative of Loss With Respect to Bias b_h and b_o

We need to find the derivatives of bias in the hidden layer b_h and the bias in the output layer b_o, let's see how it can be done in the case of b_h,

$$\frac{\partial L}{\partial b_h} = \sum_{i=0}^{t} \Bigg[\underbrace{\frac{\partial L}{\partial y_{t-i}} \frac{\partial y_{t-i}}{\partial h_{t-i}}}_{R_{t-i}} \left(\prod_{j=(t-i+1)}^{t} \frac{\partial h_j}{\partial h_{j-1}} \right) \underbrace{\frac{\partial h_{t-i}}{\partial b_h}}_{f'(a_{t-i})} \Bigg]$$

Now for bias in the output layer, b_o will be,

$$\frac{\partial L}{\partial b_o} = \sum_{i=0}^{t} \frac{\partial L}{\partial y_{t-i}} \frac{\partial y_{t-i}}{\partial b_o}$$

The process of calculating derivatives with respect to bias terms in recurrent neural networks (RNNs) is similar to the backpropagation process in feed-forward neural networks, with the added consideration of hidden states due to the recurrent nature of RNNs.

Updating Weights and Bias Using Gradient Descent

The update rule is quite similar across different types of neural networks, including recurrent neural networks (RNNs), with some variations depending on the optimization algorithm used. Since we are using Gradient Descent, the update rule will be,

Updating W_{oh},

$$W_{oh} \leftarrow W_{oh} - \alpha \cdot \frac{\partial L}{\partial W_{oh}}$$

Updating W_{xh},

$$W_{xh} \leftarrow W_{xh} - \alpha \cdot \frac{\partial L}{\partial W_{xh}}$$

Updating W_{hh},

$$W_{hh} \leftarrow W_{hh} - \alpha \cdot \frac{\partial L}{\partial W_{hh}}$$

Updating b_h,

$$b_h \leftarrow b_h - \alpha \cdot \frac{\partial L}{\partial b_h}$$

Updating b_o,

$$b_o \leftarrow b_o - \alpha \cdot \frac{\partial L}{\partial b_o}$$

Where α is the learning rate.

Conclusion

So we have discussed BPTT (Backpropagation Through Time), an effective way for training Sequential Models in Deep Learning like RNNs, LSTMs, and GRUs, using gradient descent. BPTT is a variation of Backpropagation that considers the updation of parameters through different time steps, enabling these models to capture temporal dependencies and patterns in data. By leveraging the inherent memory of recurrent connections, BPTT allows the network to learn from past inputs and adjust its parameters accordingly. This is particularly valuable in tasks where the order and timing of data elements hold critical information, such as language translation, speech recognition, and financial predictions.

However, BPTT comes with its challenges, notably the vanishing and exploding gradient problems, which necessitate careful parameter tuning and architectures that alleviate these issues, such as LSTM and GRU units. So in the upcoming tutorials, we discuss how to manage the problem of vanishing and exploding gradients commonly occurring in RNNs, and LSTMs and GRUs can help.

Thanks for Reading!

Posted by Sidharth GN
https://www.linkedin.com/in/sidharth-gn-4ab311208/

You may like these posts

01 **Derivation of Backpropagation in Convolutional Neural Network (CNN)**

IntroductionThe backpropagation algorithm has emerged as a highly...

Jul 1, 2023

02 **Build A Convolutional Neural Network (CNN) From Scratch Using Python**

No image

CNN From ScratchIntroductionConvolutional...

Jul 8, 2023

03 **Coding a Neural Network Using TensorFlow & Keras to Classify Fashion MNIST**

IntroductionIn this article, we will walk through the process of creating a neur...

Apr 6, 2023

04 **Convolutional Neural Network (CNN): Architecture Explained | Deep Learning**

IntroductionDuring the 1980s, a few researchers and collaborators includin...

Jun 2, 2023

Join the conversation (1)

Youtube

Topics

Artificial Intelligence
Deep Learning
Machine Learning

- About
- Contact
- Privacy Policy

Top