<https://statquest.org/video-index/>

## **Recurrent Neural Networks (RNNs), Clearly Explained!!!**

NN that works with different amounts of sequential data

**Sequential data** refers to **data where the order or sequence of elements matters, meaning each data point is dependent on the preceding ones**

**Recurrent Neural Network(RNN)** is one way to deal with the problem of having different amount of inputs values

Recurrent Neural Networks have weights, biases, activation functions and **Feedback loops**.

**activation functions** introduce non-linearity to the model

**Feedback loops /recurrent connections,** allow the network to process sequential data by passing information from previous time steps to the current one, enabling the network to learn and remember patterns over time

Output can go 2 paths the output or the Feedback loop

**Summation** allows the connection of the sequential data/

we can **unroll** the feedback loop by making a copy of the neural network for each input value.

Regardless of how many times we unroll a recurrent neural network, the weights and biases are shared across every input.

One problem with RNN is that the more we unroll the harder it is to train :

This problem is called **The Vanishing/Exploding Gradient Problem: When the gradient becomes either extremely small (vanishing) or large (exploding) as they propagate through multiple layers, hindering effective training**

when we optimize neural networks with Backpropagation, we first find the derivatives, or Gradients, for each parameter.

**Gradient descent** is an iterative optimization algorithm used in machine learning to find the minimum of a function

* We then plug those Gradients into the Gradient Descent algorithm to find the parameter values that minimize a **Loss Function**, like the **sum of the squared residuals**.

Cool project idea: Check my definition:

You give it a word or prompt and checks if the definition is correct.

### **summary**

* Basic, vanilla Recurrent Neural Networks are hard to train because the gradients can explode, or vanish.

## **Long Short-Term Memory (LSTM), Clearly Explained**

**Long Short-Term Memory (LSTM)**, is a type of Recurrent Neural Network that is designed to avoid the exploding/ vanishing gradient problem.

LSTM uses multiple paths to make prediction:

* One path for Long-Term Memories
* Another path for Short-Term Memories

Unlike the networks we've used before in this series, Long Short-Term Memory uses **Sigmoid Activation Functions** and Thanh Activation Functions

* Sigmoid Activation Functions takes any X-axis coordinate and turns it into a y-axis coordinate between 0 and 1
* The tanh, or **Hyperbolic Tangent**, Activation Function takes any x-axis coordinate and turns it into a y-axis coordinate between -1 and 1.

Short:

* Sigmoid Activation Function has a range from 0 to 1
* Tanh Activation Function has a range from -1 to 1

\*\*\*Note: Focus more detail on how Activation Functions work

**Cell state** —> **Long-Term Memory** : No weights and biases that can modify it directly, so it can flow through a series of unrolled units without causing the gradient to explode or vanish.

**Hidden State** —> **Short-Term Memory** : Directly connect to weights that can modify them

We have values for the Long-Term Memory, Short-Term Memory and input and their corresponding weights to get to the output.

The output of the Sigmoid activation function of the first stage determines what percentage of the Long-Term Memory is remembered.

**Forget Gate**: **determines whether to keep the current value of the memory or flush it**

Combines Short-Term Memory + input —> **Potential Long-Term Memory** that uses the Tanh Activation Function with an another Unit that determents what **percentage of the Long-Term Memory is remembered** that uses the Sigmoid Activation function —> That gives us a New Long Term Memory when we sum it with the Forget Gate result of the Long-Term Memory

**Input Gate**: **It controls what information will be passed to the memory cell based on previous output and current sensor measurement data**

The Final Stage updates the Short-Term Memory

New Long-Term Memory(Current LTM) and use it as input to the Tanh Activation Function —> Potential Short-Term Memory **X** a Unit that determents what **percentage of the Memory is remembered** that uses the Sigmoid Activation function —> New Short-Term Memory

**Output Gate**: **Controls what data to pass as the output hidden state**

The reason the LSTM reuses the exact same Weights and Biases is so it can handle data sequences of different lengths. —> Unroll the LSTM

### **Summary:**

Using different paths for Long-Term Memory and Short-Term Memory, LSTM Networks avoid the exploiting/Vanishing gradient problem, So we can Unroll them more times in accordance to the amount of sequences of input data then the Vanilla Recurrent Neural Networks

## **Neural Networks Part 5: ArgMax and SoftMax**

**ArgMax** sets the simply sets the largest value to 1 and all of the other values to 0.

* Makes the value easy to interpret
* ArgMax cant be used to optimize the Weights and Biases in the Neural Network as they are constant, 0 and 1
* Has a terrible derivative of 0 we cant use it with **Backpropagation**

**SoftMax**: transforms a vector of real numbers into a probability distribution, where each element represents the probability of belonging to a specific class, ensuring the probabilities sum to 1

* Preserves the original order/ranking of the raw output values
* Range is between 0 to 1
* The different random selected weights and biases values result in the different predicted probabilities
* Accuracy is not that good

## **Neural Networks Part 6: Cross Entropy**

We use the **Sum of Squared Residuals (SSR)** in cases where we need to evaluate how well a neural network fits the given data

**Cross-entropy** is a commonly used loss function for **classification problems**, particularly in **multi-class** and **binary classification** tasks. It measures how well the predicted probability distribution aligns with the actual class labels

Neural networks often use a numerically stable and simplified version of the cross-entropy loss to prevent computational issues like underflow when dealing with very small probability values. The simplification comes from how the softmax function is combined with cross-entropy

To get the total error of Entropy we add the results for each cross Entropy done.

**Residuals** = observed - Predicted

Note: That Step Size for Backpropagation depends in part, on the derivatives of these functions

Cross Entropy has a greater loss when we get closer to zero while the Sum of Squared Residuals does not , which it is a lot more smaller as it the loss is symmetric (treats both small and large errors equally).

## **Word Embedding and Word2Vec, Clearly Explained!!!**

One super easy way to convert words into numbers is to just assign each word to a random number. —> We can train a simple Neural Network to assign number for us.

* Advantage is that it can use context of words in the training dataset to optimize the weights that lead to similar words ending up with similar embeddings

**Identity Activation function** , also known as the **linear activation function** or "**no activation**," simply outputs the same value as its input, without any transformation or modification

The **embedding layer** maps discrete, categorical data (like words) into a continuous, lower-dimensional vector space

Goal is train the NN so that it correctly predicts the next word in a phrase

2 methods that word2vec, (Word embeddings )

1st method: **Continuous Bag of words** : uses the surrounding words to predict what occurs in the middle

2nd method: **Skip Gram** : uses the word in the middle to predict the surrounding words

Note: People use 100+ activation functions to create a lot of embeddings per word and a large amount of inputs —> Leads to slow training

We can speed up the process with **Negative Sampling**, works by randomly selecting a subset of words we don't want to predict for optimization.

## **Sequence-to-Sequence (seq2seq) Encoder-Decoder Neural Networks, Clearly Explained!!!**

**sequence-to- sequence**, or **seq2seq**, problems: **involve transforming one sequence into another**

One solution is to use **seq2seq Encoder-Decoder model** :

Ex: English to Spanish model

Requirement:

* we need our seq2seq Encoder- Decoder model to be able to handle variable input and variable output lengths.

Embedding layer convert words into numbers

In natural language processing (NLP) and neural network-based language models, **tokens** are the smallest units of text that a model processes.

* Because the vocabulary contains a mix of words and symbols, we call these individual elements tokens instead of just words

From the Embedding network we use it as a input, with the indicated word with a value of 1 and everything else 0, to the LSTM network and as we unroll the network we change the word we want to focus by indicating it with 1

Note: We reuse the exact same Weights and Biases for the LSTM and the Embedding Layer no matter how many times we unroll them

However, in practice, in order to have more Weights and Biases to fit the model to our data, people often add **additional LSTM cells** to the input.

Also, to add even more Weights and Biases to fit the model to our data, people often add additional **layers of LSTMs**.

**Encoder** contains 2 layers of LSTM, with 2 LSTM Cells per Layer

* In essence, the Encoder encodes the input sentence, Let's go, into a collection of long and short term memories (cell and hidden states).

The last long and short term memories (the cell and hidden states) from both layers of the LSTM cells in the Encoder are called the **Context Vector**.

Thus, the Encoder encodes the input sentence, Let's go, into the Context Vector.

Now we need to decode the context Vector : **Decoder** : Contains a 2 layers of LSTM, with 2 LSTM Cells per Layer (withs its own weights and biases)

As we use the context Vector to initialize the cell and hidden state in the LSTMs in the Decoder , so we can decode the context Vector into the output Sentence

So the input to the LSTM cells comes from the Embedding Layer for the other sequence we want

EOS (End of Sentence) or SOS (Start of sentence )are used to help models process text more effectively

The output values of the top LSTM cells layer .are transformed by additional Weights and Biases in what is called a Fully Connected Layer.

A **Fully Connected Layer** is just another name for a basic, vanilla neural network.

Decoder does not stop until it outputs an <EOS> token. So we unroll the Decoder step by step, feeding its previous output as the next input, until the <EOS> token is generated, signaling the end of the sequence.

NOTE: By decoupling the Encoder from the Decoder, the input text and the translated output text can be different lengths.

NOTE: Just like for all Neural Networks, all of these Weights and Biases are trained using Backpropagation.

In Training an Encoder-Decoder, instead of using the predicted token as input to the Decoder LSTMs, we use the known, **correct token**.

Also, in training instead of predicting into we reach the token <EOS> each output phrase ends where the known phase ends.

**Teacher Forcing**: an algorithm for training the weights of recurrent neural networks (RNNs).

* instead of using the model’s previous prediction as the next input, we use the **actual correct output (ground truth)** from the training data.

## **Resources:**

* <https://www.youtube.com/watch?v=AsNTP8Kwu80&t=2s>
* <https://www.youtube.com/watch?v=YCzL96nL7j0>
* <https://www.youtube.com/watch?v=viZrOnJclY0>
  + <https://www.youtube.com/watch?v=KpKog-L9veg>
  + <https://www.youtube.com/watch?v=6ArSys5qHAU>
* <https://www.youtube.com/watch?v=L8HKweZIOmg&t=223s>

<https://github.com/bentrevett/pytorch-seq2seq/tree/rewrite>