DL Lab

Exp 1

Introduction to keras and tenserflow

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries, and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence Research organization to conduct machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

TensorFlow provides stable Python and C++ APIs, as well as non-guaranteed backward compatible API for other languages.

TensorFlow is a Python-friendly open source library for numerical computation that makes machine learning and developing neural networks faster and easier.

Machine learning is a complex discipline but implementing machine learning models is far less daunting than it used to be, thanks to machine learning frameworks—such as Google's TensorFlow—that ease the process of acquiring data, training models, serving predictions, and refining future results.

Created by the Google Brain team and initially released to the public in 2015, TensorFlow is an open source library for numerical computation and large-scale machine learning. TensorFlow bundles together a slew of machine learning and deep learning models and algorithms (*aka* neural networks) and makes them useful by way of common programmatic metaphors. It uses Python or JavaScript to provide a convenient front-end API for building applications, while executing those applications in high-performance C++.

TensorFlow, which competes with frameworks such as PyTorch and Apache MXNet, can train and run deep neural networks for handwritten digit classification, image recognition, word embeddings, recurrent neural networks, sequence-to-sequence models for machine translation, natural language processing, and PDE (partial differential equation)-based

simulations. Best of all, TensorFlow supports production prediction at scale, with the same models used for training.

How TensorFlow works

TensorFlow allows developers to create *dataflow graphs*—structures that describe how data moves through a graph, or a series of processing nodes. Each node in the graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array, or *tensor*.

TensorFlow applications can be run on most any target that's convenient: a local machine, a cluster in the cloud, iOS and Android devices, CPUs or GPUs. If you use Google's own cloud, you can run TensorFlow on Google's custom TensorFlow Processing Unit (TPU) silicon for further acceleration. The resulting models created by TensorFlow, though, can be deployed on most any device where they will be used to serve predictions.

TensorFlow 2.0, released in October 2019, revamped the framework in many ways based on user feedback, to make it easier to work with (as an example, by using the relatively simple Keras API for model training) and more performant. Distributed training is easier to run thanks to a new API, and support for TensorFlow Lite makes it possible to deploy models on a greater variety of platforms. However, code written for earlier versions of TensorFlow must be rewritten—sometimes only slightly, sometimes significantly—to take maximum advantage of new TensorFlow 2.0 features.

Keras

Keras is an open-source high-level Neural Network library, which is written in Python is capable enough to run on Theano, TensorFlow, or CNTK. It was developed by one of the Google engineers, Francois Chollet. It is made user-friendly, extensible, and modular for facilitating faster experimentation with deep neural networks. It not only supports Convolutional Networks and Recurrent Networks individually but also their combination.

It cannot handle low-level computations, so it makes use of the **Backend** library to resolve it. The backend library act as a high-level API wrapper for the low-level API, which lets it run on TensorFlow, CNTK, or Theano.

Initially, it had over 4800 contributors during its launch, which now has gone up to 250,000 developers. It has a 2X growth ever since every year it has grown. Big companies like Microsoft, Google, NVIDIA, and Amazon have actively contributed to the development of Keras. It has an amazing industry interaction, and it is used in the development of popular firms likes Netflix, Uber, Google, Expedia, etc.

What makes Keras special?

- Focus on user experience has always been a major part of Keras.
- Large adoption in the industry.
- It is a multi backend and supports multi-platform, which helps all the encoders come together for coding.
- Research community present for Keras works amazingly with the production community.
- Easy to grasp all concepts.
- It supports fast prototyping.
- It seamlessly runs on CPU as well as GPU.
- It provides the freedom to design any architecture, which then later is utilized as an API for the project.
- It is really very simple to get started with.
- Easy production of models actually makes Keras special.

Keras user experience

1. **Keras is an API designed for humans**

   Best practices are followed by Keras to decrease cognitive load, ensures that the models are consistent, and the corresponding APIs are simple.
2. **Not designed for machines**

   Keras provides clear feedback upon the occurrence of any error that minimises the number of user actions for the majority of the common use cases.

3. **Easy to learn and use.**
4. **Highly Flexible**

   Keras provide high flexibility to all of its developers by integrating low-level deep learning languages such as TensorFlow or Theano, which ensures that anything written in the base language can be implemented in Keras.

GPU-

Graphics processing technology has evolved to deliver unique benefits in the world of computing. The latest graphics processing units (GPUs) unlock new possibilities in gaming, content creation, machine learning, and more.

What Does a GPU Do?

The graphics processing unit, or GPU, has become one of the most important types of computing technology, both for personal and business computing. Designed for parallel processing, the GPU is used in a wide range of applications, including graphics and video rendering. Although they're best known for their capabilities in gaming, GPUs are becoming more popular for use in creative production and artificial intelligence (AI).

GPUs were originally designed to accelerate the rendering of 3D graphics. Over time, they became more flexible and programmable, enhancing their capabilities. This allowed graphics programmers to create more interesting visual effects and realistic scenes with advanced lighting and shadowing techniques. Other developers also began to tap the power of GPUs to dramatically accelerate additional

workloads in high performance computing (HPC), deep learning, and more.

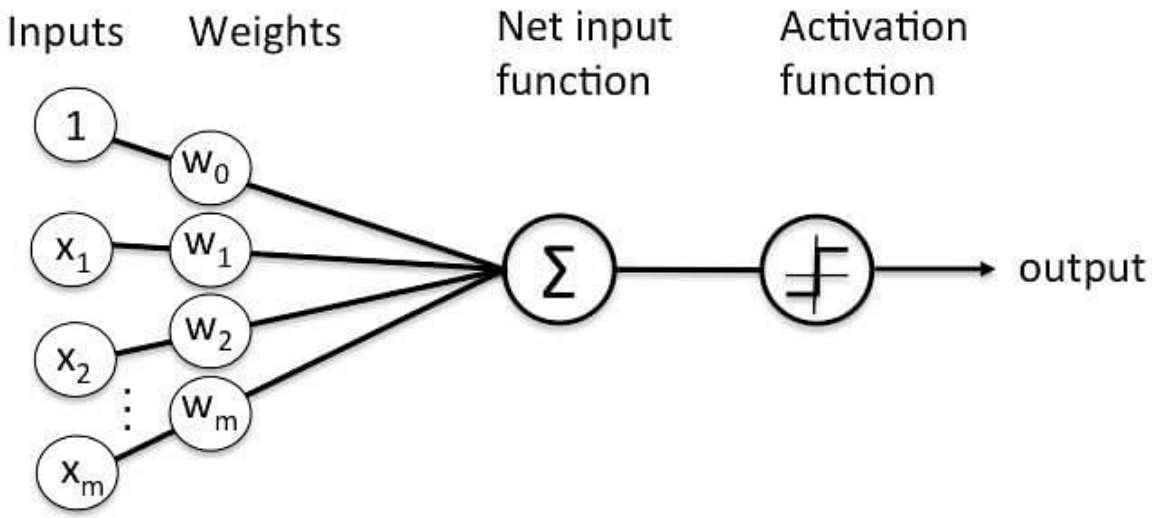
GPU and CPU: Working Together

The GPU evolved as a complement to its close cousin, the CPU (central processing unit). While CPUs have continued to deliver performance increases through architectural innovations, faster clock speeds, and the addition of cores, GPUs are specifically designed to accelerate computer graphics workloads.

Exp 2

Perceptron

Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron. A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.



Types of Perceptron:

1. **Single layer: Single layer perceptron can learn only linearly separable patterns.**
2. **Multilayer: Multilayer perceptrons can learn about two or more layers having a greater processing power.**

The Perceptron algorithm learns the weights for the input signals in order to draw a linear decision boundary.

Multi layer perceptron

Multi layer perceptron (MLP) is a supplement of feed forward neural network. It consists of three types of layers—the input layer, output

layer and hidden layer, as shown in Fig. 3. The input layer receives the input signal to be processed. The required task such as prediction and classification is performed by the output layer. An arbitrary number of hidden layers that are placed in between the input and output layer are the true computational engine of the MLP. Similar to a feed forward network in a MLP the data flows in the forward direction from input to output layer. The neurons in the MLP are trained with the back propagation learning algorithm. MLPs are designed to approximate any continuous function and can solve problems which are not linearly separable. The major use cases of MLP are pattern classification, recognition, prediction and approximation.

Exp 3,4

CNN

Exp 5 optimizers-

Gradient Descent Deep Learning Optimizer

Gradient Descent can be considered as the popular kid among the class of optimizers. This optimization algorithm uses calculus to modify the values consistently and to achieve the local minimum. Before moving ahead, you might have the question of what a gradient is?

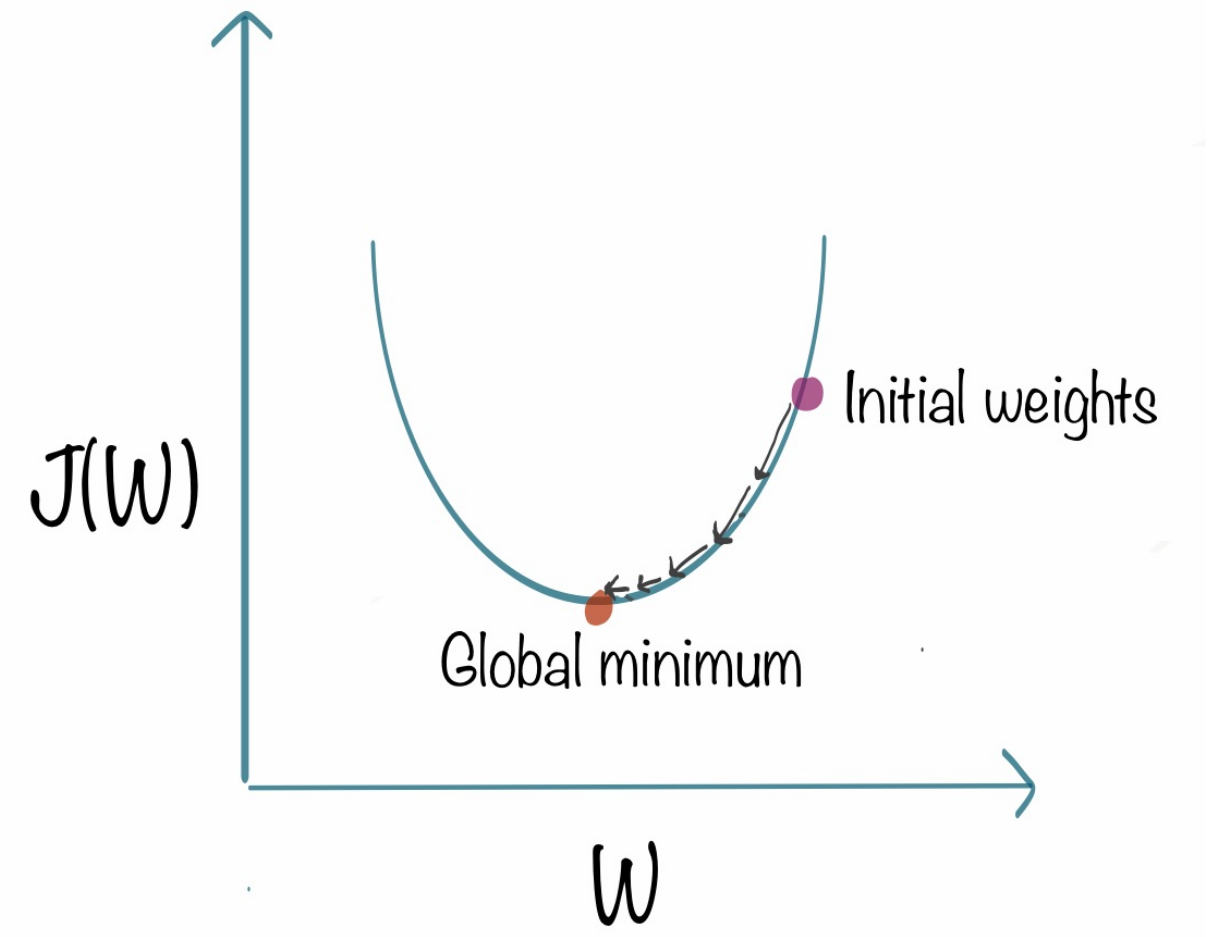
In simple terms, consider you are holding a ball resting at the top of a bowl. When you lose the ball, it goes along the steepest direction and eventually settles at the bottom of the bowl. A Gradient provides the ball in the steepest direction to reach the local minimum that is the bottom of the bowl.

$$x\_new = x - alpha * f'(x)$$

The above equation means how the gradient is calculated. Here alpha is step size that represents how far to move against each gradient with each iteration.

Gradient descent works as follows:

1. It starts with some coefficients, sees their cost, and searches for cost value lesser than what it is now.
2. It moves towards the lower weight and updates the value of the coefficients.
3. The process repeats until the local minimum is reached. A local minimum is a point beyond which it can not proceed.



KDnuggets

Gradient descent works best for most purposes. However, it has some downsides too. It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions but it doesn't know how far to travel along the gradient for nonconvex functions.

Stochastic Gradient Descent Deep Learning Optimizer

At the end of the previous section, you learned why using gradient descent on massive data might not be the best option. To tackle the problem, we have stochastic gradient descent. The term stochastic means randomness on which the algorithm is based upon. In stochastic gradient descent, instead of taking the whole dataset for each iteration, we randomly select

the batches of data. That means we only take few samples from the dataset.

$$w := w - \eta \nabla Q_i(w).$$

The procedure is first to select the initial parameters w and learning rate n. Then randomly shuffle the data at each iteration to reach an approximate minimum.

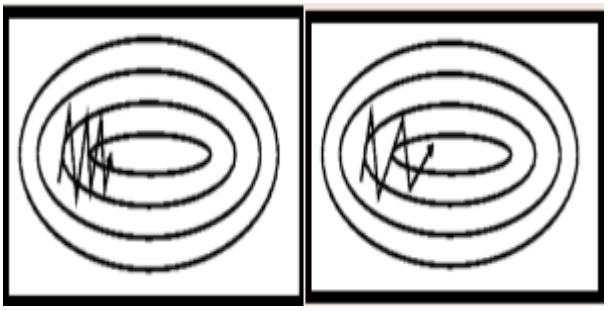
Since we are not using the whole dataset but the batches of it for each iteration, the path took by the algorithm is full of noise as compared to the gradient descent algorithm. Thus, SGD uses a higher number of iterations to reach the local minima. Due to an increase in the number of iterations, the overall computation time increases. But even after increasing the number of iterations, the computation cost is still less than that of the gradient descent optimizer. So the conclusion is if the data is enormous and computational time is an essential factor, stochastic gradient descent should be preferred over batch gradient descent algorithm.

Stochastic Gradient Descent with Momentum Deep Learning Optimizer

As discussed in the earlier section, you have learned that stochastic gradient descent takes a much more noisy path than the gradient descent algorithm. Due to this reason, it requires a more significant number of iterations to reach the optimal minimum and hence computation time is

very slow. To overcome the problem, we use stochastic gradient descent with a momentum algorithm.

What the momentum does is helps in faster convergence of the loss function. Stochastic gradient descent oscillates between either direction of the gradient and updates the weights accordingly. However, adding a fraction of the previous update to the current update will make the process a bit faster. One thing that should be remembered while using this algorithm is that the learning rate should be decreased with a high momentum term.



In the above image, the left part shows the convergence graph of the stochastic gradient descent algorithm. At the same time, the right side shows SGD with momentum. From the image, you can compare the path chosen by both the algorithms and realize that using momentum helps reach convergence in less time. You might be thinking of using a large momentum and learning rate to make the process even faster. But remember that while increasing the momentum, the possibility of passing the optimal minimum also increases. This might result in poor accuracy and even more oscillations.

Mini Batch Gradient Descent Deep Learning Optimizer

In this variant of gradient descent instead of taking all the training data, only a subset of the dataset is used for calculating the loss function. Since we are using a batch of data instead of taking the whole dataset, fewer iterations are needed. That is why the mini-batch gradient descent algorithm is faster than both stochastic gradient descent and batch gradient descent algorithms. This algorithm is more efficient and robust than the earlier variants of gradient descent. As the algorithm uses batching, all the training data need not be loaded in the memory, thus making the process more efficient to implement. Moreover, the cost function in mini-batch gradient descent is noisier than the batch gradient descent algorithm but smoother than that of the stochastic gradient descent algorithm. Because of this, mini-batch gradient descent is ideal and provides a good balance between speed and accuracy.

Despite, all that, the mini-batch gradient descent algorithm has some downsides too. It needs a hyperparameter that is “mini-batch-size”, which needs to be tuned to achieve the required accuracy. Although, the batch size of 32 is considered to be appropriate for almost every case. Also, in some cases, it results in poor final accuracy. Due to this, there needs a rise to look for other alternatives too.

Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer

The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different learning rates

for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get change, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}}$$

$$\eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithms and their variants, and it reaches convergence at a higher speed.

One downside of AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

RMS Prop(Root Mean Square) Deep Learning Optimizer

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn't been published but still very well know in the community. RMS prop is ideally an extension of the work RPPROP. RPPROP resolves the problem of varying gradients. The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea. RPPROP uses the sign of the gradient adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we're going in the right direction and hence increase the step size by a small fraction. Whereas, if they have opposite signs, we have to decrease the step size. Then we limit the step size, and now we can go for the weight update.

The problem with RPPROP is that it doesn't work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and efficiency of mini-batches at the same time was the main motivation behind the rise of RMS prop. RMS prop can also be considered an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate.

The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minima. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

$$v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2$$

where gamma is the forgetting factor. Weights are updated by the below formula

$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w)$$

In simpler terms, if there exists a parameter due to which the cost function oscillates a lot, we want to penalize the update of this parameter. Suppose you built a model to classify a variety of fishes. The model relies on the factor ‘color’ mainly to differentiate between the fishes. Due to which it makes a lot of errors. What RMS Prop does is, penalize the parameter ‘color’ so that it can rely on other features too. This prevents the algorithm from adapting too quickly to changes in the parameter ‘color’ compared to other parameters. This algorithm has several benefits as compared to earlier versions of gradient descent algorithms. The algorithm converges quickly and requires lesser tuning than gradient descent algorithms and their variants.

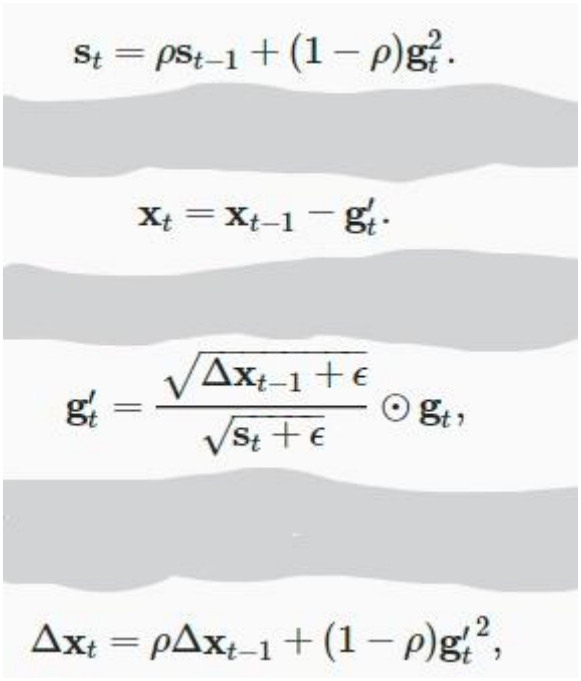
The problem with RMS Prop is that the learning rate has to be defined manually and the suggested value doesn’t work for every application.

AdaDelta Deep Learning Optimizer

AdaDelta can be seen as a more robust version of AdaGrad optimizer. It is based upon adaptive learning and is designed to deal with significant drawbacks of AdaGrad and RMS prop optimizer. The main problem with

the above two optimizers is that the initial learning rate must be defined manually. One other problem is the decaying learning rate which becomes infinitesimally small at some point. Due to which a certain number of iterations later, the model can no longer learn new knowledge.

To deal with these problems, AdaDelta uses two state variables to store the leaky average of the second moment gradient and a leaky average of the second moment of change of parameters in the model.



Here S_t and delta X_t denotes the state variables, g'_t denotes rescaled gradient, delta X_{t-1} denotes squares rescaled gradients, and epsilon represents a small positive integer to handle division by 0.

Adam Deep Learning Optimizer

The name adam is derived from adaptive moment estimation. This optimization algorithm is a further extension of stochastic gradient descent

to update network weights during training. Unlike maintaining a single learning rate through training in SGD, Adam optimizer updates the learning rate for each network weight individually. The creators of the Adam optimization algorithm know the benefits of AdaGrad and RMSProp algorithms, which are also extensions of the stochastic gradient descent algorithms. Hence the Adam optimizers inherit the features of both Adagrad and RMS prop algorithms. In adam, instead of adapting learning rates based upon the first moment(mean) as in RMS Prop, it also uses the second moment of the gradients. We mean the uncentred variance by the second moment of the gradients(we don't subtract the mean).

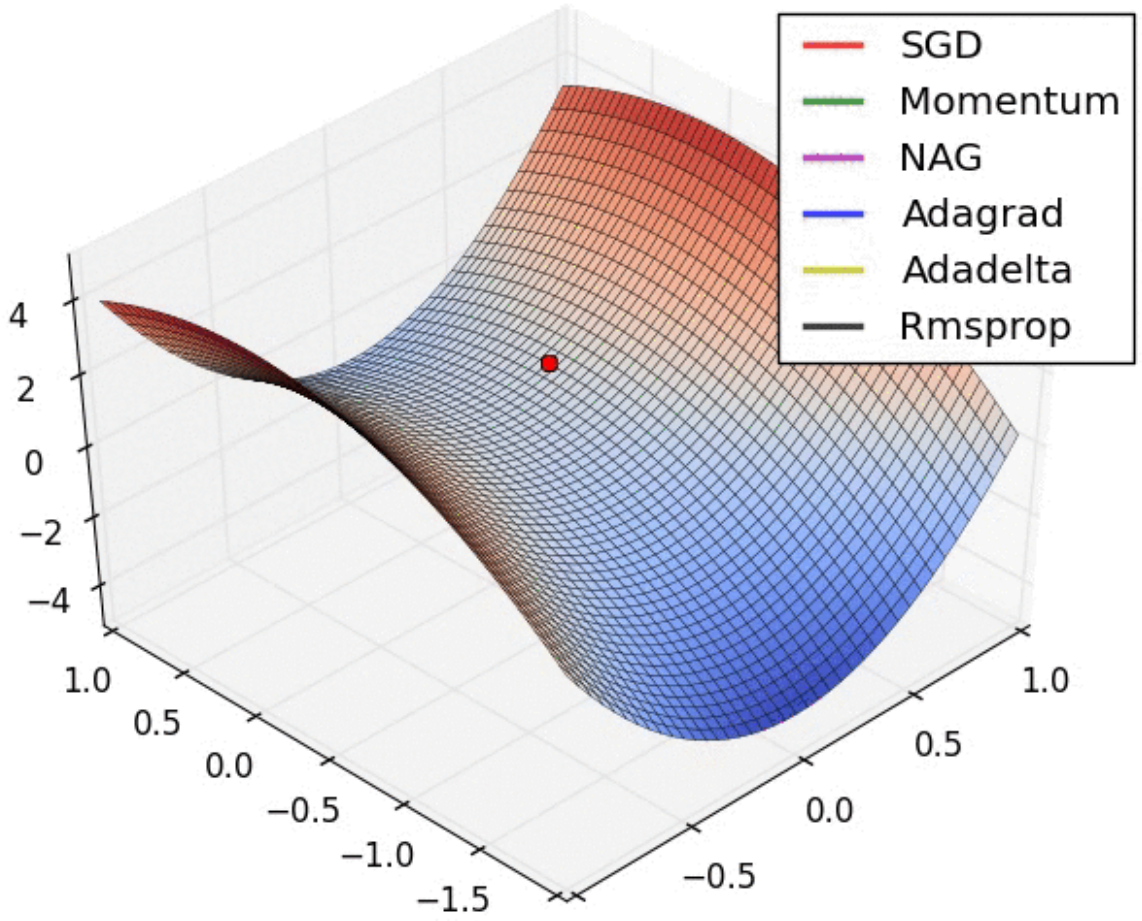
The adam optimizer has several benefits, due to which it is used widely. It is adapted as a benchmark for deep learning papers and recommended as a default optimization algorithm. Moreover, the algorithm is straightforward to implement, has faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left[\frac{\delta L}{\delta w_t}\right] v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\delta L}{\delta w_t}\right]^2$$

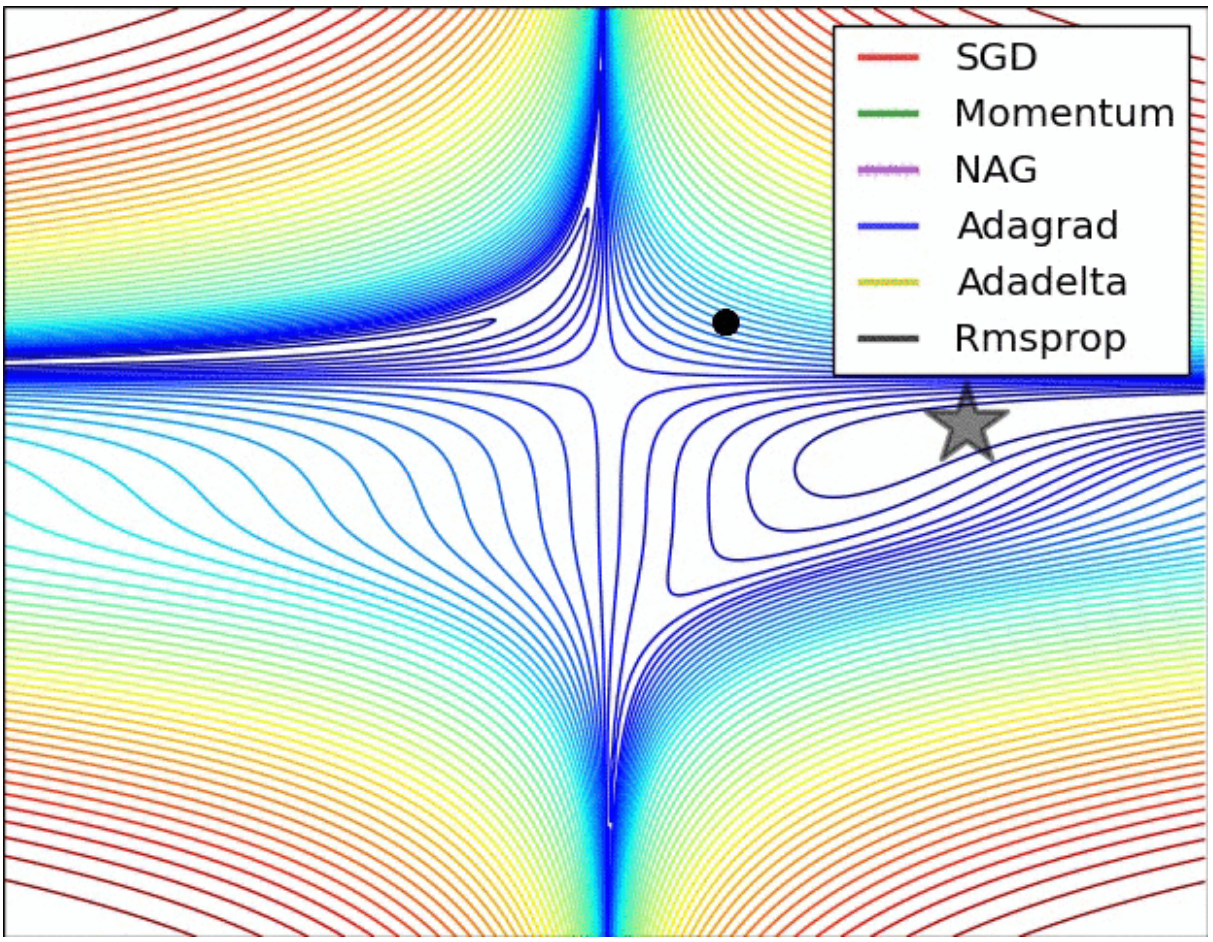
The above formula represents the working of adam optimizer. Here B1 and B2 represent the decay rate of the average of the gradients.

If the adam optimizer uses the good properties of all the algorithms and is the best available optimizer, then why shouldn't you use Adam in every application? And what was the need to learn about other algorithms in depth? This is because even Adam has some downsides. It tends to focus

on faster computation time, whereas algorithms like stochastic gradient descent focus on data points. That's why algorithms like SGD generalize the data in a better manner at the cost of low computation speed. So, the optimization algorithms can be picked accordingly depending upon the requirements and the type of data.



https://awesomeopensource.com/project/Jaewan-Yun/optimizer-visualization



https://awesomeopensource.com/project/Jaewan-Yun/optimizer-visualization

The above visualizations create a better picture in mind and help in comparing the results of various optimization algorithms.

EXP 6 transfer learning

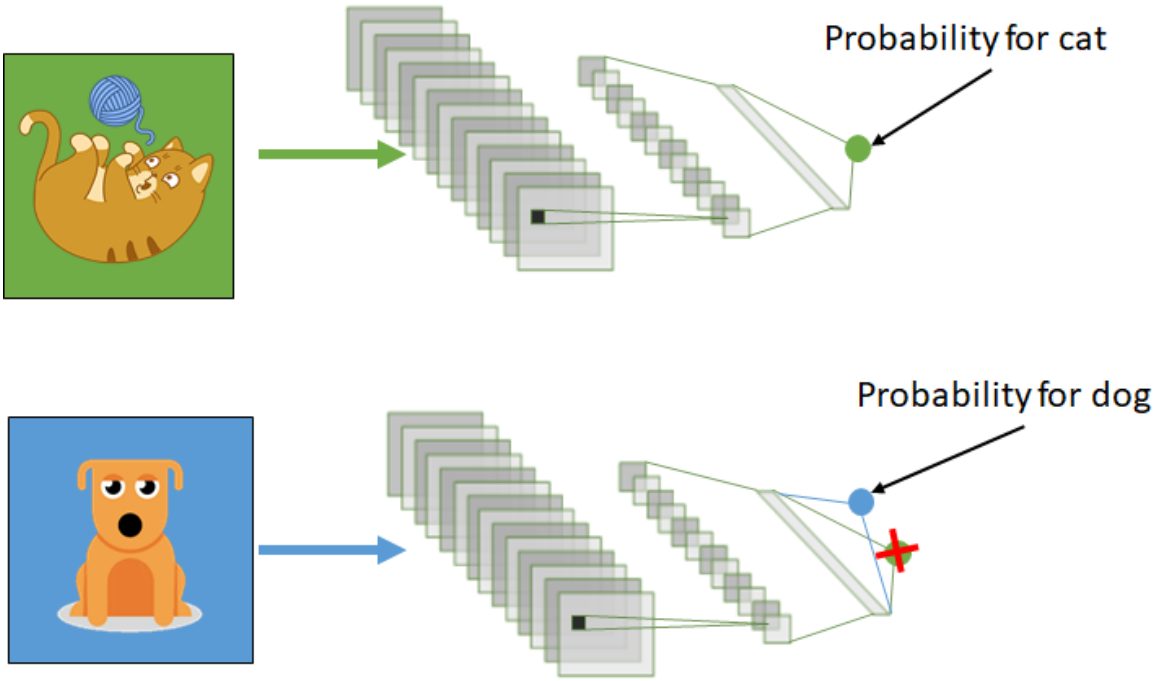
https://www.analyticsvidhya.com/blog/2021/10/understanding-transfer-learning-for-deep-learning/

EXP 7

What Is Transfer Learning and It's Working

The reuse of a pre-trained model on a new problem is known as transfer learning in machine learning. A machine uses the knowledge learned from a prior assignment to increase prediction about a new task in transfer learning. You could, for example, use the information gained during training to distinguish beverages when training a classifier to predict whether an image contains cuisine.

The knowledge of an already trained machine learning model is transferred to a different but closely linked problem throughout transfer learning. For example, if you trained a simple classifier to predict whether an image contains a backpack, you could use the model's training knowledge to identify other objects such as sunglasses.

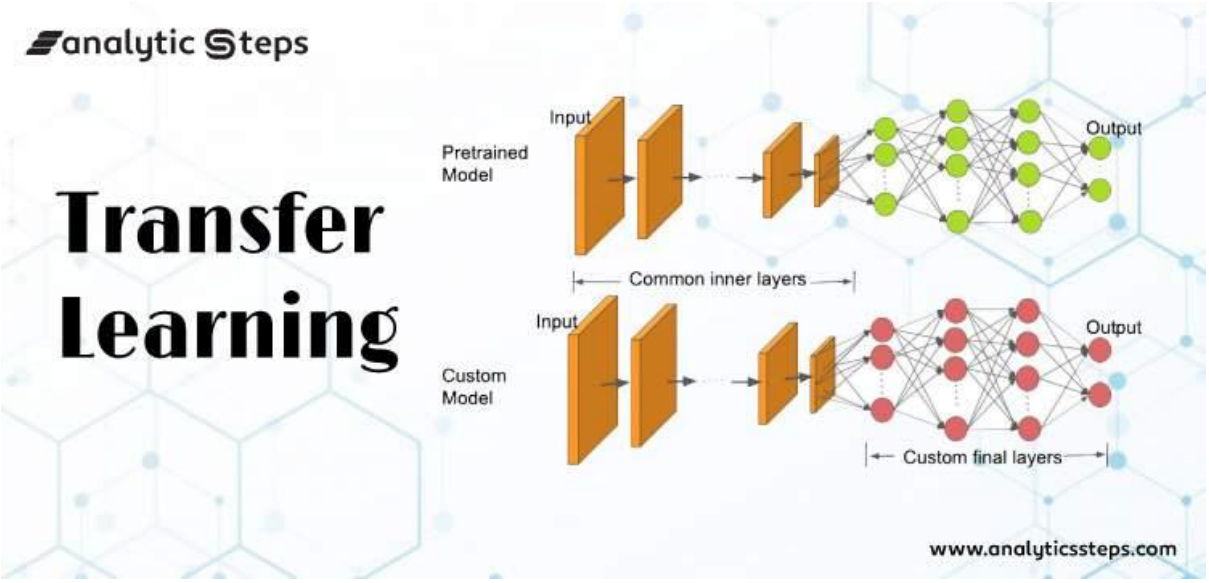


With transfer learning, we basically try to use what we've learned in one task to better understand the concepts in another. weights are being automatically being shifted to a network performing "task A" from a network that performed new "task B."

Because of the massive amount of CPU power required, transfer learning is typically applied in computer vision and natural language processing tasks like sentiment analysis.

How Transfer Learning Works

In computer vision, neural networks typically aim to detect edges in the first layer, forms in the middle layer, and task-specific features in the latter layers. The early and central layers are employed in transfer learning, and the latter layers are only retrained. It makes use of the labelled data from the task it was trained on.

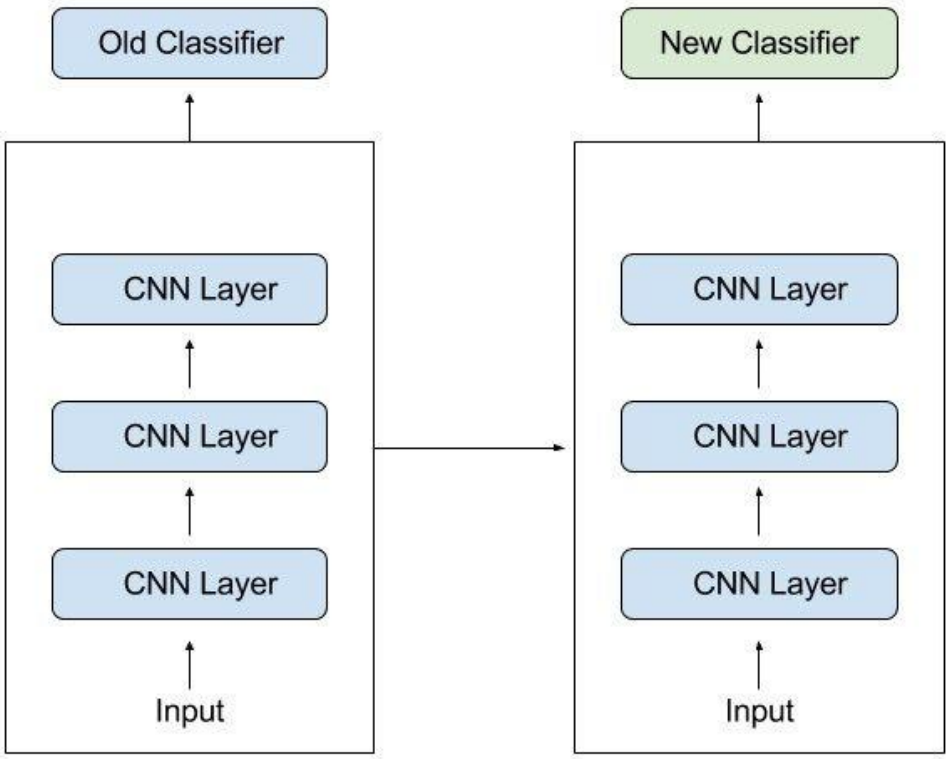


Let’s return to the example of a model that has been intended to identify a backpack in an image and will now be used to detect sunglasses. Because the model has trained to recognise objects in the earlier levels, we will simply retrain the subsequent layers to understand what distinguishes sunglasses from other objects.

Why Should You Use Transfer Learning?

Transfer learning offers a number of advantages, the most important of which are reduced training time, improved neural network performance (in most circumstances), and the absence of a large amount of data.

To train a neural model from scratch, a lot of data is typically needed, but access to that data isn't always possible – this is when transfer learning comes in handy.



Because the model has already been pre-trained, a good machine learning model can be generated with fairly little training data using transfer learning. This is especially useful in natural language processing, where huge labelled datasets require a lot of expert knowledge. Additionally,

training time is decreased because building a deep neural network from the start of a complex task can take days or even weeks.

When to Use Transfer Learning

When we don't have enough annotated data to train our model with. When there is a pre-trained model that has been trained on similar data and tasks. If you used TensorFlow to train the original model, you might simply restore it and retrain some layers for your job. Transfer learning, on the other hand, only works if the features learnt in the first task are general, meaning they can be applied to another activity. Furthermore, the model's input must be the same size as it was when it was first trained. If

If you don't have it, add a step to resize your input to the required size.

1. TRAINING A MODEL TO REUSE IT

Consider the situation in which you wish to tackle Task A but lack the necessary data to train a deep neural network. Finding a related task B with a lot of data is one method to get around this.

Utilize the deep neural network to train on task B and then use the model to solve task A. The problem you're seeking to solve will decide whether you need to employ the entire model or just a few layers.

If the input in both jobs is the same, you might reapply the model and make predictions for your new input. Changing and retraining distinct task-specific layers and the output layer, on the other hand, is an approach to investigate.

2. USING A PRE-TRAINED MODEL

The second option is to employ a model that has already been trained. There are a number of these models out there, so do some research beforehand. The number of layers to reuse and retrain is determined by the task.

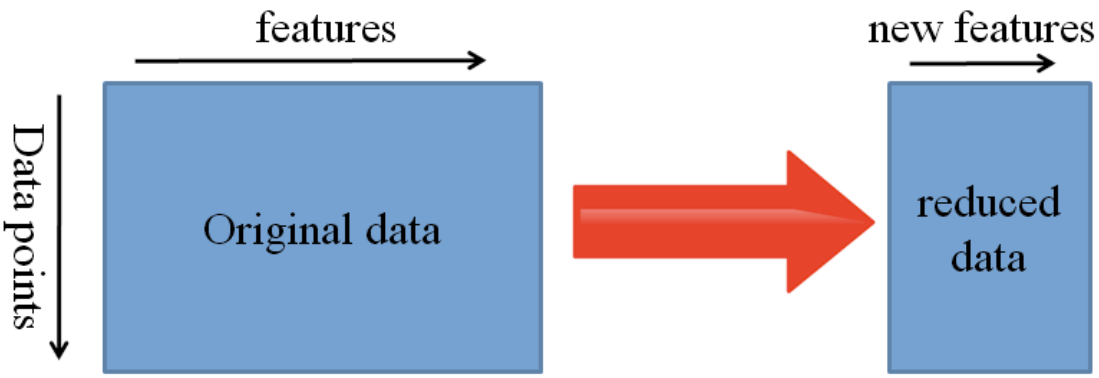
Keras consists of nine pre-trained models used in transfer learning, prediction, fine-tuning. These models, as well as some quick lessons on how to utilise them, may be found here. Many research institutions also make trained models accessible.

The most popular application of this form of transfer learning is deep learning.

3. EXTRACTION OF FEATURES

Another option is to utilise deep learning to identify the optimum representation of your problem, which comprises identifying the key features. This method is known as representation learning, and it can often produce significantly better results than hand-designed representations.

Feature creation in machine learning is mainly done by hand by researchers and domain specialists. Deep learning, fortunately, can extract features automatically. Of course, this does not diminish the importance of feature engineering and domain knowledge; you must still choose which features to include in your network.



Neural networks, on the other hand, have the ability to learn which features are critical and which aren't. Even for complicated tasks that would otherwise necessitate a lot of human effort, a representation learning algorithm can find a decent combination of characteristics in a short amount of time.

The learned representation can then be applied to a variety of other challenges. Simply utilise the initial layers to find the appropriate feature representation, but avoid using the network's output because it is too task-specific. Instead, send data into your network and output it through one of the intermediate layers.

The raw data can then be understood as a representation of this layer.

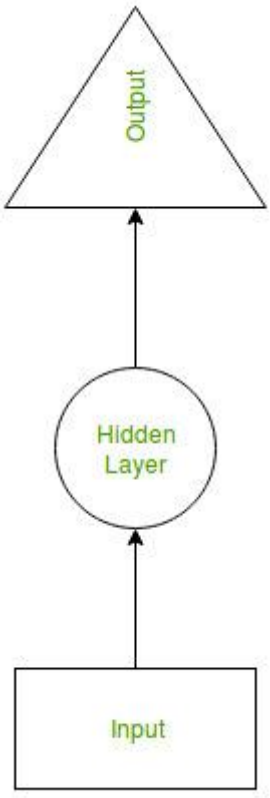
This method is commonly used in computer vision since it can shrink your dataset, reducing computation time and making it more suited for classical algorithms.

Models That Have Been Pre-Trained

There are a number of popular pre-trained machine learning models available. *The Inception-v3 model, which was developed for the ImageNet "Large Visual Recognition Challenge*," is one of them." Participants in this challenge had to categorize pictures into 1,000 subcategories such as "zebra," "Dalmatian," and "dishwasher."

EXP 7- RNN

Recurrent Neural Network(RNN) is a type of Neural Network where the **output from the previous step are fed as input to the current step**. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases like when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is **Hidden state**, which remembers some information about a sequence.

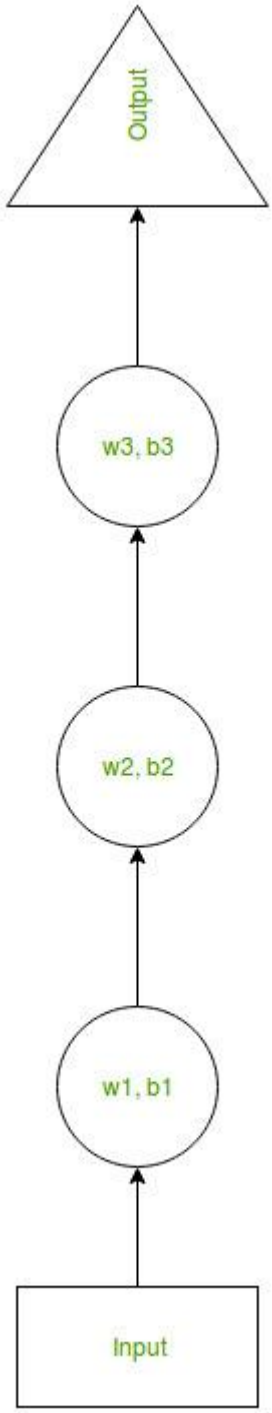


RNN have a **“memory”** which remembers all information about what has been calculated. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.

How RNN works

The working of an RNN can be understood with the help of the below example:

Example: Suppose there is a deeper network with one input layer, three hidden layers, and one output layer. Then like other neural networks, each hidden layer will have its own set of weights and biases, let’s say, for hidden layer 1 the weights and biases are (w1, b1), (w2, b2) for the second hidden layer, and (w3, b3) for the third hidden layer. This means that each of these layers is independent of the other, i.e. they do not memorize the previous outputs.

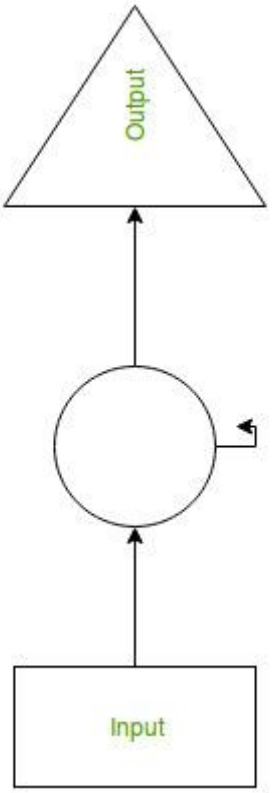


Now the RNN will do the following:

- RNN converts the independent activations into dependent activations by providing the same weights and biases to all the layers, thus reducing the complexity of increasing parameters and

memorizing each previous output by giving each output as input to the next hidden layer.

- Hence these three layers can be joined together such that the weights and bias of all the hidden layers are the same, in a single recurrent layer.



The formula for calculating current state: $h_t = f(h_{t-1}, x_t)$ where:

`ht -> current state`

`ht-1 -> previous state`

`xt -> input state`

Formula for applying Activation function(tanh):

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

where:

`Whh` -> weight at recurrent neuron

`Wxh` -> weight at input neuron

The formula for calculating output: $$y_t = W_{hy}h_t$$

`Yt` -> output

`Why` -> weight at output layer

Training through RNN

1. A single-time step of the input is provided to the network.
2. Then calculate its current state using a set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

Advantages of Recurrent Neural Network

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages of Recurrent Neural Network

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

Applications of Recurrent Neural Network

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting

EXP 8 - autoencoder

The Architecture of Encoder-Decoder models

My main goal is to help you understand the architecture used in the paper Sequence to Sequence Learning with Neural Networks by Ilya Sutskever, et al. This was one of the first papers to introduce the Encoder-Decoder model for machine translation and more generally sequence-to-sequence models. The model was applied to English to French translation.

NOTE: To get to the final model, I will attempt to simplify things and introduce the concepts one by one to hopefully make it easier for people to understand without too much in-depth knowledge of the subject matter, filling in the gaps as and when needed.

The Neural Machine Translation Problem

To help you understand better, I will be taking Neural Machine Translation as a running example throughout this post. In neural machine translation, the input is a series of words, processed one after another. The output is, likewise, a series of words.

Task: Predict the French translation for every English sentence used as input.

For simplicity let's consider that there is only a single sentence in our data corpus. So let our data be:

- **Input: English sentence: "nice to meet you"**
- **Output: French translation: "ravi de vous rencontrer"**

Terms Used

To avoid any confusion:

- I will refer to the Input sentence **"nice to meet you"** as **X/input-sequence**

- I will refer to the output sentence **“ravi de vous rencontrer”** as **Y_true/target-sequence** → This is what we want our model to predict (the ground truth).
- I will refer to the predicted output sentence of the model as **Y_pred/predicted-sequence**
- The individual words of the English and French sentence are referred to as **tokens**

Hence, given the input-sequence “nice to meet you”, we want our model to predict the target-sequence/Y_true i.e “ravi de vous rencontrer”

High-Level Overview

At a very high level, an encoder-decoder model can be thought of as two blocks, the encoder and the decoder connected by a vector which we will refer to as the ‘context vector’.

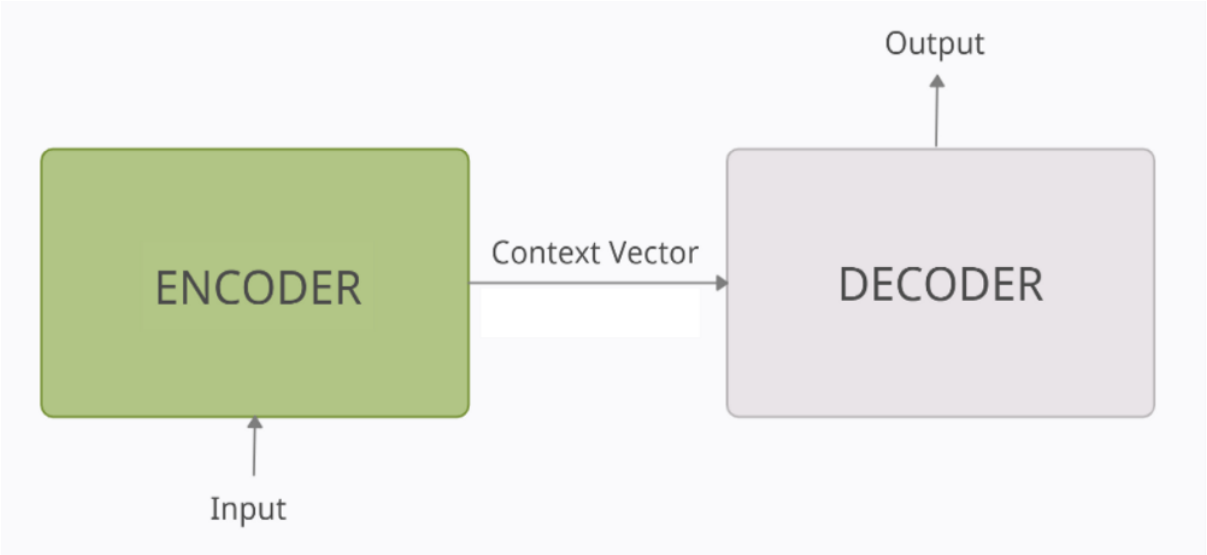


Image by author

- **Encoder**: The encoder processes each token in the input-sequence. It tries to cram all the information about the input-sequence into a vector of fixed length i.e. the ‘context vector’. After going through all the tokens, the encoder passes this vector onto the decoder.
- **Context vector**: The vector is built in such a way that it's expected to encapsulate the whole meaning of the input-sequence and help the decoder make accurate predictions. We will see later that this is the final internal states of our encoder block.
- **Decoder**: The decoder reads the context vector and tries to predict the target-sequence token by token.

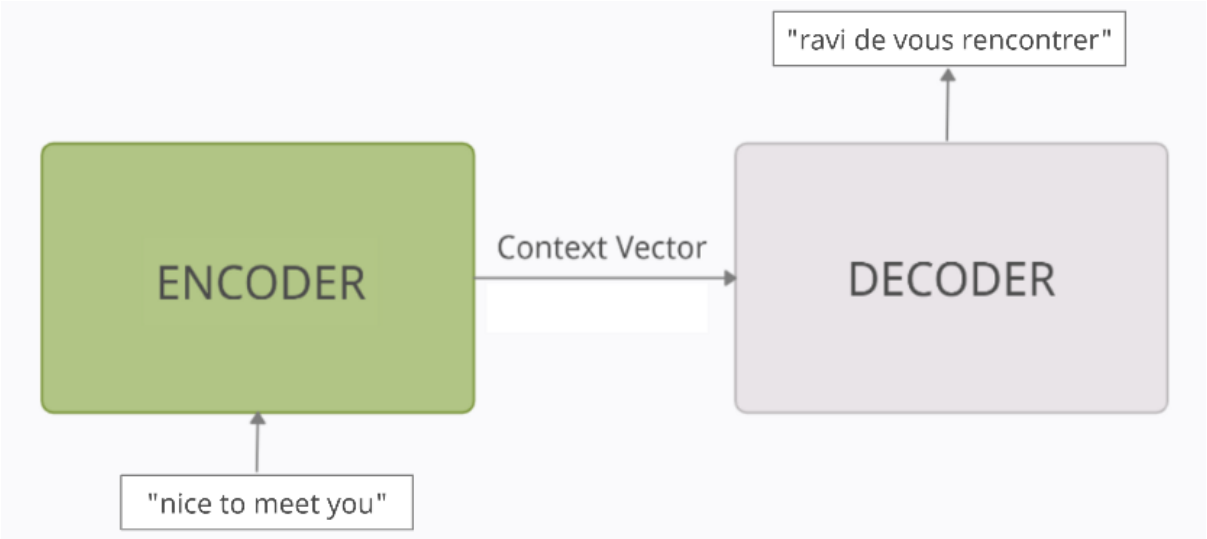


Image by author

What's under the hood?

The internal structure of both the blocks would look something like this:

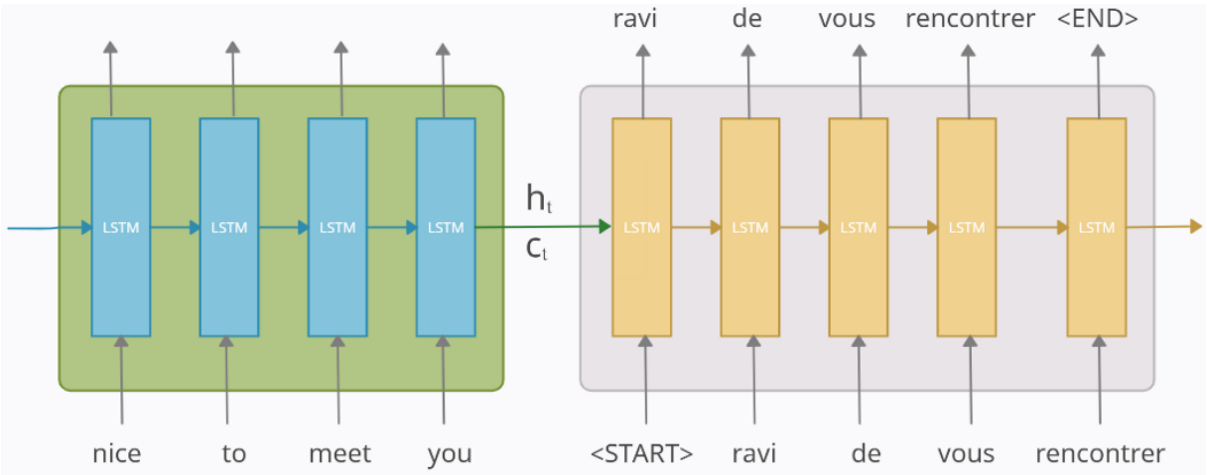


Image by author

As far as architecture is concerned, it's quite straightforward. The model can be thought of as two LSTM cells with some connection between them. The main thing here is how we deal with the inputs and the outputs. I will be explaining each part one by one.

The Encoder Block

The encoder part is an LSTM cell. It is fed in the input-sequence over time and it tries to encapsulate all its information and store it in its final internal states **h** (hidden state) and **c** (cell state). The internal states are then passed onto the decoder part, which it will use to try to produce the target-sequence. This is the 'context vector' which we were earlier referring to.

The outputs at each time-step of the encoder part are all discarded

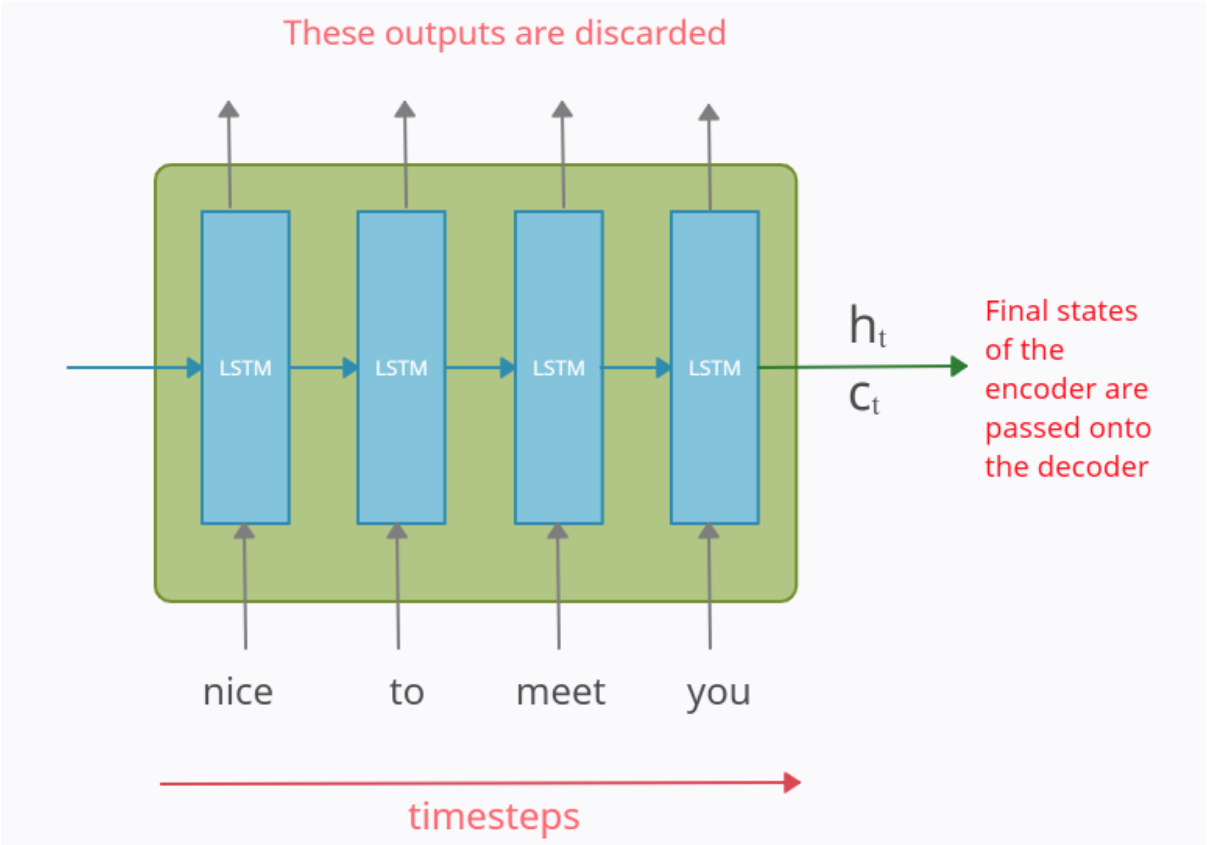


Image by author

Note: Above diagram is what an LSTM/GRU cell looks like when we unfold it on the time axis. i.e. it is the single LSTM/GRU cell that takes a single word/token at each timestamp.

In the paper, they have used LSTMs instead of classical RNNs because they work better with long-term dependencies. I have seen people use GRUs as well.

The Decoder Block

So after reading the whole input-sequence, the encoder passes the internal states to the decoder and this is where the prediction of output-sequence begins.

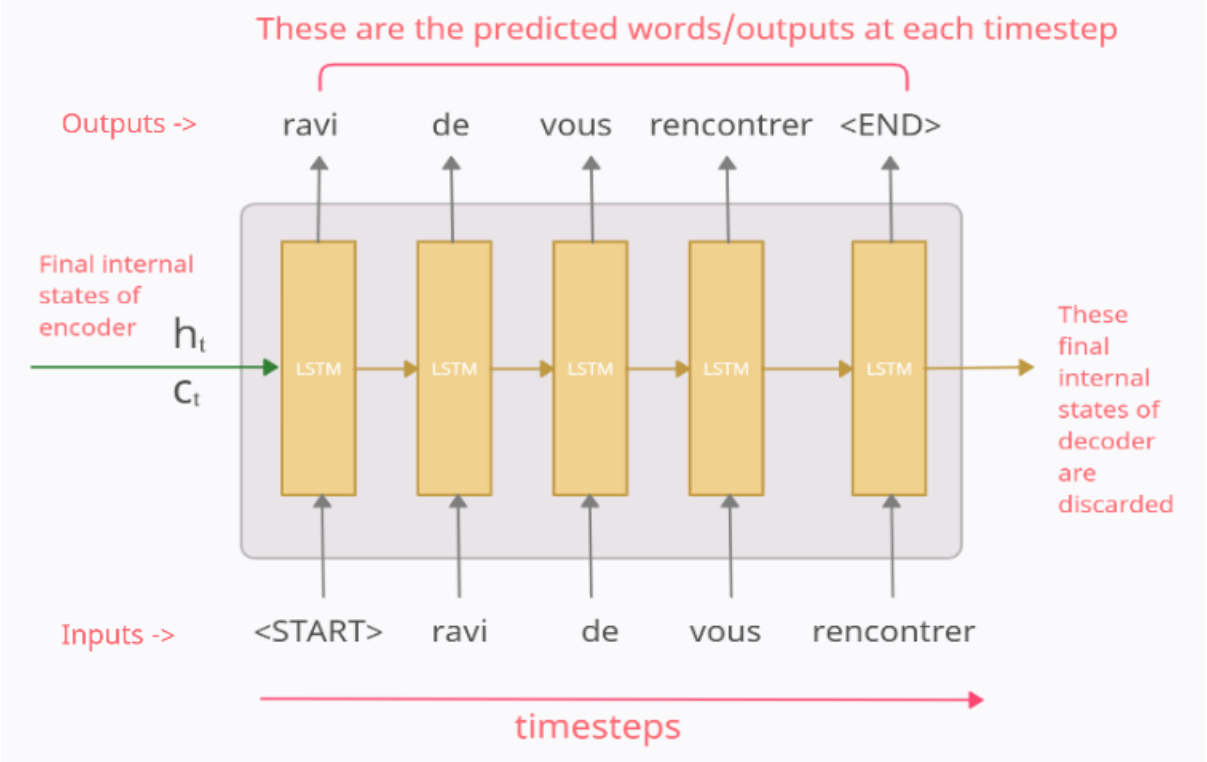


Image by author

The decoder block is also an LSTM cell. The main thing to note here is that the initial states **(h_0, c_0)** of the decoder are set to the final states **(h , c)** of the encoder. These act as the 'context' vector and help the decoder produce the desired target-sequence.

Now the way decoder works, is, that its output at any time-step **t** is supposed to be the **t^{th}** word in the target-sequence/Y_true ("ravi de vous rencontrer"). To explain this, let's see what happens at each time-step.

At time-step 1

The input fed to the decoder at the first time-step is a special symbol **"<START>"**. This is used to signify the start of the output-sequence. Now the decoder uses this input and the internal states **(h , c)** to produce the output in the 1st time-step which is supposed to be the 1st word/token in the target-sequence i.e. **'ravi'**.

At time-step 2

At time-step 2, the output from the 1st time-step **"ravi"** is fed as input to the 2nd time-step. The output in the 2nd time-step is supposed to be the 2nd word in the target-sequence i.e. **'de'**

And similarly, the output at each time-step is fed as input to the next time-step. This continues till we get the “<END>” symbol which is again a special symbol used to mark the end of the output-sequence. The final internal states of the decoder are discarded.

Note that these special symbols need not necessarily be “<START>” and “<END>” *only. These can be any strings given that these are not present in our data corpus so the model doesn’t confuse them with any other word. In the paper, they have used the symbol* “<EOS>” *and in a slightly different manner. I will be talking a little bit more about this later.*

NOTE: The process mentioned above is how an **ideal** decoder will work in the testing phase. But in the training phase, a slightly different implementation is required, to make it train faster. I have explained this in the next section.

Training and Testing Phase: Getting into the tensors

Vectorizing our data

Before getting into details of it, we first need to vectorize our data.

The raw data that we have is

- **X = “nice to meet you” → Y_true = “ravi de vous rencontrer”**

Now we put the special symbols “<START>” and “<END>” at the start and the end of our target-sequence

- **X = “nice to meet you” → Y_true = “<START> ravi de vous rencontrer <END>”**

Next the input and output data is vectorized using

one-hot-encoding (ohe). Let the input and output be represented as

- **X = (x1, x2, x3, x4) → Y_true = (y0_true, y1_true, y2_true, y3_true, y4_true, y5_true)**

where xi's and yi's represent the ohe vectors for input-sequence and output-sequence respectively. They can be shown as:

For input X

'nice' → x1 : [1 0 0 0]

'to' → x2 : [0 1 0 0]

'meet' →x3 : [0 0 1 0]

'you' → x4 : [0 0 0 1]

For Output Y_true

'<START>' → y0_true : [1 0 0 0 0 0]

'ravi' → y1_true : [0 1 0 0 0 0]

'de' → y2_true : [0 0 1 0 0 0]

‘vous’ → y3_true : [0 0 0 1 0 0]

‘rencontrer’ → y4_true : [0 0 0 0 1 0]

‘<END>’ → y5_true : [0 0 0 0 0 1]

Note: I have used this representation for easier explanation. Both the terms ‘true sequence’ and ‘target-sequence’ have been used to refer to the same sentence “ravi de vous rencontrer” which we want our model to learn.

Training and Testing of Encoder

The working of the encoder is the same in both the training and testing phase. It accepts each token/word of the input-sequence one by one and sends the final states to the decoder. Its parameters are updated using backpropagation over time.

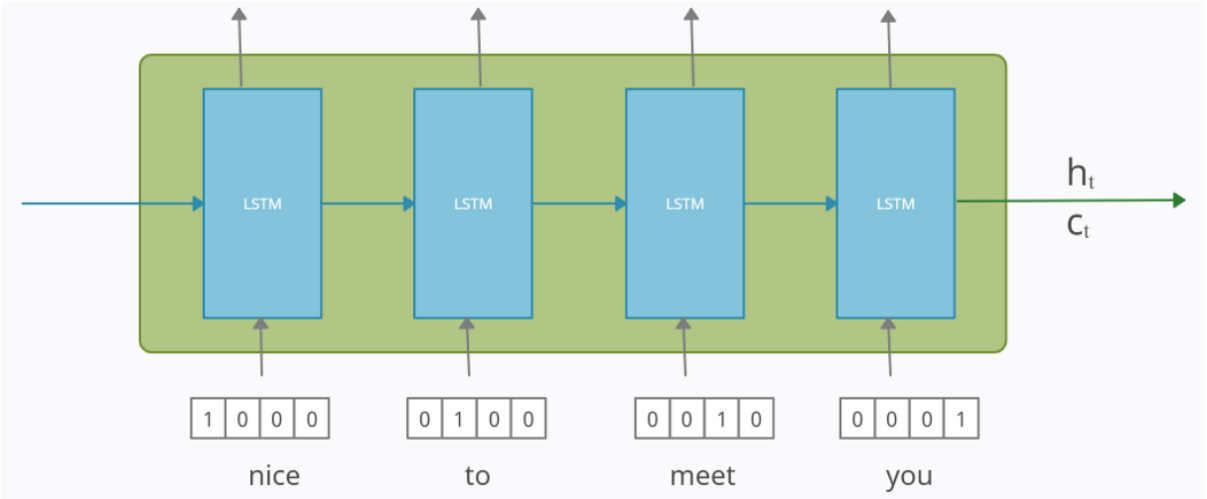


Image by author

The Decoder in Training Phase: Teacher Forcing

The working of the decoder is different during the training and testing phase, unlike the encoder part. Hence we will see both separately.

To train our decoder model, we use a technique called "**Teacher Forcing**" in which we feed the **true** output/token (and **not the predicted** output/token) from the previous time-step as input to the current time-step.

To explain, let's look at the 1st iteration of training. Here, we have fed our input-sequence to the encoder, which processes it and

passes its final internal states to the decoder. Now for the decoder part, refer to the diagram below.

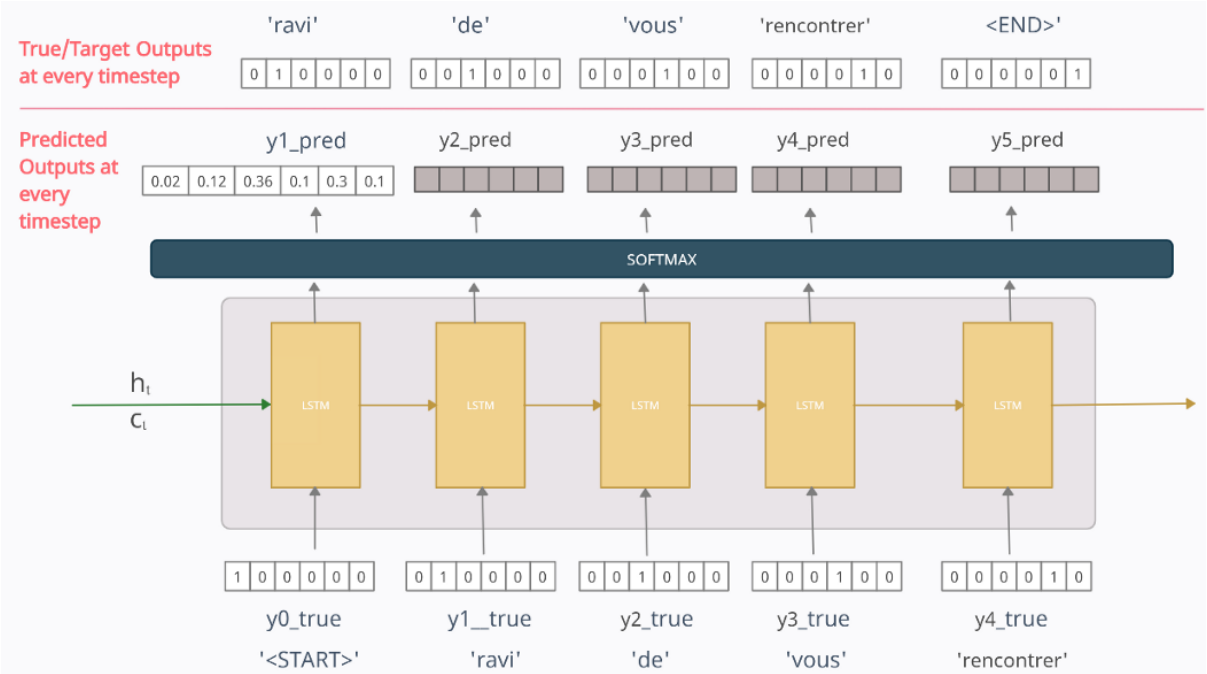


Image by author

Before moving on, note that in the decoder, at any time-step **t**, the output **yt_pred** is the probability distribution over the entire vocabulary in the output dataset which is generated by using the Softmax activation function. The token with the maximum probability is chosen to be the predicted word.

For eg. referring to the above diagram, y1_pred = [0.02 0.12 0.36 0.1 0.3 0.1] tells us that our model thinks that the probability of 1st token in the output-sequence being '<START>' is 0.02, 'ravi' is 0.12, 'de' is 0.36 and so on. We take the predicted word to be the one with the highest probability. Hence here the predicted word/token is **'de'** with a probability of 0.36

Moving on...

At time-step 1

The vector [1 0 0 0 0 0] for the word '<START>' is fed as the input vector. Now here I want my model to predict the output as y1_true=[0 1 0 0 0 0] but since my model has just started training, it will output something random. Let the predicted value at time-step 1 be y1_pred=[0.02 0.12 0.36 0.1 0.3 0.1] meaning it predicts the 1st token to be **'de'**. Now, should we use this y1_pred as the input at time-step 2?. We can do that, but in practice, it was

seen that this leads to problems like slow convergence, model instability, and poor skill which is quite logical if you think.

Thus, **teacher forcing** was introduced to rectify this. in which we feed the true output/token (and not the predicted output) from the previous time-step as input to the current time-step. That means the input to the time-step 2 will be y1_true=[0 1 0 0 0 0] and not y1_pred.

Now the output at time-step 2 will be some random vector y2_pred. But at time-step 3 we will be using input as y2_true=[0 0 1 0 0 0] and not y2_pred. Similarly at each time-step, we will use the true output from the previous time-step.

Finally, the loss is calculated on the predicted outputs from each time-step and the errors are backpropagated through time to update the parameters of the model. The loss function used is the categorical cross-entropy loss function between the

target-sequence/**Y_true** and the predicted-sequence/**Y_pred** such that

- Y_true = [y0_true, y1_true, y2_true, y3_true, y4_true, y5_true]
- Y_pred = ['<START>', y1_pred, y2_pred, y3_pred, y4_pred, y5_pred]

The final states of the decoder are discarded

The Decoder in Test Phase

In a real-world application, we won't have Y_true but only X. Thus we can't use what we did in the training phase as we don't have the target-sequence/Y_true. Thus when we are testing our model, the predicted output (and not the true output unlike the training phase) from the previous time-step is fed as input to the current time-step. Rest is all same as the training phase.

So let's say we have trained our model and now we test it on the single sentence we trained it on. **Now If we trained the model**

well and that too only on a single sentence then it should perform almost perfectly but for the sake of explanation say our model is not trained well or partially trained and now we test it. Let the scenario be depicted by the diagram below

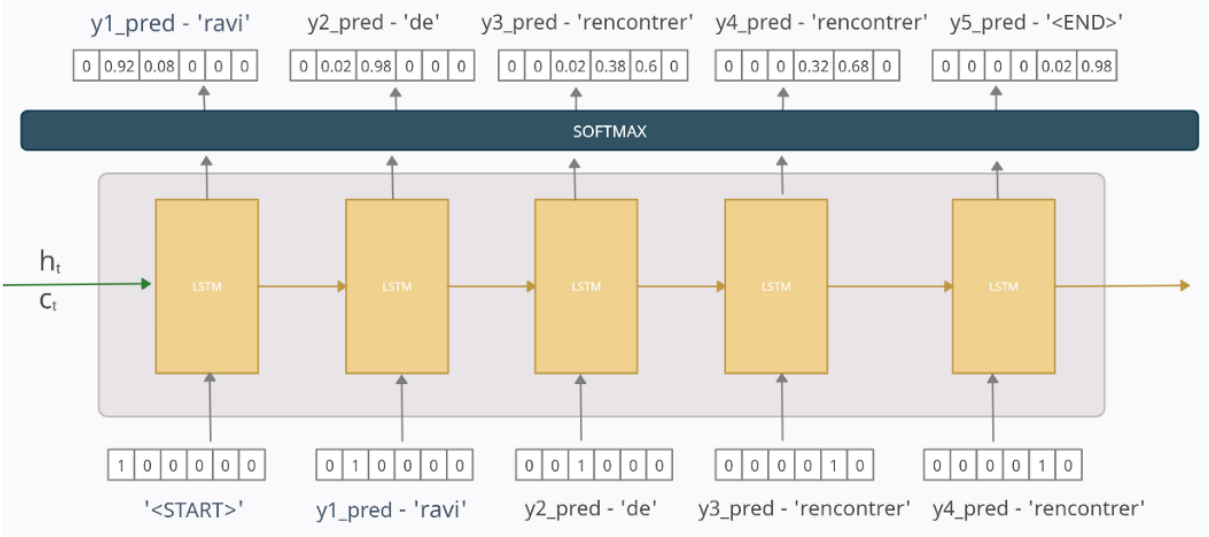


Image by author

At time-step 1

y1_pred = [0 0.92 0.08 0 0 0] tells that the model is predicting the 1st token/word in the output-sequence to be ‘ravi’ with a probability of 0.92 and so now at the next time-step this predicted word/token will only be used as input.

At time-step 2

The predicted word/token **“ravi”** from 1st time-step is used as input here. Here the model predicts the next word/token in the output-sequence to be **‘de’** with a probability of 0.98 which is then used as input at time-step 3

And the similar process is repeated at every time-step till the ‘<END>’ token is reached

Better visualization for the same would be:

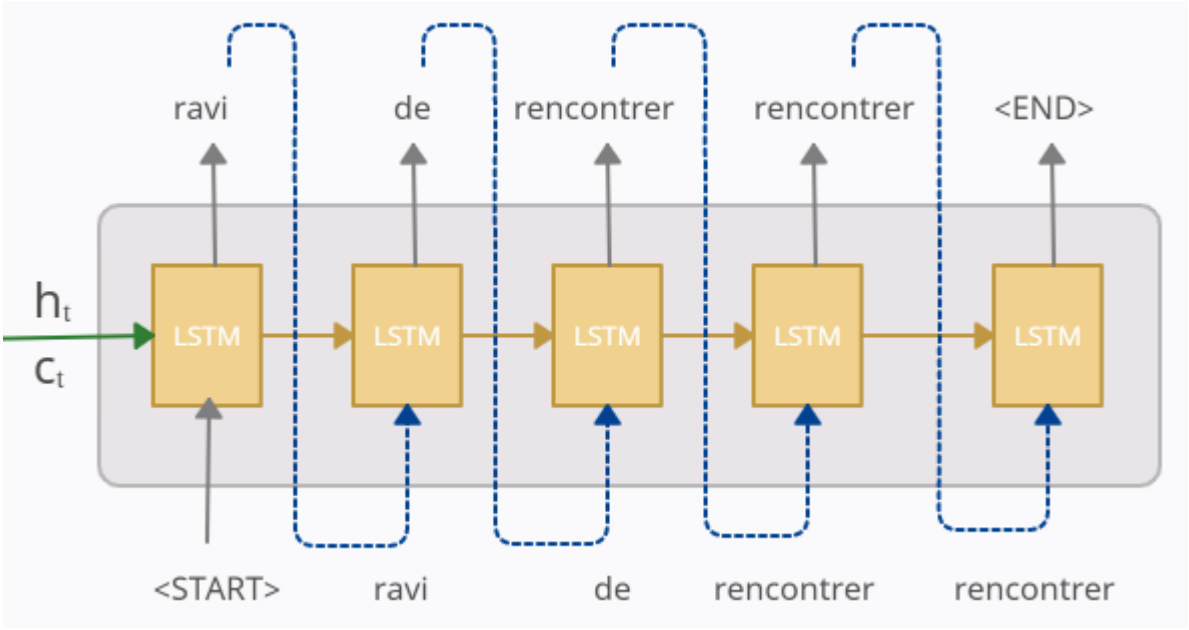


Image by author

So according to our trained model, the predicted-sequence at test time is “ravi de rencontrer rencontrer”. Hence though the model was incorrect on the 3rd prediction, we still fed it as input to the next time-step. The correctness of the model depends on the amount of data available and how well it has been trained. The model may predict a wrong output but nevertheless, the same output is only fed to the next time-step in the test phase.

The Embedding Layer

One little detail that I didn’t tell you before was that the input-sequence in both the decoder and the encoder is passed through an embedding layer to reduce the dimensions of the input word vectors because in practice, the one-hot-encoded vectors can be very large and the embedded vectors are a much better representation of words. For the encoder part, this can be shown below where the embedding layer reduces the dimensions of the word vectors from four to three.

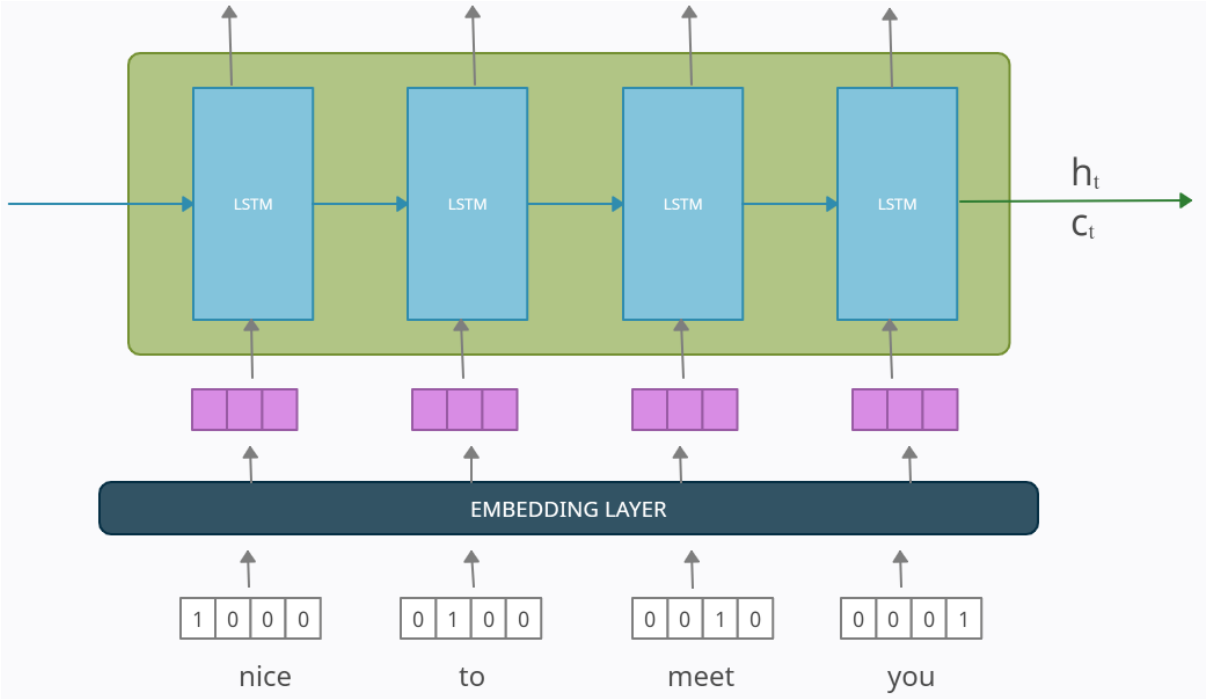


Image by author

This embedding layer can be pre-trained like Word2Vec embeddings or can be trained with the model itself.

The Final Visualization at test time

Somehow tried to capture everything in one diagram

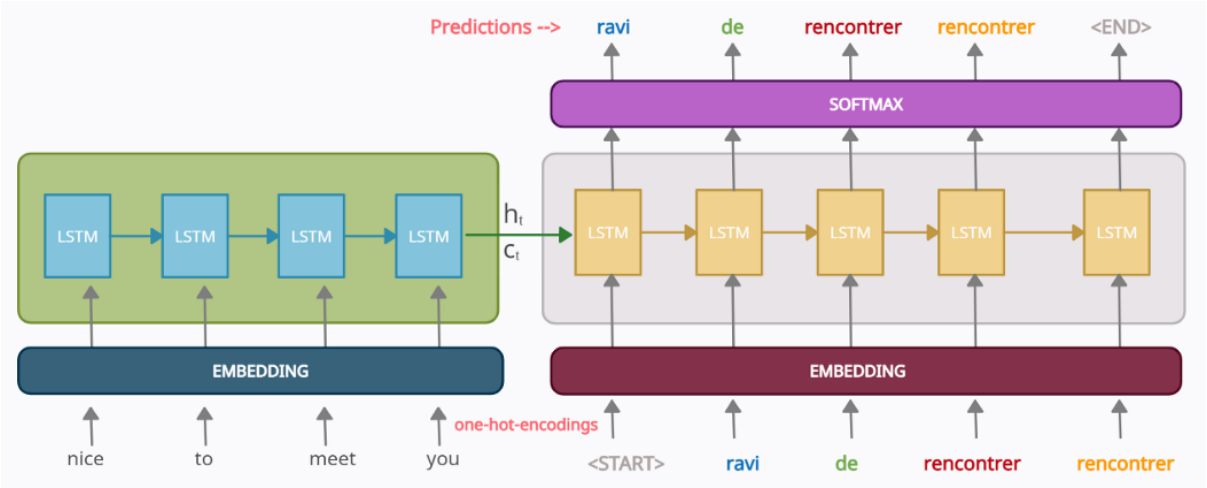


Image by author

You might find some slight differences in implementations here and there but the main idea remains the same.