1. Can you think of a few applications for a sequence-to-sequence RNN? What about a sequence-to-vector RNN, and a vector-to-sequence RNN?

In Sequence to Sequence Learning, RNN is trained to map an input sequence to an output sequence which is not necessarily of the same length. Applications are speech recognition, machine translation, image captioning and question answering.

In Sequence to Sequence Learning, RNN is trained to map an input sequence to an output sequence which is not necessarily of the same length.

Applications are speech recognition, machine translation, image captioning and question answering.

ARCHITECTURE

Encoder — Decoder Architecture

The Encoder RNN reads the input sequence and generates the ﬁxed-size context vector which represents a semantic summary of the input sequence.

The fixed-size context vector is given as input to the decoder RNN.

The fixed-size context can be provided as the initial state of the Decoder RNN, or it can be connected to the hidden units at each time step. These two ways can also be combined.

The number of time steps in the Encoder and Decoder need not to be equal.

Limitation

One limitation of this architecture is that it is difficult to summarize a long sequence with a context vector which has a small dimension.

2.How many dimensions must the inputs of an RNN layer have? What does each dimension represent? What about its outputs?

Before we get down to business, an important thing to note is that the RNN input needs to have 3 dimensions. Typically it would be batch size, the number of steps and number of features.

Researchers came up with neural networks to model the behaviour of a human brain. But if you actually think about it, normal neural networks don’t really do that much justice to its original intention. The reason for this statement is that feedforward vanilla neural networks cannot remember the things it learns. Each iteration you train the network it starts fresh, it doesn’t remember what it saw in the previous iteration when you are processing the current set of data. This is a big disadvantage when identifying correlations and data patterns. This is where Recurrent Neural Networks (RNN)came into the picture. RNNs have a very unique architecture that helps them to model memory units (hidden state) that enable them to persist data, thus being able to model short term dependencies. Due to this reason, RNNs are extensively used in time-series forecasting to identify data correlations and patterns.

Even though RNNs have been around for some time, everyone seems to have their own confusing way of explaining it’s architecture and no one really explains what happens behind the scenes. So let’s bridge the gap, shall we? This post is aimed at explaining the RNN architecture in a more granular level by going through its functionality.

Is this for you?

If you have blindly made simple RNN models using TensorFlow before and if you have been finding it hard to understand about what the inner workings of a RNN look like, then this article is just for you.

End Goal?

We will basically be explaining what happens behind the curtains when these two lines of TensorFlow code that are responsible for the declaration of the RNN and initiating the execution is run.

cell = tf.contrib.rnn.BasicRNNCell(rnn\_size,activation=tf.nn.tanh)  
  
val1, state = tf.nn.dynamic\_rnn(cell, inputs, dtype=tf.float32)

RNN Architecture

If you ever searched for architectural information about RNNs, the architecture diagrams you might get are rather confusing if you start looking into them as a beginner. I will use an example approach to explain the RNN architecture.

Before we get down to business, an important thing to note is that the RNN input needs to have 3 dimensions. Typically it would be batch size, the number of steps and number of features. The number of steps depicts the number of time steps/segments you will be feeding in one line of input of a batch of data that will be fed into the RNN.

The RNN unit in TensorFlow is called the “RNN cell”. This name itself has created a lot of confusion among people. There are many questions on Stackoverflow that inquire if “RNN cell” refers to one single cell or the whole layer. Well, it’s more like the whole layer. The reason for this is that the connections in RNNs are recurrent, thus following a “feeding to itself” approach. Basically, the RNN layer is comprised of a single rolled RNN cell that unrolls according to the “number of steps” value (number of time steps/segments) you provide.

As we mentioned earlier the main speciality in RNNs is the ability to model short term dependencies. This is due to the hidden state in the RNN. It retains information from one time step to another flowing through the unrolled RNN units. Each unrolled RNN unit has a hidden state. The current time steps hidden state is calculated using information of the previous time step’s hidden state and the current input. This process helps to retain information on what the model saw in the previous time step when processing the current time steps information. Also, something to note is that all the connections in RNN have weights and biases. The biases can be optional in some architectures. This process will be explained further in later parts of the article.

Since you now have a basic idea, let’s break down the execution process with an example. Say your batch size is 6, RNN size is 7, the number of time steps/segments you would include in one input line is 5 and the number of features in one time step is 3. If this is the case, your input tensor (matrix) shape for one batch would look something like this:

Tensor shape of one batch = (6,5,3)

Processing a batch:

When you feed a batch of data into the RNN cell it starts the processing from the 1st line of input. Likewise, the RNN cell will sequentially process all the input lines in the batch of data that was fed and give one output at the end which includes all the outputs of all the input lines.

Processing a single line of input:

In order to process a line of input, the RNN cell unrolls “number of steps” times. You can see this in the above figure (Fig 03). Since we defined “number of steps” as 5, the RNN cell has been unrolled 5 times.

The execution process is as follows:

First, the initial hidden state (S), which is typically a vector of zeros and the hidden state weight (h) is multiplied and then the hidden state bias is added to the result. In the meantime, the input at the time step t ([1,2,3]) and the input weight (i) is multiplied and the input bias is added to that result. We can obtain the hidden state at time step t by sending the addition of the above two results through an activation function, typically tanh (f).

Then, to obtain the output at time step t, the hidden state (S) at time step t is multiplied by the output weight (O) at time step t and then the output bias is added to the result.

When calculating the hidden state at time step t+1, the hidden state (S) at time step t is multiplied by the hidden state weight (h) and the hidden state bias is added to the result. Then as mentioned before the input at time step t+1 ([4,5,6]) will get multiplied by the input weight (i) and the input bias will be added to the result. These two results will then be sent through an activation function, typically tanh (f).

Then, to obtain the output at time step t+1, the hidden state (S) at time step t+1 is multiplied by the output weight (O) at time step t+1 and then the output bias is added to the result. As you can see, when producing the output of time step t+1 it not only uses the input data of time step t+1 but also uses information of data in time step t via the hidden state at time step t+1.

This process will repeat for all the time steps

After processing all time steps in one line of input in the batch, we will have 5 outputs of shape (1,7). So when all these outputs are concatenated together. the shape becomes (1,5,7). When all the input lines of the batch are done processing we get 6 outputs of size (1,5,7). Thus, the final output of the whole batch would be (6,5,7).

Note: All the hidden state weights, output weights and input weights have the same value throughout all the connections in a RNN.

Coming back to the 2 lines of code we stated earlier:

cell = tf.contrib.rnn.BasicRNNCell(rnn\_size,activation=tf.nn.tanh)  
  
val1, state = tf.nn.dynamic\_rnn(cell, inputs, dtype=tf.float32)

The 1st line basically defines the activation function and the RNN size of the RNN cell that we want to create. The 2nd line executes the processing procedure of the input data by feeding it into the RNN. The processing will happen according to what we discussed earlier. Finally, the output (value with shape (6,5,7) ) of that batch will be assigned to the “val1” variable. The final value of the hidden state will be assigned to the “state” variable.

We have now come to the end of the article. In this article, we discussed the data manipulation and representation process inside of a RNN in TensorFlow. With all the provided information, I hope that now you have a good understanding of how RNNs work in TensorFlow.

3.If you want to build a deep sequence-to-sequence RNN, which RNN layers should have return\_sequences=True? What about a sequence-to-vector RNN?

Sequence-to-sequence learning (Seq2Seq) is about training models to convert sequences from one domain (e.g. sentences in English) to sequences in another domain (e.g. the same sentences translated to French).

"the cat sat on the mat" -> [Seq2Seq model] -> "le chat etait assis sur le tapis"

This can be used for machine translation or for free-from question answering (generating a natural language answer given a natural language question) -- in general, it is applicable any time you need to generate text.

There are multiple ways to handle this task, either using RNNs or using 1D convnets. Here we will focus on RNNs.

The trivial case: when input and output sequences have the same length

When both input sequences and output sequences have the same length, you can implement such models simply with a Keras LSTM or GRU layer (or stack thereof). This is the case in [this example script](https://github.com/fchollet/keras/blob/master/examples/addition_rnn.py) that shows how to teach a RNN to learn to add numbers, encoded as character strings:

One caveat of this approach is that it assumes that it is possible to generate target[...t] given input[...t]. That works in some cases (e.g. adding strings of digits) but does not work for most use cases. In the general case, information about the entire input sequence is necessary in order to start generating the target sequence.

The general case: canonical sequence-to-sequence

In the general case, input sequences and output sequences have different lengths (e.g. machine translation) and the entire input sequence is required in order to start predicting the target. This requires a more advanced setup, which is what people commonly refer to when mentioning "sequence to sequence models" with no further context. Here's how it works:

A RNN layer (or stack thereof) acts as "encoder": it processes the input sequence and returns its own internal state. Note that we discard the outputs of the encoder RNN, only recovering the state. This state will serve as the "context", or "conditioning", of the decoder in the next step.

Another RNN layer (or stack thereof) acts as "decoder": it is trained to predict the next characters of the target sequence, given previous characters of the target sequence. Specifically, it is trained to turn the target sequences into the same sequences but offset by one timestep in the future, a training process called "teacher forcing" in this context. Importantly, the encoder uses as initial state the state vectors from the encoder, which is how the decoder obtains information about what it is supposed to generate. Effectively, the decoder learns to generate targets[t+1...] given targets[...t], conditioned on the input sequence.

In inference mode, i.e. when we want to decode unknown input sequences, we go through a slightly different process:

1) Encode the input sequence into state vectors.

2) Start with a target sequence of size 1 (just the start-of-sequence character).

3) Feed the state vectors and 1-char target sequence to the decoder to produce predictions for the next character.

4) Sample the next character using these predictions (we simply use argmax).

5) Append the sampled character to the target sequence

6) Repeat until we generate the end-of-sequence character or we hit the character limit.

The same process can also be used to train a Seq2Seq network without "teacher forcing", i.e. by reinjecting the decoder's predictions into the decoder.

A Keras example

Let's illustrate these ideas with actual code.

For our example implementation, we will use a dataset of pairs of English sentences and their French translation, which you can download from [manythings.org/anki](http://www.manythings.org/anki/). The file to download is called fra-eng.zip. We will implement a character-level sequence-to-sequence model, processing the input character-by-character and generating the output character-by-character. Another option would be a word-level model, which tends to be more common for machine translation. At the end of this post, you will find some notes about turning our model into a word-level model using Embedding layers.

The full script for our example [can be found on GitHub](https://github.com/fchollet/keras/blob/master/examples/lstm_seq2seq.py).

Here's a summary of our process:

1) Turn the sentences into 3 Numpy arrays, encoder\_input\_data, decoder\_input\_data, decoder\_target\_data:

encoder\_input\_data is a 3D array of shape (num\_pairs, max\_english\_sentence\_length, num\_english\_characters) containing a one-hot vectorization of the English sentences.

decoder\_input\_data is a 3D array of shape (num\_pairs, max\_french\_sentence\_length, num\_french\_characters) containg a one-hot vectorization of the French sentences.

decoder\_target\_data is the same as decoder\_input\_data but offset by one timestep. decoder\_target\_data[:, t, :] will be the same as decoder\_input\_data[:, t + 1, :].

2) Train a basic LSTM-based Seq2Seq model to predict decoder\_target\_data given encoder\_input\_data and decoder\_input\_data. Our model uses teacher forcing.

3) Decode some sentences to check that the model is working (i.e. turn samples from encoder\_input\_data into corresponding samples from decoder\_target\_data).

Because the training process and inference process (decoding sentences) are quite different, we use different models for both, albeit they all leverage the same inner layers.

This is our training model. It leverages three key features of Keras RNNs:

The return\_state contructor argument, configuring a RNN layer to return a list where the first entry is the outputs and the next entries are the internal RNN states. This is used to recover the states of the encoder.

The inital\_state call argument, specifying the initial state(s) of a RNN. This is used to pass the encoder states to the decoder as initial states.

The return\_sequences constructor argument, configuring a RNN to return its full sequence of outputs (instead of just the last output, which the defaults behavior). This is used in the decoder.

from keras.models import Model

from keras.layers import Input, LSTM, Dense

# Define an input sequence and process it.

encoder\_inputs = Input(shape=(None, num\_encoder\_tokens))

encoder = LSTM(latent\_dim, return\_state=True)

encoder\_outputs, state\_h, state\_c = encoder(encoder\_inputs)

# We discard `encoder\_outputs` and only keep the states.

encoder\_states = [state\_h, state\_c]

# Set up the decoder, using `encoder\_states` as initial state.

decoder\_inputs = Input(shape=(None, num\_decoder\_tokens))

# We set up our decoder to return full output sequences,

# and to return internal states as well. We don't use the

# return states in the training model, but we will use them in inference.

decoder\_lstm = LSTM(latent\_dim, return\_sequences=True, return\_state=True)

decoder\_outputs, \_, \_ = decoder\_lstm(decoder\_inputs,

initial\_state=encoder\_states)

decoder\_dense = Dense(num\_decoder\_tokens, activation='softmax')

decoder\_outputs = decoder\_dense(decoder\_outputs)

# Define the model that will turn

# `encoder\_input\_data` & `decoder\_input\_data` into `decoder\_target\_data`

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

We train our model in two lines, while monitoring the loss on a held-out set of 20% of the samples.

# Run training

model.compile(optimizer='rmsprop', loss='categorical\_crossentropy')

model.fit([encoder\_input\_data, decoder\_input\_data], decoder\_target\_data,

batch\_size=batch\_size,

epochs=epochs,

validation\_split=0.2)

After one hour or so on a MacBook CPU, we are ready for inference. To decode a test sentence, we will repeatedly:

1) Encode the input sentence and retrieve the initial decoder state

2) Run one step of the decoder with this initial state and a "start of sequence" token as target. The output will be the next target character.

3) Append the target character predicted and repeat.

Here's our inference setup:

encoder\_model = Model(encoder\_inputs, encoder\_states)

decoder\_state\_input\_h = Input(shape=(latent\_dim,))

decoder\_state\_input\_c = Input(shape=(latent\_dim,))

decoder\_states\_inputs = [decoder\_state\_input\_h, decoder\_state\_input\_c]

decoder\_outputs, state\_h, state\_c = decoder\_lstm(

decoder\_inputs, initial\_state=decoder\_states\_inputs)

decoder\_states = [state\_h, state\_c]

decoder\_outputs = decoder\_dense(decoder\_outputs)

decoder\_model = Model(

[decoder\_inputs] + decoder\_states\_inputs,

[decoder\_outputs] + decoder\_states)

We use it to implement the inference loop described above:

def decode\_sequence(input\_seq):

# Encode the input as state vectors.

states\_value = encoder\_model.predict(input\_seq)

# Generate empty target sequence of length 1.

target\_seq = np.zeros((1, 1, num\_decoder\_tokens))

# Populate the first character of target sequence with the start character.

target\_seq[0, 0, target\_token\_index['\t']] = 1.

# Sampling loop for a batch of sequences

# (to simplify, here we assume a batch of size 1).

stop\_condition = False

decoded\_sentence = ''

while not stop\_condition:

output\_tokens, h, c = decoder\_model.predict(

[target\_seq] + states\_value)

# Sample a token

sampled\_token\_index = np.argmax(output\_tokens[0, -1, :])

sampled\_char = reverse\_target\_char\_index[sampled\_token\_index]

decoded\_sentence += sampled\_char

# Exit condition: either hit max length

# or find stop character.

if (sampled\_char == '\n' or

len(decoded\_sentence) > max\_decoder\_seq\_length):

stop\_condition = True

# Update the target sequence (of length 1).

target\_seq = np.zeros((1, 1, num\_decoder\_tokens))

target\_seq[0, 0, sampled\_token\_index] = 1.

# Update states

states\_value = [h, c]

return decoded\_sentence

We get some nice results -- unsurprising since we are decoding samples taken from the training test.

Input sentence: Be nice.

Decoded sentence: Soyez gentil !

-

Input sentence: Drop it!

Decoded sentence: Laissez tomber !

-

Input sentence: Get out!

Decoded sentence: Sortez !

This concludes our ten-minute introduction to sequence-to-sequence models in Keras. Reminder: the full code for this script [can be found on GitHub](https://github.com/fchollet/keras/blob/master/examples/lstm_seq2seq.py).

References

[Sequence to Sequence Learning with Neural Networks](https://arxiv.org/abs/1409.3215)

[Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation](https://arxiv.org/abs/1406.1078)

Bonus FAQ

What if I want to use a GRU layer instead of a LSTM?

It's actually a bit simpler, because GRU has only one state, whereas LSTM has two states. Here's how to adapt the training model to use a GRU layer:

encoder\_inputs = Input(shape=(None, num\_encoder\_tokens))

encoder = GRU(latent\_dim, return\_state=True)

encoder\_outputs, state\_h = encoder(encoder\_inputs)

decoder\_inputs = Input(shape=(None, num\_decoder\_tokens))

decoder\_gru = GRU(latent\_dim, return\_sequences=True)

decoder\_outputs = decoder\_gru(decoder\_inputs, initial\_state=state\_h)

decoder\_dense = Dense(num\_decoder\_tokens, activation='softmax')

decoder\_outputs = decoder\_dense(decoder\_outputs)

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

What if I want to use a word-level model with integer sequences?

What if your inputs are integer sequences (e.g. representing sequences of words, encoded by their index in a dictionary)? You can embed these integer tokens via an Embedding layer. Here's how:

# Define an input sequence and process it.

encoder\_inputs = Input(shape=(None,))

x = Embedding(num\_encoder\_tokens, latent\_dim)(encoder\_inputs)

x, state\_h, state\_c = LSTM(latent\_dim,

return\_state=True)(x)

encoder\_states = [state\_h, state\_c]

# Set up the decoder, using `encoder\_states` as initial state.

decoder\_inputs = Input(shape=(None,))

x = Embedding(num\_decoder\_tokens, latent\_dim)(decoder\_inputs)

x = LSTM(latent\_dim, return\_sequences=True)(x, initial\_state=encoder\_states)

decoder\_outputs = Dense(num\_decoder\_tokens, activation='softmax')(x)

# Define the model that will turn

# `encoder\_input\_data` & `decoder\_input\_data` into `decoder\_target\_data`

model = Model([encoder\_inputs, decoder\_inputs], decoder\_outputs)

# Compile & run training

model.compile(optimizer='rmsprop', loss='categorical\_crossentropy')

# Note that `decoder\_target\_data` needs to be one-hot encoded,

# rather than sequences of integers like `decoder\_input\_data`!

model.fit([encoder\_input\_data, decoder\_input\_data], decoder\_target\_data,

batch\_size=batch\_size,

epochs=epochs,

validation\_split=0.2)

What if I don't want to use teacher forcing for training?

In some niche cases you may not be able to use teacher forcing, because you don't have access to the full target sequences, e.g. if you are doing online training on very long sequences, where buffering complete input-target pairs would be impossible. In that case, you may want to do training by reinjecting the decoder's predictions into the decoder's input, just like we were doing for inference.

4.Suppose you have a daily univariate time series, and you want to forecast the next seven days. Which RNN architecture should you use?

LSTM methodology, while introduced in the late 90’s, has only recently become a viable and powerful forecasting technique. Classical forecasting methods like ARIMA and HWES are still popular and powerful but they lack the overall generalizability that memory-based models like LSTM offer.

The main objective of this article is to lead you through building a working LSTM model. However, while this article’s goal isn’t necessarily to compare new and classic modeling techniques, I will discuss some advantages and disadvantages of classical vs. RNN-based techniques in the conclusion.

The full code is provided below. Given you have the dataset and the necessary classes imported, the results should be entirely reproducible.

What Are LSTMs?

LSTM (Long Short-Term Memory) is a Recurrent Neural Network (RNN) based architecture that is widely used in natural language processing and time series forecasting. [Brandon Rohrer’s video](https://www.youtube.com/watch?v=WCUNPb-5EYI) offers a great, intuitive introduction.

The LSTM rectifies a huge issue that recurrent neural networks suffer from: short-memory. Using a series of ‘gates,’ each with its own RNN, the LSTM manages to keep, forget or ignore data points based on a probabilistic model.

LSTMs also help solve exploding and vanishing gradient problems. In simple terms, these problems are a result of repeated weight adjustments as a neural network trains. With repeated epochs, gradients become larger or smaller, and with each adjustment, it becomes easier for the network’s gradients to compound in either direction. This compounding either makes the gradients way too large or way too small. While exploding and vanishing gradients are huge downsides of using traditional RNN’s, LSTM architecture severely mitigates these issues.

After a prediction is made, it is fed back into the model to predict the next value in the sequence. With each prediction, some error is introduced into the model. To avoid exploding gradients, values are ‘squashed’ via (typically) sigmoid & tanh activation functions prior to gate entrance & output.

5.What are the main difficulties when training RNNs? How can you handle them?

There are two widely known issues with properly training Recurrent Neural Networks, the vanishing and the exploding gradient problems detailed in Bengio et al. (1994).

The traditional feed-forward neural networks are not good with time-series data and other sequences or sequential data. This data can be something as volatile as stock prices or a continuous video stream from an on-board camera of an autonomous car.

Handling time series data is where RNNs excel. They were designed to grasp the information contained in these sequences/time series data.

RNNs are built on the recursive formula, where the new state is a function of the old state and the input.

RNN applications extend beyond natural language processing and speech recognition. They’re used in language translation, stock predictions and algorithmic trading as well.

However, training RNNs come with their own set of challenges. For instance, the mathematical investigation of RNNs with respect to reliable information representation and generalization ability when dealing with complex data structures is still a challenge.

This has led to diverse approaches and architectures including specific training modes such as echo and liquid-state-machines, backpropagation decorrelation, or long short term memory, specific architectures such as recursive and graph networks, and hybrid systems.

Exploding And Vanishing Gradients

A gradient in the context of a neural network refers to the gradient of the loss function with respect to the weights of the network.

This gradient is calculated using backpropagation. The goal here is to find the optimal weight for each connection that would minimise the overall loss of the network.

While in principle the recurrent network is a simple and powerful model, in practice, it is, unfortunately, hard to train properly.  The recurrent connections in the hidden layer allow information to persist from one input to another.

The exploding gradients problem refers to the large increase in the norm of the gradient during training. Such events are caused by the explosion of the long term components, which can grow exponentially more than short term ones.

The vanishing gradients problem refers to the opposite behaviour, when long term components go exponentially fast to norm 0, making it impossible for the model to learn the correlation between temporally distant events.

Why Bother About Gradients Vanishing At All?

Since the weights are updated proportional to the gradient, a vanishing gradient or a small value will result in a small change in the value of weight. The value of a weight, which is multiplied with the input, decides whether a certain input needs to be taken seriously or not. No change in value means the network isn’t learning anything. In short, there is no point in training.

6.Can you sketch the LSTM cell’s architecture?

This article talks about the problems of conventional RNNs, namely, the vanishing and exploding gradients and provides a convenient solution to these problems in the form of Long Short Term Memory (LSTM). Long Short-Term Memory is an advanced version of recurrent neural network (RNN) architecture that was designed to model chronological sequences and their long-range dependencies more precisely than conventional RNNs. The major highlights include the interior design of a basic LSTM cell, the variations brought into the LSTM architecture, and few applications of LSTMs that are highly in demand. It also makes a comparison between LSTMs and GRUs. The article concludes with a list of disadvantages of the LSTM network and a brief introduction of the upcoming attention-based models that are swiftly replacing LSTMs in the real world.

Introduction:

LSTM networks are an extension of recurrent neural networks (RNNs) mainly introduced to handle situations where RNNs fail. Talking about RNN, it is a network that works on the present input by taking into consideration the previous output (feedback) and storing in its memory for a short period of time (short-term memory). Out of its various applications, the most popular ones are in the fields of speech processing, non-Markovian control, and music composition. Nevertheless, there are drawbacks to RNNs. First, it fails to store information for a longer period of time. At times, a reference to certain information stored quite a long time ago is required to predict the current output. But RNNs are absolutely incapable of handling such “long-term dependencies”. Second, there is no finer control over which part of the context needs to be carried forward and how much of the past needs to be ‘forgotten’. Other issues with RNNs are exploding and vanishing gradients (explained later) which occur during the training process of a network through backtracking. Thus, Long Short-Term Memory (LSTM) was brought into the picture. It has been so designed that the vanishing gradient problem is almost completely removed, while the training model is left unaltered. Long time lags in certain problems are bridged using LSTMs where they also handle noise, distributed representations, and continuous values. With LSTMs, there is no need to keep a finite number of states from beforehand as required in the hidden Markov model (HMM). LSTMs provide us with a large range of parameters such as learning rates, and input and output biases. Hence, no need for fine adjustments. The complexity to update each weight is reduced to O(1) with LSTMs, similar to that of Back Propagation Through Time (BPTT), which is an advantage.

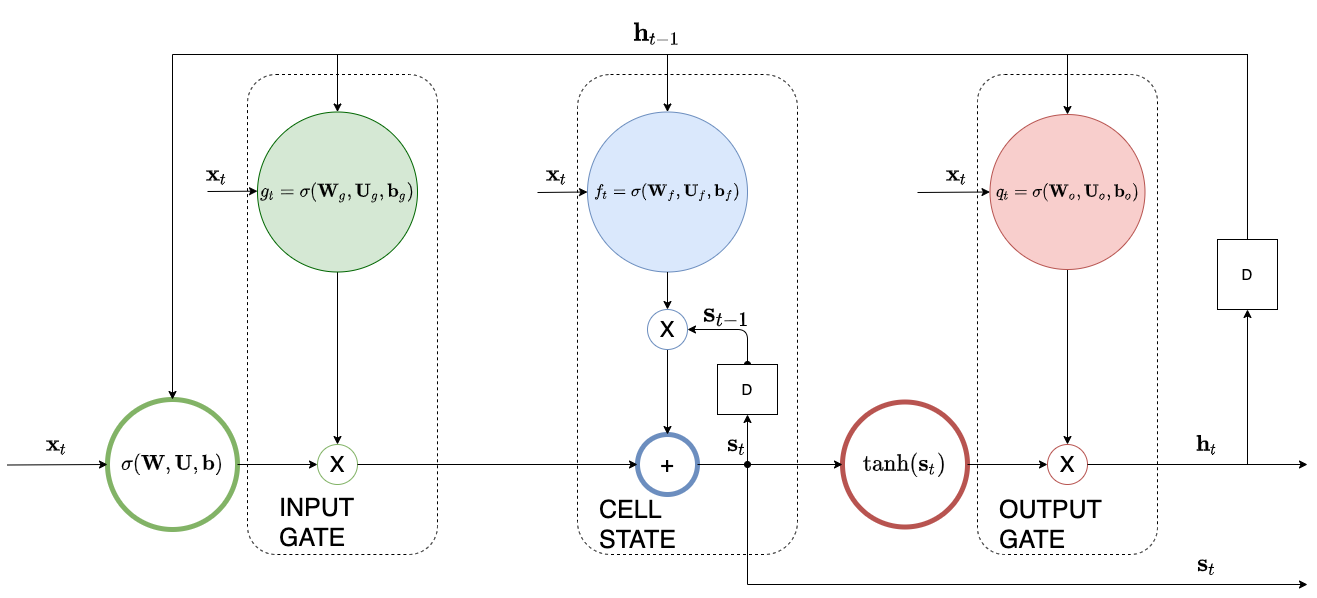
During the training process of a network, the main goal is to minimize loss (in terms of error or cost) observed in the output when training data is sent through it. We calculate the gradient, that is, loss with respect to a particular set of weights, adjust the weights accordingly and repeat this process until we get an optimal set of weights for which loss is minimum. This is the concept of backtracking. Sometimes, it so happens that the gradient is almost negligible. It must be noted that the gradient of a layer depends on certain components in the successive layers. If some of these components are small (less than 1), the result obtained, which is the gradient, will be even smaller. This is known as the scaling effect. When this gradient is multiplied with the learning rate which is in itself a small value ranging between 0.1-0.001, it results in a smaller value. As a consequence, the alteration in weights is quite small, producing almost the same output as before. Similarly, if the gradients are quite large in value due to the large values of components, the weights get updated to a value beyond the optimal value. This is known as the problem of exploding gradients. To avoid this scaling effect, the neural network unit was re-built in such a way that the scaling factor was fixed to one. The cell was then enriched by several gating units and was called LSTM.

Architecture:

The basic difference between the architectures of RNNs and LSTMs is that the hidden layer of LSTM is a gated unit or gated cell. It consists of four layers that interact with one another in a way to produce the output of that cell along with the cell state. These two things are then passed onto the next hidden layer. Unlike RNNs which have got the only single neural net layer of tanh, LSTMs comprises of three logistic sigmoid gates and one tanh layer. Gates have been introduced in order to limit the information that is passed through the cell. They determine which part of the information will be needed by the next cell and which part is to be discarded. The output is usually in the range of 0-1 where ‘0’ means ‘reject all’ and ‘1’ means ‘include all’.

Hidden layers of LSTM :

Each LSTM cell has three inputs ,and and two outputs and . For a given time t, is the hidden state, is the cell state or memory, is the current data point or input. The first sigmoid layer has two inputs–and where is the hidden state of the previous cell. It is known as the forget gate as its output selects the amount of information of the previous cell to be included. The output is a number in [0,1] which is multiplied (point-wise) with the previous cell state .

\*LSTM architecture: It is divided into three areas: input (green), cell state (blue) and output (red). You can clearly see the outer (\bm h\_{t-1}ht−1​ )and the inner (\bm s\_{t-1}st−1​) recurrence loops.\*

7.Why would you want to use 1D convolutional layers in an RNN?

In this thesis, an effort has been made to explain what exactly CNNs are learning by training the network with carefully selected input data. The data considered here are one dimensional time varying signals and hence the 1-D convolutional neural networks are used to train, test and to analyze the learned weights.

Human activity recognition is the problem of classifying sequences of accelerometer data recorded by specialized harnesses or smart phones into known well-defined movements.

Classical approaches to the problem involve hand crafting features from the time series data based on fixed-sized windows and training machine learning models, such as ensembles of decision trees. The difficulty is that this feature engineering requires deep expertise in the field.

Recently, deep learning methods such as recurrent neural networks and one-dimensional [convolutional neural networks](https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/), or CNNs, have been shown to provide state-of-the-art results on challenging activity recognition tasks with little or no data feature engineering, instead using feature learning on raw data.

In this tutorial, you will discover how to develop one-dimensional convolutional neural networks for time series classification on the problem of human activity recognition.

After completing this tutorial, you will know:

How to load and prepare the data for a standard human activity recognition dataset and develop a single 1D CNN model that achieves excellent performance on the raw data.

How to further tune the performance of the model, including data transformation, filter maps, and kernel sizes.

How to develop a sophisticated multi-headed one-dimensional convolutional neural network model that provides an ensemble-like result.

8.Which neural network architecture could you use to classify videos?

Specifically, we'll use a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN) consisting of GRU layers. This kind of hybrid architecture is popularly known as a CNN-RNN.

Getting Started with Video Classification Using Deep Learning

Video classification is similar to image classification, in that the algorithm uses feature extractors, such as convolutional neural networks (CNNs), to extract feature descriptors from a sequence of images and then classify them into categories. Video classification using deep learning provides a means to analyze, classify, and track activity contained in visual data sources, such as a video stream. Video classification has many applications, such as human activity recognition, gesture recognition, anomaly detection, and surveillance.

Video classification methodology includes these steps:

Prepare training data

Choose a video classifier

Train and evaluate the classifier

Use the classifier to process video data

You can train a classifier using a video classifier pretrained on a large activity recognition video data set, such as the Kinetics-400 Human Action Dataset, which is a large-scale and high-quality data set collection. Start by providing the video classifier with labeled video or video clips. Then, using a deep learning video classifier that consists of convolution neural networks that match the nature of the video input, you can predict and classify the videos. Ideally, your workflow should include the evaluation of your classifier. Finally, you can use the classifier to classify activity in a collection of videos or a streaming video from a webcam.

Computer Vision Toolbox™ provides the slow and fast pathway (SlowFast), ResNet with (2+1)D convolutions, and two-stream Inflated-3D techniques for training a classifier of video classification.

Create Training Data for Video Classification

To train a classifier network, you need a collection of videos and its corresponding collection of scene labels. A scene label is a label applied to a time range in a video. For example, you could label a range of frames "jumping".

You can use the [Video Labeler](https://www.mathworks.com/help/vision/ref/videolabeler-app.html) or [Ground Truth Labeler](https://www.mathworks.com/help/driving/ref/groundtruthlabeler-app.html) (Automated Driving Toolbox) to interactively label ground truth data in a video, image sequence, or custom data source with scene labels. For a summary all labelers, see [Choose an App to Label Ground Truth Data](https://www.mathworks.com/help/vision/ug/choose-a-labeling-app.html).

Create Video Classifier

Choose one of the listed video classifier objects to create deep learning classification networks using models pretrained models using the Kinetics-400 data set (which contains 400 class labels):

The [slowFastVideoClassifier](https://www.mathworks.com/help/vision/ref/slowfastvideoclassifier.html) model is pretrained on the Kinetics-400 data set which contains the residual network ResNet-50 model as the backbone architecture with slow and fast pathways. This functionality requires the Computer Vision Toolbox Model for SlowFast Video Classification.

The [r2plus1dVideoClassifier](https://www.mathworks.com/help/vision/ref/r2plus1dvideoclassifier.html) model is pretrained on the Kinetics-400 data set which contains 18 spatio-temporal (ST) residual layers. This functionality requires the Computer Vision Toolbox Model for R(2+1)D Video Classification.

The [inflated3dVideoClassifier](https://www.mathworks.com/help/vision/ref/inflated3dvideoclassifier.html) model contains two subnetworks: the video network and the optical flow network. These networks are trained on the Kinetics-400 data set with RGB data and optical flow data, respectively. This functionality requires the Computer Vision Toolbox Model for Inflated-3D Video Classification.

9.Train a classification model for the SketchRNN dataset, available in TensorFlow Datasets.

Simple Vector Drawing Datasets

This repo contains a set of optional, extra datasets for training sketch-rnn, a generative model for vector drawings. You can learn more about the model by reading this [blog post](https://research.googleblog.com/2017/04/teaching-machines-to-draw.html) or the [paper](https://arxiv.org/abs/1704.03477). An open-source TensorFlow implementation of sketch-rnn is available [here](https://magenta.tensorflow.org/sketch_rnn). Although the datasets had been created in the format customized for training sketch-rnn, it can, and should be used for training newer and better models to advance the state of generative vector image modelling.

We have organized 3 datasets in this repo:

[Aaron Koblin Sheep Market](https://github.com/hardmaru/sketch-rnn-datasets/tree/master/aaron_sheep)

[Kanji Stroke Data](https://github.com/hardmaru/sketch-rnn-datasets/tree/master/kanji)

[Omniglot Stroke Data](https://github.com/hardmaru/sketch-rnn-datasets/tree/master/omniglot)

Stroke-3 Vector Format

In each dataset, each sample is stored as list of coordinate offsets: ∆x, ∆y, and a binary value representing whether the pen is lifted away from the paper. This format, we refer to as stroke-3, is described in Alex Graves' [paper](https://arxiv.org/abs/1308.0850) on sequence generation. Note that the data format described in the sketch-rnn paper has 5 elements (stroke-5 format), and we need to perform live conversion of the data to the stroke-5 format during mini-batch construction.

We show a single example to illustrate this data format in the figure above. This below-average turtle drawing is sketched by the author. The turtle is represented as a sequence of (∆x, ∆y, binary pen state) points. In the rendered form, the line color corresponds to the sequential stroke ordering to illustrate the ordering.

Each dataset is stored as a python list of examples. Each example is represented as a np.array with np.int16 datatype, and can be easily loaded using the following commands in a Jupyter notebook:

filename = "sketch-rnn-datasets/kanji/short\_kanji.npz"

load\_data = np.load(filename)

train\_set = load\_data['train']

valid\_set = load\_data['valid']

test\_set = load\_data['test']

print len(train\_set)

Output: 10000

print len(valid\_set)

Output: 600

print len(test\_set)

Output: 500

# draw a random example (see draw\_strokes.py)

draw\_strokes(random.choice(train\_set))

For some of the datasets, we also performed simple stroke simplification to preprocess the data, using the [Ramer-Douglas-Peucker](https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm) (RDP) algorithm. We choose the epsilon parameter to be a value between 1.0 to 3.0, depending on how aggressively we want to simply the lines.

If you have a large set of simple SVG images, and want to construct a similar dataset, there are some available [libraries](https://pypi.python.org/pypi/svg.path) to convert subsets of SVGs into line segments. You can then apply RDP on the line segments before converting the data to stroke-3 format. There is some easy-to-use open source code for applying this algorithm [here](https://github.com/fhirschmann/rdp).