A Gentle Introduction to Backpropagation Through Time

by **Jason Brownlee** on August 14, 2020 in **Long Short-Term Memory Networks** **37**

Backpropagation Through Time, or BPTT, is the training algorithm used to update weights in recurrent neural networks like LSTMs.

To effectively frame sequence prediction problems for recurrent neural networks, you must have a strong conceptual understanding of what Backpropagation Through Time is doing and how configurable variations like Truncated Backpropagation Through Time will affect the skill, stability, and speed when training your network.In this post, you will get a gentle introduction to Backpropagation Through Time intended for the practitioner (no equations!).

In this post, you will get a gentle introduction to Backpropagation Through Time intended for the practitioner (no equations!).

After reading this post, you will know:

- What Backpropagation Through Time is and how it relates to the Backpropagation training algorithm used by Multilayer Perceptron networks.
- The motivations that lead to the need for Truncated Backpropagation Through Time, the most widely used variant in deep learning for training LSTMs.
- A notation for thinking about how to configure Truncated Backpropagation Through Time and the canonical configurations used in research and by deep learning libraries.

Kick-start your project with my new book Long Short-Term Memory Networks With Python, including *step-by-step tutorials* and the *Python source code* files for all examples.

Let's get started.

A Gentle Introduction to Backpropagation Through Time
Photo by Jocelyn Kinghorn, some rights reserved.

Backpropagation Training Algorithm

Backpropagation refers to two things:

- The mathematical method used to calculate derivatives and an application of the derivative chain rule.
- The training algorithm for updating network weights to minimize error.

It is this latter understanding of backpropagation that we are using here.

The goal of the backpropagation training algorithm is to modify the weights of a neural network in order to minimize the error of the network outputs compared to some expected output in response to corresponding inputs.

It is a supervised learning algorithm that allows the network to be corrected with regard to the specific errors made.

The general algorithm is as follows:

1. Present a training input pattern and propagate it through the network to get an output.
2. Compare the predicted outputs to the expected outputs and calculate the error.
3. Calculate the derivatives of the error with respect to the network weights.
4. Adjust the weights to minimize the error.
5. Repeat.

For more on Backpropagation, see the post:

- How to Implement the Backpropagation Algorithm from Scratch In Python

The Backpropagation training algorithm is suitable for training feed-forward neural networks on fixed-sized input-output pairs, but what about sequence data that may be temporally ordered?

Backpropagation Through Time

Backpropagation Through Time, or BPTT, is the application of the Backpropagation training algorithm to recurrent neural network applied to sequence data like a time series.

A recurrent neural network is shown one input each timestep and predicts one output.

Conceptually, BPTT works by unrolling all input timesteps. Each timestep has one input timestep, one copy of the network, and one output. Errors are then calculated and accumulated for each timestep. The network is rolled back up and the weights are updated.

Spatially, each timestep of the unrolled recurrent neural network may be seen as an additional layer given the order dependence of the problem and the internal state from the previous timestep is taken as an input on the subsequent timestep.

We can summarize the algorithm as follows:

1. Present a sequence of timesteps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across each timestep.
3. Roll-up the network and update weights.
4. Repeat.

BPTT can be computationally expensive as the number of timesteps increases.

If input sequences are comprised of thousands of timesteps, then this will be the number of derivatives required for a single update weight update. This can cause weights to vanish or explode (go to zero or overflow) and make slow learning and model skill noisy.

Truncated Backpropagation Through Time

Truncated Backpropagation Through Time, or TBPTT, is a modified version of the BPTT training algorithm for recurrent neural networks where the sequence is processed one timestep at a time and periodically (k1 timesteps) the BPTT update is performed back for a fixed number of timesteps (k2 timesteps).

Ilya Sutskever makes this clear in his dissertation:

> *Truncated backpropagation is arguably the most practical method for training RNNs.*
>
> …

> *One of the main problems of BPTT is the high cost of a single parameter update, which makes it impossible to use a large number of iterations.*
>
> *…*
>
> *The cost can be reduced with a naive method that splits the 1,000-long sequence into 50 sequences (say) each of length 20 and treats each sequence of length 20 as a separate training case. This is a sensible approach that can work well in practice, but it is blind to temporal dependencies that span more than 20 timesteps.*
>
> *…*
>
> *Truncated BPTT is a closely related method. It processes the sequence one timestep at a time, and every k1 timesteps, it runs BPTT for k2 timesteps, so a parameter update can be cheap if k2 is small. Consequently, its hidden states have been exposed to many timesteps and so may contain useful information about the far past, which would be opportunistically exploited.*

— Ilya Sutskever, Training Recurrent Neural Networks, Thesis, 2013

We can summarize the algorithm as follows:

1. Present a sequence of k1 timesteps of input and output pairs to the network.
2. Unroll the network then calculate and accumulate errors across k2 timesteps.
3. Roll-up the network and update weights.
4. Repeat

The TBPTT algorithm requires the consideration of two parameters:

- **k1**: The number of forward-pass timesteps between updates. Generally, this influences how slow or fast training will be, given how often weight updates are performed.
- **k2**: The number of timesteps to which to apply BPTT. Generally, it should be large enough to capture the temporal structure in the problem for the network to learn. Too large a value results in vanishing gradients.

To make this clearer:

> *…one can use a bounded-history approximation to it in which relevant information is saved for a fixed number h of time steps and any information older than that is forgotten. In general, this should be regarded as a heuristic technique for simplifying the computation, although, as discussed below, it can sometimes serve as an adequate approximation to the true gradient and may also be more appropriate in those situations where weights are adjusted as the network runs. Let us call this algorithm truncated backpropagation through time. With h representing the number of prior time steps saved, this algorithm will be denoted BPTT(h).*
>
> *…*
>
> *Note that in BPTT(h) a backward pass through the most recent h time steps is performed anew each time the network is run through an additional time step. To generalize this, one may consider letting the network run through h0 additional time steps before performing the next BPTT computation, where h0 <= h.*
>
> *…*
>
> *The key feature of this algorithm is that the next backward pass is not performed until time step t + h0; in the intervening time the history of network input, network state, and target values are saved in the history buffer, but no processing is performed on this data. Let us denote this algorithm BPTT(h; h0). Clearly BPTT(h) is the same as BPTT(h; 1), and BPTT(h; h) is the epochwise BPTT algorithm.*

— Ronald J. Williams and Jing Peng, An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories, 1990

We can borrow the notation from Williams and Peng and refer to different TBPTT configurations as TBPTT(k1, k2).

Using this notation, we can define some standard or common approaches:

Note, here n refers to the total number of timesteps in the sequence:

- **TBPTT(n,n)**: Updates are performed at the end of the sequence across all timesteps in the sequence (e.g. classical BPTT).
- **TBPTT(1,n)**: timesteps are processed one at a time followed by an update that covers all timesteps seen so far (e.g. classical TBPTT by Williams and Peng).
- **TBPTT(k1,1)**: The network likely does not have enough temporal context to learn, relying heavily on internal state and inputs.
- **TBPTT(k1,k2), where k1<k2<n**: Multiple updates are performed per sequence which can accelerate training.
- **TBPTT(k1,k2), where k1=k2**: A common configuration where a fixed number of timesteps are used for both forward and backward-pass timesteps (e.g. 10s to 100s).

We can see that all configurations are a variation on TBPTT(n,n) that essentially attempt to approximate the same effect with perhaps faster training and more stable results.

Canonical TBPTT reported in papers may be considered TBPTT(k1,k2), where k1=k2=h and h<=n, and where the chosen parameter is small (tens to hundreds of timesteps).

In libraries like TensorFlow and Keras, things look similar and h defines the vectorized fixed length of the timesteps of the prepared data.

Further Reading

This section provides some resources for further reading.

Books

- Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks, 1999
- Deep Learning, 2016

Papers

- Learning Representations by back-propagating errors, 1986
- Backpropagation through time: what it does and how to do it, 1990
- An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories, 1990
- Training Recurrent Neural Networks, Thesis, 2013
- Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity, 1995

Articles

- Backpropagation on Wikipedia
- Backpropagation through time on Wikipedia
- Styles of Truncated Backpropagation
- Answer to question "RNNs: When to apply BPTT and/or update weights?" on CrossValidated

Summary

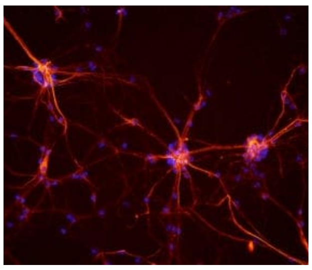
In this post, you discovered the Backpropagation Through Time for training recurrent neural networks.

Specifically, you learned:

- What Backpropagation Through Time is and how it relates to the Backpropagation training algorithm used by Multilayer Perceptron networks.
- The motivations that lead to the need for Truncated Backpropagation Through Time, the most widely used variant in deep learning for training LSTMs.
- A notation for thinking about how to configure Truncated Backpropagation Through Time and the canonical configurations used in research and by deep learning libraries.

Do you have any questions about Backpropagation Through Time?
Ask your questions in the comments below and I will do my best to answer.

More On This Topic


How to Code a Neural Network with Backpropagation In…

8 Tricks for Configuring Backpropagation to Train…

Difference Between Backpropagation and Stochastic…

How to Work Through a Time Series Forecast Project

Template for Working through Machine Learning…

How To Work Through A Problem Like A Data Scientist

About Jason Brownlee

Jason Brownlee, PhD is a machine learning specialist who teaches developers how to get results with modern machine learning methods via hands-on tutorials.
View all posts by Jason Brownlee →

< How to Handle Missing Timesteps in Sequence Prediction Problems with Python

Techniques to Handle Very Long Sequences with LSTMs >

37 Responses to *A Gentle Introduction to Backpropagation Through Time*

Sam Taha June 23, 2017 at 5:28 am # REPLY

Is TBPTT supported in Keras/Tensorflow?

Jason Brownlee June 23, 2017 at 6:44 am # REPLY

Yes, but it is not very flexible.

You must split your sequence into subsequences in order to achieve "truncated" BPTT and you are limited to BPTT(k1,k2) where k1=k2.

Pranav Goel June 27, 2017 at 2:34 am # REPLY

Hello Jason,

The post is very well written, and I feel quite clear with the concept of TBPTT. However, I am a bit confused in implementing in Keras. I understand that we are limited to BPTT(k1,k1), but even so, is there a parameter that we can set to fix the number of timesteps (h) to unroll for?

In Tensorflow – is the "num_steps" options basically BPTT(k1,k2) with k1=k2? Do you know its equivalent for Keras?

Also for Theano, we have the truncate_gradient option – do we have an equivalent option in Keras?

Do we have to split into subsequences in any case?
Thank you!

Jason Brownlee June 27, 2017 at 8:34 am # REPLY

As far as I know, no there is not.

You must achieve this by the number of time steps specified in each sample.

It is quite a pain I agree.

Pranav Goel June 27, 2017 at 10:58 pm #

Thanks for the response! I also went through your post on Stateful LSTMs in keras.
For more clarity, suppose I have really long sequences (let's say 1500 length sequences after zero padding to make all sequences of same length). And I want to perform truncated BPTT such that we use only the last 20 timesteps (it would be computationally intractable to backpropagate through the entire 1500 length sequence).
Can this be done using stateful LSTMs in Keras? If so, could you please give an idea of how? Even a broad suggestion would be much appreciated.

Thank you!

Jason Brownlee June 28, 2017 at 6:26 am #

Yes, split your sequence into subsequences of length 20.

Anh Huynh June 26, 2017 at 1:15 pm # REPLY

Hi Jason,

I have a question regarding the implementation of LSTM in Keras.

In Keras, the input_shape is (batch_size, timesteps, input_dim) and the output_shape is (batch_size, units). So we only need to provide one output for each sequence of timesteps of input.

This design of Keras seems to not go well with your statement that "Present a sequence of timesteps of input and output pairs to the network."

Can you help me to clarify this matter? Thanks.

Jason Brownlee June 27, 2017 at 8:24 am # REPLY

For sequence classification, you would provide your time steps and the output is a single prediction at the end of the sequence:

```
1, 2, 3, 4, 5 => 0
```

For sequence regression, e.g. predicting the next real value in the sequence, you can formulate the series as a supervised learning problem of predicting the next value given an input sequence and multiple "samples" can be drawn from your one series:

```
1, 2, 3 => 4
2, 3, 4 => 5
...
```

Does that help?

Anh Huynh June 27, 2017 at 6:30 pm # REPLY

Thanks for your answer. I understand your examples but I still cannot relate it the basic scheme of BPTT(k1,k2).

In tensorflow, we have k1=k2=num_steps, and we need to provide one output for each input to the network. The errors are backpropagated after every k1 time steps and go back no more than k2 steps. In keras, we provide one output for a sequence of inputs. Then what are k1 and k2 in the case of Keras?

Jason Brownlee June 28, 2017 at 6:21 am # REPLY

Great question.

In Keras, k1=k2=time_steps if you are making a prediction for each time step and k1=n,k2=time_steps if you are making a prediction at the end of the sequence (e.g. sequence classification).

Does that help?

Anh Huynh June 30, 2017 at 5:11 pm #

Great. Thanks a lot.

Jason Brownlee July 1, 2017 at 6:28 am #

You're welcome.

De Handschutter Pierre July 23, 2018 at 9:39 pm # REPLY

Hello Jason,

You say, through the citation: "This is a sensible approach that can work well in practice, but it is blind to temporal dependencies that span more than 20 timesteps". Except if you use a stateful LSTM, right ?

Thanks for your article, anyway !

Jason Brownlee July 24, 2018 at 6:17 am # REPLY

Kind of. The internal state is a noisy learned accumulation, not a perfect memory.

You can force it to memorize though, I have a post somewhere showing this capability. Here:
https://machinelearningmastery.com/memory-in-a-long-short-term-memory-network/

Martin August 2, 2018 at 6:00 pm # REPLY

I wonder whether BPTT can be used to train dynamic system (dx/dt = f(x,y,t)), instead of RNN. The model is non-linear equivalent circuit (EC) model with 1 state (1 derivative). The point would be to uncover EC model parameters, which have physical meaning (opposed to RNN's hyper-parameters).

Could you please point out some literature, or share experience on this matter ? Is there any fundamental difference between non-linear dynamic model and RNN, which would preclude the BPTT from converging ?

P.S. lets say there exists reasonable initial guess for the state of the dynamic system, as well as some rough estimate of parameters. Also, lets assume the system is SISO and the time range is 50-100 samples.

Jason Brownlee August 3, 2018 at 5:59 am # REPLY

Interesting idea.

Perhaps search on scholar.google.com for some of the early papers?

Simran Agarwal March 6, 2019 at 6:50 am # REPLY

Hi, I really liked your post. My problem is I have more than 9 sequences and each sequence has more than 10,000 steps. Now , how should I divide my sequence. Should I use stateful LSTMs?

Jason Brownlee March 6, 2019 at 8:01 am # REPLY

Good question, see this post:
https://machinelearningmastery.com/prepare-univariate-time-series-data-long-short-term-memory-networks/

Also, these tutorials:
https://machinelearningmastery.com/start-here/#deep_learning_time_series

Simran March 6, 2019 at 9:48 pm # REPLY

Hi, thanks for your reply. I saw all the tutorials and the post. I saw 'https://machinelearningmastery.com/truncated-backpropagation-through-time-in-keras/'
post and how to use stateful lstms. But my problem is all the sequences have different length. Then how should I divide my sequences into sub-sequences and train?

Jason Brownlee March 7, 2019 at 6:49 am # REPLY

There is no best way, you must choose an approach that makes sense for your project – e.g. what you want to achieve with your model.

Simran Agarwal March 7, 2019 at 7:16 am #

I want to make a LSTM Autoencoder. But the length of my sequence is really large, hence I am using stateful LSTM. I am really sorry to bother you but I have one more doubt. I have trouble in understanding what happens when batch_size>1 in stateful LSTM. So does it means that the state from sample 1 from batch 1 is transferred to sample 11 in batch 2 if my batch size is 10. And state of sample 2 is send to sample 12? Also are the states passed between batches? For example if my batch size is 10, so will the state from sample 1 be passed to sample 2 within the batch?

Jason Brownlee March 7, 2019 at 2:29 pm #

When the LSTM is staetful, it means that the state is not reset at the end of the batch. This means it is carried across batches.

Yes, as you say, state from the end of the last batch is used at the start of the next batch. This may or may not be useful to your modeling problem – test to find out.

Simran March 7, 2019 at 1:29 am #

REPLY

Hi, I am really sorry to bother you but I am really confused. As per your tutorials, I used stateful LSTMs for creating LSTM Autoencoders.

The shape of my data is (1,2000000,4) and I divided into subsequences as:
(5000,400,4). But my program is still crashing and I don't understand why. Here is a part of my code:

data= data(5000,400,4)
n_batch = data.shape[0]
timesteps= data.shape[1]
input_dim = data.shape[2]
latent_dim = 64

inputs = Input(batch_shape=(n_batch,timesteps, input_dim))
encoded = LSTM(latent_dim, stateful=True)(inputs)
decoded = RepeatVector(timesteps)(encoded)
decoded = LSTM(latent_dim, return_sequences=True, stateful=True,
batch_input_shape=(n_batch,timesteps, input_dim))(decoded)
decoded = TimeDistributed(Dense(input_dim, activation='sigmoid'))(decoded)
autoencoder= Model(inputs, decoded)
encoder = Model(inputs, encoded)

autoencoder.compile(loss='mse', optimizer='adam', metrics=['accuracy'])

But when I am using fit as:
n_epochs = 10
for i in range(n_epochs):
autoencoder.fit(data, data, epochs=1, batch_size=n_batch, verbose=2, shuffle=False)
autoencoder.reset_states()
My program is getting crashed. Please help me with the same.

Jason Brownlee March 7, 2019 at 6:55 am #

REPLY

I have some suggestions here:
https://machinelearningmastery.com/faq/single-faq/can-you-read-review-or-debug-my-code

Simran Agarwal March 7, 2019 at 7:17 am #

REPLY

Thanks for the suggestion. I could easily debug my code. 🙂

Mark West July 11, 2019 at 2:08 am #

REPLY

Hello and thanks for this very helpful blogpost!

I do have a couple of questions regarding BPTT :

(1) When is the Loss Function applied?
(2) When is the Optimisation Function applied?

Are these both applied after the RNN is unrolled? Or maybe after the RNN is rolled up again?

Jason Brownlee July 11, 2019 at 9:51 am # REPLY

Optimization is a process, it uses loss to estimate the performance of the model, estimate an error gradient and update model weights.

Loss is calculated and weights are updated at the end of each batch of samples.

Mark West July 11, 2019 at 11:08 pm # REPLY

Thanks for a quick answer. I think I got a little caught up in the Keras Loss and Optimize functions, which confused me a bit 🙂

Jason Brownlee July 12, 2019 at 8:43 am # REPLY

No problem.

Michael Przystupa July 22, 2019 at 10:30 am # REPLY

Good post, but shouldn't the 3rd scenario be
TBPTT(k1, k2) = k2 <= k1 k1 calculate the graph

```
for j in range(k1- k, k1):
        preds, hidden = model(inputs[j])
        calcloss(preds).backward()
```

Hope this question makes sense and thanks!

Jason Brownlee July 22, 2019 at 2:05 pm # REPLY

Why do you say that exactly?

sandra demirki January 20, 2020 at 3:43 am # REPLY

Hello Jason, thanks for the tutorial
Though I did not understand how the loss function is used. In the case of return_sequences=true in Keras(input and output shapes are (batch_size, timesteps,n_features)), is the loss function supposed to have a shape of (batch_size, timesteps)? Or can it be scalar for each sequence? Since you mentioned that updating the weights is done after rolling, is the loss for each timestep used separately to update the weights? Or is the loss function averaged through the timesteps and implemented at the end of each sequence?
The reason I'm asking this is that I want to implement a custom loss function in Keras, and what I mean to do is to find the correlation between the input sequence and the output sequence, so a single scalar is returned for each sequence(T timesteps).

Jason Brownlee January 20, 2020 at 8:44 am # REPLY

A single loss value is calculated for each sample.

Jenny March 20, 2020 at 4:49 pm # REPLY

Dear Dr Brownlee

thank you for the wonderful post but, I am not quite understanding the updating the weights.
you said the weights are updated after the network it rolled up.
if I understood RNN correctly I know that 3 weights (for input ,hidden layer and outputs) are the same for every time step and they share the 3weights through out the time during forward processing. then for back propagation as the gradients get added up fore earlier time step the updated weights value will be different for each time step? as I right or am i just miss understanding the whole concept? I am talking about the general simple RNN case.

Jason Brownlee March 21, 2020 at 8:18 am # REPLY

We weights are not the same at each time, they are changed sequentially over time steps.

Andrew May 23, 2024 at 6:57 am #

REPLY

Thanks for your work, sir. Your articles explain a hard topics in easy and understandable words. You are an expert!

James Carmichael May 23, 2024 at 7:51 am #

REPLY

Thank you for your feedback and support Andrew! We appreciate it!

Leave a Reply

Name (required)

Email (will not be published) (required)

SUBMIT COMMENT

Welcome!
I'm *Jason Brownlee* PhD
and I **help developers** get results with **machine learning**.
Read more

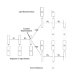
Never miss a tutorial:

Picked for you:

How to Reshape Input Data for Long Short-Term Memory Networks in Keras

How to Develop an Encoder-Decoder Model for Sequence-to-Sequence Prediction in Keras

How to Develop an Encoder-Decoder Model with Attention in Keras


A Gentle Introduction to LSTM Autoencoders

How to Use the TimeDistributed Layer in Keras