1. What are the pros and cons of using a stateful RNN versus a stateless RNN?

The Keras Python deep learning library supports both stateful and stateless Long Short-Term Memory (LSTM) networks.

When using stateful LSTM networks, we have fine-grained control over when the internal state of the LSTM network is reset. Therefore, it is important to understand different ways of managing this internal state when fitting and making predictions with LSTM networks affect the skill of the network.

In this tutorial, you will explore the performance of stateful and stateless LSTM networks in Keras for time series forecasting.

After completing this tutorial, you will know:

How to compare and contrast stateful and stateless LSTM networks for time series forecasts.

How the batch size in stateless LSTMs relate to stateful LSTM networks.

How to evaluate and compare different state resetting regimes for stateful LSTM networks.

Setting an RNN to be stateful means that it can build a state across its training sequence and even maintain that state when doing predictions. The benefits of using stateful RNNs are smaller network sizes and/or lower training times.

RNNs can be stateful, which means that they can maintain state across batches during training. That is, the hidden state computed for a batch of training data will be used as the initial hidden state for the next batch of training data. However, this needs to be explicitly set, since Keras RNNs are stateless by default and resets the state after each batch. Setting an RNN to be stateful means that it can build a state across its training sequence and even maintain that state when doing predictions.

The benefits of using stateful RNNs are smaller network sizes and/or lower training times. The disadvantage is that we are now responsible for training the network with a batch size that reflects the periodicity of the data, and resetting the state after each epoch. In addition, data should not be shuffled while training the network, since the order in which the data is presented is relevant for stateful networks.

2.Why do people use Encoder–Decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?

This two-step model, called an Encoder–Decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented on the top left), since the last words of a sentence can affect the first words of the translation, so you need to wait until you have heard the whole ..

Chapter 4. Recurrent Neural Networks

The batter hits the ball. You immediately start running, anticipating the ball’s trajectory. You track it and adapt your movements, and finally catch it (under a thunder of applause). Predicting the future is what you do all the time, whether you are finishing a friend’s sentence or anticipating the smell of coffee at breakfast. In this chapter, we are going to discuss recurrent neural networks (RNN), a class of nets that can predict the future (well, up to a point, of course). They can analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have discussed so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing (NLP) systems such as automatic translation, speech-to-text, or sentiment analysis (e.g., reading movie reviews and extracting the rater’s feeling about the movie).

Moreover, RNNs’ ability to anticipate also makes them capable of surprising creativity. You can ask them to predict which are the most likely next notes in a melody, then randomly pick one of these notes and play it. Then ask the net for the next most likely notes, play it, and repeat the process again and again. Before you know it, your net will compose a melody such as [the one](http://goo.gl/IxIL1V) produced by Google’s [Magenta project](https://magenta.tensorflow.org/). Similarly, RNNs can [generate sentences](http://goo.gl/onkPNd), [image captions](http://goo.gl/Nwx7Kh), and much more. The result is not exactly Shakespeare or Mozart yet, but who knows what they will produce a few years from now?

In this chapter, we will look at the fundamental concepts underlying RNNs, the main problem they face (namely, vanishing/exploding gradients, discussed in [Chapter 2](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch02.html#deep_chapter)), and the solutions widely used to fight it: LSTM and GRU cells. Along the way, as always, we will show how to implement RNNs using TensorFlow. Finally, we will take a look at the architecture of a machine translation system.

Recurrent Neurons

Up to now we have mostly looked at feedforward neural networks, where the activations flow only in one direction, from the input layer to the output layer. A recurrent neural network looks very much like a feedforward neural network, except it also has connections pointing backward. Let’s look at the simplest possible RNN, composed of just one neuron receiving inputs, producing an output, and sending that output back to itself, as shown in [Figure 4-1](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#simple_rnn_diagram) (left). At each time step t (also called a frame), this recurrent neuron receives the inputs x(t) as well as its own output from the previous time step, y(t–1). We can represent this tiny network against the time axis, as shown in [Figure 4-1](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#simple_rnn_diagram) (right). This is called unrolling the network through time.

You can easily create a layer of recurrent neurons. At each time step t, every neuron receives both the input vector x(t) and the output vector from the previous time step y(t–1), as shown in [Figure 4-2](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#rnn_layer_diagram). Note that both the inputs and outputs are vectors now (when there was just a single neuron, the output was a scalar).

Each recurrent neuron has two sets of weights: one for the inputs x(t) and the other for the outputs of the previous time step, y(t–1). Let’s call these weight vectors wx and wy. If we consider the whole recurrent layer instead of just one recurrent neuron, we can place all the weight vectors in two weight matrices, Wx and Wy. The output vector of the whole recurrent layer can then be computed pretty much as you might expect, as shown in [Equation 4-1](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#rnn_output_equation) (b is the bias vector and ϕ(·) is the activation function, e.g., ReLU[1](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#idm139624960131296)).

Just like for feedforward neural networks, we can compute a recurrent layer’s output in one shot for a whole mini-batch by placing all the inputs at time step t in an input matrix X(t) (see [Equation 4-2](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#rnn_output_vectorized_equation)).

the layer’s outputs at time step t for each instance in the mini-batch (m is the number of instances in the mini-batch and nneurons is the number of neurons).

X(t) is an m × ninputs matrix containing the inputs for all instances (ninputs is the number of input features).

Wx is an ninputs × nneurons matrix containing the connection weights for the inputs of the current time step.

Wy is an nneurons × nneurons matrix containing the connection weights for the outputs of the previous time step.

b is a vector of size nneurons containing each neuron’s bias term.

The weight matrices Wx and Wy are often concatenated vertically into a single weight matrix W of shape (ninputs + nneurons) × nneurons (see the second line of [Equation 4-2](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#rnn_output_vectorized_equation)).

The notation [X(t) Y(t–1)] represents the horizontal concatenation of the matrices X(t) and Y(t–1).

Notice that Y(t) is a function of X(t) and Y(t–1), which is a function of X(t–1) and Y(t–2), which is a function of X(t–2) and Y(t–3), and so on. This makes Y(t) a function of all the inputs since time t = 0 (that is, X(0), X(1), …, X(t)). At the first time step, t = 0, there are no previous outputs, so they are typically assumed to be all zeros.

Memory Cells

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of memory. A part of a neural network that preserves some state across time steps is called a memory cell (or simply a cell). A single recurrent neuron, or a layer of recurrent neurons, is a very basic cell, but later in this chapter we will look at some more complex and powerful types of cells.

In general a cell’s state at time step t, denoted h(t) (the “h” stands for “hidden”), is a function of some inputs at that time step and its state at the previous time step: h(t) = f(h(t–1), x(t)). Its output at time step t, denoted y(t), is also a function of the previous state and the current inputs. In the case of the basic cells we have discussed so far, the output is simply equal to the state.

Input and Output Sequences

An RNN can simultaneously take a sequence of inputs and produce a sequence of outputs (see [Figure 4-4](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#seq_to_seq_diagram), top-left network). For example, this type of network is useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future (i.e., from N – 1 days ago to tomorrow).

Alternatively, you could feed the network a sequence of inputs, and ignore all outputs except for the last one (see the top-right network). In other words, this is a sequence-to-vector network. For example, you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score (e.g., from –1 [hate] to +1 [love]).

Conversely, you could feed the network a single input at the first time step (and zeros for all other time steps), and let it output a sequence (see the bottom-left network). This is a vector-to-sequence network. For example, the input could be an image, and the output could be a caption for that image.

Lastly, you could have a sequence-to-vector network, called an encoder, followed by a vector-to-sequence network, called a decoder (see the bottom-right network). For example, this can be used for translating a sentence from one language to another. You would feed the network a sentence in one language, the encoder would convert this sentence into a single vector representation, and then the decoder would decode this vector into a sentence in another language. This two-step model, called an Encoder–Decoder, works much better than trying to translate on the fly with a single sequence-to-sequence RNN (like the one represented on the top left), since the last words of a sentence can affect the first words of the translation, so you need to wait until you have heard the whole sentence before translating it.

Sounds promising, so let’s start coding!

Basic RNNs in TensorFlow

First, let’s implement a very simple RNN model, without using any of TensorFlow’s RNN operations, to better understand what goes on under the hood. We will create an RNN composed of a layer of five recurrent neurons (like the RNN represented in [Figure 4-2](https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html#rnn_layer_diagram)), using the tanh activation function. We will assume that the RNN runs over only two time steps, taking input vectors of size 3 at each time step.

3.How can you deal with variable-length input sequences? What about variable-length output sequences?

The first and simplest way of handling variable length input is to set a special mask value in the dataset, and pad out the length of each input to the standard length with this mask value set for all additional entries created. Then, create a Masking layer in the model, placed ahead of all downstream layers.

Deep learning libraries assume a vectorized representation of your data.

In the case of variable length sequence prediction problems, this requires that your data be transformed such that each sequence has the same length.

This vectorization allows code to efficiently perform the matrix operations in batch for your chosen deep learning algorithms.

In this tutorial, you will discover techniques that you can use to prepare your variable length sequence data for sequence prediction problems in Python with Keras.

After completing this tutorial, you will know:

How to pad variable length sequences with dummy values.

How to pad variable length sequences to a new longer desired length.

How to truncate variable length sequences to a shorter desired length.

Environment

This tutorial assumes you have a Python SciPy environment installed. You can use either Python 2 or 3 with this example.

This tutorial assumes you have Keras (v2.0.4+) installed with either the TensorFlow (v1.1.0+) or Theano (v0.9+) backend.

This tutorial also assumes you have scikit-learn, Pandas, NumPy, and Matplotlib installed.

If you need help setting up your Python environment, see this post:

Sequence Padding

The [pad\_sequences() function](https://keras.io/preprocessing/sequence/) in the Keras deep learning library can be used to pad variable length sequences.

The default padding value is 0.0, which is suitable for most applications, although this can be changed by specifying the preferred value via the “value” argument. For example:

|  |  |
| --- | --- |
| 1 | pad\_sequences(..., value=99) |

The padding to be applied to the beginning or the end of the sequence, called pre- or post-sequence padding, can be specified by the “padding” argument, as follows.

Pre-Sequence Padding

Pre-sequence padding is the default (padding=’pre’)

The example below demonstrates pre-padding 3-input sequences with 0 values.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from keras.preprocessing.sequence import pad\_sequences  # define sequences  sequences = [  [1, 2, 3, 4],     [1, 2, 3],       [1]  ]  # pad sequence  padded = pad\_sequences(sequences)  print(padded) |

Running the example prints the 3 sequences pre-pended with zero values.

|  |  |
| --- | --- |
| 1  2  3 | [[1 2 3 4]  [0 1 2 3]  [0 0 0 1] |

Post-Sequence Padding

Padding can also be applied to the end of the sequences, which may be more appropriate for some problem domains.

Post-sequence padding can be specified by setting the “padding” argument to “post”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | from keras.preprocessing.sequence import pad\_sequences  # define sequences  sequences = [  [1, 2, 3, 4],     [1, 2, 3],       [1]  ]  # pad sequence  padded = pad\_sequences(sequences, padding='post')  print(padded) |

Running the example prints the same sequences with zero-values appended.

|  |  |
| --- | --- |
| 1  2  3 | [[1 2 3 4]  [1 2 3 0]  [1 0 0 0]] |

4.What is beam search and why would you use it? What tool can you use to implement it?

Beam search is an algorithm used in many NLP and speech recognition models as a final decision making layer to choose the best output given target variables like maximum probability or next output character.

Beam search is an algorithm used in many NLP and speech recognition models as a final decision making layer to choose the best output given target variables like maximum probability or next output character. First used for speech recognition in 1976, beam search is used often in models that have encoders and decoders with LSTM or Gated Recurrent Unit modules built in. To understand where this algorithm is used a little more let's take a look at how NLP models generate output, to see where Beam search comes into play.

‍

Sequence To Sequence NLP Model

Simple sequence to sequence model used to respond to prompts like a chatbot

‍

Sequence to sequence is a deep learning based NLP model used for machine translation and speech recognition that predicts the probability of the next token in a sequence of words. In speech recognition, the input audio sequence would be encoded with a recurrent neural network and feed into a decoder for prediction or speech classification using different ending layers. For this example, we're going to look at text language translation, where the decoder produces a new sentence in a new language. The encoded text from the original language sentence is fed into the decoder along with what we call a "Start" token to seed this runs output.

This new representation is passed through an output layer, which contains a softmax function to output a probability of the likelihood of each word in the new sentence appearing, as well as the word appearing in that position in the output sequence.

Of course, the goal of this model in our mind is a correctly translated final sentence. How does the model reach that final score and sentence that it believes to be the most accurate translated sequence? Given the number of combinations of probabilities of sequence positioning as well as probabilities of which word to place in a given position, there has to be an algorithm to decide, right?

Greedy Search example showing how it takes the best solution at each level in the tree, regardless of previous leaves or future leaves in the sequence

Greedy search will simply take the highest probability word at each position in the sequence and predict that in the output sequence. Choosing just one candidate at a step might be optimal at the current spot in the sequence, but as we move through the rest of the full sentence, it might turn out to be worse than we thought, given we couldn't see later predicted positions. As we'll see later and you can probably predict, as our outputs become longer the greedy search algorithm begins to struggle.

Beam Search: Using Conditional Probability

The beam search algorithm selects multiple tokens for a position in a given sequence based on conditional probability. The algorithm can take any number of N best alternatives through a hyperparameter know as Beam width. In greedy search we simply took the best word for each position in the sequence, where here we broaden our search or "width" to include other words that might fit better.

Greedy search looks at each position in the output sequence in isolation. A word is decided based on highest probability and we continue moving down the rest of the sentence, not going back to earlier ones. With Beam search, we also take the N best output sequences and look at the current preceding words and the probabilities compared to the current position we are decoding in the sequence. Let's walk through an example to see the steps we must take to use beam search effectively.

Step 1: Set Beam Width & Decode

Passing a sentence into the Encoder to translate

Lets set our beam width to 3 and grab the top three predicted words at each position in a given sequence. The encoded audio sequence is passed to a decoder, where a softmax function is applied to all the words in a set vocabulary (would be previously defined no matter if we're working with audio sequencing or text translation).

‍

Step 2: Pass Tokens and Predict

Lets set our beam width to 3 and grab the top three predicted words at each position in a given sequence. The encoded audio sequence is passed to a decoder, where a softmax function is applied to all the words in a set vocabulary (would be previously defined no matter if we're working with audio sequencing or text translation).

Source: https://towardsdatascience.com/an-intuitive-explanation-of-beam-search-9b1d744e7a0f - A look at selecting the top three words for the first position in the sequence. The hindu text is translated to English in the first position from any words in the vocab

For the second word in a sequence we pass the first three selected words as input into the second position. As we did before, we apply the same softmax output layer function to the set vocabulary find the next 3 words we could use for the second position. While this happens, we use conditional probability to decide on the best combination of first position words and second position words. We run these 3 input words against all words in the vocabulary to find the best 3 combinations and will pass them to the next layer as input again. Words from the first position can get dropped moving forward if another input token has a higher probability with two different sequences. For instance, if "I will" and "I am" were higher than any combination with "Us" we can drop the "Us" token and continue with our new top three sequences. We repeat this process until we reach an END token and have now generated 3 different sequences.

Step 3: Beam Search Final Output

We now have 3 different text translations or audio sequence results that we still have to decide between. These output sequences can be different in length and total tokens, which can create nice variation in our results. We simply pick the decoder output with the highest probability at the end.

In this example you can see we have 4 different beams where we have found a portion of the sequence. Each beam can have a different length and include a token N, where N is in the set of vocab words

Beam Search Analysis

Beam search doesn't have to be used for sequence based models where we use encoders and decoders to build large text and audio systems. Beam search can be broken down to a graph search where the points are possible are tokens in the input and we order all partial solutions according to some optimization function. Beam search builds its search tree using breadth-first search where the lower leaves of any given leaf are all possible states past the current leaf. We can use the beam width control how many leaves we can remove from the entire tree, where the higher a beam width the more leaves we leave and fewer are pruned. We can then use this as a breath-first search that uses the beam width as a memory control, in situations we want to account for run-time and tree size.

In terms of text sequence analysis like we saw above, a lower beam width will produce worse translations as the width moves towards 1 (which is the exact same as a Greedy Search), but becomes faster and more efficient. Many implementations start with a lower number for the beam width and progressively move it up as output results keep improving.

5.What is an attention mechanism? How does it help?

The idea behind the attention mechanism was to permit the decoder to utilize the most relevant parts of the input sequence in a flexible manner, by a weighted combination of all the encoded input vectors, with the most relevant vectors being attributed the highest weights.

The attention mechanism was introduced to improve the performance of the encoder-decoder model for machine translation. The idea behind the attention mechanism was to permit the decoder to utilize the most relevant parts of the input sequence in a flexible manner, by a weighted combination of all the encoded input vectors, with the most relevant vectors being attributed the highest weights.

In this tutorial, you will discover the attention mechanism and its implementation.

After completing this tutorial, you will know:

How the attention mechanism uses a weighted sum of all the encoder hidden states to flexibly focus the attention of the decoder on the most relevant parts of the input sequence

How the attention mechanism can be generalized for tasks where the information may not necessarily be related in a sequential fashion

How to implement the general attention mechanism in Python with NumPy and SciPy

The Attention Mechanism

The attention mechanism was introduced by [Bahdanau et al. (2014)](https://arxiv.org/abs/1409.0473) to address the bottleneck problem that arises with the use of a fixed-length encoding vector, where the decoder would have limited access to the information provided by the input. This is thought to become especially problematic for long and/or complex sequences, where the dimensionality of their representation would be forced to be the same as for shorter or simpler sequences.

[Note](https://machinelearningmastery.com/how-does-attention-work-in-encoder-decoder-recurrent-neural-networks/) that Bahdanau et al.’s attention mechanism is divided into the step-by-step computations of the alignment scores, the weights, and the context vector:

Alignment scores: The alignment model takes the encoded hidden states, hi, and the previous decoder output, st−1, to compute a score, et,i, that indicates how well the elements of the input sequence align with the current output at the position, t. The alignment model is represented by a function, a(.), which can be implemented by a feedforward neural network:

et,i=a(st−1,hi)

Weights: The weights, αt,i, are computed by applying a softmax operation to the previously computed alignment scores:

αt,i=softmax(et,i)

Context vector: A unique context vector, ct, is fed into the decoder at each time step. It is computed by a weighted sum of all, T, encoder hidden states:

ct=∑i=1Tαt,ihi

Bahdanau et al. implemented an RNN for both the encoder and decoder.

However, the attention mechanism can be re-formulated into a general form that can be applied to any sequence-to-sequence (abbreviated to seq2seq) task, where the information may not necessarily be related in a sequential fashion.

6.What is the most important layer in the Transformer architecture? What is its purpose?

The most important part here is the “Residual Connections” around the layers. This is very important in retaining the position related information which we are adding to the input representation/embedding across the network.

Transformer Architecture: Attention Is All You Need

In this post, we are going to explore the concept of attention and look at how it powers the “Transformer Architecture” which thus demonstrates why “Attention Is All You Need!”

Introduction:

Whenever long-term dependencies (natural language processing problems) are involved, we know that RNNs (even with using hacks like bi-directional, multi-layer, memory-based gates — LSTMs/GRUs) suffer from vanishing gradient problem. Also, they handle the sequence of inputs 1 by 1, word by word this resulting in an obstacle towards parallelization of the process.

Especially when it comes to seq2seq models, is one hidden state really enough to capture global information pertaining to the translation?

The problem with this approach was (as famously said at the ACL 2014 workshop):

“You can’t cram the meaning of a whole %&!$# sentence into a single $&!#\* vector!”

Here’s where attention comes in!

Attention, in general, can be thought of as follows:

The idea here is to learn a context vector (say U), which gives us global level information on all the inputs and tells us about the most important information (this could be done by taking a cosine similarity of this context vector U w.r.t the input hidden states from the fully connected layer. We do this for each input x\_i and thus obtain a theta\_i (attention weights).

i.e. : theta\_i = cosine\_similarity(U, x\_i)

For each of the input hidden states x\_1 … x\_k, we learn a set of weights theta\_1 to theta\_k which measures how much of the inputs answer the query and this generates an output

The weights for the inputs of attention are learned to understand which inputs it should attend to

For an encoder-decoder architecture:

For every single target decoder output ( say, t\_j ), all hidden state source inputs (say s\_i’s) are taken into account to compute the cosine similarity with the source inputs s\_i, to generate the theta\_i’s (attention weights) for every s\_i.

i.e. theta\_i = cosine\_similarity(t\_j, s\_i)

The context vector (out — refer to the above equation) is now computed for every source input s\_i and theta\_i (generated for the corresponding target decoder word t\_j). The context vector (out) and target word (t\_j) are used to predict the output in the decoder architecture, which is then daisy chained and continued from here on in the above manner using attention.

Attention mechanism solves this problem by allowing the decoder to “look-back” at the encoder’s hidden states based on its current state. This allows the decoder to extract only relevant information about the input tokens at each decoding, thus learning more complicated dependencies between the input and the output.

This allows the decoder to capture global information rather than to rely solely based on one hidden state!

Here, we see that the dependencies are learned between the inputs and outputs.

But, in the Transformer architecture this idea is extended to learn intra-input and intra-output dependencies as well (we’ll get to that soon!)

Attention Definition according to the Transformer paper:

Attention as explained by the Transformer Paper

An attention function can be described as mapping a query (Q) and a set of key-value pairs (K, V) to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

7.When would you need to use sampled softmax?

I'd probably consider using sampled softmax if I have over 100,000 classes, or if my final classification layer dominates overall execution time or memory use. An obvious application is large word vocabularies, for example in language modelling.

Sampled Softmax Loss

Sampled Softmax is a drop-in replacement for softmax cross entropy which improves scalability e.g. when there are millions of classes. It is very similar to Noise Contrastive Estimation (NCE) and Negative Sampling, both of which are popular in natural language processing, where the vocabulary size can be very large.

In this article, we'll think through the core idea of the sampled softmax loss function, see how to implement it in PyTorch and finally look at what happens when we use this loss. This is part of our series on [training objectives](https://douglasorr.github.io/2021-10-training-objectives/1-xent/article.html), and if you're not familiar with softmax cross entropy, [our introduction](https://douglasorr.github.io/2021-10-training-objectives/1-xent/article.html#softmax-cross-entropy-loss) to that would be a useful pre-read.

Core idea

When we looked at softmax cross entropy loss gradient updates, we saw both a narrow "spike" and a wider "forest". The spike is a strong negative gradient for the correct label e.g. "horse". The forest contains positive gradients for all other classes. We observed that the spike and the forest contain the same probability mass, but whereas the spike is concentrated on a single class, the forest is spread out over N−1 classes.

This looks fine for 10 classes, but if we have over 100,000, it suddenly looks quite inefficient. We end up making 1 large increasing (negative gradient) update to the correct class, and 99,999 small decreasing (positive gradient) updates to all other classes. Ideally we wouldn't have to compute scores or gradients for all of the incorrect classes for each and every update.

Sampled softmax, NCE and negative sampling are related sampling-based solutions to this problem. Instead of always using every negative class, the idea is to sample a smaller set of negative classes. With a slight adjustment to the loss function, this gives a biased but usable estimator of the gradient we would have obtained using plain softmax cross entropy.

The benefit of this is that the unused classes require no computation at all, as they can be ignored in both forward and backward passes. So if we have 100,000 classes but only draw 100 negative samples, we only need to compute 101 scores (100 negative samples, 1 positive target) & backpropagate 101 gradients.

The equation

Let's look at the equation. First, to recap softmax cross entropy:

L(x,t)=−xt+log⁡∑iexi

Where x is the predicted distribution from the model, and t is the target label. If we wanted to estimate the sum using samples, our first attempt might be:

L(x,t)=−xt+log⁡∑c~∼qcexc~/(kqc~)

Here q is a fixed vector of probabilities for each class, and c~ are the negative samples (drawn k times). This is [importance sampling](https://en.wikipedia.org/wiki/Importance_sampling). Briefly, this means: sampling c~ from any distribution q that we choose, then dividing out the bias caused by q, to estimate the log-sum-exp.

This is OK, but we can do better. We can lower the variance of this log-sum-exp estimation by separating out the target label from the rest of the distribution. Now we'll always use the target label in the log-sum-exp, and exclude it from sampling. This is useful because we have to compute the target score anyway, and if our model is any good it will often have high score, making it a useful "sample" for our estimator.

With this tweak (and a slight rearrangement of terms into the exp), our sampled softmax looks like this:

(1)L(x,t)=−xt+log⁡[ext+∑c~∼qc|c≠texc~−log⁡(kqc~/(1−qt))]

This still looks quite like a plain softmax cross-entropy loss. The key difference is that the sum is over the target and a fixed number of samples, rather than every class. The only other difference is an adjustment of class scores given to the softmax normaliser (x′=x−log⁡(kqc~/(1−qt))). We'll see what this adjustment does when we look at the gradients later.

PyTorch implementation

It's slightly fiddly to implement sampled softmax. To get the most out of it, we need to avoid computing scores for classes that aren't needed by the loss. Recall that loss only needs the predicted score for the target and a fixed number of negative samples. This means that the implementation depends on the final layer of the model.

In this example, our model ends with a linear projection up to output classes, which is quite common. So our model's core returns a "predicted embedding" vector of size embedding\_size for each input in a batch. We then use an "unembedding" projection of the predicted embedding up to size num\_classes to get the score. I.e.

xi=∑jPijej

If we were using full softmax cross entropy, we would compute all scores with a full matrix multiplication. With sampled softmax we can save computation and memory by selecting only the rows of P that are needed for the loss.

One optional tweak is to share noise samples between elements of the batch. Instead of drawing 100 negative samples independently for each element of a batch, we can draw 100 samples to be used for every example in the batch. This improves compute efficiency without harming gradient estimates too much.

Let's see it in action:

inputs, labels = ...

model = ... # returns (batch\_size x embedding\_size)

projection = ... # shape (n\_classes x embedding\_size)

n\_samples = ...

batch\_size = inputs.shape[0]

n\_classes = projection.shape[0]

# 1. Generate predicted embeddings

predicted\_embeddings = model(inputs)

# 2. Get target label scores

label\_scores = (predicted\_embeddings \* projection[labels, :]).sum(-1)

# 3. Sample shared noise & get scores

samples = T.randint(high=n\_classes, size=(n\_samples,))

noise\_scores = predicted\_embeddings @ projection[samples, :].T

noise\_scores += np.log(n\_classes - 1)

# 4. Reject samples matching target label & correct for remaining samples

reject\_samples = labels[:, np.newaxis] == samples[np.newaxis, :]

noise\_scores -= 1e6 \* reject\_samples

noise\_scores -= T.log((n\_samples - reject\_samples.sum(-1, keepdims=True)).float())

# 5. Apply regular softmax cross entropy

scores = T.cat([label\_scores[:, np.newaxis], noise\_scores], dim=1)

loss = T.nn.functional.cross\_entropy(scores, T.zeros(batch\_size, dtype=T.long))

print(float(loss))

loss.backward()

Since this is somewhat complex, let's walk through step-by-step:

Generate predicted embeddings: We run our model as usual, but stop before the final projection. In this demo it's a small matrix, but often it's very large, e.g. 106×256 for a language model of embedding size 256 and a vocabulary of 1 million words.

Get target label scores: Instead of using a simple dense multiplication to get all scores (simply predicted\_embeddings @ projection.T), we only compute scores for the target label & noise samples. Here we use a batched dot product to compute target label scores (note that we only need a score for the associated batch element, labels[0] for predicted\_embeddings[0, :], etc.)

Sample shared noise & get scores: In this case, we've chosen a uniform noise distribution over output classes, since our dataset has balanced classes, drawn using T.randint. We use the samples to index rows of projection, and then, since we're sharing noise samples between batch elements, use a full matrix product with the predicted embeddings to compute noise scores for each batch element. The noise scores get an adjustment for the flat sampling distribution (excluding the target), -log(1/(n\_classes-1)) == log(n\_classes-1).

Reject samples matching target label & correct for remaining samples: We must reject samples matching the target label, so subtract a large value from their score. Then, to correct noise scores (−log⁡k) we count the "actual" k after rejecting matches.

Apply regular softmax cross entropy: The final stage looks slightly odd - we build a full noise & labels scores matrix with the target labels always at index 0 (using T.cat). Then we can use standard softmax cross entropy loss, with the target label fixed at 0.

What does it do?

Let's look at an example. At each point, we'll compare against a full softmax equivalent (for the same example).

On the left, there's the regular full set of scores for a regular softmax, which is the model output for each class. On the right, we have our sampled softmax scores. In this case, we've taken 3 negative samples, which are {bird, cat, dog}. Each sample score has two portions - the portion from the model (dark blue) and a portion from the correction term in our loss equation (1) (light blue).

If we drew more samples, the correction would be decreased, but if we had more classes (so a lower probability of sampling each class) the correction term would increase. This is because the goal of the correction term is to cancel out the sampling procedure, on average. In contrast, the target score for "horse" doesn't have a correction term, as it's always included exactly once.

We won't look at softmax probabilities or loss value for this example, because they're not hugely meaningful. Instead, we'll move straight on to the backward pass.

The backward pass

There isn't anything special about the backward pass when using sampled softmax. As with the forward pass, it's just running on a restricted set of classes - negative samples plus target.

For our example, the gradients look like this:

First, notice that there are only gradients for the target class and noise samples, everything else is zero and does not need to be computed.

Second, we see remaining class gradients are all larger (positive or negative) values. In particular the gradient for "cat" is considerably larger. This is due both to the adjustment and the missing not-sampled classes.

We could now imagine what would happen if we kept repeating this with different samples. We'd get different gradients each time. Sometimes we'd sample "cat" and it would get an extra large update, but sometimes we wouldn't sample it and it would not get an update at all. The compensation in the height of each bar makes the average look similar to the reference full softmax case.

Wrap up

That's sampled softmax. It's a simple idea - we can save memory and computation by randomly sampling a few incorrect labels, rather than computing scores for all labels every single time. The equations weren't too bad, the regular softmax just needed a slight adjustment to balance out the sampling. Perhaps the worst bit was the code, which depends on the final layer of the model to make the most of the computational savings.

I'd probably consider using sampled softmax if I have over 100,000 classes, or if my final classification layer dominates overall execution time or memory use. An obvious application is large word vocabularies, for example in language modelling. However, this has fallen slightly out of favour recently with the rise of subword tokenisation schemes such as byte pair encoding and wordpiece. But it's still a useful tool to have in your box of loss functions.