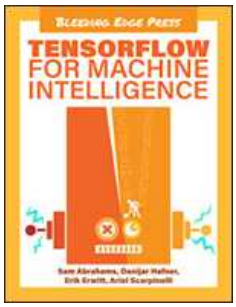


TensorFlow for Machine Intelligence: A Hands-On Introduction to Learning Algorithms

by Sam Abrahams, Danijar Hafner, Erik Erwitt and Ariel Scarpinelli
Bleeding Edge Press. (c) 2016. Copying Prohibited.

Chapter 6: Recurrent Neural Networks and Natural Language Processing

Overview

In the previous chapter, we learned to classify static images. This is a huge application of machine learning, but there is more. In this chapter, we will take a look at sequential models. Those models are model powerful in a way, allowing us to classify or label sequential inputs, generate sequences of text or translate one sequence into another.

What we learn here is not distinct from static classification and regression. Recurrent neural networks provide building blocks that fit well into the toolkit of fully connected and convolutional layers. But let's start with the basics.

Introduction to Recurrent Networks

Let's get started by examining all aspects of Recurrent Networks.

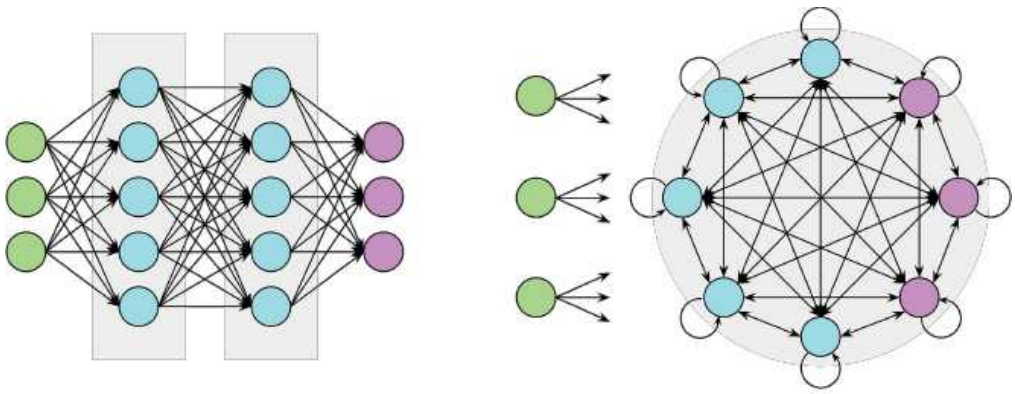
We live in a Temporal World

Many real-world problems are sequential in nature. This includes almost all problems in natural language processing (NLP). Paragraphs are sequences of sentences, sentences are sequences of words, and words are sequences of characters. Closely related, audio and video clips are sequences of frames changing over time. And even stock market prices only make sense when analyzed over time (if at all).

In all of these applications, the order of observations matters. For example, the sentence "I had cleaned my car" can be changed to "I had my car cleaned," meaning that you arranged someone else to do the work. This temporal dependence is even stronger in spoken language since several words can share a very similar sound. For example, "wreck a nice beach" sounds like "recognize speech" and the words can only be reconstructed from context.

If you think about feed-forward neural networks (including convnets) from this perspective, they seem quite limited. Those networks process incoming data in a single forward-pass, like a reflex. These networks assume all of their inputs being independent missing out many patterns in the data. While it is possible to pass inputs equal length and feed the whole sequence into the network, this does not capture the nature of sequences very well.

Recurrent Neural Networks (RNN) are a family of networks that explicitly model time. RNNs build on the same neurons summing up weighted inputs from other neurons. However, neurons are allowed to connect both forward to higher layers and backward to lower layers and form cycles. The hidden activations of the network are remembered between inputs of the same sequence.



Feed Forward Network and Recurrent Network

Various variants of RNNs have been around since the 1980s but were not widely used until recently because of insufficient computation power and difficulties in trainng. Since the invention of architectures like LSTM in 2006 we have seen very powerful applications of RNNs. They work well for sequential tasks in many domains, for example speech recognition, speeach synthesis, connected handwriting recognition, time-series forecasting, image caption generation, and end-to-end machine translation.

In the following sections of this chapter, we first take an in-depth look at RNNs and how to optimize them, including the required mathematical background. We then introduce variations of RNNs that help overcome certain limitations. With those tools at hand, we dive into four natural language processing tasks and apply RNNs to them. We will walk through all steps of the tasks including data handling, model design, implementation, and training in TensorFlow.

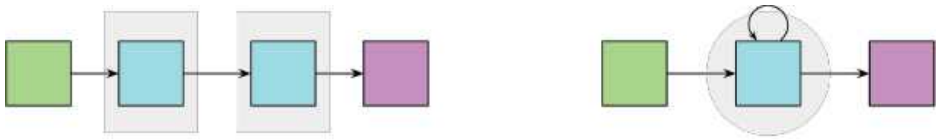
Approximating Arbitrary Programs

Let's start by introducing RNNs and gaining some intuition. Previously introduced feed-forward networks operate on fixed size vectors. For example, they map the pixels of 28x28 image to the probabilities of 10 possible classes. The computation happens in a fixed number of steps,

namely the number of layers. In contrast, recurrent networks can operate on variable length sequences of vectors, either as input, output or both.

RNNs are basically arbirary directed graphs of neurons and weights. *Input neurons* have on incoming connections because their activation is set by the input data anyway. The *output neurons* are just set of neurons in the graph that we read the prediction from. All other neurons in the graph are referred to as *hidden neurons*.

The computation performed by an RNN is analogous to a normal neural network. At each time step, we show the network the next frame of the input sequence by setting the input neurons. In contrast to forward networks, we cannot discard hidden activations because they serve as additional inputs at the next time step. The current hidden activations of an RNN are called *state*. At the beginning of each sequence, we usually start with an empty state, initialized to zeros.



Short Notation of FFNN and RNN

The state of an RNN depends on the current input and the previous state, which in turn depends on the input and state before that. Therefore, the state has indirect access to all previous inputs of the sequence and can be interpreted as a working memory.

Let's make an analogy to computer programs. Say we want to recognize the letters from an image of handwritten text. We would try and solve this with computer program in Python using variables, loops, conditionals. Feel free to try this, but I think it would be very hard to get this to work robustly.

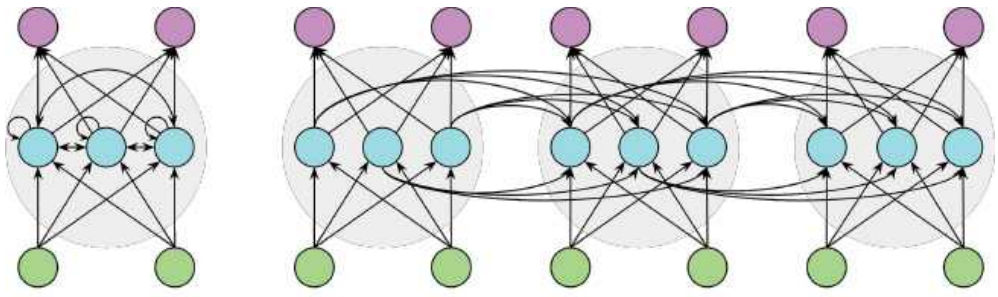
The good news is that we can train an RNN from example data instead. As we would store intermediate information in variables, the RNN learns to store intermediate information in its state. Similarly, the weight matrix of an RNN defines defines the programm it executes, deciding what inputs to store in hidden activation and how to combine activations to new activations and outputs.

In fact, RNNs with sigmoid activations were proven to be Turing-complete by Schäfer and Zimmermann in 2006. Given the right weights, RNNs can thus compute any computable program. This is a theoreticaly property since there is no method to find the perfect weights for a task. However, we can already get very good results using gradient descent, as described in the next section.

Before we look into optimizing RNNs, you might ask why we even need RNNs if we can write Python programs instead? Well, the space of possible weight matrices is much easier to explore automatically than the space of possible C programs.

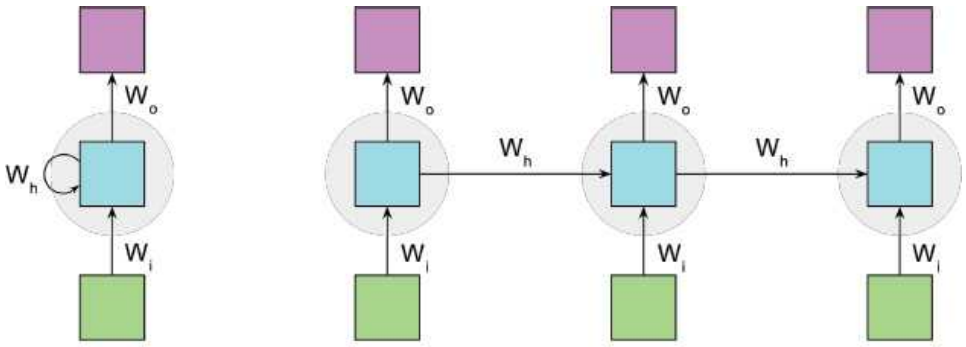
Backpropagation Through Time

Now that we have an idea of what an RNN is and why it is a cool architecture, let's take a look at how to find a good weight matrix, or how to optimize the weights. As with forward networks, the most pupular optimization method is based on Gradient Descent. However, it is not straight-forward how to backpropagate the error in this dynamic system.



Unfolding a Recurrent Network in Time

The trick for optimizing RNNs is that we can unfold them in time (also referred to as unrolling) to optimize them the same way we optimize forward networks. Let's say we want to operate on sequence of length ten. We can then copy the hidden neurons ten times spanning their connections from one copy to the next one. By doing this, we get rid of recurrent connections without changing the semantics of the computation. This yields a forward network now, with the corresponding weights between time steps being tied to the same strengths. Unfolding an RNN in time does not change the computation, it is just another view.

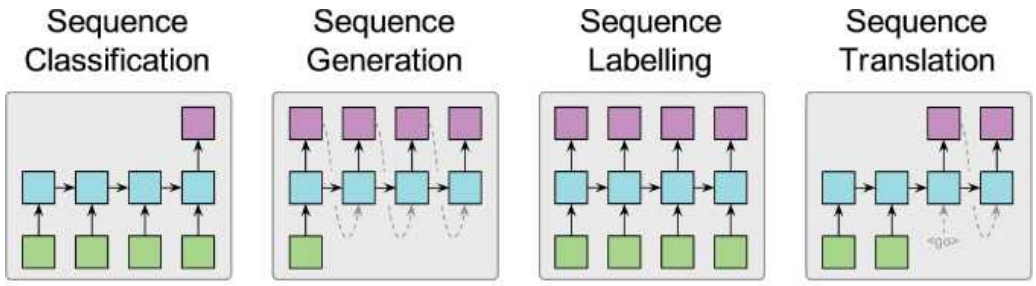


Unfolding an RNN in Time in Short Notation

We can now apply standard backpropagation through this unfolded RNN in order to compute the gradient of the error with respect to the weights. This algorithm is called *Back-Propagation Through Time* (BPTT). It will return a derivative for each weight in time, including those that are tied together. To keep tied weights at the same value, we handle them as tied weights are normally handeled, that is, summing up their gradients. Note that this equals the way convolutional filters are handeled in convents.

Encoding and Decoding Sequences

The unfolded view of RNNs from the last chapter is not only useful for optimization. It also provides an intuitive way for visualizing RNNs and their input and output data. Before we get to the implementation, we will take a quick look at what mappings RNNs can perform. Sequential tasks can come in several forms: Sometimes, the input is a sequence and the output is a single vector, or the other way around. RNNs can handle those and more complicated cases well.



Common Mappings with Recurrent Networks

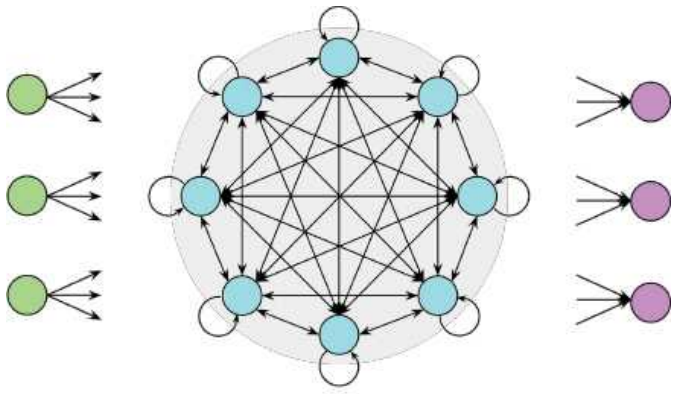
Sequence labelling is the case you probably though of during the ealier sections. We have sequences as input and train the network to produce the right output for each frame. We are basically mapping from one sequence to another sequence of the same length.

In the *sequence classification* setting, we have sequential inputs that each have a class. We can train RNNs in this setting by only selecting the output at the last time frame. During optimization, the errors will flow back through all time steps to update the weights in order to collect and integrate useful information at each time step.

Sequence generation is the opposite case where we have a single starting point, for example a class label, that we want to generate sequences from. To generate sequences, we feed the output back into the network as next input. This makes sense since the actual output is often different from the neural network output. For example, the network might output a distribution over all classes but we only choose the most likely one.

In both sequence classification and sequence generation, we can see the single vector as dense representations of information. In first case, we encode the sequence into a dense vector to predict a class. In the second case, we decode a dense vector back into a sequence.

We can combine these approaches for *sequence translation* where we first encode a sequence of one domain, for example English. We then decode the last hidden activation back into a sequence of another domain, for example French. This works with a single RNN but when input and output are conceptually different, it can make sense to use two different RNNs and initialize the second one with the last activation of the first one. When using a single network, we need to pass a special token as input after the sequence so that the network can learn when it should stop encoding and start decoding.



Recurrent Network with Output Projection

Most often, we will use a network architecture called RNNs with *output projections*. This is an RNN with fully-connected hidden units and inputs and output mapping to or from them, respectively. Another way to look at this is that we have an RNN where all hidden units are outputs and another feed-forward layer stacked on top. You will see that this is how we implement RNNs in TensorFlow because it both is convenient and allows us to specify different activation functions to the hidden and output units.

Implementing Our First Recurrent Network

Let's implement what we have learned so far. TensorFlow supports various variants of RNNs that can be found in the `tf.nn.rnn_cell` module. With the `tf.nn.dynamic_rnn()` operation, TensorFlow also implements the RNN dynamics for us.

There is also a version of this function that adds the unfolded operations to the graph instead of using a loop. However, this consumes considerably more memory and has no real benefits. We therefore use the newer `dynamic_rnn()` operation.

As parameters, `dynamic_rnn()` takes a recurrent network definition and the batch of input sequences. For now, the sequences all have the same length. The function creates the required computations for the RNN to the compute graph and returns two tensors holding the outputs and hidden states at each time step.

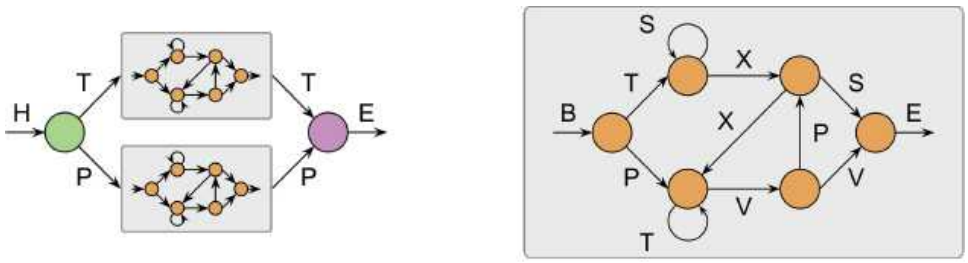
```
import tensorflow as tf

# The input data has dimensions batch_size * sequence_length * frame_size.
# To not restrict ourselves to a fixed batch size, we use None as size of
# the first dimension.
sequence_length = ...
frame_size = ...
data = tf.placeholder(tf.float32, [None, sequence_length, frame_size])

num_neurons = 200
network = tf.nn.rnn_cell.BasicRNNCell(num_neurons)

# Define the operations to simulate the RNN for sequence_length steps.
outputs, states = tf.nn.dynamic_rnn(network, data, dtype=tf.float32)
```

Now that we have defined the RNN and unfolded it in time, we can just load some data and train the network using one of TensorFlow's optimizers, for example `tf.train.RMSPropOptimizer` or `tf.train.AdamOptimizer`. We will see examples of this in the later sections of this chapter where we approach practical problems with the help of RNNs.

Vanishing and Exploding Gradients

In the last section, we defined RNNs and unfolded them in time in order to backpropagate errors and apply gradient descent. However, this model would not perform very well as it stands, especially it fails to capture long-term dependencies between input frames as needed for NLP tasks for example.

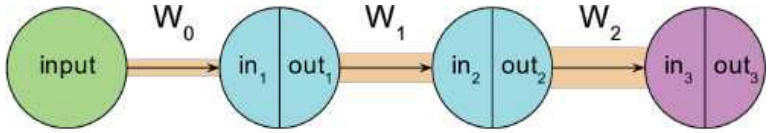
Below is an example task involving a long-term dependency where an RNN should classify whether the input sequence is part of the given grammar or not. In order to perform this task, the network has to remember the very first frame of the sequence with many unrelated frames following. Why is this a problem for the conventional RNNs we have seen so far?



A Grammar Including Long-Term Dependencies

The reason why it is difficult for an RNN to learn such long-term dependencies lies in how errors are propagated trough the network during optimization. Remember that we propagate the errors through the unfolded RNN in order to compute the gradients. For long sequences, the unfolded network gets very deep and has many layers. At each layer, backpropagation scales the error from above in the network by the local derivatives.

If most of local derivatives are much smaller than the value of one, the gradient gets scaled down at every layer causing it to shrink exponentially and eventually, *vanish*. Analogously, many local derivatives being greater than one cause the gradient to *explode*.



Unstable Gradients in Deep Networks

Let's compute the gradient of this example network with just one hidden neuron per layer in order to get a better understanding of this problem.

For each layer ***i*** the local derivatives $f'(in_i) * W_i^T$ get multiplied together:

$$\frac{\partial C}{\partial in_1} = \frac{\partial C}{\partial out_3}\frac{\partial out_3}{\partial in_3} * \frac{\partial in_3}{\partial out_2}\frac{\partial out_2}{\partial in_2} * \frac{\partial in_2}{\partial out_1}\frac{\partial out_1}{\partial in_1}$$

Resolving the derivatives yields:

$$cost'(f(out_3)) * f'(in_3) * W_2^T * f'(in_2) * W_1^T * f'(in_1)$$

As you can see, the error term contains the transposed weight matrix several times as a multiplicative term. In our toy network, the weight matrix contains just one element and it's easy to see that the terms gets close to zero or infinity when most weights are smaller or larger than one. In a larger network with real weight matrices, the same problem occurs when the eigen values of thei weight matrices are smaller or larger than one.

This problem actually exists in any deep networks, not just recurrent ones. However, in RNNs the connections between time steps are tied each. Therefore, local derivatives of such weights are either all lesser or all greater than one and the gradient is alawys scaled in the same direction for each weight in the original (not unfolded) RNN. Therefore, the problem of vanishing and exploding gradients is *more prominent in RNNs than in forward networks*.

There are a couple of problems with very small or very large gradients. With elements of the gradient close to zero or infinity, learning stagnates or diverges, respectively. Moreover, we are optimizing numerically and floating point precision comes into play distorting the gradient. This problem, also known as the *fundamental problem of deep learning* has been studied and approached by many researchers in the last years. The most popular solution is an RNN architecture called *Long-Short Term Memory* (LSTM) that we will look at in the next section.

Long-Short Term Memory

Proposed in 1997 by Hochreiter & Schmidhuber, LSTM is a special form of RNN that is designed to overcome the *vanishing and exploding gradient problem*. It works significantly better for learning long-term dependencies and has become a defacto-standard for RNNs. Since then several variations of LSTM have been proposed that are also implemented in TensorFlow and that we will highlight later in this section.

To cope with the problem of vanishing and exploding gradients, the LSTM architecture replaces the normal neurons in an RNN with so-called *LSTM cells* that have a little memory inside. Those cells are wired together as they are in a usual RNN but they have an internal state that helps to remember errors over many time steps.

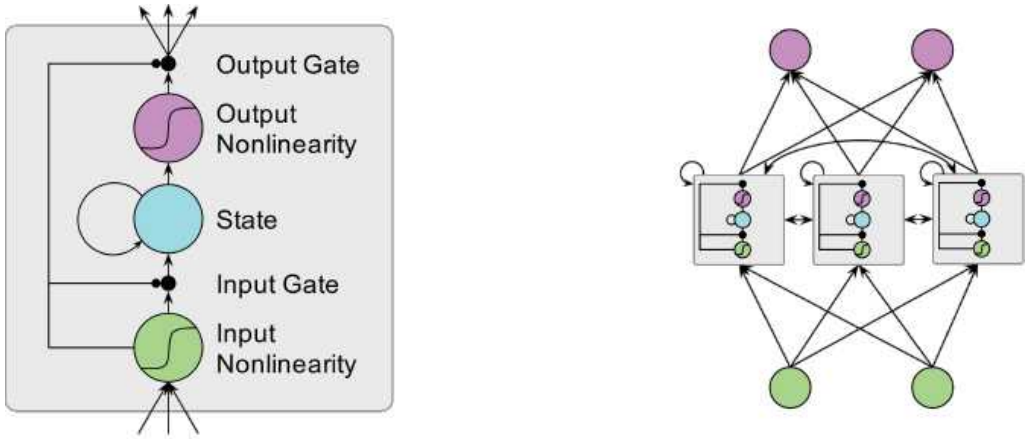
The trick of LSTM is that this internal state has a self-connection with a fixed weight of one and a linear activation function, so that its local derivative is always one. During backpropagation, this so called *constant error carousel* can carry errors over many time steps without having the gradient vanish or explode.

$$e_t = f'(in_t) * w * e_{t+1} = 1.0$$

While the purpose of the internal state is to deliver errors over many time steps, the LSTM architecture leaves learning to the *sourrounding gates* that have non-linear, usually sigmoid, activation functions. In the original LSTM cell, there are two gates: One learns to scale the incoming activation and one learns to scale the outgoing activation. The cell can thus learn when to incorporate or ignore new inputs and when to release the feature it represents to other cells. The input to a cell is feeded into all gates using individual weights.

We also refer to recurrent networks as layers because we we can use them as part of larger architectures. For example, we could first feed the time steps through several convolution and pooling layers, then process these outputs with an LSTM layer and add a soft-max layer ontop of the LSTM activation at the last time step.

TensorFlow provides such an LSTM network with the `LSTMCell` class that can be used as a drop-in replacement for `BasicRNNCell` but also provides some additional switches. Despite its name, this class represents a whole LSTM layer. In the later sections we will see how to connect LSTM layers to other networks in order to form larger architectures.

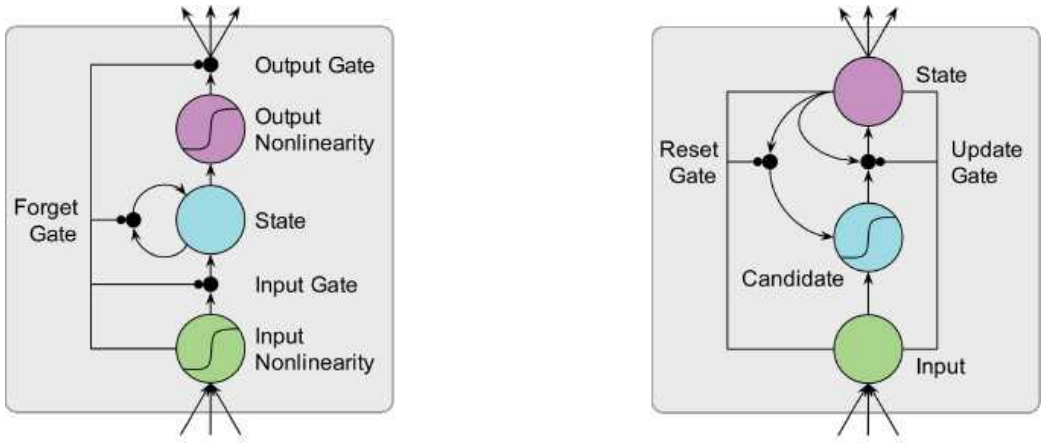


Long Short-Term Memory (LSTM)

Architecture Variations

A popular extension to LSTM is to add a *forget gate* scaling the internal recurrent connection, allowing the network to learn to forget (*Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. "Learning to forget: Continual prediction with LSTM." Neural computation 12.10 (2000): 2451-2471*.). The derivative of the internal recurrent connection is now the activation of the forget gate and can differ from the value of one. The network can still learn to leave the forget gate closed as long as remembering the cell context is important.

It is important to initialize the forget gate to a value of one so that the cell starts in a remembering state. Forget gates are the default is almost all implementations nowadays. In TensorFlow, we can initialize the bias values of the forget gates by specifying the `forget_bias` parameter to the LSTM layer. The default is the value one and usually its best to leave it that way.

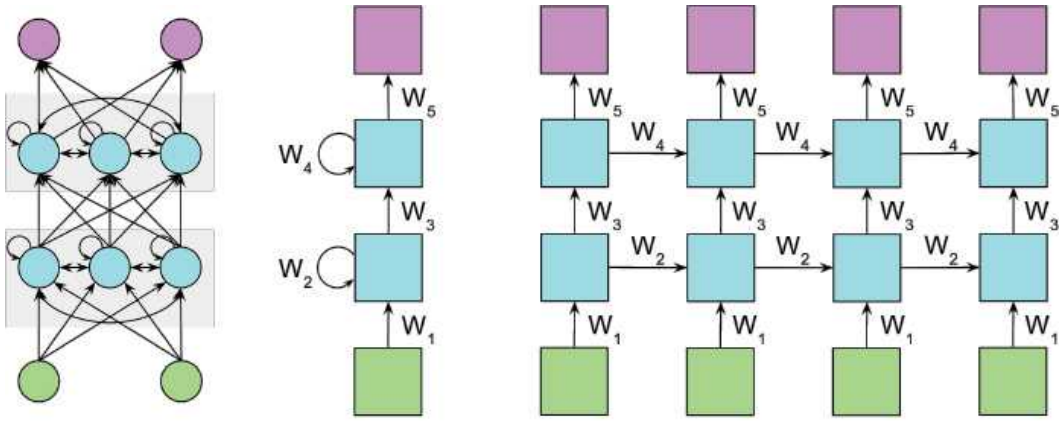


LSTM with Forget Gate and Gated Recurrent Unit (GRU)

Another extension are so called *peephole connections*, which allows the gates to look at the cell state (*Gers, Felix A., Nicol N. Schraudolph, and Jürgen Schmidhuber. "Learning precise timing with LSTM recurrent networks." The Journal of Machine Learning Research 3 (2003): 115-143*.). The authors claim that peephole connections are benefitial when the task involves precise timing and intervals. TensorFlow's LSTM layer supports peephole connections. They can be activated by passing the `use_peepholes=True` flag to the LSTM layer.

Based on the idea of LSTM, an alternative memory cell called *Gated Recurrent Unit* (GRU) has been proposed in 2014 (*Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014)*.). In contrast to LSTM, GRU has a simpler architecture and requires less computation while yielding very similar results. GRU has no output gate and combines the input and forget gates into a single *update gate*.

This update gate determines how much the internal state is blended with a candidate activation. The candidate activation is computed from a fraction of the hidden state determined by the so-called *reset gate* and the new input. The TensorFlow GRU layer is called `GRUCell` and have no parameters other than the number of cells in the layer. For further reading, we suggest the 2015 paper by Jozefowicz et al. who empirically explored recurrent cell architectures (*Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. "An empirical exploration of recurrent network architectures." Proceedings of the 32nd International Conference on Machine Learning (ICML-15). 2015*.).



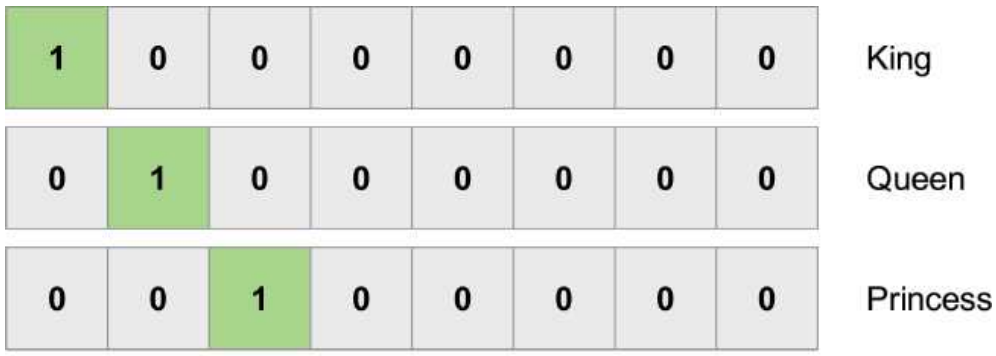
Multiple Layers in Recurrent Networks

So far we looked at RNNs with fully connected hidden units. This is the most general architecture since the network can learn to set unneeded weights to zero during trainng. However, it is common to stack two or more layers of fully-connected RNNs on top of each other. This can still be seen as one RNN that has some structure in its connections. Since nformation can only flow upward between layers, multi-layer RNNs have less weights than a huge fully connected RNN and tend to learn more abstract features.

Word Vector Embeddings

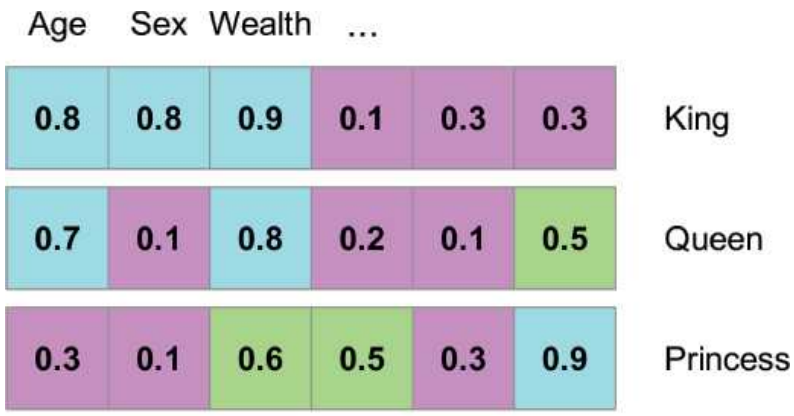
In this section we will implement a model to learn word embeddings, a very powerful way to represent words for NLP tasks. The topic of word vector embeddings has recently gained popularity since methods became efficient enough to run on large text corpora. We do not use an RNN for this task yet but we will rely on this section in all further tasks. If you are familiar with the concept of word vectors and tools like word2vec but are not interested in implementing it yourself, feel free to skip ahead to the next section.

Why to represent words as vectors? The most straight-forward way to feed words into a learning system is *one-hot encoded*, that is, as a vector of the vocabulary size with all elements zero except for the position of that word set to one. There are two problems with this approach: First, the vectors are very long for real applications since there are many different words in a natural language. Second, this one-hot representation does not provide any semantic relatedness between words that certainly exists.



One-Hot Encoded Representation of Words

As a solution to the semantic relatedness, the idea of representing words by their cooccurrences has been around for a long time. Basically, we run over a large corpus of text and for each word count the surrounding words within a distance of, say, five. Each word is then represented by the normalized counts of nearby words. The idea behind this is that words that are used in similar contexts are similar in a semantic way. We could then compress the occurance vectors to fewer dimensions by applying PCA or similar a similar method to get denser representations. While this approach leads to quite good performance, it requires us to keep track of the whole cooccurrence that is a square matrix of the size of our vocabulary.



Distributed Representation of Words

In 2013, Mikolov et al. came up with a practical and efficient way to compute word representations from context. The paper is: *Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013)*. Their *skip-gram model* starts with random representations and has a simple classifier that tries to predict a context word from the current word. The errors are propagated through both the classifier weights and the word representations and we adjust both to reduce the prediction error. It has been found that training this model over a large corpus makes the representation vectors approximate compressed co-occurance vectors. We will now implement the skip-gram model in TensorFlow.

Preparing the Wikipedia Corpus

Before going into details of the skip-gram model, we prepare our dataset, an English Wikipedia dump in this case. The default dumps contain the full revision history of all pages but we already have enough data with the about 100GB of text from current page versions. This exercise also works for other languages and you can access an overview of the available dumps at the Wikimedia Downloads website: https://dumps.wikimedia.org/backup-index.html.

```
import bz2
import collections
import os
import re


class Wikipedia:

    def __init__(self, url, cache_dir, vocabulary_size=10000):
        pass

    def __iter__(self):
```

```
        """Iterate over pages represented as lists of word indices."""
        pass

    @property
    def vocabulary_size(self):
        pass

    def encode(self, word):
        """Get the vocabulary index of a string word."""
        pass

    def decode(self, index):
        """Get back the string word from a vocabulary index."""
        pass

    def _read_pages(self, url):
        """
        Extract plain words from a Wikipedia dump and store them to the pages
        file. Each page will be a line with words separated by spaces.
        """
        pass
    def _build_vocabulary(self, vocabulary_size):
        """
        Count words in the pages file and write a list of the most frequent
        words to the vocabulary file.
        """
        pass

    @classmethod
    def _tokenize(cls, page):
        pass
```

There are a couple of steps to perform in order to get the data into the right format. As you might have seen earler in this book, data collection and cleaning is both a demanding and important task. Ultimately, we would like to iterate over Wikipedia pages represented as one-hot encoded words. We do this is in multiple steps:

1. Download the dump and extract pages and their words.
2. Count words to form a vocabulary of the most common words.
3. Encode the extracted pages using the vocabulary.

The whole corpus does not fit into main memory easily, so we have to perform these operations on data streams by reading the file line by line and write the intermediate results back to disk. This way, we have checkpoints between the steps so that we don't have to start all over if something crashes. We use the following class to handle the Wikipedia processing. In the `__init__()` you can see the checkpointing logic using file-existance checks.

```
def __init__(self, url, cache_dir, vocabulary_size=10000):
    self._cache_dir = os.path.expanduser(cache_dir)
    self._pages_path = os.path.join(self._cache_dir, 'pages.bz2')
    self._vocabulary_path = os.path.join(self._cache_dir, 'vocabulary.bz2')
    if not os.path.isfile(self._pages_path):
        print('Read pages')
        self._read_pages(url)
    if not os.path.isfile(self._vocabulary_path):
        print('Build vocabulary')
        self._build_vocabulary(vocabulary_size)
    with bz2.open(self._vocabulary_path, 'rt') as vocabulary:
        print('Read vocabulary')
        self._vocabulary = [x.strip() for x in vocabulary]
    self._indices = {x: i for i, x in enumerate(self._vocabulary)}

def __iter__(self):
    """Iterate over pages represented as lists of word indices."""
    with bz2.open(self._pages_path, 'rt') as pages:
        for page in pages:
            words = page.strip().split()
            words = [self.encode(x) for x in words]
            yield words
```

```
    @property
    def vocabulary_size(self):
        return len(self._vocabulary)

    def encode(self, word):
        """Get the vocabulary index of a string word."""
        return self._indices.get(word, 0)

    def decode(self, index):
        """Get back the string word from a vocabulary index."""
        return self._vocabulary[index]
```

As you have noticed, we still have to implement two important functions of this Wclass. The first one, `_read_pages()` will download the Wikipedia dump which comes as a compressed XML file, iterate over the pages and extract the plain words to get rid of any formatting. To read the compressed file, we need the `bz2` module that provides an `open()` function that works similar to its standard equivalent but takes care of compression and decompression, even when streaming the file. To save some disk space, we will also use this compression for the intermediate results. The regex used to extract words just captures any sequence of consecutive letter and individual occurences of some special characters.

```
from lxml import etree

TOKEN_REGEX = re.compile(r'[A-Za-z]+|[!?.:,()]')

def _read_pages(self, url):
    """
    Extract plain words from a Wikipedia dump and store them to the pages
    file. Each page will be a line with words separated by spaces.
    """
    wikipedia_path = download(url, self._cache_dir)
    with bz2.open(wikipedia_path) as wikipedia, \
            bz2.open(self._pages_path, 'wt') as pages:
        for _, element in etree.iterparse(wikipedia, tag='{*}page'):
            if element.find('./{*}redirect') is not None:
                continue
            page = element.findtext('./{*}revision/{*}text')
            words = self._tokenize(page)
            pages.write(' '.join(words) + '\n')
            element.clear()

@classmethod
def _tokenize(cls, page):
    words = cls.TOKEN_REGEX.findall(page)
    words = [x.lower() for x in words]
    return words
```

We need a vocabulary of words to use for the one-hot encoding. We can then encode each word by its intex in the vocabulary. To remove some misspelled or very unkommon words, the vocabulary only contains the the `vocabulary_size - 1` most common words and an `<unk>` token that will be used for every word that is not in the vocabulary. This token will also give us a word-vector that we can use for unseen words later.

```
def _build_vocabulary(self, vocabulary_size):
    """
    Count words in the pages file and write a list of the most frequent
    words to the vocabulary file.
    """

    counter = collections.Counter()
    with bz2.open(self._pages_path, 'rt') as pages:
        for page in pages:
            words = page.strip().split()
            counter.update(words)
    common = ['<unk>'] + counter.most_common(vocabulary_size - 1)
    common = [x[0] for x in common]
    with bz2.open(self._vocabulary_path, 'wt') as vocabulary:
        for word in common:
            vocabulary.write(word + '\n')
```

Since we extracted the plain text and defined the encoding for the words, we can form training examples of it one the fly. This is nice since storing the examples would require a lot of storage space. Most of the time will be spent for the training anyway, so this doesn't impact

performance by much. We also want to group the resulting examples into batches to train them more efficiently. We will be able to use very large batches with this model because the classifier does not require a lot of memory.

So how do we form the training examples? Remember that the *skip-gram model* predicts context words from current words. While iterating over the text, we create training examples with the current word as data and its surrounding words as targets. For a context size of $R = 5$, we would thus generate ten training examples per word, with the five words to the left and right being the targets. However, one can argue that close neighbors are more important to the semantic context than far neighbors. We thus create less training examples with far context words by randomly choosing a context size in range $[1, D = 10]$ at each word.

```
def skipgrams(pages, max_context):
    """Form training pairs according to the skip-gram model."""
    for words in pages:
        for index, current in enumerate(words):
            context = random.randint(1, max_context)
            for target in words[max(0, index - context): index]:
                yield current, target
            for target in words[index + 1: index + context + 1]:
                yield current, target


def batched(iterator, batch_size):
    """Group a numerical stream into batches and yield them as Numpy ar-
rays."""
    while True:
        data = np.zeros(batch_size)
        target = np.zeros(batch_size)
        for index in range(batch_size):
            data[index], target[index] = next(iterator)
        yield data, target
```

Model structure

Now that we got the Wikipedia corpus prepared, we can define a model to compute the word embeddings.

```
class EmbeddingModel:

    def __init__(self, data, target, params):
        self.data = data
        self.target = target
        self.params = params
        self.embeddings
        self.cost
        self.optimize

    @lazy_property
    def embeddings(self):
        pass

    @lazy_property
    def optimize(self):
        pass

    @lazy_property
    def cost(self):
        pass
```

Each word starts off being represented by a random vector. From the intermediate representation of a word, a classifier will then try to predict the current representation of one of its context words. We will then propagate the errors to tweak both the weights and the representation of the input word. The thus use a `tf.Variable` for the representations.

```
@lazy_property
def embeddings(self):
    initial = tf.random_uniform(
        [self.params.vocabulary_size, self.params.embedding_size],
        -1.0, 1.0)
    return tf.Variable(initial)
```

We use the `MomentumOptimizer` that is not very clever but has the advantage of being very fast. This makes it play nicely with our large Wikipedia corpus and the idea behind skip-gram to prefer more data over clever algorithms.

```
@lazy_property
def optimize(self):
    optimizer = tf.train.MomentumOptimizer(
        self.params.learning_rate, self.params.momentum)
    return optimizer.minimize(self.cost)
```

The only missing part of our model is the classifier. This is the heart of the successful *skip-gram* model and we will now take a look at how it works.

Noise Contrastive Classifier

There are multiple cost functions for the skip-gram model but one that has been found to work very well is *noise-constrastive estimation loss*. Ideally, we not only want the predictions to be close to the targets but also far from words that are not targets for the current word. This could be nicely modelled as a softmax classifier but we do not want to compute and train the outputs for all words in the alphabet every time. The idea is to always use some new random vectors as negative examples, also called contrastive examples. Over enough training iterations this averages to the softmax classifier while only requiring tens of classes. TensorFlow provides a convenient `tf.nn.nce_loss` function for this.

```
@lazy_property
def cost(self):
    embedded = tf.nn.embedding_lookup(self.embeddings, self.data)
    weight = tf.Variable(tf.truncated_normal(
        [self.params.vocabulary_size, self.params.embedding_size],
        stddev=1.0 / self.params.embedding_size ** 0.5))
    bias = tf.Variable(tf.zeros([self.params.vocabulary_size]))
    target = tf.expand_dims(self.target, 1)
    return tf.reduce_mean(tf.nn.nce_loss(
        weight, bias, embedded, target,
        self.params.contrastive_examples,
        self.params.vocabulary_size))
```

Training the model

We prepared the corpus and defined the model. Here is the remaining code to put things together. After training, we store the final embeddings into another file. The example below uses only a subset of Wikipedia that already takes about 5 hours to train on an average CPU. To use the full corpus, just switch the url to https://dumps.wikimedia.org/enwiki/20160501/enwiki-20160501- pages-meta-current.xml.bz2.

As you can see, we make use of a `AttrDict` class. This is equivalent to a Python `dict` except that we can access keys as if they were attributes, for example `params.batch_size`. For more details, please see the chapter *Code Structure and Utilities*.

```
params = AttrDict(
    vocabulary_size=10000,
    max_context=10,
    embedding_size=200,
    contrastive_examples=100,
    learning_rate=0.5,
    momentum=0.5,
    batch_size=1000,
)

data = tf.placeholder(tf.int32, [None])
target = tf.placeholder(tf.int32, [None])
model = EmbeddingModel(data, target, params)

corpus = Wikipedia(
    'https://dumps.wikimedia.org/enwiki/20160501/' \
    'enwiki-20160501-pages-meta-current1.xml-p000000010p000030303.bz2',
    '/home/user/wikipedia',
    params.vocabulary_size)
examples = skipgrams(corpus, params.max_context)
batches = batched(examples, params.batch_size)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
average = collections.deque(maxlen=100)
for index, batch in enumerate(batches):
    feed_dict = {data: batch[0], target: batch[1]}
    cost, _ = sess.run([model.cost, model.optimize], feed_dict)
    average.append(cost)
    print('{}: {:5.1f}'.format(index + 1, sum(average) / len(average)))
```

```
embeddings = sess.run(model.embeddings)
np.save('/home/user/wikipedia/embeddings.npy', embeddings)
```

After about five hours of training, this we will get the learned embeddings as a stored Numpy array. While we will use the embeddings in the later chapter, you don't have to compute them yourself, if you don't want to. There are pre-trained word embeddings available online and we will point to them later when we need them.

Sequence Classification

Sequence classification is a problem setting where we predict a class for the whole input sequence. Such problems are common in many fields including genomics and finance. A prominent from NLP is sentiment analysis: Predicting the attitude towards a given topic from user-written text. For example, one could predict the sentiment of tweets mentioning a certain candidate in an election and use that to forecast the election results. Another example is predicting product or movie ratings from written reviews. This is used as a benchmark task in the NLP community because reviews often contain numerical ratings that make for convenient target values.

We will use a dataset of movie reviews from the *International Movie Database* with the binary targets *positive* or *negative*. On this dataset, navive methods that just look at the existance of words tend to fail because of negations, irony and ambiguity in language in general. We will build a recurrent model operating on the word vectors from the last section. The recurrent network will see a review word-by-word. From the *activation at the last word*, we will train a classifier to predict the sentiment of the whole review. Because we train the architecture end-to-end, the RNN will to collect and encode the useful information from the words that will be most valuable for the later classification.

Imdb Movie Review Dataset

The movie review dataset is offered by Stanford University's AI department: http://ai.stanford.edu/~amaas/data/sentiment/. It comes as a compressed `tar` archive where positive and negative reviews can be found as text files in two according folders. We apply the same pre-processing to the text as in the last section: Extracting plain words using a regular expression and converting to lower case.

```
import tarfile
import re


class ImdbMovieReviews:

    DEFAULT_URL = \
        'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz'
    TOKEN_REGEX = re.compile(r'[A-Za-z]+|[!?.:,()]')

    def __init__(self, cache_dir, url=None):
        self._cache_dir = cache_dir
        self._url = url or type(self).DEFAULT_URL
    def __iter__(self):
        filepath = download(self._url, self._cache_dir)
        with tarfile.open(filepath) as archive:
            for filename in archive.getnames():
                if filename.startswith('aclImdb/train/pos/'):
                    yield self._read(archive, filename), True
                elif filename.startswith('aclImdb/train/neg/'):
                    yield self._read(archive, filename), False

    def _read(self, archive, filename):
        with archive.extractfile(filename) as file_:
            data = file_.read().decode('utf-8')
            data = type(self).TOKEN_REGEX.findall(data)
            data = [x.lower() for x in data]
            return data
```

Using the Word Embeddings

As explained in the *Word Vector Embeddings* section, embeddings are semantically richer that one-hot encoded words. We can help our RNN by letting it operate on the embedded words of movie reviews rather than one-hot encoded words. For this, we will use the vocabulary and embeddings that we computed computed in the referenced section. The code should be straight forward. We just use the vocabulary to determine the index of a word and use that index to find the right embedding vector. The following class also padds the sequences to the same length so we can easily fit batches of multiple reviews into your network later.

```
import bz2
import numpy as np
```

```
class Embedding:

    def __init__(self, vocabulary_path, embedding_path, length):
        self._embedding = np.load(embedding_path)
        with bz2.open(vocabulary_path, 'rt') as file_:
            self._vocabulary = {k.strip(): i for i, k in enumerate(file_)}
        self._length = length

    def __call__(self, sequence):
        data = np.zeros((self._length, self._embedding.shape[1]))
        indices = [self._vocabulary.get(x, 0) for x in sequence]
        embedded = self._embedding[indices]
        data[:len(sequence)] = embedded
        return data
    @property
    def dimensions(self):
        return self._embedding.shape[1]
```

Sequence Labelling Model

We want to classify the sentiment of text sequences. Because this is a supervised setting, we pass two placeholders to the model: one for the input `data`, or the sequence, and one for the `target` value, or the sentiment. We also pass in the `params` object that contains configuration parameters like the size of the recurrent layer, its cell architecture (LSTM, GRU, etc), and the optimizer to use. We will now implement the properties and discuss them in detail.

```
class SequenceClassificationModel:

    def __init__(self, data, target, params):
        self.data = data
        self.target = target
        self.params = params
        self.prediction
        self.cost
        self.error
        self.optimize

    @lazy_property
    def length(self):
        pass

    @lazy_property
    def prediction(self):
        pass

    @lazy_property
    def cost(self):
        pass

    @lazy_property
    def error(self):
        pass

    @lazy_property
    def optimize(self):
        pass

    @staticmethod
    def _last_relevant(output, length):
        pass
```

First, we obtain the *lengths of sequences* in the current data batch. We need this since the data comes as a single tensor, padded with zero vectors to the longest review length. Instead of keeping track of the sequence lengths of every review, we just compute it dynamically in TensorFlow. To get the length per sequence, we collapse the word vectors using the maximum on the absolute values. The resulting scalars will be zero for zero vectors and larger than zero for any real word vector. We then discretize these values to zero or one using `tf.sign()` and sum up the results along the time steps to obtain the length of each sequence. The resulting tensor has the length of batch size and contains a scalar length for each sequence.

```
@lazy_property
def length(self):
```

```
    used = tf.sign(tf.reduce_max(tf.abs(self.data), reduction_indices=2))
    length = tf.reduce_sum(used, reduction_indices=1)
    length = tf.cast(length, tf.int32)
    return length
```

Softmax from last relevant activation

For the prediction, we define an RNN as usual. However, this time we want to augment it by *stacking a softmax layer ontop of its last activation*. For the RNN, we use a cell type and cell count defined in the `params` object. We use the already defined `length` property to only show rows of the batch to the RNN up to their length. We can then fetch the last output activation of each sequence and feed that into a softmax layer. Defining the softmax layer should be pretty straight forward if you've followed the book up to this section.

Note that the last relevant output activation of the RNN has a different index for each sequence in the training batch. This is because each review has a different length. We already know the length of each sequence, so how do we use that to select the last activations? The problem is that we want to index in the dimension of time steps, which is the second dimension in the batch of shape `sequences x time_steps x word_vectors`.

```
@lazy_property
def prediction(self):
    # Recurrent network.
    output, _ = tf.nn.dynamic_rnn(
        self.params.rnn_cell(self.params.rnn_hidden),
        self.data,
        dtype=tf.float32,
        sequence_length=self.length,
    )
    last = self._last_relevant(output, self.length)
    # Softmax layer.
    num_classes = int(self.target.get_shape()[1])
    weight = tf.Variable(tf.truncated_normal(
        [self.params.rnn_hidden, num_classes], stddev=0.01))
    bias = tf.Variable(tf.constant(0.1, shape=[num_classes]))
    prediction = tf.nn.softmax(tf.matmul(last, weight) + bias)
    return prediction
```

As of now, TensorFlow only supports indexing along the first dimension, using `tf.gather()`. We thus flatten the first two dimensions of the output activations from their shape of `sequences x time_steps x word_vectors` and construct an index into this resulting tensor. The index takes into account the start indices for each sequence in the flat tensor and adds the sequence length to it. Actually, we only add `length - 1` so that we select the last valid time step.

```
@staticmethod
def _last_relevant(output, length):
    batch_size = tf.shape(output)[0]
    max_length = int(output.get_shape()[1])
    output_size = int(output.get_shape()[2])
    index = tf.range(0, batch_size) * max_length + (length - 1)
    flat = tf.reshape(output, [-1, output_size])
    relevant = tf.gather(flat, index)
    return relevant
```

We will be able to train the whole model end-to-end with TensorFlow propagating the errors through the softmax layer and the used time steps of the RNN. The only thing that is missing for training is a cost function.

Gradient clipping

For sequence classification, we can use any cost function that makes sense for classification because the model output is just a probability distribution over the available classes. In our example, the two classes are *positive* and *negative* sentiment and we will use a standard cross-entropy cost as explain in the previous chapter on object recognition and classification.

To minimize the cost function, we use the optimizer defined in the configuration. However, we will improve on what we've learned so far by adding *gradient clipping*. RNNs are quite hard to train and weights tend to diverge if the hyper parameters do not play nicely together. The idea of gradient clipping is to restrict the the values of the gradient to a sensible range. This way, we can limit the maximum weight updates.

```
@lazy_property
def cost(self):
    cross_entropy = -tf.reduce_sum(self.target * tf.log(self.prediction))
    return cross_entropy

@lazy_property
def optimize(self):
```

```
        gradient = self.params.optimizer.compute_gradients(self.cost)
        if self.params.gradient_clipping:
            limit = self.params.gradient_clipping
            gradient = [
                (tf.clip_by_value(g, -limit, limit), v)
                if g is not None else (None, v)
                for g, v in gradient]
        optimize = self.params.optimizer.apply_gradients(gradient)
        return optimize

    @lazy_property
    def error(self):
        mistakes = tf.not_equal(
            tf.argmax(self.target, 1), tf.argmax(self.prediction, 1))
        return tf.reduce_mean(tf.cast(mistakes, tf.float32))
```

TensorFlow supports this szenario with the `compute_gradients()` function that each optimizer instance provides. We can then modify the gradients and apply the weight changes with `apply_gradients()`. For gradient clipping, we set elements to `-limit` if they are lower than that or to `limit` if they are larger than that. The only tricky part is that derivatives in TensorFlow can be `None` which means there is no relation between a variable and the cost function. Mathematically, those derivatives should be zero vectors, but using `None` allows for internal performance optimizations. We handle those cases by just passing the `None` value back as in the tuple.

Training the model

Let's now train the advanced model we defined above. As we said, we will feed the the movie reviews into the recurrent network word-by-word so each time step is a batch of word vectors. We adapt the `batched()` function from the last section to lookup the word vectors and padd all sequences to the same length as follows:

```
def preprocess_batched(iterator, length, embedding, batch_size):
    iterator = iter(iterator)
    while True:
        data = np.zeros((batch_size, length, embedding.dimensions))
        target = np.zeros((batch_size, 2))
        for index in range(batch_size):
            text, label = next(iterator)
            data[index] = embedding(text)
            target[index] = [1, 0] if label else [0, 1]
        yield data, target
```

We can easily train the model now. We define the hyper parameters, load the dataset and embeddings, and run the model on the preprocessed training batches.

```
params = AttrDict(
    rnn_cell=GRUCell,
    rnn_hidden=300,
    optimizer=tf.train.RMSPropOptimizer(0.002),
    batch_size=20,
)

reviews = ImdbMovieReviews('/home/user/imdb')
length = max(len(x[0]) for x in reviews)

embedding = Embedding(
    '/home/user/wikipedia/vocabulary.bz2',
    '/home/user/wikipedia/embedding,npy', length)
batches = preprocess_batched(reviews, length, embedding, params.batch_size)

data = tf.placeholder(tf.float32, [None, length, embedding.dimensions])
target = tf.placeholder(tf.float32, [None, 2])
model = SequenceClassificationModel(data, target, params)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
for index, batch in enumerate(batches):
    feed = {data: batch[0], target: batch[1]}
    error, _ = sess.run([model.error, model.optimize], feed)
    print('{}: {:3.1f}%'.format(index + 1, 100 * error))
```

This time, the training success of this model will not only depend on the network structure and hyper parameter, but also on the quality of the word embeddings. If you did not train your own word embeddings as described in the last section, you can load pre-trained embeddings from

the word2vec project: https://code.google.com/archive/p/word2vec/ that implements the skip-gram model, or the very similar Glove model from the Stanford NLP group: http://nlp.stanford.edu/projects/glove/. In both cases you will be able to find Python loaders on the web.

We have this model now, so what can you do with it? There is an open learning competition on Kaggle, a famous website hosting data science challenges. It uses the same IMDB movie review dataset as we did in this section. So if you are interested how your results compare to others, you can run the model on their testset and upload your results at https://www.kaggle.com/c/word2vec-nlp-tutorial.

Sequence Labelling

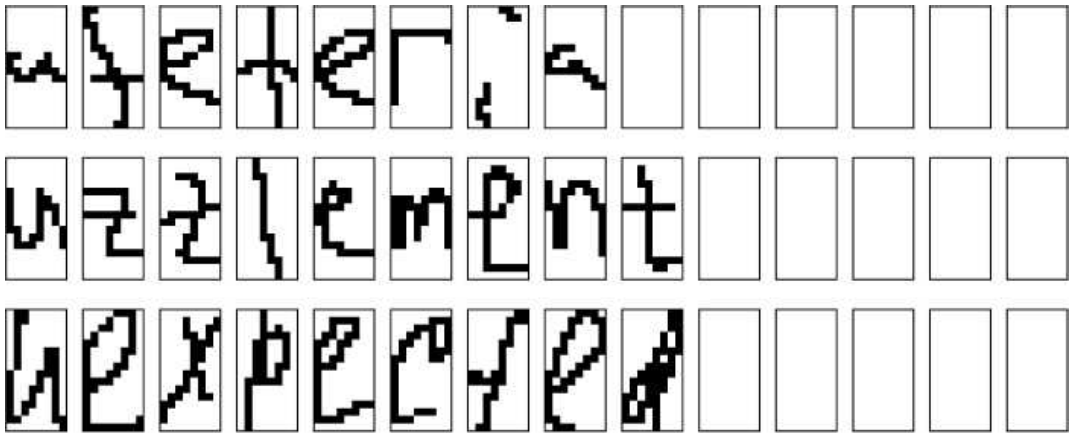
In the last section, we built a sequence classification model that uses an LSTM network and stacked a softmax layer ontop of the last activation. Building on this, we will now tackle the slightly harder problem of *sequence labelling*. This setting differs from sequence classification in that we want to predict an individual class for each frame in the input sequence.

For example, let's think about regocnizing handwritten text. Each word is a sequence of letters and while we could classify each letter independendly, human language has a strong structure that we can take advantage of. If you take a look at a handwritten sample, there are often letters that are hard to read on their own, for example "n", "m", and "u". They can be regocnized from the context of nearby letters however. In this section, we will use RNNs to make use of the dependencies between letters and built a more robust OCR (Optical Character Recognition) system.

Optical Character Recognition Dataset

As an example of sequence labelling problems, we will take a look at the OCR dataset collected by Rob Kassel at the MIT Spoken Language Systems Group and preprocessed by Ben Taskar at the Stanford AI group. The dataset contains individual hand-written letters as binary images of 16 times 8 pixels. The letters are arranged into sequences that where each sequence forms a word. In the whole dataset, there are about 6800 words of length up to 14.

Here are three example sequence from the OCR dataset. The words are "cafeteria", "puzzlement", and "unexpected". The first letters are not included in the dataset since they were uppercase. All sequences are padded to maximal length of 14. To make it a little easier, the dataset contains only lowercase letters. This is why some words miss their first letter.



The dataset is available at http://ai.stanford.edu/~btaskar/ocr/ and comes as a gzipped tab separated textfile that we can read using Python's `csv` module. Each line represents a letter of the dataset and holds attributes like and id, the target letter, the pixel values and the id of the following letter of the word.

```
import gzip
import csv
import numpy as np
class OcrDataset:
    """
    Dataset of handwritten words collected by Rob Kassel at the MIT Spoken
    Language Systems Group. Each example contains the normalized letters of
the
    word, padded to the maximum word length. Only contains lower case letter,
    capitalized letters were removed.
    From: http://ai.stanford.edu/~btaskar/ocr/
    """

    URL = 'http://ai.stanford.edu/~btaskar/ocr/letter.data.gz'

    def __init__(self, cache_dir):
        path = download(type(self).URL, cache_dir)
        lines = self._read(path)
```

```
        data, target = self._parse(lines)
        self.data, self.target = self._pad(data, target)

    @staticmethod
    def _read(filepath):
        with gzip.open(filepath, 'rt') as file_:
            reader = csv.reader(file_, delimiter='\t')
            lines = list(reader)
            return lines

    @staticmethod
    def _parse(lines):
        lines = sorted(lines, key=lambda x: int(x[0]))
        data, target = [], []
        next_ = None
        for line in lines:
            if not next_:
                data.append([])
                target.append([])
            else:
                assert next_ == int(line[0])
            next_ = int(line[2]) if int(line[2]) > -1 else None
            pixels = np.array([int(x) for x in line[6:134]])
            pixels = pixels.reshape((16, 8))
            data[-1].append(pixels)
            target[-1].append(line[1])
        return data, target

    @staticmethod
    def _pad(data, target):
        max_length = max(len(x) for x in target)
        padding = np.zeros((16, 8))
        data = [x + ([padding] * (max_length - len(x))) for x in data]
        target = [x + ([''] * (max_length - len(x))) for x in target]
        return np.array(data), np.array(target)
```

We first sort by those following id values so that we can read the letters of each word in the correct order. Then, we continue collecting letters until the field of the next id is not set in which case we start a new sequence. After reading the target letters and their data pixels, we pad the sequences with zero images so that they fit into two big Numpy arrays containing the target letters and all the pixel data.

Softmax shared between time steps

This time, not only the data but also the target array contains sequences, one target letter for each image frame. The easiest approach to get a prediction at each frame is to augment our RNN with a softmax classifier on top of the output eat each letter. This is very similar to our model for sequence classification from the last section, except that we classifier is evaluated at each frame rather than just at the last one.

```
class SequenceLabellingModel:

    def __init__(self, data, target, params):
        self.data = data
        self.target = target
        self.params = params
        self.prediction
        self.cost
        self.error
        self.optimize

    @lazy_property
    def length(self):
        pass

    @lazy_property
    def prediction(self):
        pass

    @lazy_property
    def cost(self):
        pass
```

```
    @lazy_property
    def error(self):
        pass

    @lazy_property
    def optimize(self):
        pass
```

Let's implement the methods of our sequence labelling mode. First off, we again need to compute the sequence lengths. We already did this in the last section so there is not much to add here.

```
@lazy_property
def length(self):
    used = tf.sign(tf.reduce_max(tf.abs(self.data), reduction_indices=2))
    length = tf.reduce_sum(used, reduction_indices=1)
    length = tf.cast(length, tf.int32)
    return length
```

Now, we come to the prediction, were the main different to the sequence classification model lies. There would be two ways to add a softmax layer to all frames. We could either add several different classifiers or share the same among all frames. Since classifying the third letter should not be very different from classifying the eighth letter, it makes sense to take the latter approach. This way, the classifier weights are also trainer more often because each letter of the word contributes to training them.

In order to implement a shared layer in TensorFlow, we have to apply a little trick. A weight matrix of a fully-connected layer always has the dimentions `batch_size x in_size x out_size`. But we now have two input dimensions along which we want to apply the matrix, `batch_size` and `sequence_steps`.

What we can do to circumvent this problem is to flatten the input to the layer, in this case the ougoing activation of the RNN, to shape `batch_size * sequence_steps x in_size`. This way, it just looks like a large batch to the weight matrix. Of course we have to reshape the results bach to unflatten them.

```
@lazy_property
def prediction(self):
    output, _ = tf.nn.dynamic_rnn(
        GRUCell(self.params.rnn_hidden),
        self.data,
        dtype=tf.float32,
        sequence_length=self.length,
    )
    # Softmax layer.
    max_length = int(self.target.get_shape()[1])
    num_classes = int(self.target.get_shape()[2])
    weight = tf.Variable(tf.truncated_normal(
        [self.params_rnn_hidden, num_classes], stddev=0.01))
    bias = tf.Variable(tf.constant(0.1, shape=[num_classes]))
    # Flatten to apply same weights to all time steps.
    output = tf.reshape(output, [-1, self.params.rnn_hidden])
    prediction = tf.nn.softmax(tf.matmul(output, weight) + bias)
    prediction = tf.reshape(prediction, [-1, max_length, num_classes])
    return prediction
```

The cost and error function change slightly compared to what we had for sequence classification. Namely, there is now an prediction-target pair for each frame in the sequence, so we have to average over that dimension as well. However, `tf.reduce_mean()` doesn't work here since it would normalize by the tensor length which is the maximum sequence length. Instead, we want to normalize by the actual sequence lengths computed earlier. Thus, we manually use `tf.reduce_sum()` and a division to obtain the correct mean.

```
@lazy_property
def cost(self):
    # Compute cross entropy for each frame.
    cross_entropy = self.target * tf.log(self.prediction)
    cross_entropy = -tf.reduce_sum(cross_entropy, reduction_indices=2)
    mask = tf.sign(tf.reduce_max(tf.abs(self.target), reduction_indices=2))
    cross_entropy *= mask
    # Average over actual sequence lengths.
    cross_entropy = tf.reduce_sum(cross_entropy, reduction_indices=1)
    cross_entropy /= tf.cast(self.length, tf.float32)
    return tf.reduce_mean(cross_entropy)
```

Analogously to the cost, we have to adjust the error function. The axis that `tf.argmax()` operates on is axis two rather than axis one now. Then we mask padding frames and compute the average over the actual sequence length. The last `tf.reduce_mean()` averages over the words in the data batch.

```
    @lazy_property
    def error(self):
        mistakes = tf.not_equal(
            tf.argmax(self.target, 2), tf.argmax(self.prediction, 2))
        mistakes = tf.cast(mistakes, tf.float32)
        mask = tf.sign(tf.reduce_max(tf.abs(self.target), reduction_indices=2))
        mistakes *= mask
        # Average over actual sequence lengths.
        mistakes = tf.reduce_sum(mistakes, reduction_indices=1)
        mistakes /= tf.cast(self.length, tf.float32)
        return tf.reduce_mean(mistakes)
```

The nice thing about TensorFlow's automatic gradient computation is that we can use the same optimization operation for this model as we used for sequence classification, just plugging in the new cost function. We will apply gradient clipping in all RNNs from now on, since it can prevent divergence during training while it does not have any negative impace.

```
    @lazy_property
    def optimize(self):
        gradient = self.params.optimizer.compute_gradients(self.cost)
        if self.params.gradient_clipping:
            limit = self.params.gradient_clipping
            gradient = [
                (tf.clip_by_value(g, -limit, limit), v)
                if g is not None else (None, v)
                for g, v in gradient]
        optimize = self.params.optimizer.apply_gradients(gradient)
        return optimize
```

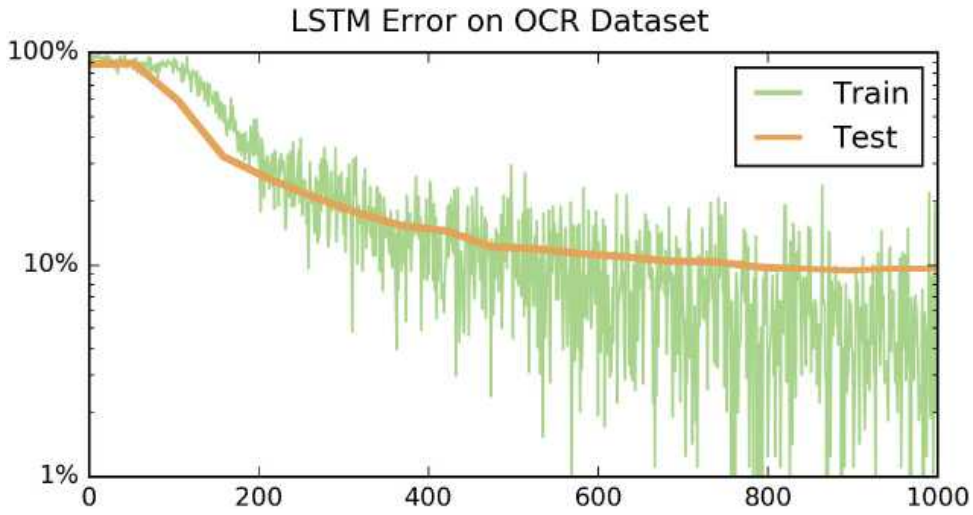
Training the Model

Now we can put together the pieces described so far and train the model. The imports and configuration parameters should be familiar to you from the previous section. We then use `get_dataset()` to download and preprocess the handwritten images. This is also where we encode the targets from lower-case letters to one-hot vectors. After the encoding, we shuffle the data so that we get unbiased splits for training and testing.

```
import random

params = AttrDict(
    rnn_cell=tf.nn.rnn_cell.GRUCell,
    rnn_hidden=300,
    optimizer=tf.train.RMSPropOptimizer(0.002),
    gradient_clipping=5,
    batch_size=10,
    epochs=20,
    epoch_size=50,
)

def get_dataset():
    dataset = OcrDataset('~/.dataset/book/ocr')
    # Flatten images into vectors.
    dataset.data = dataset.data.reshape(dataset.data.shape[:2] + (-1,))
    # One-hot encode targets.
    target = np.zeros(dataset.target.shape + (26,))
    for index, letter in np.ndenumerate(dataset.target):
        if letter:
            target[index][ord(letter) - ord('a')] = 1
    dataset.target = target
    # Shuffle order of examples.
    order = np.random.permutation(len(dataset.data))
    dataset.data = dataset.data[order]
    dataset.target = dataset.target[order]
    return dataset

# Split into training and test data.
dataset = get_dataset()
split = int(0.66 * len(dataset.data))
train_data, test_data = dataset.data[:split], dataset.data[split:]
train_target, test_target = dataset.target[:split], dataset.target[split:]
# Compute graph.
```

```
_, length, image_size = train_data.shape
num_classes = train_target.shape[2]
data = tf.placeholder(tf.float32, [None, length, image_size])
target = tf.placeholder(tf.float32, [None, length, num_classes])
model = SequenceLabellingModel(data, target, params)
```

After training of 1000 words our model classifies about 9% of all letter in the test set correctly. That's not too bad, but there is also room for improvement here.



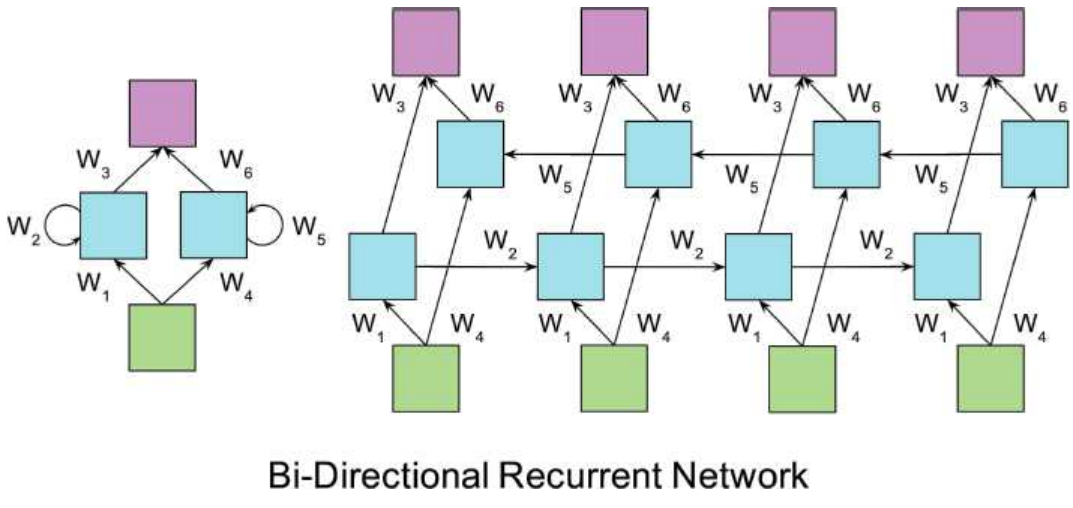
Our current model very similar to the model for sequence classification. This was by intent so that you see the differences needed to apply in order to adapt existing models to new tasks. What worked on another problem is more likely to work well on a new problem than if you would make a wild guess. However, we can do better! In the next section, we will try and improve on our results using a more advanced recurrent architecture.

Bidirectional RNNs

How can we improve the results on the OCR dataset that we got with the RNN plus Softmax architecture? Well, let's take a look at our motivation to use RNNs. The reason we chose them for the OCR dataset was that there are dependencies, or mutual information, between the letters within one word. The RNN stores information about all the previous inputs of the same word in its hidden activation.

If you think about it, the recurrency in our model doesn't help much for classifying the first few letters because the network hasn't had many inputs yet to infer additional information from. In sequence classification, this wasn't a problem since the network sees all frames before making a descision. In sequence labelling, we can address this shortcoming using *bidirectional RNNs*, a technique that holds state or the art in several classification problems.

The idea of bidirectional RNNs is simple. There are two RNNs that take a look at the input sequence, one going from the left reading the word in normal order, and one going from the right reading the letters in reverse order. At each time step, we now got two output activations that we concatenate before passing them up into the shared softmax layer. Using this architecture, the classifier can access information of the whole word at each letter.



How can we implement bidirectional RNNs in TensorFlow? There is actually an implementation available with `tf.model.rnn.bidirectional_rnn`. However, we want to learn how to build complex models ourselves and so let's build our own implementation. I'll walk you through the steps. First, we split the prediction property into two functions so we can focus on smaller parts at the time.

```
@lazy_property
def prediction(self):
    output = self._bidirectional_rnn(self.data, self.length)
    num_classes = int(self.target.get_shape()[2])
    prediction = self._shared_softmax(output, num_classes)
    return prediction

def _bidirectional_rnn(self, data, length):
    pass

def _shared_softmax(self, data, out_size):
    pass
```

The `_shared_softmax()` function above is easy; we already had the code in the prediction property before. The difference is that this time, we infer the input size from the `data` tensor that gets passed into the function. This way, we can reuse the function for other architectures if needed. Then we use the same flattening trick to share the same softmax layer accross all time steps.

```
aef _shared_softmax(self, data, out_size):
    max_length = int(data.get_shape()[1])
    in_size = int(data.get_shape()[2])
    weight = tf.Variable(tf.truncated_normal(
        [in_size, out_size], stddev=0.01))
    bias = tf.Variable(tf.constant(0.1, shape=[out_size]))
    # Flatten to apply same weights to all time steps.
    flat = tf.reshape(data, [-1, in_size])
    output = tf.nn.softmax(tf.matmul(flat, weight) + bias)
    output = tf.reshape(output, [-1, max_length, out_size])
    return output
```

Here somes the interesting part, the implementation of bidirectional RNNs. As you can see, we have create two RNNs using `tf.nn.dynamic_rnn`. The forward network should look familiar while the backward network is new.

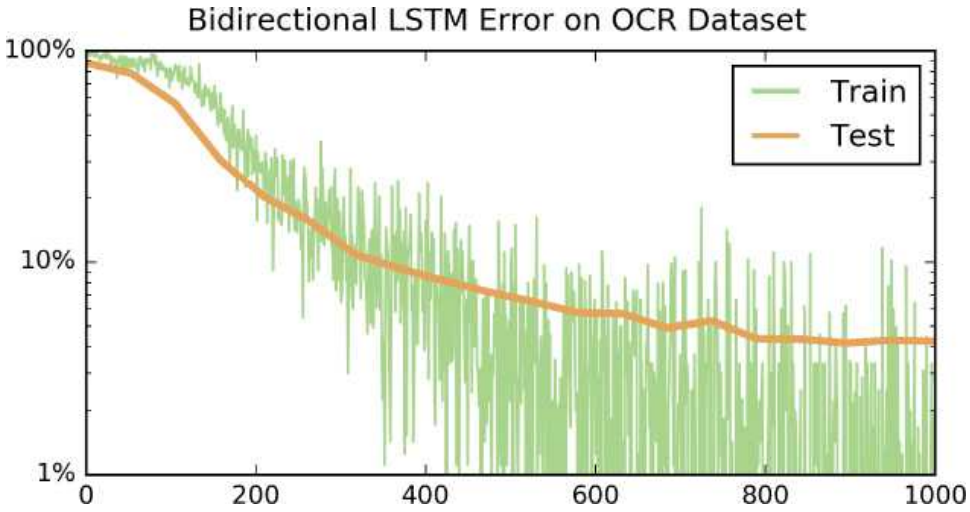
Instead of just feeding in the data into the backward RNN, we first reverse the sequences. This is easier than implementing a new RNN operation that would go backwards. TensorFlow helps us with with the `tf.reverse_sequence()` functions that takes care of only reversing the used frames up to `sequence_length`. Note that at the moment of writing this, the function expects the lengths to be a 64-bit integer tensor. It's likely that it will also work with 32-bit tensors and you can just pass in `self.length`.

```
def _bidirectional_rnn(self, data, length):
    length_64 = tf.cast(length, tf.int64)
    forward, _ = tf.nn.dynamic_rnn(
        cell=self.params.rnn_cell(self.params.rnn_hidden),
        inputs=data,
        dtype=tf.float32,
        sequence_length=length,
        scope='rnn-forward')
    backward, _ = tf.nn.dynamic_rnn(
        cell=self.params.rnn_cell(self.params.rnn_hidden),
        inputs=tf.reverse_sequence(data, length_64, seq_dim=1),
        dtype=tf.float32,
        sequence_length=self.length,
        scope='rnn-backward')
    backward = tf.reverse_sequence(backward, length_64, seq_dim=1)
    output = tf.concat(2, [forward, backward])
    return output
```

We also use the `scope` parameter this time. Why do we need this? As explained in the *TensorFlow Fundamentals* chapter, nodes in the compute graph have names. `scope` is the name of the variable scope used by `rnn.dynamic_cell` and it detauls to `RNN`. This time we have two RNNs that have different parameters so they have to live in different scopes.

After feeding the reversed sequence into the backward RNN, we again reverse the network outputs to align with the forward outputs. Then we concatenate both tensors along the dimension of the output neurons of the RNNs and return this. For example, with a batch size of 50, 300 hidden units per RNN and words of up to 14 letters, the resulting tensor would have the shape `50 × 14 × 600`.

Okay cool, we built our first architecture that is composed of multiple RNNs! Let's see how it performs using the training code from the last section. As you can see from comparing the graphs, the bidirectional model performs significantly better. After seeing 1000 words, it only misclassifies 4% of the letters in the test split.



To summarize, in this section we learned how to use RNNs for sequence labelling and the differences to the sequence classification setting. Namely, we want a classifier that takes the RNN outputs and is shared accross all time steps.

This architecture can be improved drastically by adding a second RNN that visits the sequence from back to front and combining the outputs at each time step. This is because now information of the whole sequence are available for the classification of each letter.

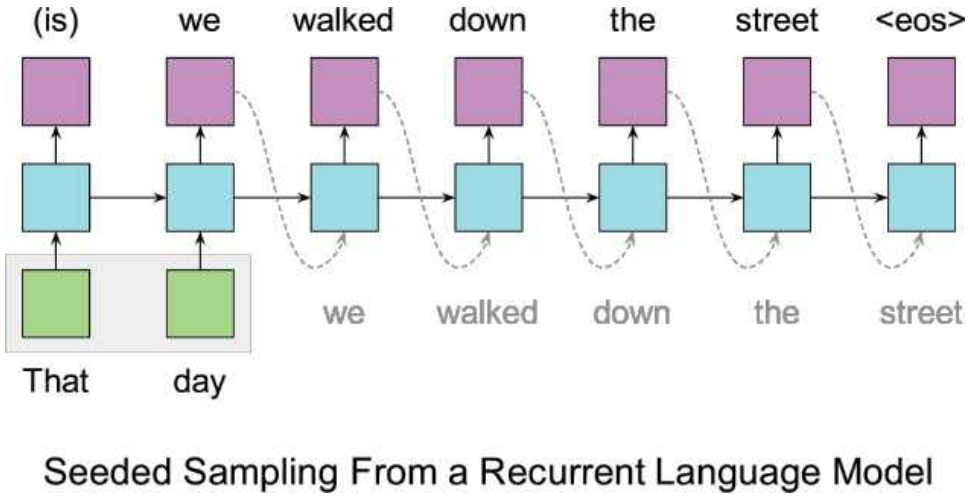
In the next section, we will take a look at training an RNN in an unsupervised fashion in order to learn language.

Predictive coding

We already learned how to use RNNs to classify the sentiment of movie reviews, and to recognize hand-written words. These applications have been supervised, meaning that we needed a labelled dataset. Another interesting learning setting is called *predictive coding*. We just show the RNN a lot of sequences and train it to predict the next frame of the sequence.

Let's take text as an example, where predicting the likelihood of the next word in a sentence is called *language modelling*. Why would it be useful to predict the next word in a sentence? One groups of applications is recongnizing language. Let's say you want to build a handwriting recognizer that translates scans of handwriting to typed text. While you can try to recover all the words from the input scans only, knowing the distribution of likely next words already narrows down the candidate words to decide between. It's the difference between dumb recognition of shapes and reading, basically.

Besides improving performance in tasks involving natural language, we can also sample from the distribution of what the network thinks should follow next in order to generate text. After training, we can start feeding a seed word into the RNN and look at the next word prediction. Then we feed the most likely word back into the RNN as the next input so see what it thinks should follow now. Doing this repeatedly, we can generate new content looking similar to the training data.



Seeded Sampling From a Recurrent Language Model

The interesting thing is that predictive coding trains the network to compress all the important information of any sequence. The next words in a sentence usually depends on the previous words, their order and relations between each other. A network that is able to accurately predict the next character in natural language thus needs to capture the rules of grammar and language well.

Character-level language modelling

We will now build a predictive coding language model using an RNN. Instead of the traditional approach to operate on words though, we will have our RNN operate on *individual characters*. So instead of word embeddings as inputs, we have a little over 26 one-hot encoded characters to represent the alphabet and some punctuation and whitespace.

It's known yet whether word-level or character-level language modelling is the better approach. The beauty of of the character approach is that the network does not only learn how to combine words, but also how to spell them. In addition, the input to our network is lower-dimensional than if we would use word embeddings of size 300 or even one-hot encoded words. As a bonus, we don't have to account for unknown words anymore, because they are composed of letters that the network already knows about. This, in theory, even allows the network could invent new words.

Andrew Karpathy experimented with RNNs operating on chracters in 2015 and was able to generate surprisingly nice samples of Shakespeare scripts, Linux kernel and driver code, and Wikipedia articles including correct markup syntax. The project is available on Github under https://github.com/karpathy/char-rnn. We will now train a similar model on the abstracts of machine learning publications and see if we can generate some more-or-less plausible new abstracts!

ArXiv abstracts API

ArXiv.org is an online library hosting many research papers from computer science, maths, physics and biology. You probably already heard of it if your are following machine learning research. Fortunately, the platform provides a web-based API to retrieve publications. Let's write a class that fetches the abstracts from ArXiv for a given search query.

```
import requests
import os
from bs4 import BeautifulSoup

class ArxivAbstracts:

    def __init__(self, cache_dir, categories, keywords, amount=None):
        pass

    def _fetch_all(self, amount):
        pass

    def _fetch_page(self, amount, offset):
        pass

    def _fetch_count(self):
        pass
    def _build_url(self, amount, offset):
        pass
```

In the constructor, we first check if there is already a previous dump of abstracts available. If it is, we will use that instead of hitting the API again. You could imagine more complicated logic to check if the existing file matches the new categories and keywords, but for now it is sufficient to delete or move the old dump manually to perform a new query. If no dump is available, we call the `_fetch_all()` method and write the lines it yields to disk.

```
def __init__(self, cache_dir, categories, keywords, amount=None):
    self.categories = categories
    self.keywords = keywords
    cache_dir = os.path.expanduser(cache_dir)
    ensure_directory(cache_dir)
    filename = os.path.join(cache_dir, 'abstracts.txt')
    if not os.path.isfile(filename):
        with open(filename, 'w') as file_:
            for abstract in self._fetch_all(amount):
                file_.write(abstract + '\n')
    with open(filename) as file_:
        self.data = file_.readlines()
```

Since we are interested in machine learning papers, we will search within the categories *Machine Learning*, *Neural and Evolutionary Computing*, and *Optimization and Control*. We further restrict the results to those containing any of the words *neural*, *network* or *deep* in the metadata. This gives us about 7 MB of text which is a fair amount of data to learn a simple RNN language model. It would be reasonable to use more data and get better results, but we don't want to wait for too many hours of training to pass before seeing some results. Feel free to use a broader search query and train this model on more data though.

```
ENDPOINT = 'http://export.arxiv.org/api/query'

def _build_url(self, amount, offset):
```

```
        categories = ' OR '.join('cat:' + x for x in self.categories)
        keywords = ' OR '.join('all:' + x for x in self.keywords)
        url = type(self).ENDPOINT
        url += '?search_query=(({}) AND ({}))'.format(categories, keywords)
        url += '&max_results={}&offset={}'.format(amount, offset)
        return url

    def _fetch_count(self):
        url = self._build_url(0, 0)
        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'lxml')
        count = int(soup.find('opensearch:totalresults').string)
        print(count, 'papers found')
        return count
```

The `_fetch_all()` method basically performs *pagination*. The API only gives us a certain amount of abstracts per request and we can specify an offset to get results of the second, third, etc "page". As you can see, we can specify the page size which gets passed into the next function, `_fetch_page()`. In theory, we could set the page size to a huge number and try to get all results at once. In practice however, this makes the request very slow. Fetching in pages is also more fault tolerant and most importantly, does not stress the Arxiv API too much.

```
    PAGE_SIZE = 100

    def _fetch_all(self, amount):
        page_size = type(self).PAGE_SIZE
        count = self._fetch_count()
        if amount:
            count = min(count, amount)
        for offset in range(0, count, page_size):
            print('Fetch papers {}/{}'.format(offset + page_size, count))
            yield from self._fetch_page(page_size, count)
```

Here we perform the actual fetching. The result comes in XML and we use the popular and powerful BeautifulSoup library to extract the abstracts. If you haven't installed it already, you can issue a `sudo -H pip3 install beautifulsoup4`. BeautifulSoup parses the XML result for us so that we can easily iterate over the tags that are of our interest. First, we look for `<entry>` tags corresponding to publications and within each of them, we read our the `<summary>` tag containing the abstract text.

```
    def _fetch_page(self, amount, offset):
        url = self._build_url(amount, offset)
        response = requests.get(url)
        soup = BeautifulSoup(response.text)
        for entry in soup.findAll('entry'):
            text = entry.find('summary').text
            text = text.strip().replace('\n', ' ')
            yield text
```

Preprocessing the data

```
import random
import numpy as np


class Preprocessing:

    VOCABULARY = \
        " $%'()+,-./0123456789:;=?ABCDEFGHIJKLMNOPQRSTUVWXYZ" \
        "\\^_abcdefghijklmnopqrstuvwxyz{|}"
    def __init__(self, texts, length, batch_size):
        self.texts = texts
        self.length = length
        self.batch_size = batch_size
        self.lookup = {x: i for i, x in enumerate(self.VOCABULARY)}

    def __call__(self, texts):
        batch = np.zeros((len(texts), self.length, len(self.VOCABULARY)))
        for index, text in enumerate(texts):
            text = [x for x in text if x in self.lookup]
            assert 2 <= len(text) <= self.length
            for offset, character in enumerate(text):
                code = self.lookup[character]
```

```
                batch[index, offset, code] = 1
        return batch

    def __iter__(self):
        windows = []
        for text in self.texts:
            for i in range(0, len(text) - self.length + 1, self.length // 2):
                windows.append(text[i: i + self.length])
        assert all(len(x) == len(windows[0]) for x in windows)
        while True:
            random.shuffle(windows)
            for i in range(0, len(windows), self.batch_size):
                batch = windows[i: i + self.batch_size]
                yield self(batch)
```

Predictive coding model

By now, you already know the procedure: We have defined our task, have written a parser to obtain a dataset and now we will implement the neural network model in TensorFlow. Because for predictive coding we try and predict the next character of the input sequence, there is only one input to the model, which is the `sequence` parameter in the constructor.

Moreover, the constructor takes a parameter object to change options in a central place and make our experiments reproducible. The third parameter `initial=None` is the initial inner activation of the recurrent layer. While we want to TensorFlow to initialize the hidden state to zero tensors for us, it will become handy to define it when we will sample from the learned language model later.

```
import tensorflow as tf
from utility import lazy_property


class PredictiveCodingModel:
    def __init__(self, params, sequence, initial=None):
        self.params = params
        self.sequence = sequence
        self.initial = initial
        self.prediction
        self.state
        self.cost
        self.error
        self.logprob
        self.optimize

    @lazy_property
    def data(self):
        pass

    @lazy_property
    def target(self):
        pass

    @lazy_property
    def mask(self):
        pass

    @lazy_property
    def length(self):
        pass

    @lazy_property
    def prediction(self):
        pass

    @lazy_property
    def state(self):
        pass

    @lazy_property
    def forward(self):
        pass
```

```
    @lazy_property
    def cost(self):
        pass

    @lazy_property
    def error(self):
        pass

    @lazy_property
    def logprob(self):
        pass
    @lazy_property
    def optimize(self):
        pass

    def _average(self, data):
        pass
```

In the code example above, you can see an overview of the functions that our model will implemnet. Don't worry it that looks overwhelming at first: We just want to expose some more values of our model than we did in the previous chapters.

Let's start with the data processing. As we said, the model just takes a one block of sequences as input. First, we use that to construct input data and target sequences from it. This is where we introduce a temporal difference because at timestep t, the model should have character S_t as input but S_{t+1} as target. As easy way to obtain `data` or `target` is to slice the provided sequence and cut away the last or the first frame, respectively.

We do this slicing using `tf.slice()` which takes the sequence to slice, a tuple of start indices for each dimension, and a tuple of sizes for each dimension. For the sizes -1 means to keep all elemnts from the start index in that dimension until the end. Since we want to slices frames, we only care about the second dimension.

```
@lazy_property
def data(self):
    max_length = int(self.sequence.get_shape()[1])
    return tf.slice(self.sequence, (0, 0, 0), (-1, max_length - 1, -1))

@lazy_property
def target(self):
    return tf.slice(self.sequence, (0, 1, 0), (-1, -1, -1))

@lazy_property
def mask(self):
    return tf.reduce_max(tf.abs(self.target), reduction_indices=2)

@lazy_property
def length(self):
    return tf.reduce_sum(self.mask, reduction_indices=1)
```

We also define two properties on the target sequence as we already discussed in earler sections: `mask` is a tensor of size `batch_size x max_length` where elements are zero or one depending on wheter the respective frame is used or a padding frame. The `length` property sums up the mask along the time axis in order to obtain the length of each sequence.

Note that the mask and length properties are also valid for the data sequence since conceptually, they are of the same length as the target sequence. However, we couldn't compute them on the data sequence since it still contains the last frame that is not needed since there is no next character to predict. You are right, we sliced away the last frame of the data tensor, but that didn't contained the actual last frame of most sequences but mainly padding frames. This is the reason why we will use `mask` below to mask our cost function.

Now we will define the actual network that consists of a recurrent network and a shared softmax layer, just like we used for sequence labelling in the previous section. We don't show the code for the shared softmax layer here again but you can find it in the previous section.

```
@lazy_property
def prediction(self):
    prediction, _ = self.forward
    return prediction

@lazy_property
def state(self):
    _, state = self.forward
    return state
```

```
@lazy_property
def forward(self):
    cell = self.params.rnn_cell(self.params.rnn_hidden)
    cell = tf.nn.rnn_cell.MultiRNNCell([cell] * self.params.rnn_layers)
    hidden, state = tf.nn.dynamic_rnn(
        inputs=self.data,
        cell=cell,
        dtype=tf.float32,
        initial_state=self.initial,
        sequence_length=self.length)
    vocabulary_size = int(self.target.get_shape()[2])
    prediction = self._shared_softmax(hidden, vocabulary_size)
    return prediction, state
```

The new part about the neural network code above is that we want to get both the prediction and the last recurrent activation. Previously, we only returned the prediction but the last activation allows us to generate sequences more effectively later. Since we only want to construct the graph for the recurrent network once, there is a `forward` property that return the tuple of both tensors and `prediction` and `state` are just there to provide easy access from the outside.

The next part of our model is the cost and evaluation functions. At each time step, the model predicts the next character out of the vocabulary. This is a classification problem and we use the cross entropy cost, accordingly. We can easily compute the error rate of character predictions as well.

The `logprob` property is new. It describes the probability that our model assigned to the correct next character in logarithmic space. This is basically the negative cross entropy transformed into logarithmic space and averaged there. Converting the result back into linear space yields the so-called *perplexity*, a common measure to evaluate the performance of language models.

The perplexity is defined as $2^{\frac{1}{n}\sum_{i=1}^{n}\log p(y_i)}$. Intuitively, it represents the number of options the model had to guess between at each time step. A perfect model has a perplexity of `1` while a model that always outputs the same probability for each of the ***n*** classes has a perplexity of ***n***. The perplexity can even become infinity when the model assigns a zero probability to the next character once. We prevent this extreme case by clamping the prediction probabilities within a very small positive number and one.

```
@lazy_property
def cost(self):
    prediction = tf.clip_by_value(self.prediction, 1e-10, 1.0)
    cost = self.target * tf.log(prediction)
    cost = -tf.reduce_sum(cost, reduction_indices=2)
    return self._average(cost)

@lazy_property
def error(self):
    error = tf.not_equal(
        tf.argmax(self.prediction, 2), tf.argmax(self.target, 2))
    error = tf.cast(error, tf.float32)
    return self._average(error)

@lazy_property
def logprob(self):
    logprob = tf.mul(self.prediction, self.target)
    logprob = tf.reduce_max(logprob, reduction_indices=2)
    logprob = tf.log(tf.clip_by_value(logprob, 1e-10, 1.0)) / tf.log(2.0)
    return self._average(logprob)

def _average(self, data):
    data *= self.mask
    length = tf.reduce_sum(self.length, 1)
    data = tf.reduce_sum(data, reduction_indices=1) / length
    data = tf.reduce_mean(data)
    return data
```

All the three properties above are averaged over the frames of all sequences. With fixed-length sequences, this would be a single `tf.reduce_mean()`, but as we work with variable-length sequences, we have to be a bit more careful. First, we mask out padding frames by multiplying with the mask. Then we aggregate along the frame size. Because the three functions above all multiply with the target, each frame has just one element set and we use `tf.reduce_sum()` to aggregate each frame into a scalar.

Next, we want to average along the frames of each sequence using the actual sequence length. To protect against division by zero in case of empty sequences, we use the maximum of each sequence length and one. Finally, we can use `tf.reduce_mean()` to average over the examples in the batch.

We will directly head to training this model. Note that we did not the define the `optimize` operation. It is identical to those used for sequence classification or sequence labelling ealier in the chapter.

Training the model

Before sampling from our language model, we have to put together the blocks we just built: The dataset, the preprocessing step and the neural model. Let's write a class for that that puts together these steps, prints the newly introduced perplexity measure and regularly stores trainng progress. This checkpointing is useful to continue training later but also to load the trained model for sampling, which we will do shortly.

```
import os


class Training:

    @overwrite_graph
    def __init__(self, params):
        self.params = params
        self.texts = ArxivAbstracts('/home/user/dataset/arxiv')()
        self.prep = Preprocessing(
            self.texts, self.params.max_length, self.params.batch_size)
        self.sequence = tf.placeholder(
            tf.float32,
            [None, self.params.max_length, len(self.prep.VOCABULARY)])
        self.model = PredictiveCodingModel(self.params, self.sequence)
        self._init_or_load_session()

    def __call__(self):
        print('Start training')
        self.logprobs = []
        batches = iter(self.prep)
        for epoch in range(self.epoch, self.params.epochs + 1):
            self.epoch = epoch
            for _ in range(self.params.epoch_size):
                self._optimization(next(batches))
            self._evaluation()
        return np.array(self.logprobs)
    def _optimization(self, batch):
        logprob, _ = self.sess.run(
            (self.model.logprob, self.model.optimize),
            {self.sequence: batch})
        if np.isnan(logprob):
            raise Exception('training diverged')
        self.logprobs.append(logprob)

    def _evaluation(self):
        self.saver.save(self.sess, os.path.join(
            self.params.checkpoint_dir, 'model'), self.epoch)
        self.saver.save(self.sess, os.path.join(
            self.params.checkpoint_dir, 'model'), self.epoch)
        perplexity = 2 ** -(sum(self.logprobs[-self.params.epoch_size:]) /
                            self.params.epoch_size)
        print('Epoch {:2d} perplexity {:5.1f}'.format(self.epoch, perplexi-
ty))

    def _init_or_load_session(self):
        self.sess = tf.Session()
        self.saver = tf.train.Saver()
        checkpoint = tf.train.get_checkpoint_state(self.params.check-
point_dir)
        if checkpoint and checkpoint.model_checkpoint_path:
            path = checkpoint.model_checkpoint_path
            print('Load checkpoint', path)
            self.saver.restore(self.sess, path)
            self.epoch = int(re.search(r'-(\d+)$', path).group(1)) + 1
        else:
            ensure_directory(self.params.checkpoint_dir)
            print('Randomly initialize variables')
            self.sess.run(tf.initialize_all_variables())
```

```
            self.epoch = 1
```

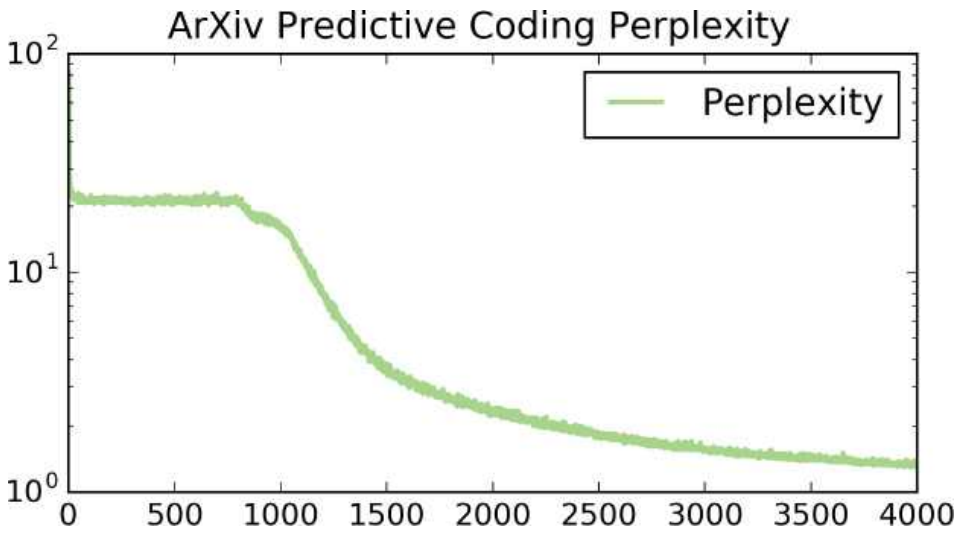
The constructor, `__call__()`, `_optimization()` and `_evaluation()` should be easy to understand. We load the dataset and define inputs to the compute graph, train on the preprocessed dataset and keep track of the logarithmic probabilities. We use those at evaluation time between each training epoch to compute and print the perplexity.

In `_init_or_load_session()` we introduce a `tf.train.Saver()` that stores the current values of all `tf.Variable()` in the graph to a checkpoint file. While the actual checkpointing is done in `_evaluation()`, here we create the class and look for existing checkpoints to load. The `tf.train.get_checkpoint_state()` looks for TensorFlow's meta data file in our checkpoint directory. As of writing, it only contains the file of the least recent checkpoint file.

Checkpoint files are prepended by a number that we can specify, in our case the epoch number. When loading a checkpoint, we apply a regular experession with Python's `re` package to extract that epoch number. With the checkpointing logic set up, we can start training. Here is the configuration:

```
def get_params():
    checkpoint_dir = '/home/user/model/arxiv-predictive-coding'
    max_length = 50
    sampling_temperature = 0.7
    rnn_cell = GRUCell
    rnn_hidden = 200
    rnn_layers = 2
    learning_rate = 0.002
    optimizer = tf.train.AdamOptimizer
    gradient_clipping = 5
    batch_size = 100
    epochs = 20
    epoch_size = 200
    return AttrDict(**locals())
```

To run the code, you can just call `Training(get_params())()`. On my notebook, it takes about one hour for the 20 epochs. During this training, the model saw `20 epochs * 200 batches * 100 examples * 50 characters = 20M` characters.



As you can see on the graph, the model converges at a perplexity of about 1.5 per character. This means that with our model, we could compress a text at an average of 1.5 bits per character.

For comparison with word-level language models, we would have to average by the number of words rather than the number of characters. As a rough estimate, we can multipy it by the average number of characters per word, which is … on our test set.

Generating similiar sequences

After all the work, we can now use the trained model to sample new seuquences. We will write a small class that work similar to our `Training` class in that it loads the latest model checkpoint from disk and defines placeholders to feed data into the compute graph. Of course, this time we don't train the model but use it to generate new data.

```
class Sampling:

    @overwrite_graph
    def __init__(self, params):
        pass
```

```
    def __call__(self, seed, length):
        pass

    def _sample(self, dist):
        pass
```

In the constructor, we create an instance of our preprocessing class that we will use convert the current generated sequence into a Numpy vector to feed into the graph. The `sequence` placeholder for this is only has one sequence per batch because we don't want to generate multiple sequences at the same time.

One thing to explain is the sequence length of two. Remember that the model use all but the last characters as input data and all but the first characters as targets. We feed in the last character of the current text and any second character as sequence. The network will predict the target for the first character. The second character is used as target but since we don't train anything, it will be ignored.

You may wonder how we can get along with only passing the last character of the current text into the network. The trick here is that we will get the last activation of the recurrent network and use that to initialize the state in the next run. For this, we make use of the initial state argument of our model. For the `GRUCell` that we used, the state is a vector of size `rnn_layers * rnn_units`.

```
@overwrite_graph
def __init__(self, params, length):
    self.params = params
    self.prep = Preprocessing([], 2, self.params.batch_size)
    self.sequence = tf.placeholder(
        tf.float32, [1, 2, len(self.prep.VOCABULARY)])
    self.state = tf.placeholder(
        tf.float32, [1, self.params.rnn_hidden * self.params.rnn_layers])
    self.model = PredictiveCodingModel(
        self.params, self.sequence, self.state)
    self.sess = tf.Session()
    checkpoint = tf.train.get_checkpoint_state(self.params.checkpoint_dir)
    if checkpoint and checkpoint.model_checkpoint_path:
        tf.train.Saver().restore(
            self.sess, checkpoint.model_checkpoint_path)
    else:
        print('Sampling from untrained model.')
    print('Sampling temperature', self.params.sampling_temperature)
```

The `__call__()` functions defines the logic for sampling a text sequence. We start with the seed and predict one character at a time, always feeding the current text into the network. We use the same preprocessing class to convert the current texts into padded Numpy blocks and feed them into the network. Since we only have one sequence with a single output frame in the batch, we only care at the prediction at index `[0, 0]`. We then sample from the softmax output using the `_sample()` function described next.

```
def __call__(self, seed, length=100):
    text = seed
    state = np.zeros((1, self.params.rnn_hidden * self.params.rnn_layers))
    for _ in range(length):
        feed = {self.state: state}
        feed[self.sequence] = self.prep([text[-1] + '?'])
        prediction, state = self.sess.run(
            [self.model.prediction, self.model.state], feed)
        text += self._sample(prediction[0, 0])
    return text
```

How do we sample from the network output? Earlier we said we can generate sequences by taking their best bet and feeding that in as the next frame. Actually, we don't just choose the most likely next frame but randomly sample one from the probability distribution that the RNN outputs. This way, words with a high output probability are more likely to be choosen but less likely words are still possible. This results in more dynamic generated sequences. Otherwise, we might just generate the same average sentence again and again.

There is a simple mechanism to manually control how advantegerous the generation process should be. For example, if we would always choose the next word randomly (and ignore the network output completely), we would get very new and unique sentences but they would not make any sense. If we always choose the network's highest output as the next word, we would get a lot of common, but meaningless words like "the," "a," etc.

The way can control this behavior is by introducing a *temperature* parameter ***T***. We use this parameter to make the predictions of the output distribution at the softmax layer more similar or more radical. This will result in more interesting but random sequences on the one side of the spectrum, and to more plausible but boring sequences on the other side. The way it works is that we scale the scale the outputs in linear space, then transform them back into exponential space and normalize again:

$$p(x_i) = \frac{e^{\frac{x_i}{T}}}{\sum_j e^{x_j T}}$$

Since the network already outputs a softmax distribution, we undo it by applying the natural logarithm. We don't have to undo the normalization since we will normalize our results again, anyways. Then we divide each value by the choosen temperature value and re-apply the softmax function.

```
def _sample(self, dist):
    dist = np.log(dist) / self.params.sampling_temperature
    dist = np.exp(dist) / np.exp(dist).sum()
    choice = np.random.choice(len(dist), p=dist)
    choice = self.prep.VOCABULARY[choice]
    return choice
```

Let's run the code by calling `Sampling(get_params())('We', 500))` for the network to generate a new abstract. While you can certainly tell that this text is not written by a human, it is quite remarkable what the network learnes from examples.

```
We study nonconvex encoder in the networks (RFNs) hasding configurations with
non-convex large-layers of images, each directions literatic for layers. More
recent results competitive strategy, in which data at training and more
difficult to parallelize. Recent Newutic systems, the desirmally parametri-
cally
in the DNNs improves optimization technique, we extend their important and
subset of theidesteding and dast and scale in recent advances in sparse
recovery to complicated patterns of the $L_p$
```

We did not tell the RNN what a space is, but it captured statistically dependencies in the data to place whitespace accordingly in the generated text. Even between some non-existent words that the network dreamed up, the whitespace looks reasonable. Moreover, those words are composed of valid combinations of vowels and consonants, another abstract feature learned from the example texts.

Conclusion

RNNs are powerful sequential models that are applicable to a wide range of problems and are responsible for state-of-the-art results. We learned how to optimize RNNs, what problems arise doing so, and how architectures like LSTM and GRU help to overcome them. Using these building blocks, we solved several problems in natural language processing and related domains including classifying the sentiment of movie reviews, recognizing hand-written words, and generating fake scienfitic abstracts.

In the next chapter we will put our trained models in production so they can be consumed by other applications.