highly correlated to correctly producing corresponding segments of the output, attentions can dramatically improve performance.

Dissecting a Neural Translation Network

State-of-the-art neural translation networks use a number of different techniques and advancements that build on the basic seq2seq encoder-decoder architecture. Attention, as detailed in the previous section, is an important and critical architectural improvement. In this section, we will dissect a fully implemented neural machine translation system, complete with the data processing steps, building the model, training it, and eventually using it as a translation system to convert English phrases to French phrases! We'll pursue this exploration by working with a simplified version of the official TensorFlow machine translation tutorial code.[12]

The pipeline used in training and eventually using a neural machine translation system is very similar to that of most machine learning pipelines: gather data, prepare the data, construct the model, train the model, evaluate the model's progress, and eventually use the trained model to predict or infer something useful. We review each of these steps here.

We first gather the data from the WMT'15 repository, which houses large corpora used in training translation systems. For our use case, we'll be using the English-to-French data. Note that if we want to be able to translate to or from different languages, we would have to train a model from scratch with the new data. We then preprocess our data into a format that is easily usable by our models during training and inference time. This will involve some amount of cleaning and tokenizing the sentences in each of the English and French phrases. What follows now is a set of techniques used in preparing the data, and later we will present the implementations of the techniques.

The first step is to parse sentences and phrases into formats that are more compatible with the model by *tokenization*. This is the process by which we discretize a particular English or French sentence into its constituent tokens. For instance, a simple word-level tokenizer will consume the sentence "I read." to produce the array ["I", "read", "."], or it would consume the French sentence "Je lis." to produce the array ["Je", "lis", "."]. A character-level tokenizer may break the sentence into individual characters or into pairs of characters like ["I", " ", "r", "e", "a", "d", "."] and ["I ", "re", "ad", "."], respectively. One kind of tokenization may work better than the other, and each has its pros and cons. For instance, a word-level tokenizer will ensure that the model produces words that are from some dictionary, but the size of the dictionary may be too large to efficiently choose from during decoding. This is in fact a known issue and some-

12 This code can be found at: *https://github.com/tensorflow/tensorflow/tree/r0.7/tensorflow/models/rnn/translate*.

thing that we'll address in the coming discussions. On the other hand, the decoder using a character-level tokenization may not produce intelligible outputs, but the total dictionary that the decoder must choose from is much smaller, as it is simply the set of all printable ASCII characters. In this tutorial, we use a word-level tokenization, but we encourage the reader to experiment with different tokenizations to observe the effects this has. It is worth noting that we must also add a special EOS, or end-of-sequence character, to the end of all output sequences because we need to provide a definitive way for the decoder to indicate that it has reached the end of its decoding. We can't use regular punctuation because we cannot assume that we are translating full sentences. Note that we do not need EOS characters in our source sequences because we are feeding these in pre-formatted and do not need an end-of-sequence character for ourselves to denote the end of our source sequence.

The next optimization involves further modifying how we represent each source and target sequence, and we introduce a concept called *bucketing*. This is a method employed primarily in sequence-to-sequence tasks, especially machine translation, that helps the model efficiently handle sentences or phrases of different lengths. We first describe the naive method of feeding in training data and illustrate the shortcomings of this approach. Normally, when feeding in encoder and decoder tokens, the length of the source sequence and the target sequence is not always equal between pairs of examples. For example, the source sequence may have length X, and the target sequence may have length Y. It may seem that we need different seq2seq networks to accommodate each (X, Y) pair, yet this immediately seems wasteful and inefficient. Instead, we can do a little better if we *pad* each sequence up to a certain length, as shown in Figure 7-28, assuming we use a word-level tokenization and that we've appended EOS tokens to our target sequences.

I	read	.	<PAD>	<PAD>	<PAD>	<PAD>
Je	lis	.	<EOS>	<PAD>	<PAD>	<PAD>
See	you	in	a	little	while	.
A	tout	a	l'heure	<EOS>	<PAD>	<PAD>
			...			

Figure 7-28. Naive strategy for padding sequences

This step saves us the trouble of having to construct a different seq2seq model for each pair of source and target lengths. However, this introduces a different issue: if there were a very long sequence, it would mean that we would have to pad every other sequence *up to that length*. This would make a short sequence padded to the end take as much computational resources as a long one with few PAD tokens, which is wasteful and could introduce a major performance hit to our model. We could consider breaking up every sentence in the corpus into phrases such that the length of

each phrase does not exceed a certain maximum limit, but it's not clear how to break the corresponding translations. This is where bucketing helps us.

Bucketing is the idea that we can place encoder and decoder pairs into buckets of similar size, and only pad up to the maximum length of sequences in each respective bucket. For instance, we can denote a set of buckets, [(5, 10), (10, 15), (20, 25), (30, 40)], where each tuple in the list is the maximum length of the source sequence and target sequence, respectively. Borrowing the preceding example, we can place the pair of sequences (["I", "read", "."], ["Je", "lis", ".", "EOS"]) in the first bucket, as the source sequence is smaller than 5 tokens and the target sequence is smaller than 10 tokens. We would then place the (["See", "you", "in", "a", "little", "while"], ["A", "tout", "a", "l'heure", "EOS]) in the second bucket, and so on. This technique allows us to compromise between the two extremes, where we only need to pad as much as necessary, as shown in Figure 7-29.

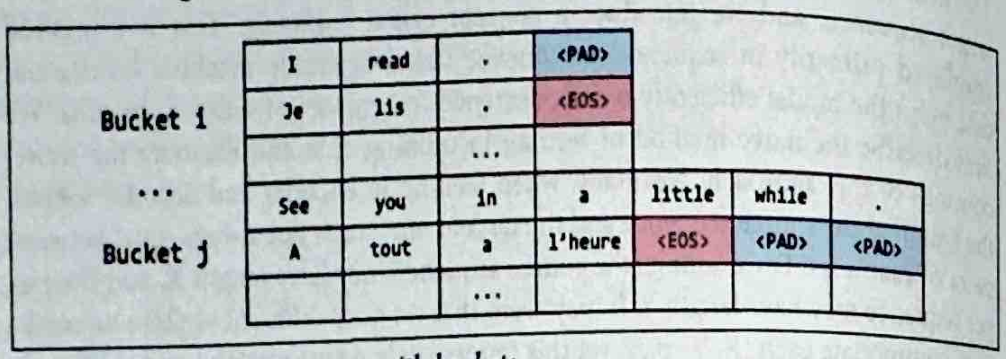

Figure 7-29. Padding sequences with buckets

Using bucketing shows a considerable speedup during training and test time, and allows developers and frameworks to write very optimized code to leverage the fact that any sequence from a bucket will have the same size and pack the data together in ways that allow even further GPU efficiency.

With the sequences properly padded, we need to add one additional token to the target sequences: *a GO token*. This GO token will signal to the decoder that decoding needs to begin, at which point it will take over and begin decoding.

The last improvement we make in the data preparation side is that we reverse the source sequences. Researchers found that doing so improved performance, and this has become a standard trick to try when training neural machine translation models. This is a bit of an engineering hack, but consider the fact that our fixed-size neural state can only hold so much information, and information encoded while processing the beginning of the sentence may be overwritten while encoding later parts of the sentence. In many language pairs, the beginning of sentences is harder to translate than the end of sentences, so this hack of reversing the sentence improves translation accuracy by giving the beginning of the sentence the last say on what final state is encoded. With these ideas in place, the final sequences look as they do in Figure 7-30.

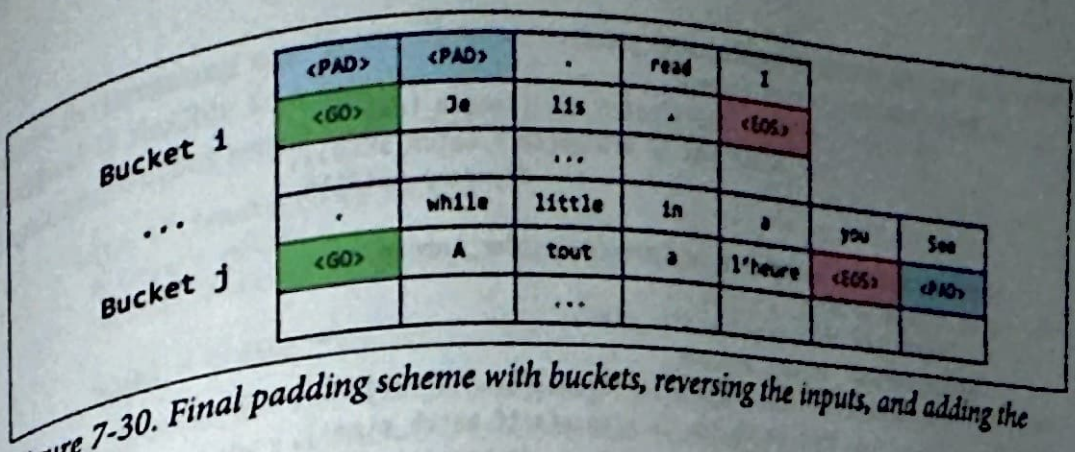

Figure 7-30. Final padding scheme with buckets, reversing the inputs, and adding the GO token

With these techniques described, we can now detail the implementation. The ideas are in a method called `get_batch()` in the code. This method collects a single batch of training data, given the `bucket_id`, which is chosen from the training loop, and the data. The result of this method includes the tokens in the source and target sequences and applies all of the techniques we just discussed, including the padding with buckets and reversing the inputs:

```
def get_batch(self, data, bucket_id):
  encoder_size, decoder_size = self.buckets[bucket_id]
  encoder_inputs, decoder_inputs = [], []
```

We first declare placeholders for each of the inputs that the encoder and decoder consume:

```
  for _ in xrange(self.batch_size):
    encoder_input, decoder_input = random.choice(data[
      bucket_id])

    # Encoder inputs are padded and then reversed.
    encoder_pad = [data_utils.PAD_ID] * (encoder_size - len(
                                         encoder_input))
    encoder_inputs.append(list(reversed(encoder_input +
                                        encoder_pad)))


    # Decoder inputs get an extra "GO" symbol,
    # and are then padded.
    decoder_pad_size = decoder_size - len(decoder_input) - 1
    decoder_inputs.append([data_utils.GO_ID] + decoder_input +
                          [data_utils.PAD_ID] *
                             decoder_pad_size)
```

Given the size of the batch, we gather that many encoder and decoder sequences:

```
    # Now we create batch-major vectors from the data selected
    # above.
  batch_encoder_inputs, batch_decoder_inputs, batch_weights =
    [], [], []

  # Batch encoder inputs are just re-indexed encoder_inputs.
```

```
    for length_idx in xrange(encoder_size):
      batch_encoder_inputs.append(
          np.array([encoder_inputs[batch_idx][length_idx]
                    for batch_idx in xrange(self.batch_size)],
                                              dtype=np.int32))

    # Batch decoder inputs are re-indexed decoder_inputs,
    # we create weights.
    for length_idx in xrange(decoder_size):
      batch_decoder_inputs.append(
          np.array([decoder_inputs[batch_idx][length_idx]
                    for batch_idx in xrange(self.batch_size)],
                                              dtype=np.int32))
```

With additional bookkeeping, we make sure that vectors are batch-major, meaning that the batch size is the first dimension in the tensor, and we resize the previously defined placeholders into the correct shape:

```
      # Create target_weights to be 0 for targets that
      # are padding.
      batch_weight = np.ones(self.batch_size, dtype=np.float32)
      for batch_idx in xrange(self.batch_size):
        # We set weight to 0 if the corresponding target is
        # a PAD symbol.
        # The corresponding target is decoder_input shifted
        # by 1 forward.
        if length_idx < decoder_size - 1:
          target = decoder_inputs[batch_idx][length_idx + 1]
        if length_idx == decoder_size - 1 or
              target == data_utils.PAD_ID:
          batch_weight[batch_idx] = 0.0
      batch_weights.append(batch_weight)
    return batch_encoder_inputs, batch_decoder_inputs,
    batch_weights
```

Finally, we set the target weights of zero to those tokens that are simply the PAD token.

With the data preparation now done, we are ready to begin building and training our model! We first detail the code used during training and test time, and abstract the model away for now. When doing so, we can make sure we understand the high-level pipeline, and we will then study the seq2seq model in more depth. As always, the first step during training is to load our data:

```
def train():
 """Train a en->fr translation model using WMT data."""
 # Prepare WMT data.
 print("Preparing WMT data in %s" % FLAGS.data_dir)
 en_train, fr_train, en_dev, fr_dev, _, _ =
   data_utils.prepare_wmt_data(
     FLAGS.data_dir, FLAGS.en_vocab_size, FLAGS.fr_vocab_size)
```

After instantiating our TensorFlow session, we first create our model. Note that this method is flexible to a number of different architectures as long as they respect the input and output requirements detailed by the `train()` method:

```
with tf.Session() as sess:
  # Create model.
  print("Creating %d layers of %d units." % (FLAGS.num_layers,
                                              FLAGS.size))

  model = create_model(sess, False)
```

We now process the data using various utility functions into buckets that are later used by `get_batch()` to fetch the data. We also create an array of real numbers from 0 to 1 that roughly dictate the likelihood of selecting a bucket, normalized by the size of buckets. When `get_batch()` selects buckets, it will do so respecting these probabilities:

```
  # Read data into buckets and compute their sizes.
  print ("Reading development and training data (limit: %d)."
         % FLAGS.max_train_data_size)
  dev_set = read_data(en_dev, fr_dev)
  train_set = read_data(en_train, fr_train,
                        FLAGS.max_train_data_size)
  train_bucket_sizes = [len(train_set[b]) for b in xrange(
                         len(_buckets))]
  train_total_size = float(sum(train_bucket_sizes))

  # A bucket scale is a list of increasing numbers
  # from 0 to 1 that we'll use to select a bucket.
  # Length of [scale[i], scale[i+1]] is proportional to
  # the size if i-th training bucket, as used later.
  train_buckets_scale = [sum(train_bucket_sizes[:i + 1]) /
                             train_total_size
                         for i in xrange(len(
                             train_bucket_sizes))]
```

With data ready, we now enter our main training loop. We initialize various loop variables, like `current_step` and `previous_losses` to 0 or empty. It is important to note that each cycle in the `while` loop denotes one epoch, which is the terminology for looping through one batch of training data. Therefore, per epoch, we select a `bucket_id`, get a batch using `get_batch`, and then *step* forward in our model with the data:

```
# This is the training loop.
step_time, loss = 0.0, 0.0
current_step = 0
previous_losses = []
while True:
  # Choose a bucket according to data distribution.
  # We pick a random number
  # in [0, 1] and use the corresponding interval
  # in train_buckets_scale.
```

```
random_number_01 = np.random.random_sample()
bucket_id = min([i for i in xrange(len(
                train_buckets_scale))
                 if train_buckets_scale[i] >
                   random_number_01])

# Get a batch and make a step.
start_time = time.time()
encoder_inputs, decoder_inputs, target_weights =
   model.get_batch(
    train_set, bucket_id)
_, step_loss, _ = model.step(sess, encoder_inputs,
                             decoder_inputs,
                             target_weights, bucket_id,
                             False)
```

We measure the loss incurred during prediction time as well as keep track of other running metrics:

```
step_time += (time.time() - start_time) /
  FLAGS.steps_per_checkpoint
loss += step_loss / FLAGS.steps_per_checkpoint
current_step += 1
```

Lastly, every so often, as dictated by a global variable, we will carry out a number of tasks. First, we print statistics for the previous batch, such as the loss, the learning rate, and the perplexity. If we find that the loss is not decreasing, it is possible that the model has fallen into a local optima. To assist the model in escaping this, we anneal the learning rate so that it won't make large leaps in any particular direction. At this point, we also save a copy of the model and its weights and activations to disk:

```
# Once in a while, we save checkpoint, print statistics,
# and run evals.
if current_step % FLAGS.steps_per_checkpoint == 0:
 # Print statistics for the previous epoch.
 perplexity = math.exp(float(loss)) if loss <
   300 else float("inf")
 print ("global step %d learning rate %.4f
        step-time %.2f perplexity "
       "%.2f" % (model.global_step.eval(),
                 model.learning_rate.eval(),
                 step_time, perplexity))
 # Decrease learning rate if no improvement was seen over
 # last 3 times.
 if len(previous_losses) > 2 and loss > max(
       previous_losses[-3:]):
  sess.run(model.learning_rate_decay_op)
 previous_losses.append(loss)
 # Save checkpoint and zero timer and loss.
 checkpoint_path = os.path.join(FLAGS.train_dir,
   "translate.ckpt")
 model.saver.save(sess, checkpoint_path,
```

```
                global_step=model.global_step)
        step_time, loss = 0.0, 0.0
```

Finally, we will measure the model's performance on a held-out development set. By doing so, we can measure the generalization of the model and see if is improving, and if so, at what rate. We again fetch data using get_batch, but this time only use bucket_id from the held-out set. We again step through the model, but this time without updating any of the weights because the last argument in the step() method is True as opposed to False during the main training loop; we will discuss the semantics of step() later. We measure this evaluation loss and display it to the user:

```
        # Run evals on development set and print
        # their perplexity.
        for bucket_id in xrange(len(_buckets)):
          if len(dev_set[bucket_id]) == 0:
            print("  eval: empty bucket %d" % (bucket_id))
            continue
          encoder_inputs, decoder_inputs,
          target_weights = model.get_batch(
              dev_set, bucket_id)
          # attns, _, eval_loss, _ = model.step(sess,
            encoder_inputs, decoder_inputs,
          _, eval_loss, _ = model.step(sess, encoder_inputs,
                                       decoder_inputs,
                                       target_weights,
                                       bucket_id,
                                       True)
          eval_ppx = math.exp(float(eval_loss)) if eval_loss <
                      300 else float(
              "inf")
          print("  eval: bucket %d perplexity %.2f" % (
                 bucket_id, eval_ppx))
        sys.stdout.flush()
```

We also have another major use case for our model: single-use prediction. In other words, we want to be able to use our trained model to translate new sentences that we, or other users, provide. To do so, we use the decode() method. This method will essentially carry out the same functions as was done in the evaluation loop for the held-out development set. However, the largest difference is that during training and evaluation, we never needed the model to translate the output embeddings to output tokens that are human-readable, which is something we do here. We detail this method now.

Because this is a separate mode of computation, we need to again instantiate the TensorFlow session and create the model, or load a saved model from a previous checkpoint step:

```
def decode():
  with tf.Session() as sess:
```

```
  # Check if the sizes match.
  encoder_size, decoder_size = self.buckets[bucket_id]
  if len(encoder_inputs) != encoder_size:
    raise ValueError("Encoder length must be equal to the one
                     in bucket,"
                     " %d != %d." % (len(
                     encoder_inputs), encoder_size))
  if len(decoder_inputs) != decoder_size:
    raise ValueError("Decoder length must be equal to the one
                     in bucket,"
                     " %d != %d." % (len(decoder_inputs),
                                     decoder_size))
  if len(target_weights) != decoder_size:
    raise ValueError("Weights length must be equal to the one
                     in bucket,"
                     " %d != %d." % (len(target_weights),
                                     decoder_size))

  # Input feed: encoder inputs, decoder inputs, target_weights,
  # as provided.
  input_feed = {}
  for l in xrange(encoder_size):
    input_feed[self.encoder_inputs[l].name] = encoder_inputs[l]
  for l in xrange(decoder_size):
    input_feed[self.decoder_inputs[l].name] = decoder_inputs[l]
    input_feed[self.target_weights[l].name] = target_weights[l]

  # Since our targets are decoder inputs shifted by one,
  # we need one more.
  last_target = self.decoder_inputs[decoder_size].name
  input_feed[last_target] = np.zeros([self.batch_size],
                                     dtype=np.int32)
```

The output feed, if a loss is computed and needs to be backpropagated through the network, contains the update operation that performs the stochastic gradient descent and computes the gradient norm and loss for the batch:

```
  # Output feed: depends on whether we do a backward step or
  # not.
  if not forward_only:
    output_feed = [self.updates[bucket_id],  # Update Op that
                                             # does SGD.
                   self.gradient_norms[bucket_id],  # Gradient
                                                    # norm.
                   self.losses[bucket_id]]  # Loss for this
                                            # batch.
  else:
    output_feed = [self.losses[bucket_id]]  # Loss for this
                                            # batch.
    for l in xrange(decoder_size):  # Output logits.
      output_feed.append(self.outputs[bucket_id][l])
```

These two feeds are passed to `session.run()`. Depending on the `forward_only` flag, either the gradient norm and loss are returned for maintaining statistics, or the outputs are returned for decoding purposes:

```
outputs = session.run(output_feed, input_feed)
if not forward_only:
  return outputs[1], outputs[2], None #, attns
    # Gradient norm, loss, no outputs.
else:
  return None, outputs[0], outputs[1:] #, attns
    # No gradient norm, loss, outputs.
```

Now, we can study the model itself. The constructor for the model sets up the computation graph using high-level constructs created. We first review the `create_model()` method briefly, which calls this constructor, and then discuss the details of this constructor.

The `create_model()` method itself is fairly straightforward: it uses a number of user-defined or default flags, such as the sizes of the English and French vocabularies and batch size, to create the model by using the constructor `seq2seq_model.Seq2SeqModel`. One particularly interesting flag is the `use_fp16` flag. With this, a lower precision is used as the type in the underlying numpy arrays; this results in faster performance at the cost of some amount of precision. However, it's often the case that 16-bit representations are sufficient for representing losses and gradient updates and often perform close to the level of using 32-bit representations. Model creation can be achieved using the following code:

```
def create_model(session, forward_only):
 """Create translation model and initialize or
    load parameters in session."""
 dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
 model = seq2seq_model.Seq2SeqModel(
     FLAGS.en_vocab_size,
     FLAGS.fr_vocab_size,
     _buckets,
     FLAGS.size,
     FLAGS.num_layers,
     FLAGS.max_gradient_norm,
     FLAGS.batch_size,
     FLAGS.learning_rate,
     FLAGS.learning_rate_decay_factor,
     forward_only=forward_only,
     dtype=dtype)
```

Before returning the model, a check is done to see if there are any previously checkpointed models from earlier training runs. If so, this model and its parameters are read into the model variable and used. This allows us to stop training at a checkpoint and later resume it without training from scratch. Otherwise, the fresh model created is returned as the main object:

```
    ckpt = tf.train.get_checkpoint_state(FLAGS.train_dir)
    if ckpt and tf.train.checkpoint_exists(
       ckpt.model_checkpoint_path):
      print("Reading model parameters from %s"
            % ckpt.model_checkpoint_path)
      model.saver.restore(session, ckpt.model_checkpoint_path)
    else:
      print("Created model with fresh parameters.")
      session.run(tf.global_variables_initializer())
    return model
```

We now review the constructor `seq2seq_model.Seq2SeqModel`. This constructor creates the entire computation graph and will occasionally call certain lower-level constructs. Before we jump to those details, we continue in our top-down investigation of the code and sketch the details of the overarching computation graph.

The same arguments passed to `create_model()` are passed to this constructor, and a few class-level fields are created:

```
class Seq2SeqModel(object):
  def __init__(self,
               source_vocab_size,
               target_vocab_size,
               buckets,
               size,
               num_layers,
               max_gradient_norm,
               batch_size,
               learning_rate,
               learning_rate_decay_factor,
               use_lstm=False,
               num_samples=512,
               forward_only=False,
               dtype=tf.float32):
    self.source_vocab_size = source_vocab_size
    self.target_vocab_size = target_vocab_size
    self.buckets = buckets
    self.batch_size = batch_size
    self.learning_rate = tf.Variable(
        float(learning_rate), trainable=False, dtype=dtype)
    self.learning_rate_decay_op = self.learning_rate.assign(
        self.learning_rate * learning_rate_decay_factor)
    self.global_step = tf.Variable(0, trainable=False)
```

The next part creates the sampled softmax and the output projection. This is an improvement over basic seq2seq models in that they allow for efficient decoding over large output vocabularies and project the output logits to the correct space:

```
# If we use sampled softmax, we need an output projection.
output_projection = None
softmax_loss_function = None
# Sampled softmax only makes sense if we sample less than
# vocabulary size.
if num_samples > 0 and num_samples <
  self.target_vocab_size:
  w_t = tf.get_variable("proj_w", [self.target_vocab_size,
                                   size], dtype=dtype)

  w = tf.transpose(w_t)
  b = tf.get_variable("proj_b", [self.target_vocab_size],
                      dtype=dtype)

  output_projection = (w, b)

  def sampled_loss(inputs, labels):
    labels = tf.reshape(labels, [-1, 1])
    # We need to compute the sampled_softmax_loss using
    # 32bit floats to avoid numerical instabilities.
    local_w_t = tf.cast(w_t, tf.float32)
    local_b = tf.cast(b, tf.float32)
    local_inputs = tf.cast(inputs, tf.float32)
    return tf.cast(
        tf.nn.sampled_softmax_loss(local_w_t, local_b,
                                   local_inputs, labels,
                                   num_samples,
                                   self.target_vocab_size),

        dtype)
  softmax_loss_function = sampled_loss
```

Based on the flags, we choose the underlying RNN cell, whether it's a GRU cell, an LSTM cell, or a multilayer LSTM cell. Production systems will rarely use single-layer LSTM cells, but they are much faster to train and may make the debugging cycle faster:

```
# Create the internal multi-layer cell for our RNN.
single_cell = tf.nn.rnn_cell.GRUCell(size)
if use_lstm:
  single_cell = tf.nn.rnn_cell.BasicLSTMCell(size)
cell = single_cell
if num_layers > 1:
  cell = tf.nn.rnn_cell.MultiRNNCell([single_cell] *
    num_layers)
```

The recurrent function `seq2seq_f()` is defined with `seq2seq.embedding_attention_seq2seq()`, which we will discuss later:

```
# The seq2seq function: we use embedding for the
# input and attention.
def seq2seq_f(encoder_inputs, decoder_inputs, do_decode):
  return seq2seq.embedding_attention_seq2seq(
      encoder_inputs,
      decoder_inputs,
```

```
            cell,
            num_encoder_symbols=source_vocab_size,
            num_decoder_symbols=target_vocab_size,
            embedding_size=size,
            output_projection=output_projection,
            feed_previous=do_decode,
            dtype=dtype)
```

We define placeholders for the inputs and targets:

```
        # Feeds for inputs.
        self.encoder_inputs = []
        self.decoder_inputs = []
        self.target_weights = []
        for i in xrange(buckets[-1][0]):  # Last bucket is
                                          # the biggest one.
          self.encoder_inputs.append(tf.placeholder(tf.int32,
                                shape=[None],
                                name="encoder{0}".format(i)))
        for i in xrange(buckets[-1][1] + 1):
          self.decoder_inputs.append(tf.placeholder(tf.int32,
                                shape=[None],
                                name="decoder{0}".format(i)))
          self.target_weights.append(tf.placeholder(dtype,
                                shape=[None],
                                name="weight{0}".format(i)))

        # Our targets are decoder inputs shifted by one.
        targets = [self.decoder_inputs[i + 1]
                   for i in xrange(len(self.decoder_inputs) - 1)]
```

We now compute the outputs and losses from the function `seq2seq.model_with_buckets`. This function simply constructs the seq2seq model to be compatible with buckets and computes the loss either by averaging over the entire example sequence or as a weighted cross-entropy loss for a sequence of logits:

```
        # Training outputs and losses.
        if forward_only:
          self.outputs, self.losses = seq2seq.model_with_buckets(
              self.encoder_inputs, self.decoder_inputs, targets,
              self.target_weights, buckets, lambda x, y:
                seq2seq_f(x, y, True),
              softmax_loss_function=softmax_loss_function)
          # If we use output projection, we need to project outputs
          # for decoding.
          if output_projection is not None:
            for b in xrange(len(buckets)):
              self.outputs[b] = [
                  tf.matmul(output, output_projection[0]) +
                    output_projection[1]
                  for output in self.outputs[b]
              ]
        else:
```

```
self.outputs, self.losses = seq2seq.model_with_buckets(
    self.encoder_inputs, self.decoder_inputs, targets,
    self.target_weights, buckets,
    lambda x, y: seq2seq_f(x, y, False),
    softmax_loss_function=softmax_loss_function)
```

Finally, we update the parameters of the model (because they are trainable variables) using some form of gradient descent. We use vanilla SGD with gradient clipping, but we are free to use any optimizer—the results will certainly improve and training may proceed much faster. Afterward, we save all variables:

```
# Gradients and SGD update operation for training the model.
params = tf.trainable_variables()
if not forward_only:
  self.gradient_norms = []
  self.updates = []
  opt = tf.train.GradientDescentOptimizer(
        self.learning_rate)
  for b in xrange(len(buckets)):
    gradients = tf.gradients(self.losses[b], params)
    clipped_gradients, norm = tf.clip_by_global_norm(
                              gradients,
                              max_gradient_norm)
    self.gradient_norms.append(norm)
    self.updates.append(opt.apply_gradients(
        zip(clipped_gradients, params), global_step=
                                        self.global_step))

self.saver = tf.train.Saver(tf.all_variables())
```

With the high-level detail of the computation graph described, we now describe the last and lowest level of the model: the internals of `seq2seq.embedding_attention_seq2seq()`.

When initializing this model, several flags and arguments are passed as function arguments. One argument of particular note is feed_previous. When this is true, the decoder will use the outputted logit at time step T as input to time step T+1. In this way, it is sequentially decoding the next token based on all tokens thus far. We can describe this type of decoding, where the next output depends on all previous outputs, as *autoregressive decoding*:

```
def embedding_attention_seq2seq(encoder_inputs,
                                decoder_inputs,
                                cell,
                                num_encoder_symbols,
                                num_decoder_symbols,
                                embedding_size,
                                output_projection=None,
                                feed_previous=False,
                                dtype=None,
```

```
                                    scope=None,
                                    initial_state_attention=False):
```

We first create the wrapper for the encoder.

```
with variable_scope.variable_scope(
    scope or "embedding_attention_seq2seq", dtype=dtype)
     as scope:
  dtype = scope.dtype
  encoder_cell = rnn_cell.EmbeddingWrapper(
      cell,
      embedding_classes=num_encoder_symbols,
      embedding_size=embedding_size)
  encoder_outputs, encoder_state = rnn.rnn(
      encoder_cell, encoder_inputs, dtype=dtype)
```

In this following code snippet, we calculate a concatenation of encoder outputs to put attention on; this is important because it allows the decoder to attend over these states as a distribution:

```
# First calculate a concatenation of encoder outputs
# to put attention on.
top_states = [
    array_ops.reshape(e, [-1, 1, cell.output_size]) for e
     in encoder_outputs
]
attention_states = array_ops.concat(1, top_states)
```

Now, we create the decoder. If the `output_projection` flag is not specified, the cell is wrapped to be one that uses an output projection:

```
output_size = None
if output_projection is None:
  cell = rnn_cell.OutputProjectionWrapper(cell,
          num_decoder_symbols)
  output_size = num_decoder_symbols
```

From here, we compute the outputs and states using the `embedding_attention_decoder`:

```
if isinstance(feed_previous, bool):
  return embedding_attention_decoder(
      decoder_inputs,
      encoder_state,
      attention_states,
      cell,
      num_decoder_symbols,
      embedding_size,
      output_size=output_size,
      output_projection=output_projection,
      feed_previous=feed_previous,
      initial_state_attention=initial_state_attention)
```

The embedding_attention_decoder is a simple improvement over the attention_decoder described in the previous section; essentially, the inputs are projected to a learned embedding space, which usually improves performance. The loop function, which simply describes the dynamics of the recurrent cell with embedding, is invoked in this step:

```
def embedding_attention_decoder(decoder_inputs,
                                initial_state,
                                attention_states,
                                cell,
                                num_symbols,
                                embedding_size,
                                output_size=None,
                                output_projection=None,
                                feed_previous=False,
                                update_embedding_for_previous=
                                 True,
                                dtype=None,
                                scope=None,
                                initial_state_attention=False):


if output_size is None:
  output_size = cell.output_size
if output_projection is not None:
  proj_biases = ops.convert_to_tensor(output_projection[1],
    dtype=dtype)
  proj_biases.get_shape().assert_is_compatible_with(
      [num_symbols])

with variable_scope.variable_scope(
    scope or "embedding_attention_decoder", dtype=dtype)
      as scope:

  embedding = variable_scope.get_variable("embedding",
                                          [num_symbols,
                                           embedding_size])
  loop_function = _extract_argmax_and_embed(
      embedding, output_projection,
      update_embedding_for_previous) if feed_previous
        else None
  emb_inp = [
      embedding_ops.embedding_lookup(embedding, i) for i in
        decoder_inputs
  ]
  return attention_decoder(
      emb_inp,
      initial_state,
      attention_states,
      cell,
      output_size=output_size,
```

```
        loop_function=loop_function,
        initial_state_attention=initial_state_attention)
```

The last step is to study the `attention_decoder` itself. As the name suggests, the main feature of this decoder is that it computes a set of attention weights over the hidden states that the encoder emitted during encoding. After defensive checks, we reshape the hidden features to the right size:

```
def attention_decoder(decoder_inputs,
                      initial_state,
                      attention_states,
                      cell,
                      output_size=None,
                      loop_function=None,
                      dtype=None,
                      scope=None,
                      initial_state_attention=False):
  if not decoder_inputs:
    raise ValueError("Must provide at least 1 input to attention
                      decoder.")
  if attention_states.get_shape()[2].value is None:
    raise ValueError("Shape[2] of attention_states must be known:
      %s" %
                     attention_states.get_shape())
  if output_size is None:
    output_size = cell.output_size

  with variable_scope.variable_scope(
      scope or "attention_decoder", dtype=dtype) as scope:
    dtype = scope.dtype

    batch_size = array_ops.shape(decoder_inputs[0])[0]  # Needed
                                                        # for
                                                      #reshaping.
    attn_length = attention_states.get_shape()[1].value
    if attn_length is None:
      attn_length = array_ops.shape(attention_states)[1]
    attn_size = attention_states.get_shape()[2].value

    # To calculate W1 * h_t we use a 1-by-1 convolution,
    # need to reshape before.
    hidden = array_ops.reshape(attention_states,
                               [-1, attn_length, 1, attn_size])
    hidden_features = []
    v = []
    attention_vec_size = attn_size  # Size of query vectors
      for attention.
    k = variable_scope.get_variable("AttnW_0",
                                    [1, 1, attn_size,
                                     attention_vec_size])
    hidden_features.append(nn_ops.conv2d(hidden, k,
      [1, 1, 1, 1], "SAME"))
```

```
    v.append(
      variable_scope.get_variable("AttnV_0",
                                  [attention_vec_size]))

  state = initial_state
```

We now define the attention() method itself, which consumes a query vector and returns the attention-weighted vector over the hidden states. This method implements the same attention as described in the previous section:

```
  def attention(query):
    """Put attention masks on hidden using hidden_features
     and query."""
    ds = []  # Results of attention reads will be
             # stored here.
    if nest.is_sequence(query):  # If the query is a tuple,
                                 # flatten it.
      query_list = nest.flatten(query)
      for q in query_list:  # Check that ndims == 2 if
                            # specified.
        ndims = q.get_shape().ndims
        if ndims:
          assert ndims == 2
      query = array_ops.concat(1, query_list)
      # query = array_ops.concat(query_list, 1)
    with variable_scope.variable_scope("Attention_0"):
      y = linear(query, attention_vec_size, True)
      y = array_ops.reshape(y, [-1, 1, 1,
           attention_vec_size])
      # Attention mask is a softmax of v^T * tanh(...).
      s = math_ops.reduce_sum(v[0] * math_ops.tanh(
                              hidden_features[0] + y),
                              [2, 3])
      a = nn_ops.softmax(s)
      # Now calculate the attention-weighted vector d.
      d = math_ops.reduce_sum(
          array_ops.reshape(a, [-1, attn_length, 1, 1]) *
                                       hidden, [1, 2])
      ds.append(array_ops.reshape(d, [-1, attn_size]))
    return ds
```

Using the function, we compute the attention over each of the output states, starting with the initial state:

```
  outputs = []
  prev = None
  batch_attn_size = array_ops.stack([batch_size, attn_size])
  attns = [array_ops.zeros(batch_attn_size, dtype=dtype)]
  for a in attns:  # Ensure the second shape of attention
                   # vectors is set.
    a.set_shape([None, attn_size])
  if initial_state_attention:
    attns = attention(initial_state)
```

Now we loop over the rest of the inputs. We perform a defensive check to ensure that the input at the current time step is the right size. Then we run the RNN cell as well as the attention query. These two are then combined and passed to the output according to the same dynamics:

```
for i, inp in enumerate(decoder_inputs):
  if i > 0:
    variable_scope.get_variable_scope().reuse_variables()
  # If loop_function is set, we use it instead of
  # decoder_inputs.
  if loop_function is not None and prev is not None:
    with variable_scope.variable_scope("loop_function",
      reuse=True):
      inp = loop_function(prev, i)
  # Merge input and previous attentions into one vector of
  # the right size.
  input_size = inp.get_shape().with_rank(2)[1]
  if input_size.value is None:
    raise ValueError("Could not infer input size from input:
      %s" % inp.name)
  x = linear([inp] + attns, input_size, True)
  # Run the RNN.
  cell_output, state = cell(x, state)
  # Run the attention mechanism.
  if i == 0 and initial_state_attention:
    with variable_scope.variable_scope(
        variable_scope.get_variable_scope(), reuse=True):
      attns = attention(state)
  else:
    attns = attention(state)

  with variable_scope.variable_scope(
    "AttnOutputProjection"):
    output = linear([cell_output] + attns, output_size,
            True)
  if loop_function is not None:
    prev = output
  outputs.append(output)

return outputs, state
```

With this, we've successfully completed a full tour of the implementation details of a fairly sophisticated neural machine translation system. Production systems have additional tricks that are not as generalizable, and these systems are trained on huge compute servers to ensure that state-of-the-art performance is met.

For reference, this exact model was trained on eight NVIDIA Telsa M40 GPUs for four days. We show plots for the perplexity in Figure 7-31 and Figure 7-32, and show the learning rate anneal over time as well.

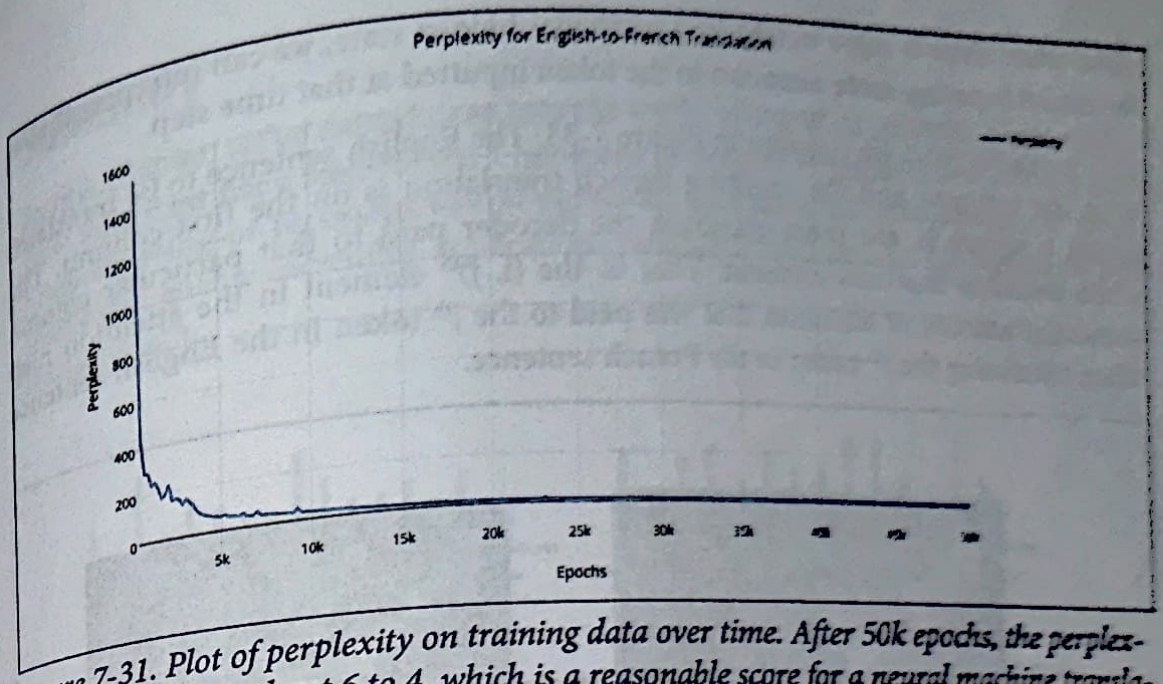

Figure 7-31. Plot of perplexity on training data over time. After 50k epochs, the perplexity decreases from about 6 to 4, which is a reasonable score for a neural machine translation system.

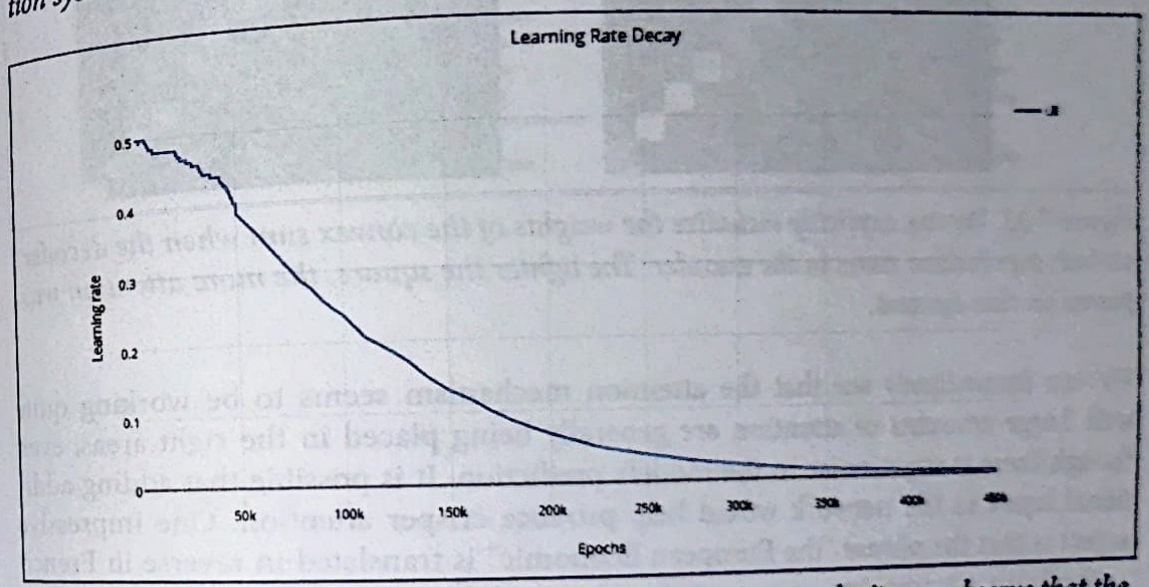

Figure 7-32. Plot of learning rate over time; as opposed to perplexity, we observe that the learning rate almost smoothly declines to 0. This means that by the time we stopped training, the model was approaching a stable state.

To showcase the attentional model more explicitly, we can visualize the attention that the decoder LSTM computes while translating a sentence from English to French. In particular, we know that as the encoder LSTM is updating its cell state in order to compress the sentence into a continuous vector representations, it also computes hidden states at every time step. We know that the decoder LSTM computes a convex sum over these hidden states, and one can think of this sum as the attention mecha-

nism; when there is more weight on a particular hidden state, we can interpret that as the model is paying more attention to the token inputted at that time step.

This is exactly what we visualize in Figure 7-33. The English sentence to be translated is on the top row, and the resulting French translation is on the first column. The lighter a square is, the more attention the decoder paid to that particular column when decoding that row element. That is, the $(i, j)^{th}$ element in the attention map shows the amount of attention that was paid to the j^{th} token in the English sentence when translating the i^{th} token in the French sentence.

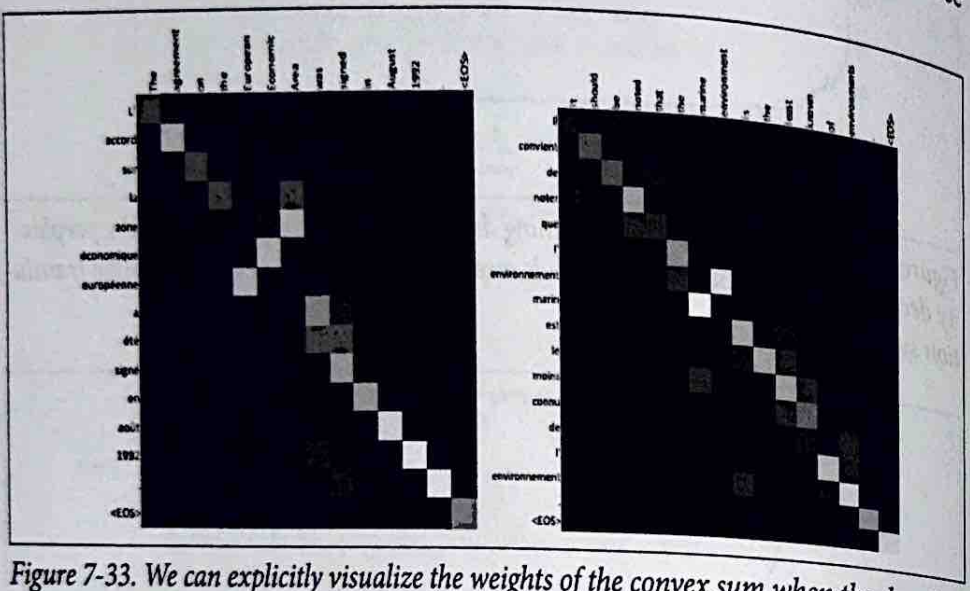

Figure 7-33. We can explicitly visualize the weights of the convex sum when the decoder attends over hidden states in the encoder. The lighter the square, the more attention was placed on that element.

We can immediately see that the attention mechanism seems to be working quite well. Large amounts of attention are generally being placed in the right areas, even though there is slight noise in the model's prediction. It is possible that adding additional layers to the network would help produce crisper attention. One impressive aspect is that the phrase "the European Economic" is translated in reverse in French as the "zone économique européenne," and as such, the attention weights reflect this flip! These kinds of attention patterns may be even more interesting when translating from English to a different language that does not parse smoothly from left to right.

With one of the most fundamental architectures understood and implemented, we now move forward to study exciting new developments with recurrent neural networks and begin a foray into more sophisticated learning.

Summary

In this chapter, we've delved deep into the world of sequence analysis. We've analyzed how we might hack feed-forward networks to process sequences, developed a strong understanding of recurrent neural networks, and explored how attentional mechanisms can enable incredible applications ranging from language translation to audio transcription.