Equation 16-1. Attention mechanisms

$$\widetilde{\mathbf{h}}_{(t)} = \sum_{i} \alpha_{(t,i)} \mathbf{y}_{(i)}$$

$$\text{with } \alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)}$$

$$\text{and } e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^{\top} \mathbf{y}_{(i)} & \textit{dot} \\ \mathbf{h}_{(t)}^{\top} \mathbf{W} \mathbf{y}_{(i)} & \textit{general} \\ \mathbf{v}^{\top} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & \textit{concat} \end{cases}$$

Here is how you can add Luong attention to an Encoder–Decoder model using TensorFlow Addons:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(
    units, encoder_state, memory_sequence_length=encoder_sequence_length)
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

We simply wrap the decoder cell in an `AttentionWrapper`, and we provide the desired attention mechanism (Luong attention in this example).

Visual Attention

Attention mechanisms are now used for a variety of purposes. One of their first applications beyond NMT was in generating image captions using visual attention:[17] a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one word at a time. At each decoder time step (each word), the decoder uses the attention model to focus on just the right part of the image. For example, in Figure 16-7, the model generated the caption "A woman is throwing a frisbee in a park," and you can see what part of the input image the decoder focused its attention on when it was about to output the word "frisbee": clearly, most of its attention was focused on the frisbee.

17 Kelvin Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.

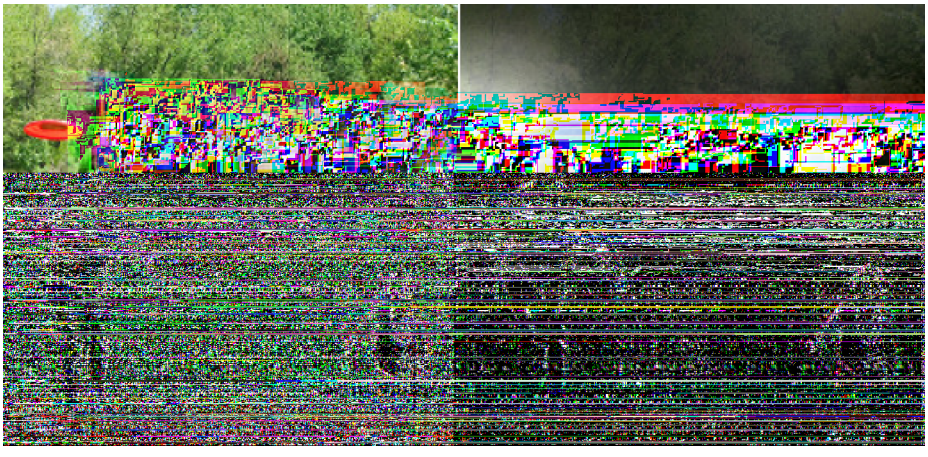


Figure 16-7. Visual attention: an input image (left) and the model's focus before producing the word "frisbee" (right)[18]

Explainability

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake: for example, if an image of a dog walking in the snow is labeled as "a wolf walking in the snow," then you can go back and check what the model focused on when it output the word "wolf." You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps the way the model learned to distinguish dogs from wolves is by checking whether or not there's a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great 2016 paper[19] by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier's prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement (think of a system deciding whether or not it should grant you a loan).

18 This is a part of figure 3 from the paper. It is reproduced with the kind authorization of the authors.

19 Marco Tulio Ribeiro et al., "'Why Should I Trust You?': Explaining the Predictions of Any Classifier," *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

Attention mechanisms are so powerful that you can actually build state-of-the-art models using only attention mechanisms.

Attention Is All You Need: The Transformer Architecture

In a groundbreaking 2017 paper,[20] a team of Google researchers suggested that "Attention Is All You Need." They managed to create an architecture called the *Transformer*, which significantly improved the state of the art in NMT without using any recurrent or convolutional layers,[21] just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). As an extra bonus, this architecture was also much faster to train and easier to parallelize, so they managed to train it at a fraction of the time and cost of the previous state-of-the-art models.

The Transformer architecture is represented in Figure 16-8.

20 Ashish Vaswani et al., "Attention Is All You Need," *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

21 Since the Transformer uses time-distributed `Dense` layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.

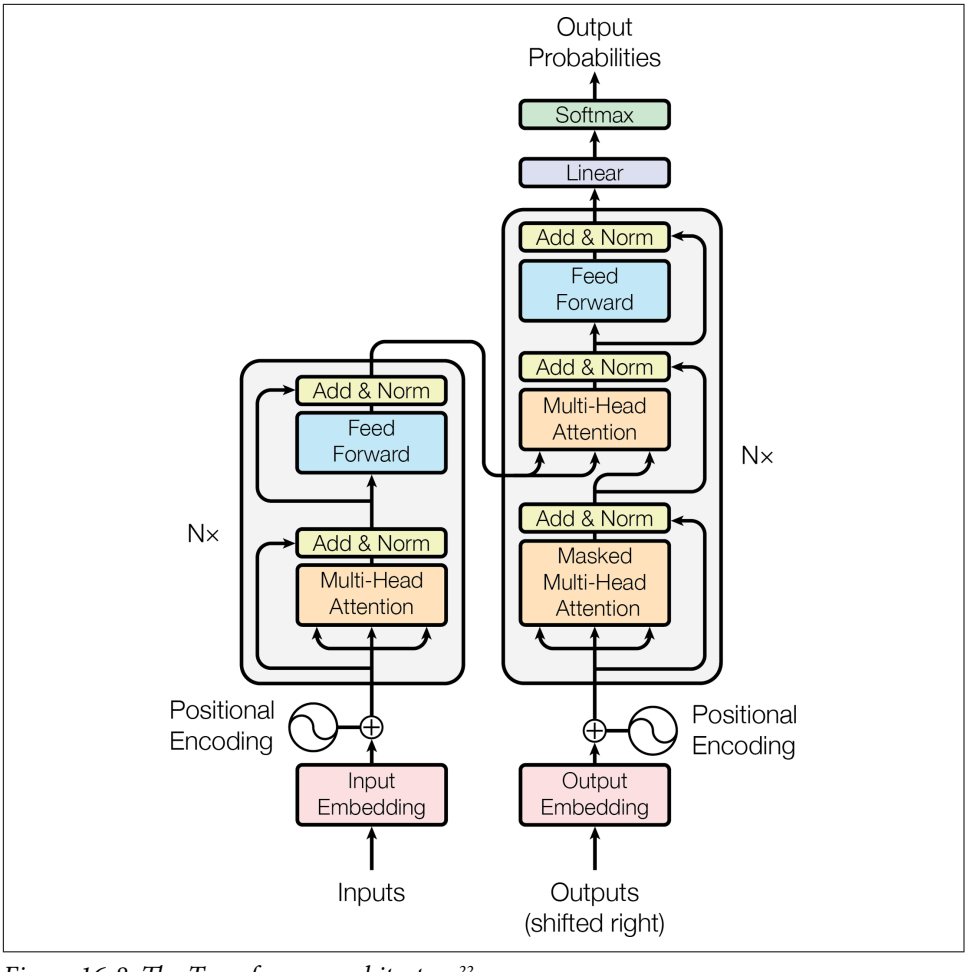


Figure 16-8. The Transformer architecture[22]

Let's walk through this figure:

- The lefthand part is the encoder. Just like earlier, it takes as input a batch of sentences represented as sequences of word IDs (the input shape is [*batch size*, *max input sentence length*]), and it encodes each word into a 512-dimensional representation (so the encoder's output shape is [*batch size*, *max input sentence length*, 512]). Note that the top part of the encoder is stacked N times (in the paper, $N = 6$).

22 This is figure 1 from the paper, reproduced with the kind authorization of the authors.

- The righthand part is the decoder. During training, it takes the target sentence as input (also represented as a sequence of word IDs), shifted one time step to the right (i.e., a start-of-sequence token is inserted at the beginning). It also receives the outputs of the encoder (i.e., the arrows coming from the left side). Note that the top part of the decoder is also stacked N times, and the encoder stack's final outputs are fed to the decoder at each of these N levels. Just like earlier, the decoder outputs a probability for each possible next word, at each time step (its output shape is [*batch size*, *max output sentence length*, *vocabulary length*]).
- During inference, the decoder cannot be fed targets, so we feed it the previously output words (starting with a start-of-sequence token). So the model needs to be called repeatedly, predicting one more word at every round (which is fed to the decoder at the next round, until the end-of-sequence token is output).
- Looking more closely, you can see that you are already familiar with most components: there are two embedding layers, $5 \times N$ skip connections, each of them followed by a layer normalization layer, $2 \times N$ "Feed Forward" modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function), and finally the output layer is a dense layer using the softmax activation function. All of these layers are time-distributed, so each word is treated independently of all the others. But how can we translate a sentence by only looking at one word at a time? Well, that's where the new components come in:
  - The encoder's *Multi-Head Attention* layer encodes each word's relationship with every other word in the same sentence, paying more attention to the most relevant ones. For example, the output of this layer for the word "Queen" in the sentence "They welcomed the Queen of the United Kingdom" will depend on all the words in the sentence, but it will probably pay more attention to the words "United" and "Kingdom" than to the words "They" or "welcomed." This attention mechanism is called *self-attention* (the sentence is paying attention to itself). We will discuss exactly how it works shortly. The decoder's *Masked Multi-Head Attention* layer does the same thing, but each word is only allowed to attend to words located before it. Finally, the decoder's upper Multi-Head Attention layer is where the decoder pays attention to the words in the input sentence. For example, the decoder will probably pay close attention to the word "Queen" in the input sentence when it is about to output this word's translation.
  - The *positional embeddings* are simply dense vectors (much like word embeddings) that represent the position of a word in the sentence. The n^{th} positional embedding is added to the word embedding of the n^{th} word in each sentence. This gives the model access to each word's position, which is needed because the Multi-Head Attention layers do not consider the order or the position of the words; they only look at their relationships. Since all the other layers are

time-distributed, they have no way of knowing the position of each word (either relative or absolute). Obviously, the relative and absolute word positions are important, so we need to give this information to the Transformer somehow, and positional embeddings are a good way to do this.

Let's look a bit closer at both these novel components of the Transformer architecture, starting with the positional embeddings.

Positional embeddings

A positional embedding is a dense vector that encodes the position of a word within a sentence: the i^{th} positional embedding is simply added to the word embedding of the i^{th} word in the sentence. These positional embeddings can be learned by the model, but in the paper the authors preferred to use fixed positional embeddings, defined using the sine and cosine functions of different frequencies. The positional embedding matrix P is defined in Equation 16-2 and represented at the bottom of Figure 16-9 (transposed), where $P_{p,i}$ is the i^{th} component of the embedding for the word located at the p^{th} position in the sentence.

Equation 16-2. Sine/cosine positional embeddings

$$P_{p,2i} = \sin\left(p/10000^{2i/d}\right)$$
$$P_{p,2i+1} = \cos\left(p/10000^{2i/d}\right)$$

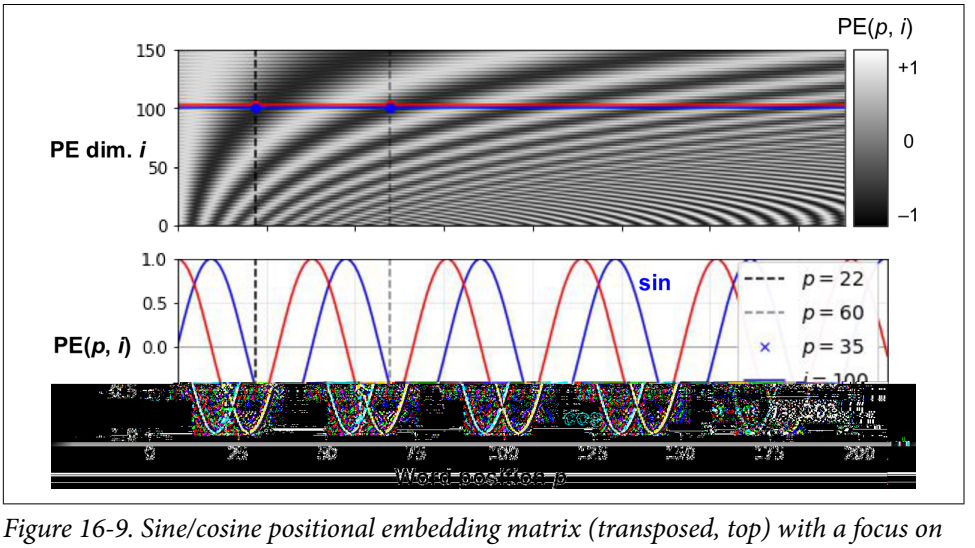


Figure 16-9. Sine/cosine positional embedding matrix (transposed, top) with a focus on two values of i (bottom)

This solution gives the same performance as learned positional embeddings do, but it can extend to arbitrarily long sentences, which is why it's favored. After the positional embeddings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional embedding for each position (e.g., the positional embedding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the bottom left of Figure 16-9, and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional embedding values in the embedding dimensions $i = 100$ and $i = 101$, as you can see in Figure 16-9. This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 25$ and $p = 35$ (marked by a cross).

There is no `PositionalEmbedding` layer in TensorFlow, but it is easy to create one. For efficiency reasons, we precompute the positional embedding matrix in the constructor (so we need to know the maximum sentence length, `max_steps`, and the number of dimensions for each word representation, `max_dims`). Then the `call()` method crops this embedding matrix to the size of the inputs, and it adds it to the inputs. Since we added an extra first dimension of size 1 when creating the positional embedding matrix, the rules of broadcasting will ensure that the matrix gets added to every sentence in the inputs:

```
class PositionalEncoding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1 # max_dims must be even
        p, i = np.meshgrid(np.arange(max_steps), np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**(2 * i / max_dims)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**(2 * i / max_dims)).T
        self.positional_embedding = tf.constant(pos_emb.astype(self.dtype))
    def call(self, inputs):
        shape = tf.shape(inputs)
        return inputs + self.positional_embedding[:, :shape[-2], :shape[-1]]
```

Then we can create the first layers of the Transformer:

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps, max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

Now let's look deeper into the heart of the Transformer model: the Multi-Head Attention layer.

Multi-Head Attention

To understand how a Multi-Head Attention layer works, we must first understand the *Scaled Dot-Product Attention* layer, which it is based on. Let's suppose the encoder analyzed the input sentence "They played chess," and it managed to understand that the word "They" is the subject and the word "played" is the verb, so it encoded this information in the representations of these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analog to a dictionary lookup: it's as if the encoder created a dictionary {"subject": "They", "verb": "played", …} and the decoder wanted to look up the value that corresponds to the key "verb." However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); it has vectorized representations of these concepts (which it learned during training), so the key it will use for the lookup (called the *query*) will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1. Then the model can compute a weighted sum of the corresponding values, so if the weight of the "verb" key is close to 1, then the weighted sum will be very close to the representation of the word "played." In short, you can think of this whole process as a differentiable dictionary lookup. The similarity measure used by the Transformer is just the dot product, like in Luong attention. In fact, the equation is the same as for Luong attention, except for a scaling factor. The equation is shown in Equation 16-3, in a vectorized form.

Equation 16-3. Scaled Dot-Product Attention

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_{keys}}}\right)\mathbf{V}$$

In this equation:

- Q is a matrix containing one row per query. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- K is a matrix containing one row per key. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.

- V is a matrix containing one row per value. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of each value.
- The shape of Q K^T is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. The output of the softmax function has the same shape, but all rows sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).
- The scaling factor scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the Masked Multi-Head Attention layer.

In the encoder, this equation is applied to every input sentence in the batch, with Q, K, and V all equal to the list of words in the input sentence (so each word in the sentence will be compared to every word in the same sentence, including itself). Similarly, in the decoder's masked attention layer, the equation will be applied to every target sentence in the batch, with Q, K, and V all equal to the list of words in the target sentence, but this time using a mask to prevent any word from comparing itself to words located after it (at inference time the decoder will only have access to the words it already output, not to future words, so during training we must mask out future output tokens). In the upper attention layer of the decoder, the keys K and values V are simply the list of word encodings produced by the encoder, and the queries Q are the list of word encodings produced by the decoder.

The `keras.layers.Attention` layer implements Scaled Dot-Product Attention, efficiently applying Equation 16-3 to multiple sentences in a batch. Its inputs are just like Q, K, and V, except with an extra batch dimension (the first dimension).

In TensorFlow, if `A` and `B` are tensors with more than two dimensions—say, of shape [2, 3, 4, 5] and [2, 3, 5, 6] respectively—then `tf.matmul(A, B)` will treat these tensors as 2×3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the i^{th} row and j^{th} column in `A` will be multiplied by the matrix at the i^{th} row and j^{th} column in `B`. Since the product of a 4×5 matrix with a 5×6 matrix is a 4×6 matrix, `tf.matmul(A, B)` will return an array of shape [2, 3, 4, 6].

If we ignore the skip connections, the layer normalization layers, the Feed Forward blocks, and the fact that this is Scaled Dot-Product Attention, not exactly Multi-Head Attention, then the rest of the Transformer model can be implemented like this:

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])

encoder_outputs = Z
Z = decoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)
```

The `use_scale=True` argument creates an additional parameter that lets the layer learn how to properly downscale the similarity scores. This is a bit different from the Transformer model, which always downscales the similarity scores by the same factor ($\sqrt{d_{\text{keys}}}$). The `causal=True` argument when creating the second attention layer ensures that each output token only attends to previous output tokens, not future ones.

Now it's time to look at the final piece of the puzzle: what is a Multi-Head Attention layer? Its architecture is shown in Figure 16-10.

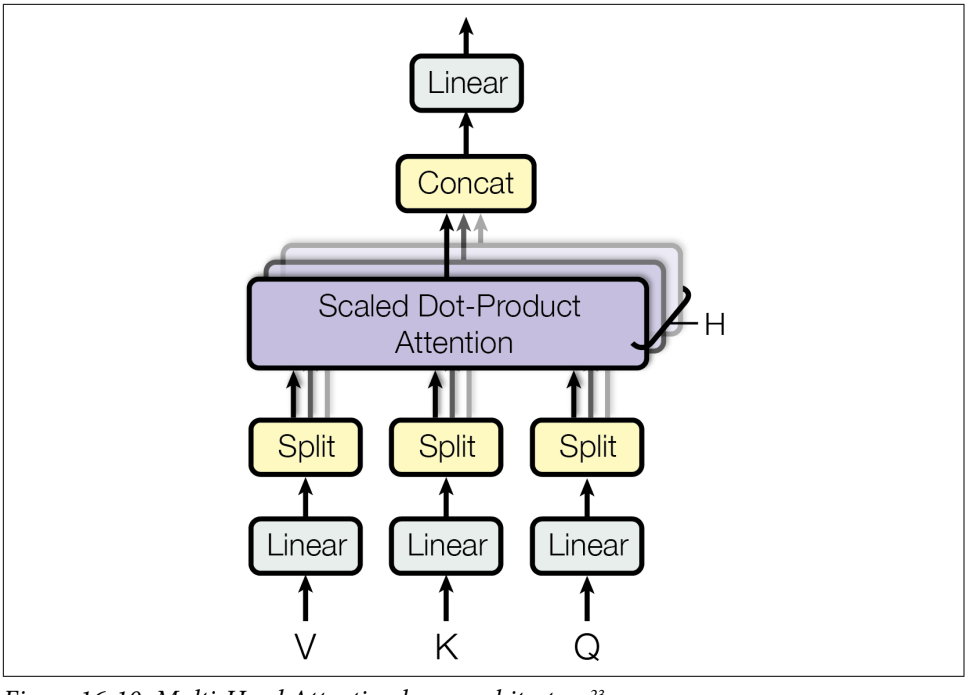


Figure 16-10. Multi-Head Attention layer architecture[23]

As you can see, it is just a bunch of Scaled Dot-Product Attention layers, each preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed `Dense` layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed). But why? What is the intuition behind this architecture? Well, consider the word "played" we discussed earlier (in the sentence "They played chess"). The encoder was smart enough to encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the past tense. In short, the word representation encodes many different characteristics of the word. If we just used a single Scaled Dot-Product Attention layer, we would only be able to query all of these characteristics in one shot. This is why the Multi-Head Attention layer applies multiple different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word's characteristics. Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a

23 This is the right part of figure 2 from the paper, reproduced with the kind authorization of the authors.

verb, another linear layer will extract just the fact that it is past tense, and so on. Then the Scaled Dot-Product Attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

At the time of this writing, there is no `Transformer` class or `MultiHeadAttention` class available for TensorFlow 2. However, you can check out TensorFlow's great tutorial for building a Transformer model for language understanding. Moreover, the TF Hub team is currently porting several Transformer-based modules to TensorFlow 2, and they should be available soon. In the meantime, I hope I have demonstrated that it is not that hard to implement a Transformer yourself, and it is certainly a great exercise!

Recent Innovations in Language Models

The year 2018 has been called the "ImageNet moment for NLP": progress was astounding, with larger and larger LSTM and Transformer-based architectures trained on immense datasets. I highly recommend you check out the following papers, all published in 2018:

- The ELMo paper[24] by Matthew Peters introduced *Embeddings from Language Models* (ELMo): these are contextualized word embeddings learned from the internal states of a deep bidirectional language model. For example, the word "queen" will not have the same embedding in "Queen of the United Kingdom" and in "queen bee."
- The ULMFiT paper[25] by Jeremy Howard and Sebastian Ruder demonstrated the effectiveness of unsupervised pretraining for NLP tasks: the authors trained an LSTM language model using self-supervised learning (i.e., generating the labels automatically from the data) on a huge text corpus, then they fine-tuned it on various tasks. Their model outperformed the state of the art on six text classification tasks by a large margin (reducing the error rate by 18–24% in most cases). Moreover, they showed that by fine-tuning the pretrained model on just 100 labeled examples, they could achieve the same performance as a model trained from scratch on 10,000 examples.
- The GPT paper[26] by Alec Radford and other OpenAI researchers also demonstrated the effectiveness of unsupervised pretraining, but this time using a

24 Matthew Peters et al., "Deep Contextualized Word Representations," *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2018): 2227–2237.

25 Jeremy Howard and Sebastian Ruder, "Universal Language Model Fine-Tuning for Text Classification," *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics* 1 (2018): 328–339.

26 Alec Radford et al., "Improving Language Understanding by Generative Pre-Training" (2018).

Transformer-like architecture. The authors pretrained a large but fairly simple architecture composed of a stack of 12 Transformer modules (using only Masked Multi-Head Attention layers) on a large dataset, once again trained using self-supervised learning. Then they fine-tuned it on various language tasks, using only minor adaptations for each task. The tasks were quite diverse: they included text classification, *entailment* (whether sentence A entails sentence B),[27] similarity (e.g., "Nice weather today" is very similar to "It is sunny"), and question answering (given a few paragraphs of text giving some context, the model must answer some multiple-choice questions). Just a few months later, in February 2019, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the GPT-2 paper,[28] which proposed a very similar architecture, but larger still (with over 1.5 billion parameters!) and they showed that it could achieve good performance on many tasks without any fine-tuning. This is called *zero-shot learning* (ZSL). A smaller version of the GPT-2 model (with "just" 117 million parameters) is available at *https://github.com/openai/gpt-2*, along with its pretrained weights.

- The BERT paper[29] by Jacob Devlin and other Google researchers also demonstrates the effectiveness of self-supervised pretraining on a large corpus, using a similar architecture to GPT but non-masked Multi-Head Attention layers (like in the Transformer's encoder). This means that the model is naturally bidirectional; hence the B in BERT (*Bidirectional Encoder Representations from Transformers*). Most importantly, the authors proposed two pretraining tasks that explain most of the model's strength:

  Masked language model (MLM)
  : Each word in a sentence has a 15% probability of being masked, and the model is trained to predict the masked words. For example, if the original sentence is "She had fun at the birthday party," then the model may be given the sentence "She <mask> fun at the <mask> party" and it must predict the words "had" and "birthday" (the other outputs will be ignored). To be more precise, each selected word has an 80% chance of being masked, a 10% chance of being replaced by a random word (to reduce the discrepancy between pretraining and fine-tuning, since the model will not see <mask> tokens during fine-tuning), and a 10% chance of being left alone (to bias the model toward the correct answer).

27 For example, the sentence "Jane had a lot of fun at her friend's birthday party" entails "Jane enjoyed the party," but it is contradicted by "Everyone hated the party" and it is unrelated to "The Earth is flat."

28 Alec Radford et al., "Language Models Are Unsupervised Multitask Learners" (2019).

29 Jacob Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that "The dog sleeps" and "It snores loudly" are consecutive sentences, while "The dog sleeps" and "The Earth orbits the Sun" are not consecutive. This is a challenging task, and it significantly improves the performance of the model when it is fine-tuned on tasks such as question answering or entailment.

As you can see, the main innovations in 2018 and 2019 have been better subword tokenization, shifting from LSTMs to Transformers, and pretraining universal language models using self-supervised learning, then fine-tuning them with very few architectural changes (or none at all). Things are moving fast; no one can say what architectures will prevail next year. Today, it's clearly Transformers, but tomorrow it might be CNNs (e.g., check out the 2018 paper[30] by Maha Elbayad et al., where the researchers use masked 2D convolutional layers for sequence-to-sequence tasks). Or it might even be RNNs, if they make a surprise comeback (e.g., check out the 2018 paper[31] by Shuai Li et al. that shows that by making neurons independent of each other in a given RNN layer, it is possible to train much deeper RNNs capable of learning much longer sequences).

In the next chapter we will discuss how to learn deep representations in an unsupervised way using autoencoders, and we will use generative adversarial networks (GANs) to produce images and more!

Exercises

1. What are the pros and cons of using a stateful RNN versus a stateless RNN?
2. Why do people use Encoder–Decoder RNNs rather than plain sequence-to-sequence RNNs for automatic translation?
3. How can you deal with variable-length input sequences? What about variable-length output sequences?
4. What is beam search and why would you use it? What tool can you use to implement it?
5. What is an attention mechanism? How does it help?

30 Maha Elbayad et al., "Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction," arXiv preprint arXiv:1808.03867 (2018).

31 Shuai Li et al., "Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018): 5457–5466.

6. What is the most important layer in the Transformer architecture? What is its purpose?
7. When would you need to use sampled softmax?
8. *Embedded Reber grammars* were used by Hochreiter and Schmidhuber in their paper about LSTMs. They are artificial grammars that produce strings such as "BPBTSXXVPSEPE." Check out Jenny Orr's nice introduction to this topic. Choose a particular embedded Reber grammar (such as the one represented on Jenny Orr's page), then train an RNN to identify whether a string respects that grammar or not. You will first need to write a function capable of generating a training batch containing about 50% strings that respect the grammar, and 50% that don't.
9. Train an Encoder–Decoder model that can convert a date string from one format to another (e.g., from "April 22, 2019" to "2019-04-22").
10. Go through TensorFlow's Neural Machine Translation with Attention tutorial.
11. Use one of the recent language models (e.g., BERT) to generate more convincing Shakespearean text.

Solutions to these exercises are available in Appendix A.

CHAPTER 17

Representation Learning and Generative Learning Using Autoencoders and GANs

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called *latent representations* or *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see Chapter 8), especially for visualization purposes. Autoencoders also act as feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in Chapter 11). Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. However, the generated images are usually fuzzy and not entirely realistic.

In contrast, faces generated by generative adversarial networks (GANs) are now so convincing that it is hard to believe that the people they represent do not exist. You can judge so for yourself by visiting *https://thispersondoesnotexist.com/*, a website that shows faces generated by a recent GAN architecture called *StyleGAN* (you can also check out *https://thisrentaldoesnotexist.com/* to see some generated Airbnb bedrooms). GANs are now widely used for super resolution (increasing the resolution of an image), colorization, powerful image editing (e.g., replacing photo bombers with realistic background), turning a simple sketch into a photorealistic image, predicting the next frames in a video, augmenting a dataset (to train other models), generating other types of data (such as text, audio, and time series), identifying the weaknesses in other models and strengthening them, and more.

Autoencoders and GANs are both unsupervised, they both learn dense representations, they can both be used as generative models, and they have many similar applications. However, they work very differently:

- Autoencoders simply learn to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the latent representations, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In short, the codings are byproducts of the autoencoder learning the identity function under some constraints.
- GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. This architecture is very original in Deep Learning in that the generator and the discriminator compete against each other during training: the generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake. *Adversarial training* (training competing neural networks) is widely considered as one of the most important ideas in recent years. In 2016, Yann LeCun even said that it was "the most interesting idea in the last 10 years in Machine Learning."

In this chapter we will start by exploring in more depth how autoencoders work and how to use them for dimensionality reduction, feature extraction, unsupervised pretraining, or as generative models. This will naturally lead us to GANs. We will start by building a simple GAN to generate fake images, but we will see that training is often quite difficult. We will discuss the main difficulties you will encounter with adversarial training, as well as some of the main techniques to work around these difficulties. Let's start with autoencoders!

Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you will notice that it is just the list of even numbers from 50 down to 14. Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to remember the pattern (i.e., decreasing even numbers) and the starting and ending numbers (i.e., 50 and 14). Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. The fact that it is hard to memorize long sequences is what makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was famously studied by William Chase and Herbert Simon in the early 1970s.[1] They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game. Noticing patterns helps them store information efficiently.

Just like the chess players in this memory experiment, an autoencoder looks at the inputs, converts them to an efficient latent representation, and then spits out something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or *recognition network*) that converts the inputs to a latent representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs (see Figure 17-1).

1 William G. Chase and Herbert A. Simon, "Perception in Chess," *Cognitive Psychology* 4, no. 1 (1973): 55–81.

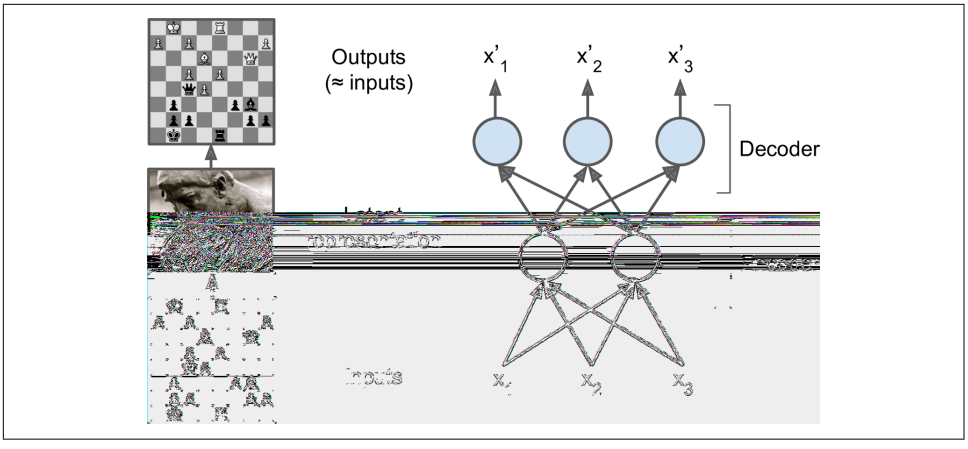


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

As you can see, an autoencoder typically has the same architecture as a Multi-Layer Perceptron (MLP; see Chapter 10), except that the number of neurons in the output layer must be equal to the number of inputs. In this example, there is just one hidden layer composed of two neurons (the encoder), and one output layer composed of three neurons (the decoder). The outputs are often called the *reconstructions* because the autoencoder tries to reconstruct the inputs, and the cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Because the internal representation has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing Principal Component Analysis (PCA; see Chapter 8).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

```
from tensorflow import keras

encoder = keras.models.Sequential([keras.layers.Dense(2, input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3, input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])

autoencoder.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=0.1))
```

This code is really not very different from all the MLPs we built in past chapters, but there are a few things to note:

- We organized the autoencoder into two subcomponents: the encoder and the decoder. Both are regular `Sequential` models with a single `Dense` layer each, and the autoencoder is a `Sequential` model containing the encoder followed by the decoder (remember that a model can be used as a layer in another model).
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform simple PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE. We will see more complex autoencoders shortly.

Now let's train the model on a simple generated 3D dataset and use it to encode that same dataset (i.e., project it to 2D):

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Note that the same dataset, `X_train`, is used as both the inputs and the targets. Figure 17-2 shows the original 3D dataset (on the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, on the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

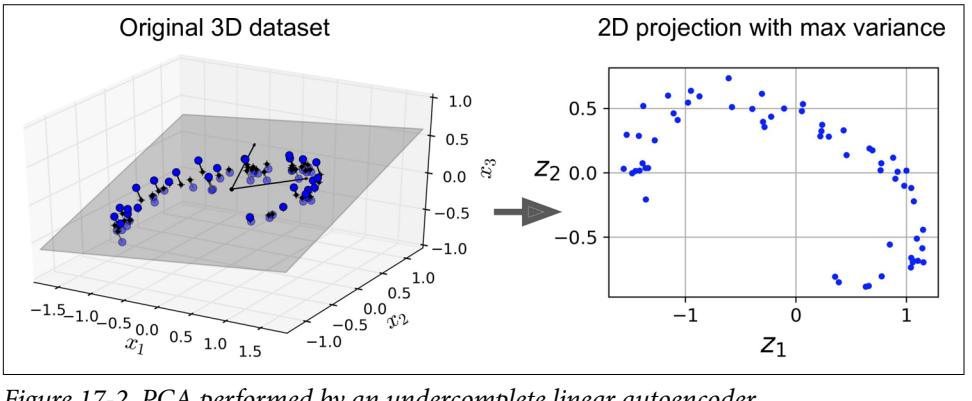


Figure 17-2. PCA performed by an undercomplete linear autoencoder

You can think of autoencoders as a form of self-supervised learning (i.e., using a supervised learning technique with automatically generated labels, in this case simply equal to the inputs).

Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. That said, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances).

The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for MNIST (introduced in Chapter 3) may have 784 inputs, followed by a hidden layer with 100 neurons, then a central hidden layer of 30 neurons, then another hidden layer with 100 neurons, and an output layer with 784 neurons. This stacked autoencoder is represented in Figure 17-3.

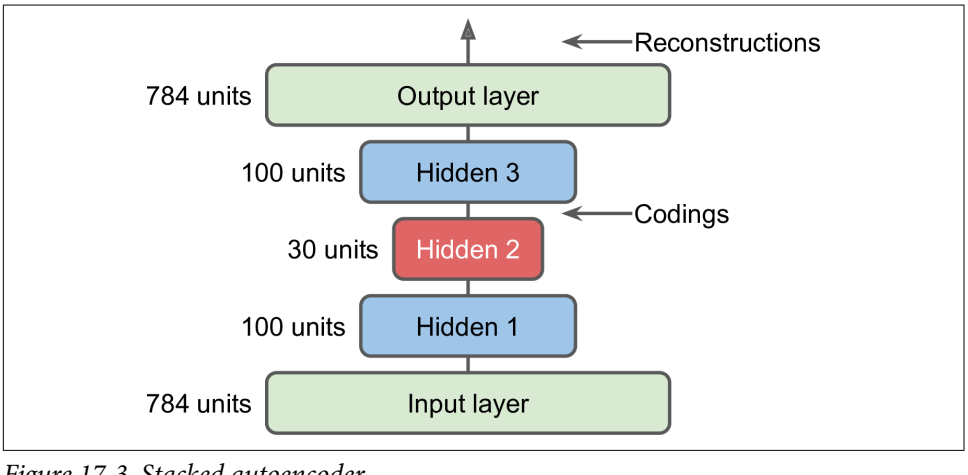


Figure 17-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using Keras

You can implement a stacked autoencoder very much like a regular deep MLP. In particular, the same techniques we used in Chapter 11 for training deep nets can be applied. For example, the following code builds a stacked autoencoder for Fashion

MNIST (loaded and normalized as in Chapter 10), using the SELU activation function:

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
```

Let's go through this code:

- Just like earlier, we split the autoencoder model into two submodels: the encoder and the decoder.
- The encoder takes 28 × 28–pixel grayscale images, flattens them so that each image is represented as a vector of size 784, then processes these vectors through two `Dense` layers of diminishing sizes (100 units then 30 units), both using the SELU activation function (you may want to add LeCun normal initialization as well, but the network is not very deep so it won't make a big difference). For each input image, the encoder outputs a vector of size 30.
- The decoder takes codings of size 30 (output by the encoder) and processes them through two `Dense` layers of increasing sizes (100 units then 784 units), and it reshapes the final vectors into 28 × 28 arrays so the decoder's outputs have the same shape as the encoder's inputs.
- When compiling the stacked autoencoder, we use the binary cross-entropy loss instead of the mean squared error. We are treating the reconstruction task as a multilabel binary classification problem: each pixel intensity represents the probability that the pixel should be black. Framing it this way (rather than as a regression problem) tends to make the model converge faster.[2]
- Finally, we train the model using `X_train` as both the inputs and the targets (and similarly, we use `X_valid` as both the validation inputs and targets).

2 You might be tempted to use the accuracy metric, but it would not work properly, since this metric expects the labels to be either 0 or 1 for each pixel. You can easily work around this problem by creating a custom metric that computes the accuracy after rounding the targets and predictions to 0 or 1.

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions:

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

Figure 17-4 shows the resulting images.

Figure 17-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder deeper, or make the codings larger. But if we make the network too powerful, it will manage to make perfect reconstructions without having learned any useful patterns in the data. For now, let's go with this model.

Visualizing the Fashion MNIST Dataset

Now that we have trained a stacked autoencoder, we can use it to reduce the dataset's dimensionality. For visualization, this does not give great results compared to other dimensionality reduction algorithms (such as those we discussed in Chapter 8), but one big advantage of autoencoders is that they can handle large datasets, with many instances and many features. So one strategy is to use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality

reduction algorithm for visualization. Let's use this strategy to visualize Fashion MNIST. First, we use the encoder from our stacked autoencoder to reduce the dimensionality down to 30, then we use Scikit-Learn's implementation of the t-SNE algorithm to reduce the dimensionality down to 2 for visualization:

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Now we can plot the dataset:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

Figure 17-5 shows the resulting scatterplot (beautified a bit by displaying some of the images). The t-SNE algorithm identified several clusters which match the classes reasonably well (each class is represented with a different color).

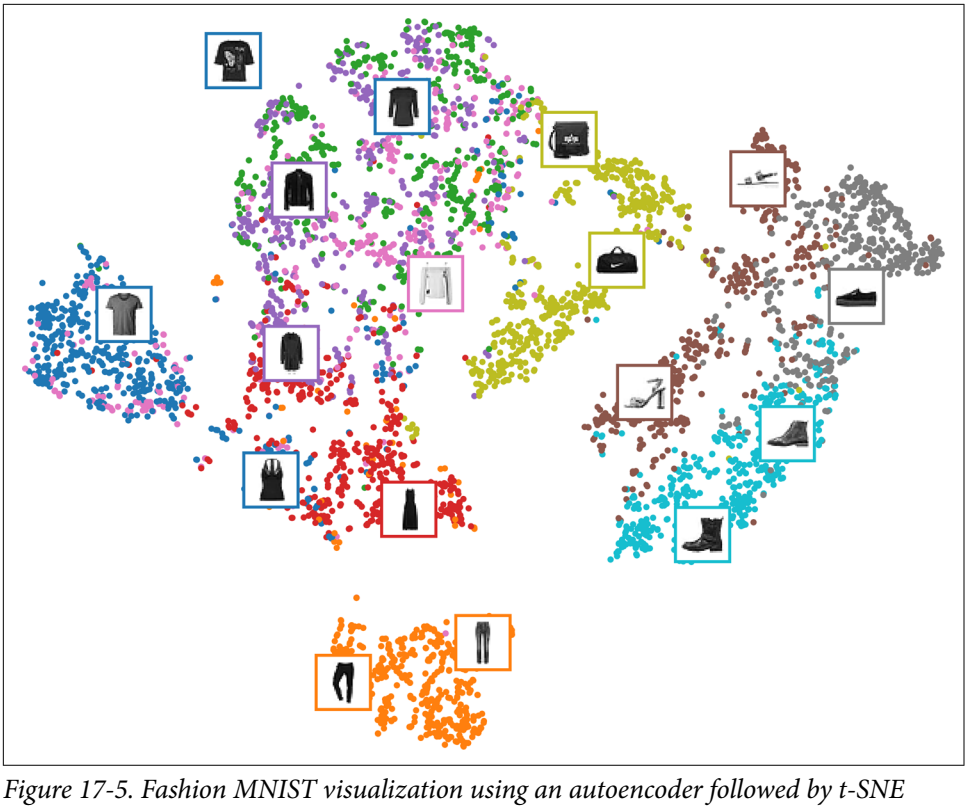


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

So, autoencoders can be used for dimensionality reduction. Another application is for unsupervised pretraining.

Unsupervised Pretraining Using Stacked Autoencoders

As we discussed in Chapter 11, if you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data. For example, Figure 17-6 shows how to use a stacked autoencoder to perform unsupervised pretraining for a classification neural network. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).

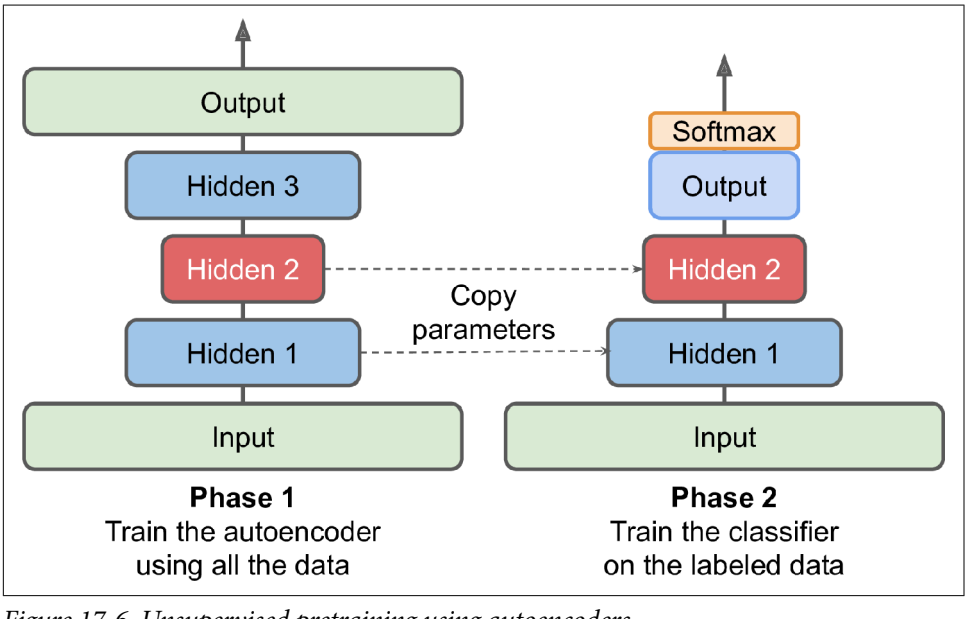


Figure 17-6. Unsupervised pretraining using autoencoders

Having plenty of unlabeled data and little labeled data is common. Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images (e.g., classifying them as cute or not) can usually be done reliably only by humans. Labeling instances is time-consuming and costly, so it's normal to have only a few thousand human-labeled instances.

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network (see the exercises at the end of this chapter for an example).

Next, let's look at a few techniques for training stacked autoencoders.

Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the L^{th} layer (e.g., layer 1 is the first hidden layer, layer $N/2$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined simply as: $\mathbf{W}_{N-L+1} = \mathbf{W}_L^\intercal$ (with $L = 1, 2, \ldots, N/2$).

To tie weights between layers using Keras, let's define a custom layer:

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

This custom layer acts like a regular `Dense` layer, but it uses another `Dense` layer's weights, transposed (setting `transpose_b=True` is equivalent to transposing the second argument, but it's more efficient as it performs the transposition on the fly within the `matmul()` operation). However, it uses its own bias vector. Next, we can build a new stacked autoencoder, much like the previous one, but with the decoder's `Dense` layers tied to the encoder's `Dense` layers:

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])
```

```
tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])
```

This model achieves a very slightly lower reconstruction error than the previous model, with almost half the number of parameters.

Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown in Figure 17-7. This technique is not used as much these days, but you may still run into papers that talk about "greedy layerwise training," so it's good to know what it means.

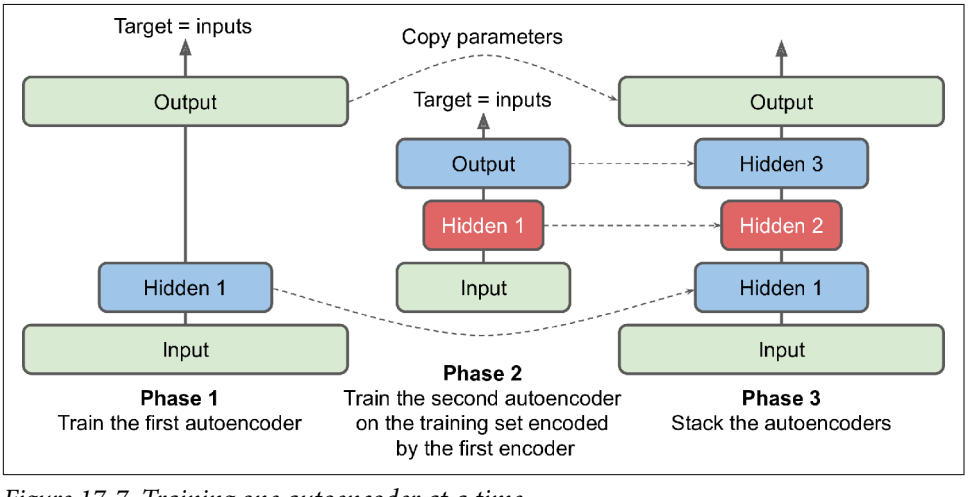


Figure 17-7. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. Then we encode the whole training set using this first autoencoder, and this gives us a new (compressed) training set. We then train a second autoencoder on this new dataset. This is the second phase of training. Finally, we build a big sandwich using all these autoencoders, as shown in Figure 17-7 (i.e., we first stack the hidden layers of each autoencoder, then the output layers in reverse order). This gives us the final stacked autoencoder (see the "Training One Autoencoder at a Time" section in the notebook for an implementation). We could easily train more autoencoders this way, building a very deep stacked autoencoder.

As we discussed earlier, one of the triggers of the current tsunami of interest in Deep Learning was the discovery in 2006 by Geoffrey Hinton et al. that deep neural networks can be pretrained in an unsupervised fashion, using this greedy layerwise approach. They used restricted Boltzmann machines (RBMs; see Appendix E) for this purpose, but in 2007 Yoshua Bengio et al. showed[3] that autoencoders worked just as well. For several years this was the only efficient way to train deep nets, until many of the techniques introduced in Chapter 11 made it possible to just train a deep net in one shot.

Autoencoders are not limited to dense networks: you can also build convolutional autoencoders, or even recurrent autoencoders. Let's look at these now.

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small): as we saw in Chapter 14, convolutional neural networks are far better suited than dense networks to work with images. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.[4] The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps). The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers). Here is a simple convolutional autoencoder for Fashion MNIST:

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                                 activation="selu",
                                 input_shape=[3, 3, 64]),
```

3 Yoshua Bengio et al., "Greedy Layer-Wise Training of Deep Networks," *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006): 153–160.

4 Jonathan Masci et al., "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction," *Proceedings of the 21st International Conference on Artificial Neural Networks* 1 (2011): 52–59.

```
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                                 activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

Recurrent Autoencoders

If you want to build an autoencoder for sequences, such as time series or text (e.g., for unsupervised learning or dimensionality reduction), then recurrent neural networks (see Chapter 15) may be better suited than dense networks. Building a *recurrent autoencoder* is straightforward: the encoder is typically a sequence-to-vector RNN which compresses the input sequence down to a single vector. The decoder is a vector-to-sequence RNN that does the reverse:

```
recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True, input_shape=[None, 28]),
    keras.layers.LSTM(30)
])
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder, recurrent_decoder])
```

This recurrent autoencoder can process sequences of any length, with 28 dimensions per time step. Conveniently, this means it can process Fashion MNIST images by treating each image as a sequence of rows: at each time step, the RNN will process a single row of 28 pixels. Obviously, you could use a recurrent autoencoder for any kind of sequence. Note that we use a `RepeatVector` layer as the first layer of the decoder, to ensure that its input vector gets fed to the decoder at each time step.

OK, let's step back for a second. So far we have seen various kinds of autoencoders (basic, stacked, convolutional, and recurrent), and we have looked at how to train them (either in one shot or layer by layer). We also looked at a couple applications: data visualization and unsupervised pretraining.

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it undercomplete. There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*. Let's look at some of those approaches now.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a 2008 paper,[5] Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a 2010 paper,[6] Vincent et al. introduced *stacked denoising autoencoders*.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout (introduced in Chapter 11). Figure 17-8 shows both options.

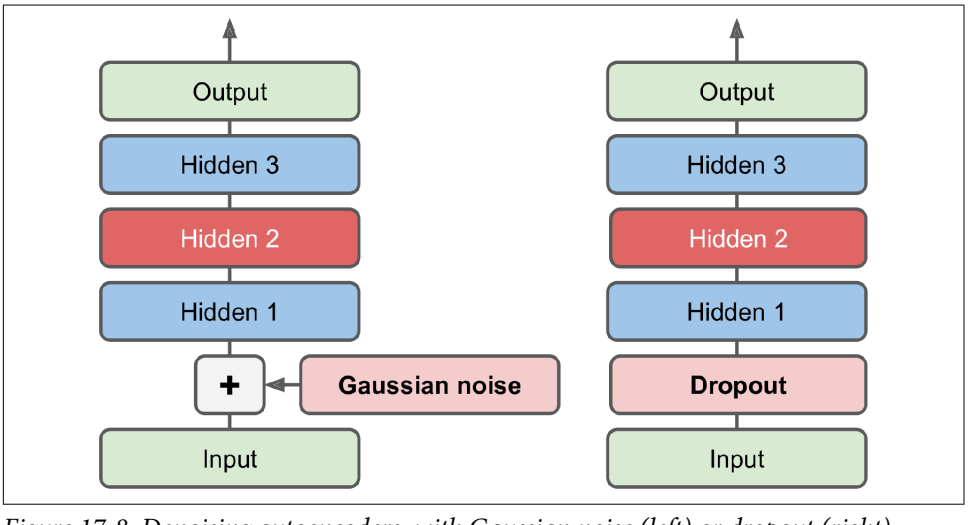


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

The implementation is straightforward: it is a regular stacked autoencoder with an additional `Dropout` layer applied to the encoder's inputs (or you could use a `GaussianNoise` layer instead). Recall that the `Dropout` layer is only active during training (and so is the `GaussianNoise` layer):

5 Pascal Vincent et al., "Extracting and Composing Robust Features with Denoising Autoencoders," *Proceedings of the 25th International Conference on Machine Learning* (2008): 1096–1103.

6 Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," *Journal of Machine Learning Research* 11 (2010): 3371–3408.

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```

Figure 17-9 shows a few noisy images (with half the pixels turned off), and the images reconstructed by the dropout-based denoising autoencoder. Notice how the autoencoder guesses details that are actually not in the input, such as the top of the white shirt (bottom row, fourth image). As you can see, not only can denoising autoencoders be used for data visualization or unsupervised pretraining, like the other autoencoders we've discussed so far, but they can also be used quite simply and efficiently to remove noise from images.

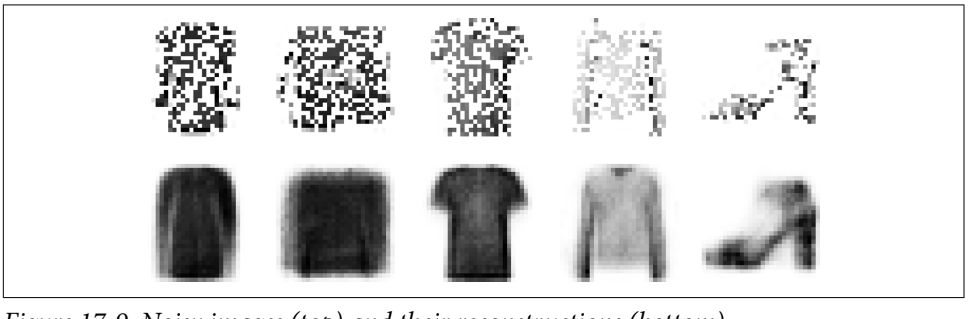


Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Sparse Autoencoders

Another kind of constraint that often leads to good feature extraction is *sparsity*: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer. For example, it may be pushed to have on average only 5% significantly active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

A simple approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer (e.g., with

300 units), and add some ℓ_1 regularization to the coding layer's activations (the decoder is just a regular decoder):

```
sparse_l1_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_l1_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

This `ActivityRegularization` layer just returns its inputs, but as a side effect it adds a training loss equal to the sum of absolute values of its inputs (this layer only has an effect during training). Equivalently, you could remove the `ActivityRegularization` layer and set `activity_regularizer=keras.regularizers.l1(1e-3)` in the previous layer. This penalty will encourage the neural network to produce codings close to 0, but since it will also be penalized if it does not reconstruct the inputs correctly, it will have to output at least a few nonzero values. Using the ℓ_1 norm rather than the ℓ_2 norm will push the neural network to preserve the most important codings while eliminating the ones that are not needed for the input image (rather than just reducing all codings).

Another approach, which often yields better results, is to measure the actual sparsity of the coding layer at each training iteration, and penalize the model when the measured sparsity differs from a target sparsity. We do so by computing the average activation of each neuron in the coding layer, over the whole training batch. The batch size must not be too small, or else the mean will not be accurate.

Once we have the mean activation per neuron, we want to penalize the neurons that are too active, or not active enough, by adding a *sparsity loss* to the cost function. For example, if we measure that a neuron has an average activation of 0.3, but the target sparsity is 0.1, it must be penalized to activate less. One approach could be simply adding the squared error $(0.3 - 0.1)^2$ to the cost function, but in practice a better approach is to use the Kullback–Leibler (KL) divergence (briefly discussed in Chapter 4), which has much stronger gradients than the mean squared error, as you can see in Figure 17-10.

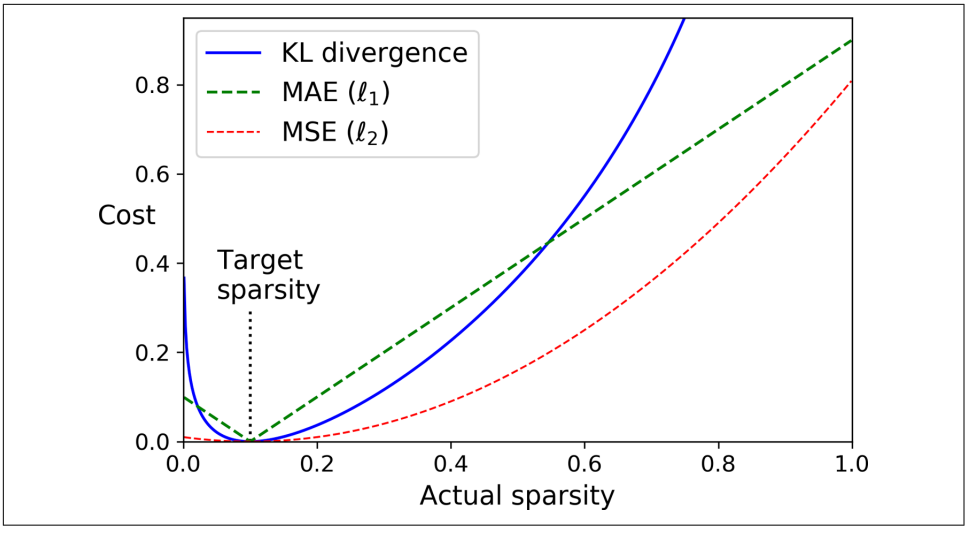


Figure 17-10. Sparsity loss

Given two discrete probability distributions P and Q, the KL divergence between these distributions, noted $D_{\text{KL}}(P \parallel Q)$, can be computed using Equation 17-1.

Equation 17-1. Kullback–Leibler divergence

$$D_{\text{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

In our case, we want to measure the divergence between the target probability p that a neuron in the coding layer will activate and the actual probability q (i.e., the mean activation over the training batch). So the KL divergence simplifies to Equation 17-2.

Equation 17-2. KL divergence between the target sparsity p and the actual sparsity q

$$D_{\text{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Once we have computed the sparsity loss for each neuron in the coding layer, we sum up these losses and add the result to the cost function. In order to control the relative importance of the sparsity loss and the reconstruction loss, we can multiply the sparsity loss by a sparsity weight hyperparameter. If this weight is too high, the model will stick closely to the target sparsity, but it may not reconstruct the inputs properly, making the model useless. Conversely, if it is too low, the model will mostly ignore the sparsity objective and will not learn any interesting features.

We now have all we need to implement a sparse autoencoder based on the KL divergence. First, let's create a custom regularizer to apply KL divergence regularization:

```
K = keras.backend
kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):
    def __init__(self, weight, target=0.1):
        self.weight = weight
        self.target = target
    def __call__(self, inputs):
        mean_activities = K.mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

Now we can build the sparse autoencoder, using the `KLDivergenceRegularizer` for the coding layer's activations:

```
kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)
sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=kld_reg)
])
sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

After training this sparse autoencoder on Fashion MNIST, the activations of the neurons in the coding layer are mostly close to 0 (about 70% of all activations are lower than 0.1), and all neurons have a mean activation around 0.1 (about 90% of all neurons have a mean activation between 0.1 and 0.2), as shown in Figure 17-11.

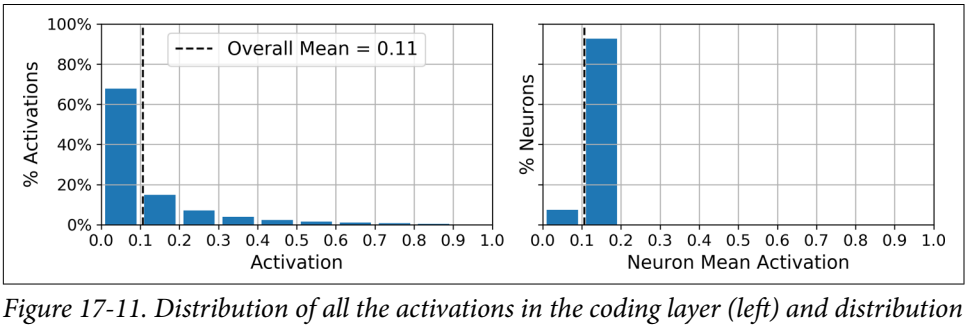


Figure 17-11. Distribution of all the activations in the coding layer (left) and distribution of the mean activation per neuron (right)

Variational Autoencoders

Another important category of autoencoders was introduced in 2013 by Diederik Kingma and Max Welling and quickly became one of the most popular types of autoencoders: *variational autoencoders*.[7]

They are quite different from all the autoencoders we have discussed so far, in these particular ways:

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.

Both these properties make them rather similar to RBMs, but they are easier to train, and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a "thermal equilibrium" before you can sample a new instance). Indeed, as their name suggests, variational autoencoders perform variational Bayesian inference (introduced in Chapter 9), which is an efficient way to perform approximate Bayesian inference.

Let's take a look at how they work. Figure 17-12 (left) shows a variational autoencoder. You can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a *mean coding* μ and a standard deviation σ. The actual coding is then sampled randomly from a Gaussian distribution with mean μ and standard deviation σ. After that the decoder decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces μ and σ, then a coding is sampled randomly (notice that it is not exactly located at μ), and finally this coding is decoded; the final output resembles the training instance.

7 Diederik Kingma and Max Welling, "Auto-Encoding Variational Bayes," arXiv preprint arXiv:1312.6114 (2013).

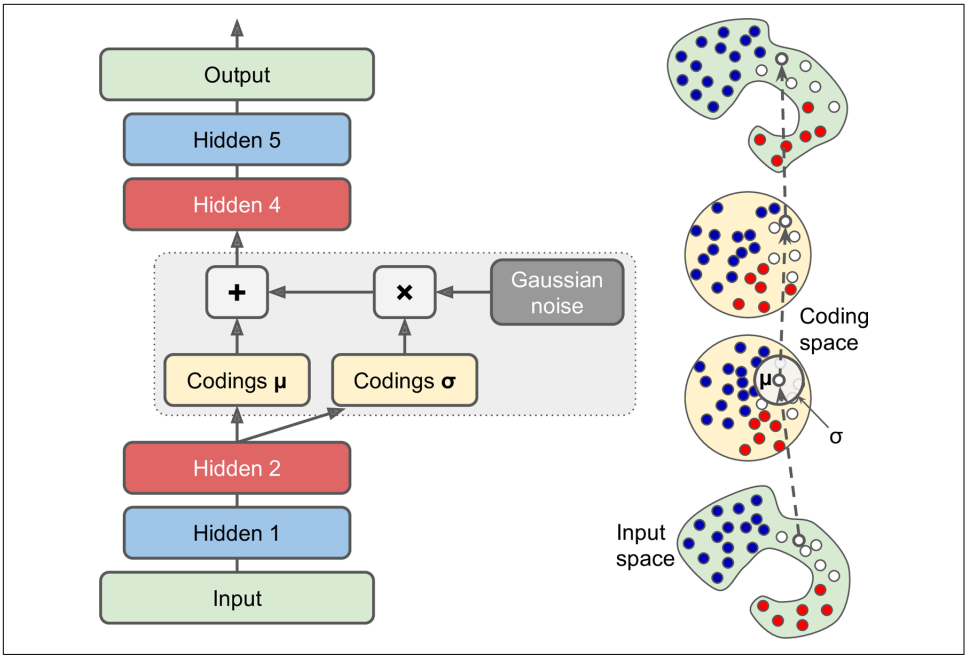


Figure 17-12. Variational autoencoder (left) and an instance going through it (right)

As you can see in the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution:[8] during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to end up looking like a cloud of Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

Now, let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs (we can use cross entropy for this, as discussed earlier). The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the target distribution (i.e., the Gaussian distribution) and the actual distribution of the codings. The math is a bit more complex than with the sparse autoencoder, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer (thus pushing the autoencoder to learn useful features). Luckily, the

8 Variational autoencoders are actually more general; the codings are not limited to Gaussian distributions.

equations simplify, so the latent loss can be computed quite simply using Equation 17-3:[9]

Equation 17-3. Variational autoencoder's latent loss

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \log\left(\sigma_i^2\right) - \sigma_i^2 - \mu_i^2$$

In this equation, $\mathcal{L}$ is the latent loss, n is the codings' dimensionality, and μ_i and σ_i are the mean and standard deviation of the i^{th} component of the codings. The vectors μ and σ (which contain all the μ_i and σ_i) are output by the encoder, as shown in Figure 17-12 (left).

A common tweak to the variational autoencoder's architecture is to make the encoder output $\gamma = \log(\sigma^2)$ rather than σ. The latent loss can then be computed as shown in Equation 17-4. This approach is more numerically stable and speeds up training.

Equation 17-4. Variational autoencoder's latent loss, rewritten using $\boldsymbol{\gamma} = \log(\boldsymbol{\sigma}^2)$

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + \gamma_i - \exp\left(\gamma_i\right) - \mu_i^2$$

Let's start building a variational autoencoder for Fashion MNIST (as shown in Figure 17-12, but using the γ tweak). First, we will need a custom layer to sample the codings, given μ and γ:

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) * K.exp(log_var / 2) + mean
```

This `Sampling` layer takes two inputs: `mean` (μ) and `log_var` (γ). It uses the function `K.random_normal()` to sample a random vector (of the same shape as γ) from the Normal distribution, with mean 0 and standard deviation 1. Then it multiplies it by exp(γ / 2) (which is equal to σ, as you can verify), and finally it adds μ and returns the result. This samples a codings vector from the Normal distribution with mean μ and standard deviation σ.

Next, we can create the encoder, using the Functional API because the model is not entirely sequential:

9 For more mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's great tutorial (2016).

```
codings_size = 10

inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z)  # μ
codings_log_var = keras.layers.Dense(codings_size)(z)  # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Note that the `Dense` layers that output `codings_mean` (μ) and `codings_log_var` (γ) have the same inputs (i.e., the outputs of the second `Dense` layer). We then pass both `codings_mean` and `codings_log_var` to the `Sampling` layer. Finally, the `variational_encoder` model has three outputs, in case you want to inspect the values of `codings_mean` and `codings_log_var`. The only output we will use is the last one (`codings`). Now let's build the decoder:

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

For this decoder, we could have used the Sequential API instead of the Functional API, since it is really just a simple stack of layers, virtually identical to many of the decoders we have built so far. Finally, let's build the variational autoencoder model:

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs], outputs=[reconstructions])
```

Note that we ignore the first two outputs of the encoder (we only want to feed the codings to the decoder). Lastly, we must add the latent loss and the reconstruction loss:

```
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var) - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

We first apply Equation 17-4 to compute the latent loss for each instance in the batch (we sum over the last axis). Then we compute the mean loss over all the instances in the batch, and we divide the result by 784 to ensure it has the appropriate scale compared to the reconstruction loss. Indeed, the variational autoencoder's reconstruction loss is supposed to be the sum of the pixel reconstruction errors, but when Keras computes the `"binary_crossentropy"` loss, it computes the mean over all 784 pixels,

rather than the sum. So, the reconstruction loss is 784 times smaller than we need it to be. We could define a custom loss to compute the sum rather than the mean, but it is simpler to divide the latent loss by 784 (the final loss will be 784 times smaller than it should be, but this just means that we should use a larger learning rate).

Note that we use the `RMSprop` optimizer, which works well in this case. And finally we can train the autoencoder!

```
history = variational_ae.fit(X_train, X_train, epochs=50, batch_size=128,
                             validation_data=[X_valid, X_valid])
```

Generating Fashion MNIST Images

Now let's use this variational autoencoder to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution and decode them:

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

Figure 17-13 shows the 12 generated images.

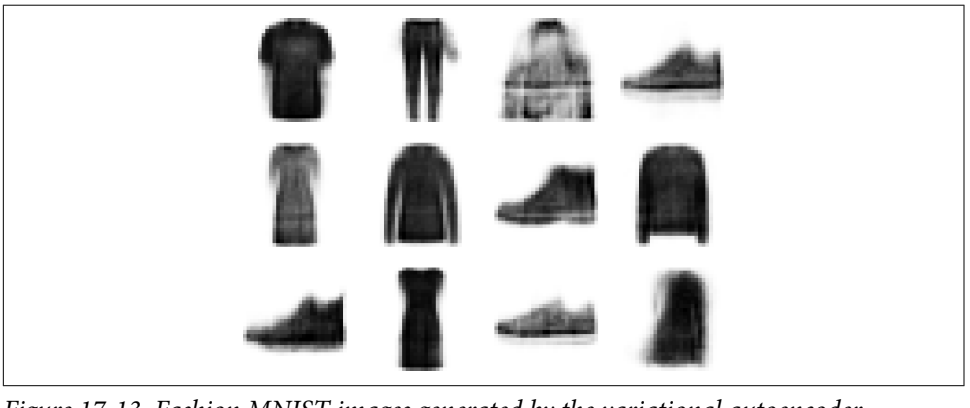


Figure 17-13. Fashion MNIST images generated by the variational autoencoder

The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder—it only had a few minutes to learn! Give it a bit more fine-tuning and training time, and those images should look better.

Variational autoencoders make it possible to perform *semantic interpolation*: instead of interpolating two images at the pixel level (which would look as if the two images were overlaid), we can interpolate at the codings level. We first run both images through the encoder, then we interpolate the two codings we get, and finally we decode the interpolated codings to get the final image. It will look like a regular Fashion MNIST image, but it will be an intermediate between the original images. In the following code example, we take the 12 codings we just generated, we organize them

in a 3 × 4 grid, and we use TensorFlow's `tf.image.resize()` function to resize this grid to 5 × 7. By default, the `resize()` function will perform bilinear interpolation, so every other row and column will contain interpolated codings. We then use the decoder to produce all the images:

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()
```

Figure 17-14 shows the resulting images. The original images are framed, and the rest are the result of semantic interpolation between the nearby images. Notice, for example, how the shoe in the fourth row and fifth column is a nice interpolation between the two shoes located above and below it.

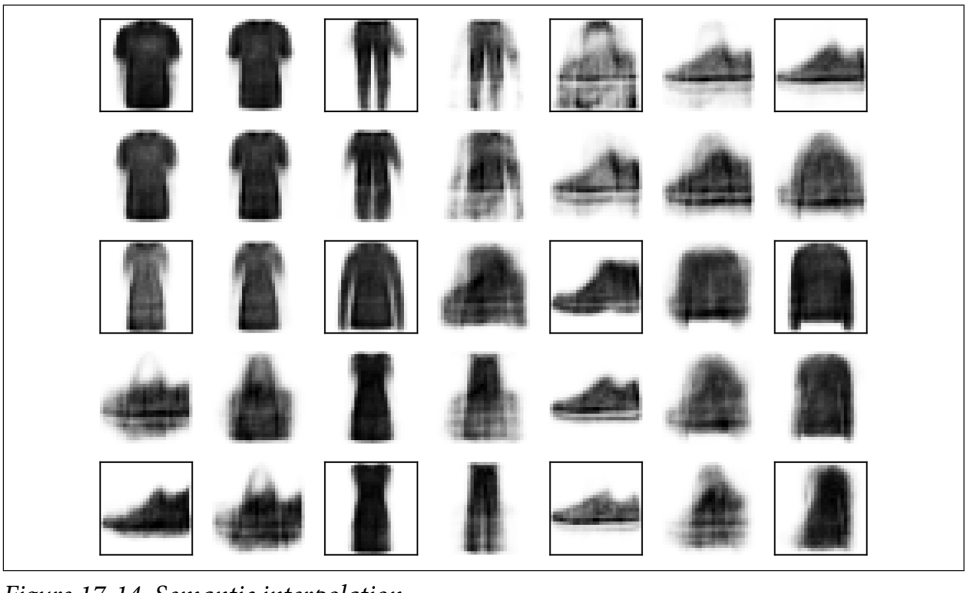


Figure 17-14. Semantic interpolation

For several years, variational autoencoders were quite popular, but GANs eventually took the lead, in particular because they are capable of generating much more realistic and crisp images. So let's turn our attention to GANs.

Generative Adversarial Networks

Generative adversarial networks were proposed in a 2014 paper[10] by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs. Like many great ideas, it seems simple in hindsight: make neural networks compete against each other in the hope that this competition will push them to excel. As shown in Figure 17-15, a GAN is composed of two neural networks:

Generator
: Takes a random distribution as input (typically Gaussian) and outputs some data —typically, an image. You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated. So, as you can see, the generator offers the same functionality as a decoder in a variational autoencoder, and it can be used in the same way to generate new images (just feed it some Gaussian noise, and it outputs a brand-new image). However, it is trained very differently, as we will soon see.

Discriminator
: Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.

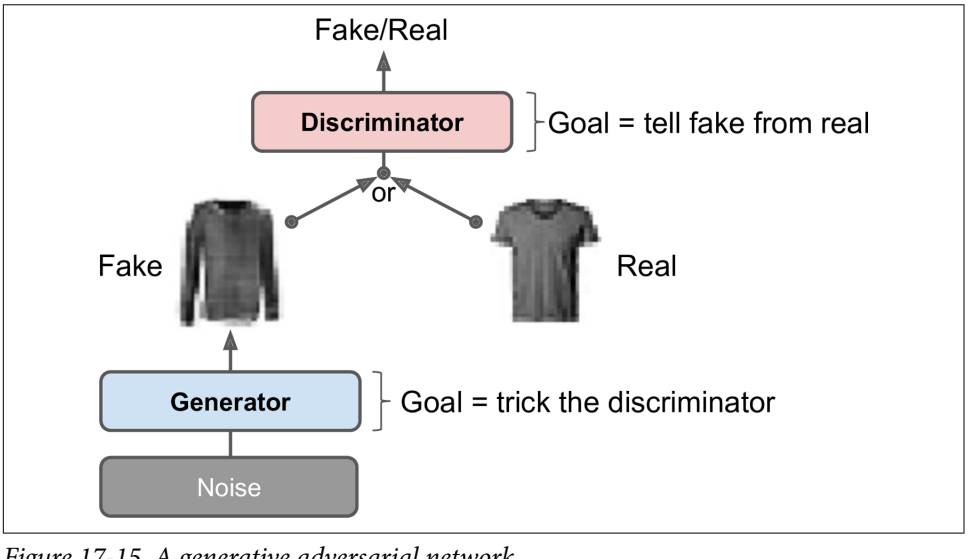


Figure 17-15. A generative adversarial network

10 Ian Goodfellow et al., "Generative Adversarial Nets," *Proceedings of the 27th International Conference on Neural Information Processing Systems* 2 (2014): 2672–2680.

During training, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator. Because the GAN is composed of two networks with different objectives, it cannot be trained like a regular neural network. Each training iteration is divided into two phases:

- In the first phase, we train the discriminator. A batch of real images is sampled from the training set and is completed with an equal number of fake images produced by the generator. The labels are set to 0 for fake images and 1 for real images, and the discriminator is trained on this labeled batch for one step, using the binary cross-entropy loss. Importantly, backpropagation only optimizes the weights of the discriminator during this phase.
- In the second phase, we train the generator. We first use it to produce another batch of fake images, and once again the discriminator is used to tell whether the images are fake or real. This time we do not add real images in the batch, and all the labels are set to 1 (real): in other words, we want the generator to produce images that the discriminator will (wrongly) believe to be real! Crucially, the weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.

The generator never actually sees any real images, yet it gradually learns to produce convincing fake images! All it gets is the gradients flowing back through the discriminator. Fortunately, the better the discriminator gets, the more information about the real images is contained in these secondhand gradients, so the generator can make significant progress.

Let's go ahead and build a simple GAN for Fashion MNIST.

First, we need to build the generator and the discriminator. The generator is similar to an autoencoder's decoder, and the discriminator is a regular binary classifier (it takes an image as input and ends with a `Dense` layer containing a single unit and using the sigmoid activation function). For the second phase of each training iteration, we also need the full GAN model containing the generator followed by the discriminator:

```
codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
```

```
discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Next, we need to compile these models. As the discriminator is a binary classifier, we can naturally use the binary cross-entropy loss. The generator will only be trained through the `gan` model, so we do not need to compile it at all. The `gan` model is also a binary classifier, so it can use the binary cross-entropy loss. Importantly, the discriminator should not be trained during the second phase, so we make it non-trainable before compiling the `gan` model:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

The `trainable` attribute is taken into account by Keras only when compiling a model, so after running this code, the `discriminator` *is* trainable if we call its `fit()` method or its `train_on_batch()` method (which we will be using), while it is *not* trainable when we call these methods on the `gan` model.

Since the training loop is unusual, we cannot use the regular `fit()` method. Instead, we will write a custom training loop. For this, we first need to create a `Dataset` to iterate through the images:

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

We are now ready to write the training loop. Let's wrap it in a `train_gan()` function:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size)
```

As discussed earlier, you can see the two phases at each iteration:

- In phase one we feed Gaussian noise to the generator to produce fake images, and we complete this batch by concatenating an equal number of real images. The targets `y1` are set to 0 for fake images and 1 for real images. Then we train the discriminator on this batch. Note that we set the discriminator's `trainable` attribute to `True`: this is only to get rid of a warning that Keras displays when it notices that `trainable` is now `False` but was `True` when the model was compiled (or vice versa).
- In phase two, we feed the GAN some Gaussian noise. Its generator will start by producing fake images, then the discriminator will try to guess whether these images are fake or real. We want the discriminator to believe that the fake images are real, so the targets `y2` are set to 1. Note that we set the `trainable` attribute to `False`, once again to avoid a warning.

That's it! If you display the generated images (see Figure 17-16), you will see that at the end of the first epoch, they already start to look like (very noisy) Fashion MNIST images.

Unfortunately, the images never really get much better than that, and you may even find epochs where the GAN seems to be forgetting what it learned. Why is that? Well, it turns out that training a GAN can be challenging. Let's see why.

Figure 17-16. Images generated by the GAN after one epoch of training

The Difficulties of Training GANs

During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game. As training advances, the game may end up in a state that game theorists call a *Nash equilibrium*, named after the mathematician John Nash: this is when no player would be better off changing their own strategy, assuming the other players do not change theirs. For example, a Nash equilibrium is reached when everyone drives on the left side of the road: no driver would be better off being the only one to switch sides. Of course, there is a second possible Nash equilibrium: when everyone drives on the *right* side of the road. Different initial states and dynamics may lead to one equilibrium or the other. In this example, there is a single optimal strategy once an equilibrium is reached (i.e., driving on the same side as everyone else), but a Nash equilibrium can involve multiple competing strategies (e.g., a predator chases its prey, the prey tries to escape, and neither would be better off changing their strategy).

So how does this apply to GANs? Well, the authors of the paper demonstrated that a GAN can only reach a single Nash equilibrium: that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake). This fact is very encouraging: it would seem that you just need to train the GAN for long enough, and it will eventually reach this equilibrium, giving you a perfect generator. Unfortunately, it's not that simple: nothing guarantees that the equilibrium will ever be reached.

The biggest difficulty is called *mode collapse*: this is when the generator's outputs gradually become less diverse. How can this happen? Suppose that the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable. Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities. And since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them.

These problems have kept researchers very busy since 2014: many papers were published on this topic, some proposing new cost functions[11] (though a 2018 paper[12] by Google researchers questions their efficiency) or techniques to stabilize training or to avoid the mode collapse issue. For example, a popular technique called *experience replay* consists in storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator). This reduces the chances that the discriminator will overfit the latest generator's outputs. Another common technique is called *mini-batch discrimination*: it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity. This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse. Other papers simply propose specific architectures that happen to perform well.

In short, this is still a very active field of research, and the dynamics of GANs are still not perfectly understood. But the good news is that great progress has been made, and some of the results are truly astounding! So let's look at some of the most successful architectures, starting with deep convolutional GANs, which were the state of the art just a few years ago. Then we will look at two more recent (and more complex) architectures.

11 For a nice comparison of the main GAN losses, check out this great GitHub project by Hwalsuk Lee.

12 Mario Lucic et al., "Are GANs Created Equal? A Large-Scale Study," *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018): 698–707.

Deep Convolutional GANs

The original GAN paper in 2014 experimented with convolutional layers, but only tried to generate small images. Soon after, many researchers tried to build GANs based on deeper convolutional nets for larger images. This proved to be tricky, as training was very unstable, but Alec Radford et al. finally succeeded in late 2015, after experimenting with many different architectures and hyperparameters. They called their architecture *deep convolutional GANs* (DCGANs).[13] Here are the main guidelines they proposed for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use Batch Normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.
- Use leaky ReLU activation in the discriminator for all layers.

These guidelines will work in many cases, but not always, so you may still need to experiment with different hyperparameters (in fact, just changing the random seed and training the same model again will sometimes work). For example, here is a small DCGAN that works reasonably well with Fashion MNIST:

13 Alec Radford et al., "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks," arXiv preprint arXiv:1511.06434 (2015).

```
codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same",
                                 activation="tanh")
])
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2),
                        input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

The generator takes codings of size 100, and it projects them to 6272 dimensions (7 * 7 * 128), and reshapes the result to get a $7 \times 7 \times 128$ tensor. This tensor is batch normalized and fed to a transposed convolutional layer with a stride of 2, which upsamples it from 7×7 to 14×14 and reduces its depth from 128 to 64. The result is batch normalized again and fed to another transposed convolutional layer with a stride of 2, which upsamples it from 14×14 to 28×28 and reduces the depth from 64 to 1. This layer uses the tanh activation function, so the outputs will range from –1 to 1. For this reason, before training the GAN, we need to rescale the training set to that same range. We also need to reshape it to add the channel dimension:

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1. # reshape and rescale
```

The discriminator looks much like a regular CNN for binary classification, except instead of using max pooling layers to downsample the image, we use strided convolutions (`strides=2`). Also note that we use the leaky ReLU activation function.

Overall, we respected the DCGAN guidelines, except we replaced the `BatchNormalization` layers in the discriminator with `Dropout` layers (otherwise training was unstable in this case) and we replaced ReLU with SELU in the generator. Feel free to tweak this architecture: you will see how sensitive it is to the hyperparameters (especially the relative learning rates of the two networks).

Lastly, to build the dataset, then compile and train this model, we use the exact same code as earlier. After 50 epochs of training, the generator produces images like those

shown in Figure 17-17. It's still not perfect, but many of these images are pretty convincing.

Figure 17-17. Images generated by the DCGAN after 50 epochs of training

If you scale up this architecture and train it on a large dataset of faces, you can get fairly realistic images. In fact, DCGANs can learn quite meaningful latent representations, as you can see in Figure 17-18: many images were generated, and nine of them were picked manually (top left), including three representing men with glasses, three men without glasses, and three women without glasses. For each of these categories, the codings that were used to generate the images were averaged, and an image was generated based on the resulting mean codings (lower left). In short, each of the three lower-left images represents the mean of the three images located above it. But this is not a simple mean computed at the pixel level (this would result in three overlapping faces), it is a mean computed in the latent space, so the images still look like normal faces. Amazingly, if you compute men with glasses, minus men without glasses, plus women without glasses—where each term corresponds to one of the mean codings—and you generate the image that corresponds to this coding, you get the image at the center of the 3 × 3 grid of faces on the right: a woman with glasses! The eight other images around it were generated based on the same vector plus a bit of noise, to illustrate the semantic interpolation capabilities of DCGANs. Being able to do arithmetic on faces feels like science fiction!

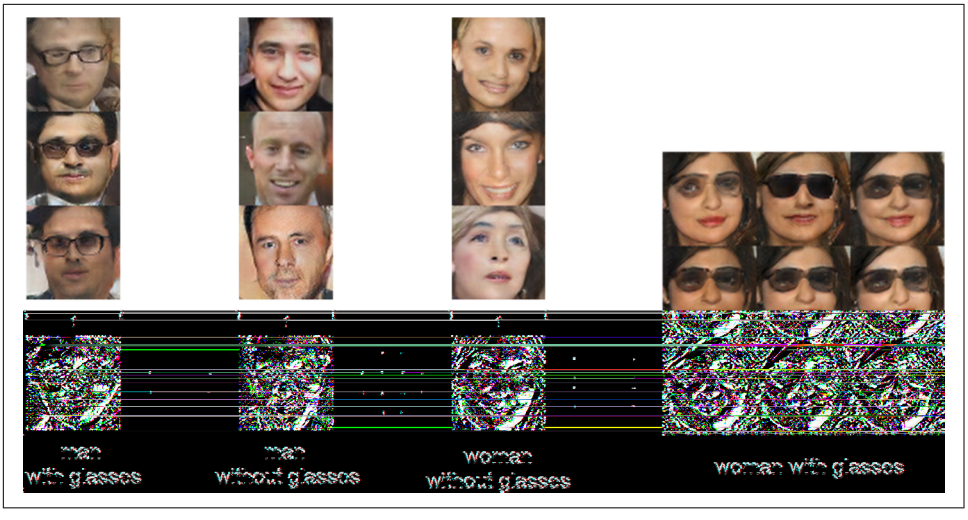


Figure 17-18. Vector arithmetic for visual concepts (part of figure 7 from the DCGAN paper)[14]

If you add each image's class as an extra input to both the generator and the discriminator, they will both learn what each class looks like, and thus you will be able to control the class of each image produced by the generator. This is called a *conditional GAN*[15] (CGAN).

DCGANs aren't perfect, though. For example, when you try to generate very large images using DCGANs, you often end up with locally convincing features but overall inconsistencies (such as shirts with one sleeve much longer than the other). How can you fix this?

Progressive Growing of GANs

An important technique was proposed in a 2018 paper[16] by Nvidia researchers Tero Karras et al.: they suggested generating small images at the beginning of training, then gradually adding convolutional layers to both the generator and the discriminator to produce larger and larger images (4×4, 8×8, 16×16, …, 512×512, $1{,}024 \times 1{,}024$). This approach resembles greedy layer-wise training of stacked autoencoders.

14 Reproduced with the kind authorization of the authors.

15 Mehdi Mirza and Simon Osindero, "Conditional Generative Adversarial Nets," arXiv preprint arXiv: 1411.1784 (2014).

16 Tero Karras et al., "Progressive Growing of GANs for Improved Quality, Stability, and Variation," *Proceedings of the International Conference on Learning Representations* (2018).

The extra layers get added at the end of the generator and at the beginning of the discriminator, and previously trained layers remain trainable.

For example, when growing the generator's outputs from 4 × 4 to 8 × 8 (see Figure 17-19), an upsampling layer (using nearest neighbor filtering) is added to the existing convolutional layer, so it outputs 8 × 8 feature maps, which are then fed to the new convolutional layer (which uses `"same"` padding and strides of 1, so its outputs are also 8 × 8). This new layer is followed by a new output convolutional layer: this is a regular convolutional layer with kernel size 1 that projects the outputs down to the desired number of color channels (e.g., 3). To avoid breaking the trained weights of the first convolutional layer when the new convolutional layer is added, the final output is a weighted sum of the original output layer (which now outputs 8 × 8 feature maps) and the new output layer. The weight of the new outputs is α, while the weight of the original outputs is $1 - \alpha$, and α is slowly increased from 0 to 1. In other words, the new convolutional layers (represented with dashed lines in Figure 17-19) are gradually faded in, while the original output layer is gradually faded out. A similar fade-in/fade-out technique is used when a new convolutional layer is added to the discriminator (followed by an average pooling layer for downsampling).

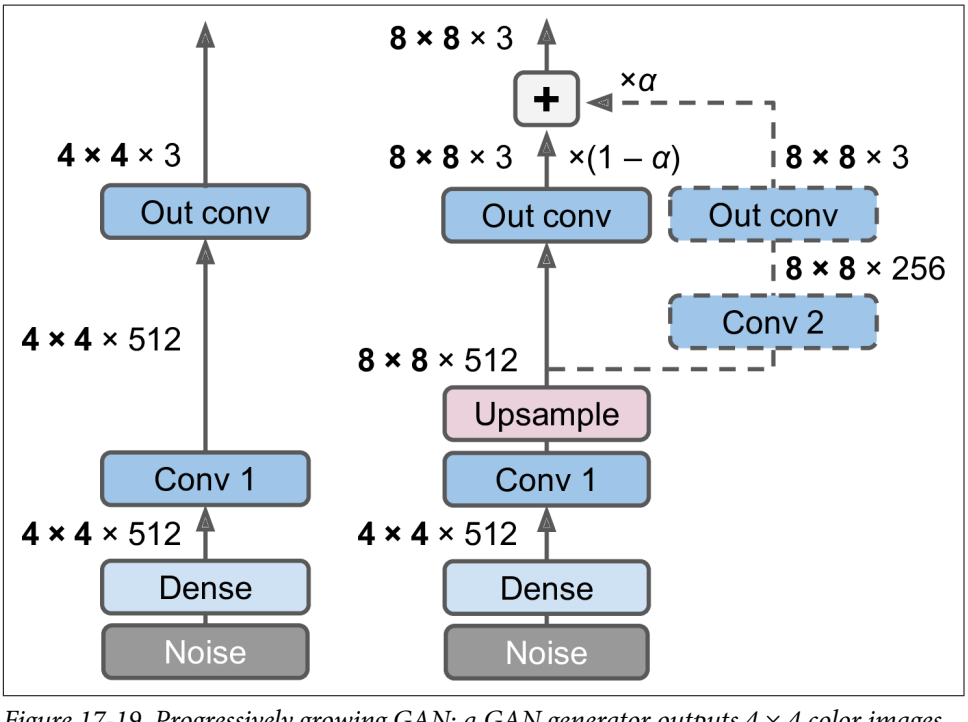


Figure 17-19. Progressively growing GAN: a GAN generator outputs 4 × 4 color images (left); we extend it to output 8 × 8 images (right)

The paper also introduced several other techniques aimed at increasing the diversity of the outputs (to avoid mode collapse) and making training more stable:

Minibatch standard deviation layer
: Added near the end of the discriminator. For each position in the inputs, it computes the standard deviation across all channels and all instances in the batch (`S = tf.math.reduce_std(inputs, axis=[0, -1])`). These standard deviations are then averaged across all points to get a single value (`v = tf.reduce_mean(S)`). Finally, an extra feature map is added to each instance in the batch and filled with the computed value (`tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)`). How does this help? Well, if the generator produces images with little variety, then there will be a small standard deviation across feature maps in the discriminator. Thanks to this layer, the discriminator will have easy access to this statistic, making it less likely to be fooled by a generator that produces too little diversity. This will encourage the generator to produce more diverse outputs, reducing the risk of mode collapse.

Equalized learning rate
: Initializes all weights using a simple Gaussian distribution with mean 0 and standard deviation 1 rather than using He initialization. However, the weights are scaled down at runtime (i.e., every time the layer is executed) by the same factor as in He initialization: they are divided by $\sqrt{2/n_{\text{inputs}}}$, where n_{inputs} is the number of inputs to the layer. The paper demonstrated that this technique significantly improved the GAN's performance when using RMSProp, Adam, or other adaptive gradient optimizers. Indeed, these optimizers normalize the gradient updates by their estimated standard deviation (see Chapter 11), so parameters that have a larger dynamic range[17] will take longer to train, while parameters with a small dynamic range may be updated too quickly, leading to instabilities. By rescaling the weights as part of the model itself rather than just rescaling them upon initialization, this approach ensures that the dynamic range is the same for all parameters, throughout training, so they all learn at the same speed. This both speeds up and stabilizes training.

Pixelwise normalization layer
: Added after each convolutional layer in the generator. It normalizes each activation based on all the activations in the same image and at the same location, but across all channels (dividing by the square root of the mean squared activation). In TensorFlow code, this is `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (the smoothing term `1e-8` is needed to

17 The dynamic range of a variable is the ratio between the highest and the lowest value it may take.

avoid division by zero). This technique avoids explosions in the activations due to excessive competition between the generator and the discriminator.

The combination of all these techniques allowed the authors to generate extremely convincing high-definition images of faces. But what exactly do we call "convincing"? Evaluation is one of the big challenges when working with GANs: although it is possible to automatically evaluate the diversity of the generated images, judging their quality is a much trickier and subjective task. One technique is to use human raters, but this is costly and time-consuming. So the authors proposed to measure the similarity between the local image structure of the generated images and the training images, considering every scale. This idea led them to another groundbreaking innovation: StyleGANs.

StyleGANs

The state of the art in high-resolution image generation was advanced once again by the same Nvidia team in a 2018 paper[18] that introduced the popular StyleGAN architecture. The authors used *style transfer* techniques in the generator to ensure that the generated images have the same local structure as the training images, at every scale, greatly improving the quality of the generated images. The discriminator and the loss function were not modified, only the generator. Let's take a look at the StyleGAN. It is composed of two networks (see Figure 17-20):

Mapping network
: An eight-layer MLP that maps the latent representations z (i.e., the codings) to a vector w. This vector is then sent through multiple *affine transformations* (i.e., `Dense` layers with no activation functions, represented by the "A" boxes in Figure 17-20), which produces multiple vectors. These vectors control the style of the generated image at different levels, from fine-grained texture (e.g., hair color) to high-level features (e.g., adult or child). In short, the mapping network maps the codings to multiple style vectors.

Synthesis network
: Responsible for generating the images. It has a constant learned input (to be clear, this input will be constant *after* training, but *during* training it keeps getting tweaked by backpropagation). It processes this input through multiple convolutional and upsampling layers, as earlier, but there are two twists: first, some noise is added to the input and to all the outputs of the convolutional layers (before the activation function). Second, each noise layer is followed by an *Adaptive Instance Normalization* (AdaIN) layer: it standardizes each feature map independently (by

18 Tero Karras et al., "A Style-Based Generator Architecture for Generative Adversarial Networks," arXiv preprint arXiv:1812.04948 (2018).

subtracting the feature map's mean and dividing by its standard deviation), then it uses the style vector to determine the scale and offset of each feature map (the style vector contains one scale and one bias term for each feature map).

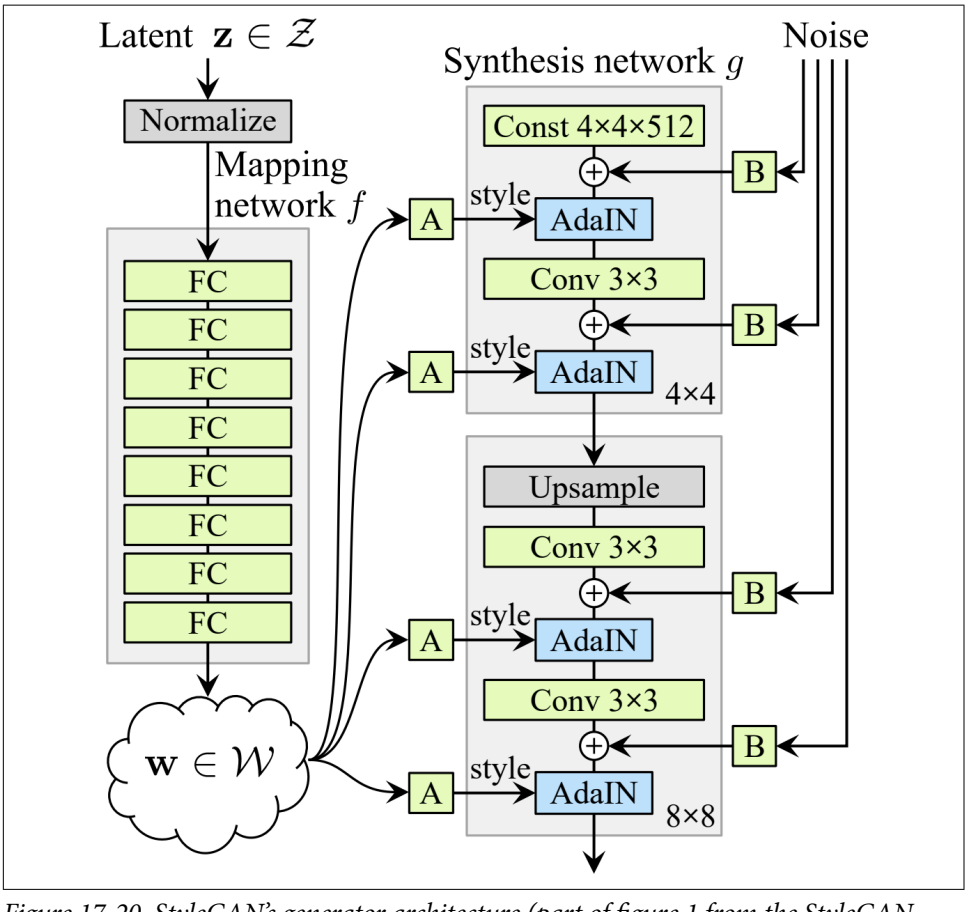


Figure 17-20. StyleGAN's generator architecture (part of figure 1 from the StyleGAN paper)[19]

The idea of adding noise independently from the codings is very important. Some parts of an image are quite random, such as the exact position of each freckle or hair. In earlier GANs, this randomness had to either come from the codings or be some pseudorandom noise produced by the generator itself. If it came from the codings, it meant that the generator had to dedicate a significant portion of the codings' representational power to store noise: this is quite wasteful. Moreover, the noise had to be

19 Reproduced with the kind authorization of the authors.

able to flow through the network and reach the final layers of the generator: this seems like an unnecessary constraint that probably slowed down training. And finally, some visual artifacts may appear because the same noise was used at different levels. If instead the generator tried to produce its own pseudorandom noise, this noise might not look very convincing, leading to more visual artifacts. Plus, part of the generator's weights would be dedicated to generating pseudorandom noise, which again seems wasteful. By adding extra noise inputs, all these issues are avoided; the GAN is able to use the provided noise to add the right amount of stochasticity to each part of the image.

The added noise is different for each level. Each noise input consists of a single feature map full of Gaussian noise, which is broadcast to all feature maps (of the given level) and scaled using learned per-feature scaling factors (this is represented by the "B" boxes in Figure 17-20) before it is added.

Finally, StyleGAN uses a technique called *mixing regularization* (or *style mixing*), where a percentage of the generated images are produced using two different codings. Specifically, the codings c_1 and c_2 are sent through the mapping network, giving two style vectors w_1 and w_2. Then the synthesis network generates an image based on the styles w_1 for the first levels and the styles w_2 for the remaining levels. The cutoff level is picked randomly. This prevents the network from assuming that styles at adjacent levels are correlated, which in turn encourages locality in the GAN, meaning that each style vector only affects a limited number of traits in the generated image.

There is such a wide variety of GANs out there that it would require a whole book to cover them all. Hopefully this introduction has given you the main ideas, and most importantly the desire to learn more. If you're struggling with a mathematical concept, there are probably blog posts out there that will help you understand it better. Then go ahead and implement your own GAN, and do not get discouraged if it has trouble learning at first: unfortunately, this is normal, and it will require quite a bit of patience before it works, but the result is worth it. If you're struggling with an implementation detail, there are plenty of Keras or TensorFlow implementations that you can look at. In fact, if all you want is to get some amazing results quickly, then you can just use a pretrained model (e.g., there are pretrained StyleGAN models available for Keras).

In the next chapter we will move to an entirely different branch of Deep Learning: Deep Reinforcement Learning.

Exercises

1. What are the main tasks that autoencoders are used for?
2. Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?
3. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?
4. What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?
5. How do you tie weights in a stacked autoencoder? What is the point of doing so?
6. What is a generative model? Can you name a type of generative autoencoder?
7. What is a GAN? Can you name a few tasks where GANs can shine?
8. What are the main difficulties when training GANs?
9. Try using a denoising autoencoder to pretrain an image classifier. You can use MNIST (the simplest option), or a more complex image dataset such as CIFAR10 if you want a bigger challenge. Regardless of the dataset you're using, follow these steps:
    - Split the dataset into a training set and a test set. Train a deep denoising autoencoder on the full training set.
    - Check that the images are fairly well reconstructed. Visualize the images that most activate each neuron in the coding layer.
    - Build a classification DNN, reusing the lower layers of the autoencoder. Train it using only 500 images from the training set. Does it perform better with or without pretraining?
10. Train a variational autoencoder on the image dataset of your choice, and use it to generate images. Alternatively, you can try to find an unlabeled dataset that you are interested in and see if you can generate new samples.
11. Train a DCGAN to tackle the image dataset of your choice, and use it to generate images. Add experience replay and see if this helps. Turn it into a conditional GAN where you can control the generated class.

Solutions to these exercises are available in Appendix A.

CHAPTER 18
Reinforcement Learning

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years,[1] particularly in games (e.g., *TD-Gammon*, a Backgammon-playing program) and in machine control, but seldom making the headline news. But a revolution took place in 2013, when researchers from a British startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch,[2] eventually outperforming humans[3] in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.[4] This was the first of a series of amazing feats, culminating in March 2016 with the victory of their system AlphaGo against Lee Sedol, a legendary professional player of the game of Go, and in May 2017 against Ke Jie, the world champion. No program had ever come close to beating a master of this game, let alone the world champion. Today the whole field of RL is boiling with new ideas, with a wide range of applications. DeepMind was bought by Google for over $500 million in 2014.

So how did DeepMind achieve all this? With hindsight it seems rather simple: they applied the power of Deep Learning to the field of Reinforcement Learning, and it worked beyond their wildest dreams. In this chapter we will first explain what

1 For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press).

2 Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning," arXiv preprint arXiv:1312.5602 (2013).

3 Volodymyr Mnih et al., "Human-Level Control Through Deep Reinforcement Learning," *Nature* 518 (2015): 529–533.

4 Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and other video games at *https://homl.info/dqn3*.

Reinforcement Learning is and what it's good at, then present two of the most important techniques in Deep Reinforcement Learning: *policy gradients* and *deep Q-networks* (DQNs), including a discussion of *Markov decision processes* (MDPs). We will use these techniques to train models to balance a pole on a moving cart; then I'll introduce the TF-Agents library, which uses state-of-the-art algorithms that greatly simplify building powerful RL systems, and we will use the library to train an agent to play *Breakout*, the famous Atari game. I'll close the chapter by taking a look at some of the latest advances in the field.

Learning to Optimize Rewards

In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see Figure 18-1):

a. The agent can be the program controlling a robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors* such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.
b. The agent can be the program controlling *Ms. Pac-Man*. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
c. Similarly, the agent can be the program playing a board game such as Go.
d. The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
e. The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

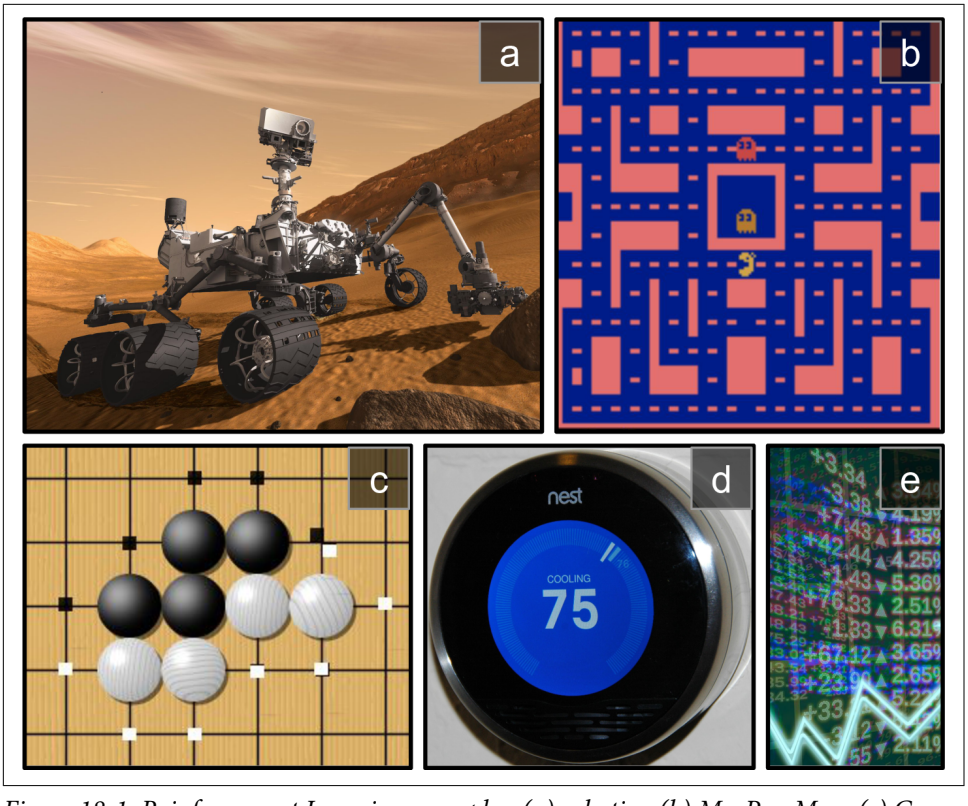


Figure 18-1. Reinforcement Learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader[5]

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it had better find the exit as quickly as possible! There are many other examples of tasks to which Reinforcement Learning is well suited, such as self-driving cars, recommender systems, placing ads on a web page, or controlling where an image classification system should focus its attention.

5 Image (a) is from NASA (public domain). (b) is a screenshot from the *Ms. Pac-Man* game, copyright Atari (fair use in this chapter). Images (c) and (d) are reproduced from Wikipedia. (c) was created by user Stevertigo and released under Creative Commons BY-SA 2.0. (d) is in the public domain. (e) was reproduced from Pixabay, released under Creative Commons CC0.

Policy Search

The algorithm a software agent uses to determine its actions is called its *policy*. The policy could be a neural network taking observations as inputs and outputting the action to take (see Figure 18-2).

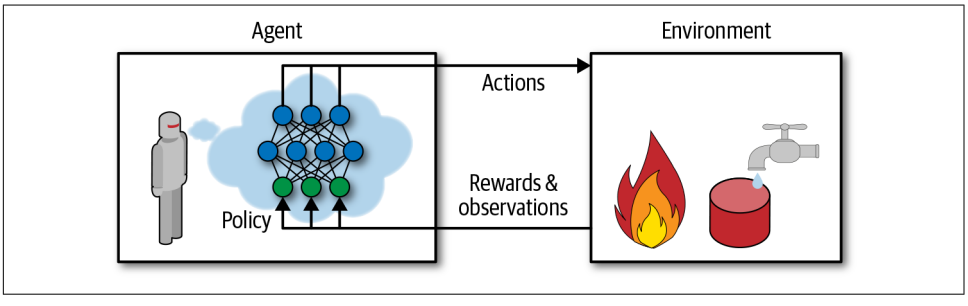


Figure 18-2. Reinforcement Learning using a neural network policy

The policy can be any algorithm you can think of, and it does not have to be deterministic. In fact, in some cases it does not even have to observe the environment! For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 – p$. The rotation angle would be a random angle between $–r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability p and the angle range r. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 18-3). This is an example of *policy search*, in this case using a brute force approach. When the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then "kill" the 80 worst policies[6] and make the 20 survivors produce 4 offspring each. An

6 It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the "gene pool."

offspring is a copy of its parent[7] plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy.[8]

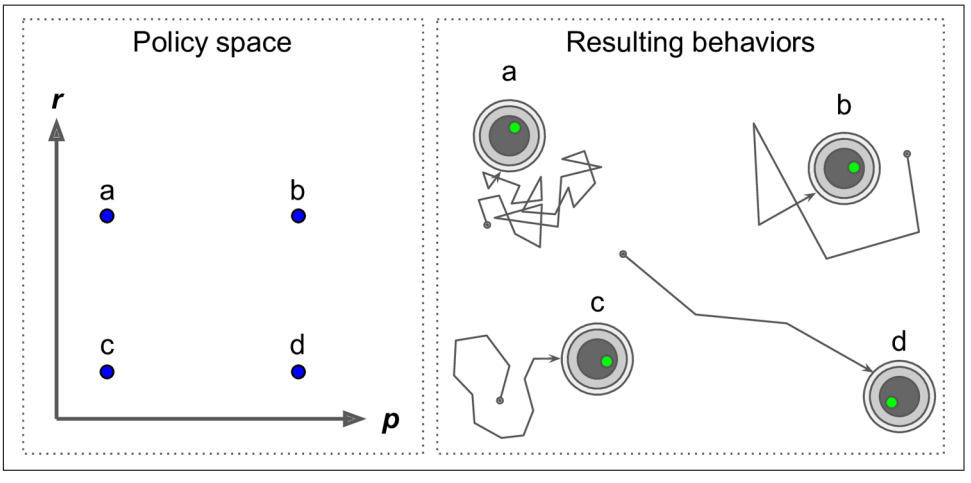


Figure 18-3. Four points in policy space (left) and the agent's corresponding behavior (right)

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards.[9] We will discuss this approach, is called *policy gradients* (PG), in more detail later in this chapter. Going back to the vacuum cleaner robot, you could slightly increase p and evaluate whether doing so increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase p some more, or else reduce p. We will implement a popular PG algorithm using TensorFlow, but before we do, we need to create an environment for the agent to live in—so it's time to introduce OpenAI Gym.

Introduction to OpenAI Gym

One of the challenges of Reinforcement Learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that

7 If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring's genome (in this case a set of policy parameters) is randomly composed of parts of its parents' genomes.

8 One interesting example of a genetic algorithm used for Reinforcement Learning is the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm.

9 This is called *Gradient Ascent*. It's just like Gradient Descent but in the opposite direction: maximizing instead of minimizing.

will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world, and you can directly train your robot in that environment, but this has its limits: if the robot falls off a cliff, you can't just click Undo. You can't speed up time either; adding more computing power won't make the robot move any faster. And it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training. For example, you may use a library like PyBullet or MuJoCo for 3D physics simulation.

OpenAI Gym[10] is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

Before installing the toolkit, if you created an isolated environment using virtualenv, you first need to activate it:

```
$ cd $ML_PATH                # Your ML working directory (e.g., $HOME/ml)
$ source my_env/bin/activate # on Linux or MacOS
$ .\my_env\Scripts\activate  # on Windows
```

Next, install OpenAI Gym (if you are not using a virtual environment, you will need to add the `--user` option, or have administrator rights):

```
$ python3 -m pip install -U gym
```

Depending on your system, you may also need to install the Mesa OpenGL Utility (GLU) library (e.g., on Ubuntu 18.04 you need to run `apt install libglu1-mesa`). This library will be needed to render the first environment. Next, open up a Python shell or a Jupyter notebook and create an environment with `make()`:

```
>>> import gym
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614,  0.04207708, -0.00180545])
```

Here, we've created a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 18-4). You can get the list of all available environments by running `gym.envs.registry.all()`. After the environment is created, you must initialize it using the `reset()` method. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a 1D NumPy array containing four floats: these floats represent the cart's horizontal

10 OpenAI is an artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate it).

position (`0.0` = center), its velocity (positive means right), the angle of the pole (`0.0` = vertical), and its angular velocity (positive means clockwise).

Now let's display this environment by calling its `render()` method (see Figure 18-4). On Windows, this requires first installing an X Server, such as VcXsrv or Xming:

```
>>> env.render()
True
```

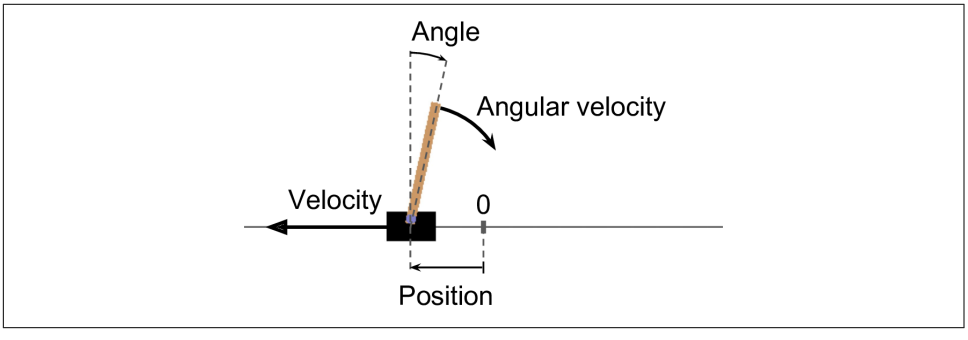


Figure 18-4. The CartPole environment

If you are using a headless server (i.e., without a screen), such as a virtual machine on the cloud, rendering will fail. The only way to avoid this is to use a fake X server such as Xvfb or Xdummy. For example, you can install Xvfb (`apt install xvfb` on Ubuntu or Debian) and start Python using the following command: `xvfb-run -s "-screen 0 1400x900x24" python3`. Alternatively, install Xvfb and the `pyvirtualdisplay` library (which wraps Xvfb) and run `pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()` at the beginning of your program.

If you want `render()` to return the rendered image as a NumPy array, you can set `mode="rgb_array"` (oddly, this environment will render the environment to screen as well):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape  # height, width, channels (3 = Red, Green, Blue)
(800, 1200, 3)
```

Let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1). Other environments may have additional discrete

actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

The `step()` method executes the given action and returns four values:

`obs`
: This is the new observation. The cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted toward the left after the next step.

`reward`
: In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running as long as possible.

`done`
: This value will be `True` when the episode is over. This will happen when the pole tilts too much, or goes off the screen, or after 200 steps (in this last case, you have won). After that, the environment must be reset before it can be used again.

`info`
: This environment-specific dictionary can provide some extra information that you may find useful for debugging or for training. For example, in some games it may indicate how many lives the agent has.

Once you have finished using an environment, you should call its `close()` method to free resources.

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

This code is hopefully self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps. Not great. If you look at the simulation in the Jupyter notebooks, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. Let's see if a neural network can come up with a better policy.

Neural Network Policies

Let's create a neural network policy. Just like with the policy we hardcoded earlier, this neural network will take an observation as input, and it will output the action to be executed. More precisely, it will estimate a probability for each action, and then we will select an action randomly, according to the estimated probabilities (see Figure 18-5). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be $1 – p$. For example, if it outputs 0.7, then we will pick action 0 with 70% probability, or action 1 with 30% probability.

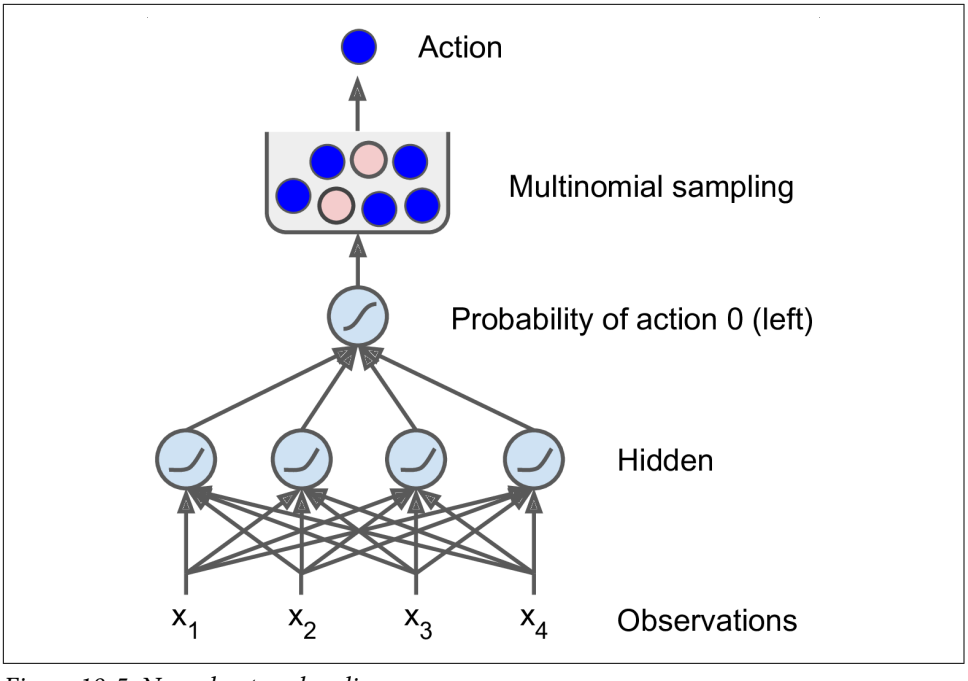


Figure 18-5. Neural network policy

You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you might need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free, and they contain the environment's full state.

Here is the code to build this neural network policy using tf.keras:

```
import tensorflow as tf
from tensorflow import keras

n_inputs = 4 # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])
```

After the imports, we use a simple `Sequential` model to define the policy network. The number of inputs is the size of the observation space (which in the case of CartPole is 4), and we have just five hidden units because it's a simple problem. Finally, we want to output a single probability (the probability of going left), so we have a single output neuron using the sigmoid activation function. If there were more than two possible actions, there would be one output neuron per action, and we would use the softmax activation function instead.

OK, we now have a neural network policy that will take observations and output action probabilities. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in Reinforcement Learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount factor* γ (gamma) at each step. This sum of discounted rewards is called the action's *return*. Consider the example in Figure 18-6). If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$. If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as

much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount factor of 0.95 seems reasonable.

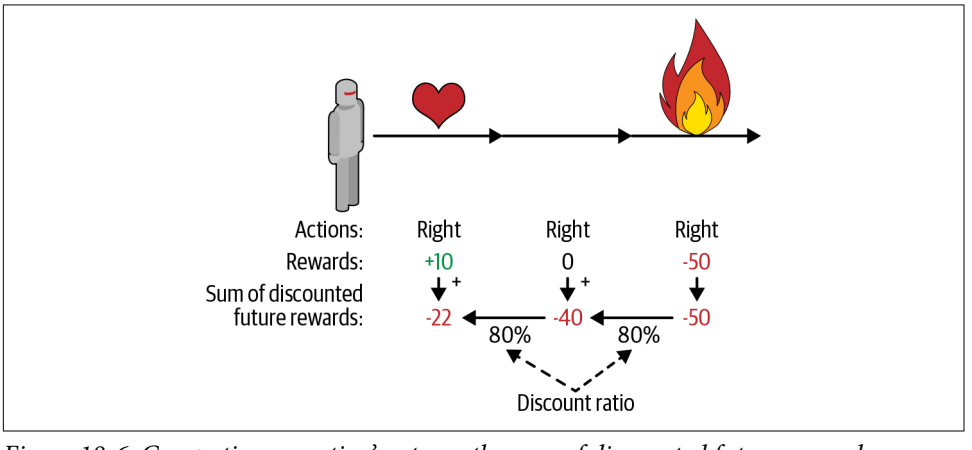


Figure 18-6. Computing an action's return: the sum of discounted future rewards

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low return (similarly, a good actor may sometimes star in a terrible movie). However, if we play the game enough times, on average good actions will get a higher return than bad ones. We want to estimate how much better or worse an action is, compared to the other possible actions, on average. This is called the *action advantage*. For this, we must run many episodes and normalize all the action returns (by subtracting the mean and dividing by the standard deviation). After that, we can reasonably assume that actions with a negative advantage were bad while actions with a positive advantage were good. Perfect—now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called

REINFORCE algorithms, was introduced back in 1992[11] by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times, and at each step, compute the gradients that would make the chosen action even more likely—but don't apply these gradients yet.
2. Once you have run several episodes, compute each action's advantage (using the method described in the previous section).
3. If an action's advantage is positive, it means that the action was probably good, and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the action's advantage is negative, it means the action was probably bad, and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is simply to multiply each gradient vector by the corresponding action's advantage.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

Let's use tf.keras to implement this algorithm. We will train the neural network policy we built earlier so that it learns to balance the pole on the cart. First, we need a function that will play one step. We will pretend for now that whatever action it takes is the right one so that we can compute the loss and its gradients (these gradients will just be saved for a while, and we will modify them later depending on how good or bad the action turned out to be):

```
def play_one_step(env, obs, model, loss_fn):
    with tf.GradientTape() as tape:
        left_proba = model(obs[np.newaxis])
        action = (tf.random.uniform([1, 1]) > left_proba)
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))
    grads = tape.gradient(loss, model.trainable_variables)
    obs, reward, done, info = env.step(int(action[0, 0].numpy()))
    return obs, reward, done, grads
```

Let's walk though this function:

- Within the `GradientTape` block (see Chapter 12), we start by calling the model, giving it a single observation (we reshape the observation so it becomes a batch containing a single instance, as the model expects a batch). This outputs the probability of going left.

11 Ronald J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Leaning," *Machine Learning* 8 (1992) : 229–256.

- Next, we sample a random float between 0 and 1, and we check whether it is greater than `left_proba`. The `action` will be `False` with probability `left_proba`, or `True` with probability `1 - left_proba`. Once we cast this Boolean to a number, the action will be 0 (left) or 1 (right) with the appropriate probabilities.
- Next, we define the target probability of going left: it is 1 minus the action (cast to a float). If the action is 0 (left), then the target probability of going left will be 1. If the action is 1 (right), then the target probability will be 0.
- Then we compute the loss using the given loss function, and we use the tape to compute the gradient of the loss with regard to the model's trainable variables. Again, these gradients will be tweaked later, before we apply them, depending on how good or bad the action turned out to be.
- Finally, we play the selected action, and we return the new observation, the reward, whether the episode is ended or not, and of course the gradients that we just computed.

Now let's create another function that will rely on the `play_one_step()` function to play multiple episodes, returning all the rewards and gradients for each episode and each step:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):
    all_rewards = []
    all_grads = []
    for episode in range(n_episodes):
        current_rewards = []
        current_grads = []
        obs = env.reset()
        for step in range(n_max_steps):
            obs, reward, done, grads = play_one_step(env, obs, model, loss_fn)
            current_rewards.append(reward)
            current_grads.append(grads)
            if done:
                break
        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
    return all_rewards, all_grads
```

This code returns a list of reward lists (one reward list per episode, containing one reward per step), as well as a list of gradient lists (one gradient list per episode, each containing one tuple of gradients per step and each tuple containing one gradient tensor per trainable variable).

The algorithm will use the `play_multiple_episodes()` function to play the game several times (e.g., 10 times), then it will go back and look at all the rewards, discount them, and normalize them. To do that, we need a couple more functions: the first will compute the sum of future discounted rewards at each step, and the second will

normalize all these discounted rewards (returns) across many episodes by subtracting the mean and dividing by the standard deviation:

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = [discount_rewards(rewards, discount_factor)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean) / reward_std
            for discounted_rewards in all_discounted_rewards]
```

Let's check that this works:

```
>>> discount_rewards([10, 0, -50], discount_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.0727777 ])]
```

The call to `discount_rewards()` returns exactly what we expect (see Figure 18-6). You can verify that the function `discount_and_normalize_rewards()` does indeed return the normalized action advantages for each action in both episodes. Notice that the first episode was much worse than the second, so its normalized advantages are all negative; all actions from the first episode would be considered bad, and conversely all actions from the second episode would be considered good.

We are almost ready to run the algorithm! Now let's define the hyperparameters. We will run 150 training iterations, playing 10 episodes per iteration, and each episode will last at most 200 steps. We will use a discount factor of 0.95:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

We also need an optimizer and the loss function. A regular Adam optimizer with learning rate 0.01 will do just fine, and we will use the binary cross-entropy loss function because we are training a binary classifier (there are two possible actions: left or right):

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

We are now ready to build and run the training loop!

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)
    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean(
            [final_reward * all_grads[episode_index][step][var_index]
             for episode_index, final_rewards in enumerate(all_final_rewards)
                 for step, final_reward in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads, model.trainable_variables))
```

Let's walk through this code:

- At each training iteration, this loop calls the `play_multiple_episodes()` function, which plays the game 10 times and returns all the rewards and gradients for every episode and step.
- Then we call the `discount_and_normalize_rewards()` to compute each action's normalized advantage (which in the code we call the `final_reward`). This provides a measure of how good or bad each action actually was, in hindsight.
- Next, we go through each trainable variable, and for each of them we compute the weighted mean of the gradients for that variable over all episodes and all steps, weighted by the `final_reward`.
- Finally, we apply these mean gradients using the optimizer: the model's trainable variables will be tweaked, and hopefully the policy will be a bit better.

And we're done! This code will train the neural network policy, and it will successfully learn to balance the pole on the cart (you can try it out in the "Policy Gradients" section of the Jupyter notebook). The mean reward per episode will get very close to 200 (which is the maximum by default with this environment). Success!

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically. For example, since you know that the pole should be as vertical as possible, you could add negative rewards proportional to the pole's angle. This will make the rewards much less sparse and speed up training. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is due to the fact that it must run multiple episodes to estimate the advantage of each action, as we have seen. However, it is the foundation of more powerful algorithms, such as *Actor-Critic* algorithms (which we will discuss briefly at the end of this chapter).

We will now look at another popular family of algorithms. Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct: the agent learns to estimate the expected return for each state, or for each action in each state, then it uses this knowledge to decide how to act. To understand these algorithms, we must first introduce *Markov decision processes*.

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s'), not on past states (this is why we say that the system has no memory).

Figure 18-7 shows an example of a Markov chain with four states.

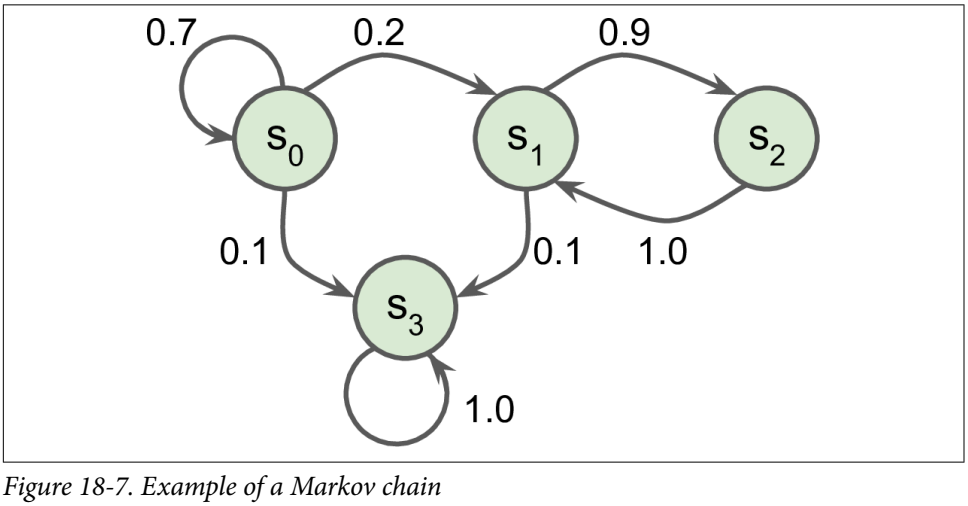


Figure 18-7. Example of a Markov chain

Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back because no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1

(with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever (this is a *terminal state*). Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

Markov decision processes were first described in the 1950s by Richard Bellman.[12] They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize reward over time.

For example, the MDP represented in Figure 18-8 has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds).

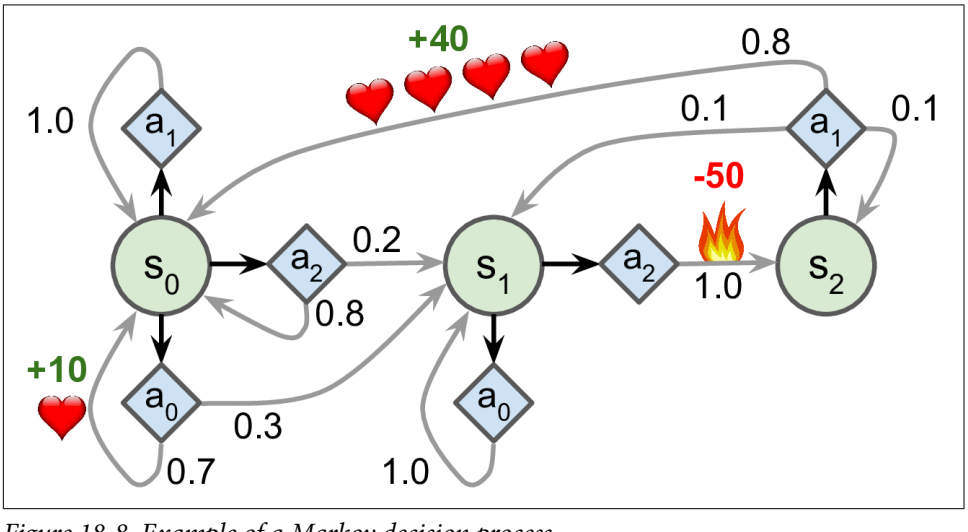


Figure 18-8. Example of a Markov decision process

If it starts in state s_0, the agent can choose between actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10 and remaining in state s_0. It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_2. It can choose to stay put by repeatedly choosing action a_0, or it can choose to move on to state s_2 and get a negative reward of –50 (ouch). In state s_2 it has no other choice than to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the

12 Richard Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.

way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

Bellman found a way to estimate the *optimal state value* of any state s, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state s, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies (see Equation 18-1). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 18-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot V^*(s')\right] \quad \text{for all } s$$

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a. For example, in Figure 18-8, $T(s_2, a_1, s_0) = 0.8$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a. For example, in Figure 18-8, $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero, and then you iteratively update them using the *Value Iteration* algorithm (see Equation 18-2). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 18-2. Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot V_k(s')\right] \quad \text{for all } s$$

In this equation, $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

This algorithm is an example of *Dynamic Programming*, which breaks down a complex problem into tractable subproblems that can be tackled iteratively.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values* (Quality Values). The optimal Q-Value of the state-action pair (s, a), noted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see Equation 18-3).

Equation 18-3. Q-Value Iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s'a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-Value for that state: $\pi^*(s) = \underset{a}{\text{argmax}}\ Q^*(s,a)$.

Let's apply this algorithm to the MDP represented in Figure 18-8. First, we need to define the MDP:

```
transition_probabilities = [ # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
        [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
        [None, [0.8, 0.1, 0.1], None]]
rewards = [ # shape=[s, a, s']
        [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
        [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
        [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

For example, to know the transition probability from s_2 to s_0 after playing action a_1, we will look up `transition_probabilities[2][1][0]` (which is 0.8). Similarly, to get the corresponding reward, we will look up `rewards[2][1][0]` (which is +40). And to get the list of possible actions in s_2, we will look up `possible_actions[2]` (in this case, only action a_1 is possible). Next, we must initialize all the Q-Values to 0 (except for the the impossible actions, for which we set the Q-Values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

Now let's run the Q-Value Iteration algorithm. It applies Equation 18-3 repeatedly, to all Q-Values, for every state and every possible action:

```
gamma = 0.90 # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

That's it! The resulting Q-Values look like this:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

For example, when the agent is in state s_0 and it chooses action a_1, the expected sum of discounted future rewards is approximately 17.0.

For each state, let's look at the action that has the highest Q-Value:

```
>>> np.argmax(Q_values, axis=1) # optimal action for each state
array([0, 0, 1])
```

This gives us the optimal policy for this MDP, when using a discount factor of 0.90: in state s_0 choose action a_0; in state s_1 choose action a_0 (i.e., stay put); and in state s_2 choose action a_1 (the only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy changes: in state s_1 the best action becomes a_2 (go through the fire!). This makes sense because the more you value future rewards, the more you are willing to put up with some pain now for the promise of future bliss.

Temporal Difference Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *Temporal Difference Learning* (TD Learning) algorithm is very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has

only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it progresses, the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 18-4).

Equation 18-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$

with $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

A more concise way of writing the first form of this equation is to use the notation $a \underset{\alpha}{\leftarrow} b$, which means $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. So, the first line of Equation 18-4 can be rewritten like this: $V(s) \underset{\alpha}{\leftarrow} r + \gamma \cdot V(s')$.

TD Learning has many similarities with Stochastic Gradient Descent, in particular the fact that it handles one sample at a time. Moreover, just like Stochastic GD, it can only truly converge if you gradually reduce the learning rate (otherwise it will keep bouncing around the optimum Q-Values).

For each state s, this algorithm simply keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later (assuming it acts optimally).

Q-Learning

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 18-5). Q-Learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-Values. Once it has

accurate Q-Value estimates (or close enough), then the optimal policy is choosing the action that has the highest Q-Value (i.e., the greedy policy).

Equation 18-5. Q-Learning algorithm

$$Q\left(s,a\right) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} \quad Q\left(s',a'\right)$$

For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-Value estimates for the next state s', since we assume that the target policy would act optimally from then on.

Let's implement the Q-Learning algorithm. First, we will need to make an agent explore the environment. For this, we need a step function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Now let's implement the agent's exploration policy. Since the state space is pretty small, a simple random policy will be sufficient. If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Next, after we initialize the Q-Values just like earlier, we are ready to run the Q-Learning algorithm with learning rate decay (using power scheduling, introduced in Chapter 11):

```
alpha0 = 0.05 # initial learning rate
decay = 0.005 # learning rate decay
gamma = 0.90 # discount factor
state = 0 # initial state

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

This algorithm will converge to the optimal Q-Values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. As you can see in Figure 18-9, the Q-Value Iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-Learning algorithm (right) takes about 8,000 iterations to converge. Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

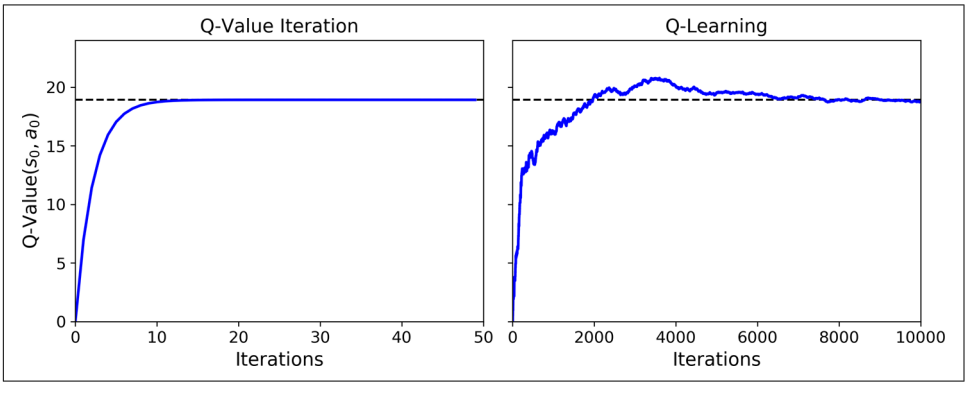


Figure 18-9. The Q-Value Iteration algorithm (left) versus the Q-Learning algorithm (right)

The Q-Learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one being executed: in the previous code example, the policy being executed (the exploration policy) is completely random, while the policy being trained will always choose the actions with the highest Q-Values. Conversely, the Policy Gradients algorithm is an *on-policy* algorithm: it explores the world using the policy being trained. It is somewhat surprising that Q-Learning is capable of learning the optimal policy by just watching an agent act randomly (imagine learning to play golf when your teacher is a drunk monkey). Can we do better?

Exploration Policies

Of course, Q-Learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy* (ε is epsilon): at each step it acts randomly with probability ε, or greedily with probability 1–ε (i.e., choosing the action with the highest Q-Value). The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-Value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in Equation 18-6.

Equation 18-6. Q-Learning using an exploration function

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} f(Q(s',a'), N(s',a'))$$

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Approximate Q-Learning and Deep Q-Learning

The main problem with Q-Learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-Learning to train an agent to play *Ms. Pac-Man* (see Figure 18-1). There are about 150 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible states is greater than $2^{150} \approx 10^{45}$. And if you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man, the number of possible states becomes larger than the number of atoms in our planet, so there's absolutely no way you can keep track of an estimate for every single Q-Value.

The solution is to find a function $Q_\theta(s, a)$ that approximates the Q-Value of any state-action pair (s, a) using a manageable number of parameters (given by the parameter vector θ). This is called *Approximate Q-Learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., distance of the closest ghosts, their directions, and so on) to estimate Q-Values, but in 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-Values is called a *Deep Q-Network* (DQN), and using a DQN for Approximate Q-Learning is called *Deep Q-Learning*.

Now, how can we train a DQN? Well, consider the approximate Q-Value computed by the DQN for a given state-action pair (s, a). Thanks to Bellman, we know we want this approximate Q-Value to be as close as possible to the reward r that we actually observe after playing action a in state s, plus the discounted value of playing optimally

from then on. To estimate this sum of future discounted rewards, we can simply execute the DQN on the next state s' and for all possible actions a'. We get an approximate future Q-Value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-Value $y(s, a)$ for the state-action pair (s, a), as shown in Equation 18-7.

Equation 18-7. Target Q-Value

$$Q_{\text{target}}\left(s, a\right) = r + \gamma \cdot \max_{a'} \; Q_{\theta}\left(s', a'\right)$$

With this target Q-Value, we can run a training step using any Gradient Descent algorithm. Specifically, we generally try to minimize the squared error between the estimated Q-Value $Q(s, a)$ and the target Q-Value (or the Huber loss to reduce the algorithm's sensitivity to large errors). And that's all for the basic Deep Q-Learning algorithm! Let's see how to implement it to solve the CartPole environment.

Implementing Deep Q-Learning

The first thing we need is a Deep Q-Network. In theory, you need a neural net that takes a state-action pair and outputs an approximate Q-Value, but in practice it's much more efficient to use a neural net that takes a state and outputs one approximate Q-Value for each possible action. To solve the CartPole environment, we do not need a very complicated neural net; a couple of hidden layers will do:

```
env = gym.make("CartPole-v0")
input_shape = [4] # == env.observation_space.shape
n_outputs = 2 # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

To select an action using this DQN, we pick the action with the largest predicted Q-Value. To ensure that the agent explores the environment, we will use an ε-greedy policy (i.e., we will choose a random action with probability ε):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Instead of training the DQN based only on the latest experiences, we will store all experiences in a *replay buffer* (or *replay memory*), and we will sample a random training batch from it at each training iteration. This helps reduce the correlations between the experiences in a training batch, which tremendously helps training. For this, we will just use a deque list:

```
from collections import deque

replay_buffer = deque(maxlen=2000)
```

A *deque* is a linked list, where each element points to the next one and to the previous one. It makes inserting and deleting items very fast, but the longer the deque is, the slower random access will be. If you need a very large replay buffer, use a circular buffer; see the "Deque vs Rotating List" section of the notebook for an implementation.

Each experience will be composed of five elements: a state, the action the agent took, the resulting reward, the next state it reached, and finally a Boolean indicating whether the episode ended at that point (`done`). We will need a small function to sample a random batch of experiences from the replay buffer. It will return five NumPy arrays corresponding to the five experience elements:

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

Let's also create a function that will play a single step using the ε-greedy policy, then store the resulting experience in the replay buffer:

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

Finally, let's create one last function that will sample a batch of experiences from the replay buffer and train the DQN by performing a single Gradient Descent step on this batch:

```
batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error
```

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Let's go through this code:

- First we define some hyperparameters, and we create the optimizer and the loss function.
- Then we create the `training_step()` function. It starts by sampling a batch of experiences, then it uses the DQN to predict the Q-Value for each possible action in each experience's next state. Since we assume that the agent will be playing optimally, we only keep the maximum Q-Value for each next state. Next, we use Equation 18-7 to compute the target Q-Value for each experience's state-action pair.
- Next, we want to use the DQN to compute the Q-Value for each experienced state-action pair. However, the DQN will also output the Q-Values for the other possible actions, not just for the action that was actually chosen by the agent. So we need to mask out all the Q-Values we do not need. The `tf.one_hot()` function makes it easy to convert an array of action indices into such a mask. For example, if the first three experiences contain actions 1, 1, 0, respectively, then the mask will start with `[[0, 1], [0, 1], [1, 0],...]`. We can then multiply the DQN's output with this mask, and this will zero out all the Q-Values we do not want. We then sum over axis 1 to get rid of all the zeros, keeping only the Q-Values of the experienced state-action pairs. This gives us the `Q_values` tensor, containing one predicted Q-Value for each experience in the batch.
- Then we compute the loss: it is the mean squared error between the target and predicted Q-Values for the experienced state-action pairs.
- Finally, we perform a Gradient Descent step to minimize the loss with regard to the model's trainable variables.

This was the hardest part. Now training the model is straightforward:

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```

We run 600 episodes, each for a maximum of 200 steps. At each step, we first compute the `epsilon` value for the ε-greedy policy: it will go from 1 down to 0.01, linearly, in a bit under 500 episodes. Then we call the `play_one_step()` function, which will use the ε-greedy policy to pick an action, then execute it and record the experience in the replay buffer. If the episode is done, we exit the loop. Finally, if we are past the 50th episode, we call the `training_step()` function to train the model on one batch sampled from the replay buffer. The reason we play 50 episodes without training is to give the replay buffer some time to fill up (if we don't wait enough, then there will not be enough diversity in the replay buffer). And that's it, we just implemented the Deep Q-Learning algorithm!

Figure 18-10 shows the total rewards the agent got during each episode.

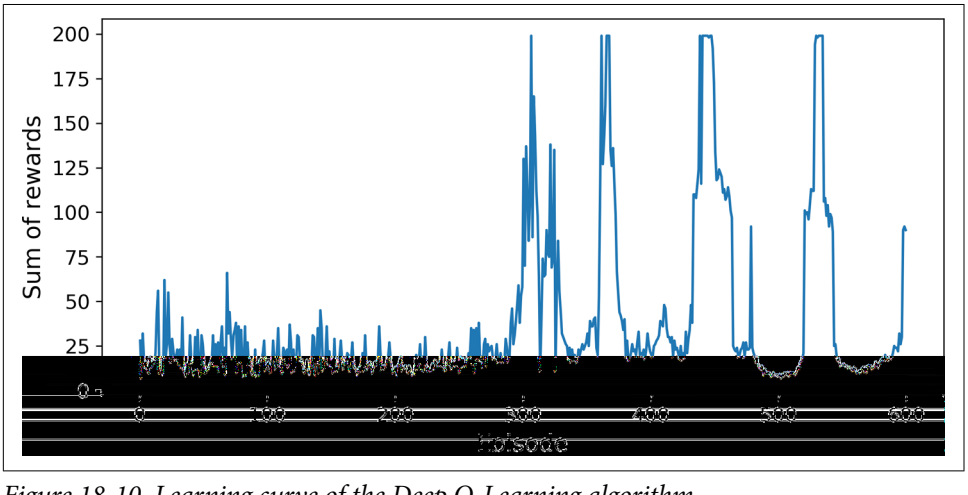


Figure 18-10. Learning curve of the Deep Q-Learning algorithm

As you can see, the algorithm made no apparent progress at all for almost 300 episodes (in part because ε was very high at the beginning), then its performance suddenly skyrocketed up to 200 (which is the maximum possible performance in this environment). That's great news: the algorithm worked fine, and it actually ran much faster than the Policy Gradient algorithm! But wait… just a few episodes later, it forgot everything it knew, and its performance dropped below 25! This is called

catastrophic forgetting, and it is one of the big problems facing virtually all RL algorithms: as the agent explores the environment, it updates its policy, but what it learns in one part of the environment may break what it learned earlier in other parts of the environment. The experiences are quite correlated, and the learning environment keeps changing—this is not ideal for Gradient Descent! If you increase the size of the replay buffer, the algorithm will be less subject to this problem. Reducing the learning rate may also help. But the truth is, Reinforcement Learning is hard: training is often unstable, and you may need to try many hyperparameter values and random seeds before you find a combination that works well. For example, if you try changing the number of neurons per layer in the preceding from 32 to 30 or 34, the performance will never go above 100 (the DQN may be more stable with one hidden layer instead of two).

Reinforcement Learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.[13] As the researcher Andrej Karpathy put it: "[Supervised learning] wants to work. […] RL must be forced to work." You will need time, patience, perseverance, and perhaps a bit of luck too. This is a major reason RL is not as widely adopted as regular Deep Learning (e.g., convolutional nets). But there are a few real-world applications, beyond AlphaGo and Atari games: for example, Google uses RL to optimize its datacenter costs, and it is used in some robotics applications, for hyperparameter tuning, and in recommender systems.

You might wonder why we didn't plot the loss. It turns out that loss is a poor indicator of the model's performance. The loss might go down, yet the agent might perform worse (e.g., this can happen when the agent gets stuck in one small region of the environment, and the DQN starts overfitting this region). Conversely, the loss could go up, yet the agent might perform better (e.g., if the DQN was underestimating the Q-Values, and it starts correctly increasing its predictions, the agent will likely perform better, getting more rewards, but the loss might increase because the DQN also sets the targets, which will be larger too).

The basic Deep Q-Learning algorithm we've been using so far would be too unstable to learn to play Atari games. So how did DeepMind do it? Well, they tweaked the algorithm!

13 A great 2018 post by Alex Irpan nicely lays out RL's biggest difficulties and limitations.

Deep Q-Learning Variants

Let's look at a few variants of the Deep Q-Learning algorithm that can stabilize and speed up training.

Fixed Q-Value Targets

In the basic Deep Q-Learning algorithm, the model is used both to make predictions and to set its own targets. This can lead to a situation analogous to a dog chasing its own tail. This feedback loop can make the network unstable: it can diverge, oscillate, freeze, and so on. To solve this problem, in their 2013 paper the DeepMind researchers used two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is the *target model* used only to define the targets. The target model is just a clone of the online model:

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

Then, in the `training_step()` function, we just need to change one line to use the target model instead of the online model when computing the Q-Values of the next states:

```
next_Q_values = target.predict(next_states)
```

Finally, in the training loop, we must copy the weights of the online model to the target model, at regular intervals (e.g., every 50 episodes):

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

Since the target model is updated much less often than the online model, the Q-Value targets are more stable, the feedback loop we discussed earlier is dampened, and its effects are less severe. This approach was one of the DeepMind researchers' main contributions in their 2013 paper, allowing agents to learn to play Atari games from raw pixels. To stabilize training, they used a tiny learning rate of 0.00025, they updated the target model only every 10,000 steps (instead of the 50 in the previous code example), and they used a very large replay buffer of 1 million experiences. They decreased `epsilon` very slowly, from 1 to 0.1 in 1 million steps, and they let the algorithm run for 50 million steps.

Later in this chapter, we will use the TF-Agents library to train a DQN agent to play *Breakout* using these hyperparameters, but before we get there, let's take a look at another DQN variant that managed to beat the state of the art once more.

Double DQN

In a 2015 paper,[14] DeepMind researchers tweaked their DQN algorithm, increasing its performance and somewhat stabilizing training. They called this variant *Double DQN*. The update was based on the observation that the target network is prone to overestimating Q-Values. Indeed, suppose all actions are equally good: the Q-Values estimated by the target model should be identical, but since they are approximations, some may be slightly greater than others, by pure chance. The target model will always select the largest Q-Value, which will be slightly greater than the mean Q-Value, most likely overestimating the true Q-Value (a bit like counting the height of the tallest random wave when measuring the depth of a pool). To fix this, they proposed using the online model instead of the target model when selecting the best actions for the next states, and using the target model only to estimate the Q-Values for these best actions. Here is the updated `training_step()` function:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # the rest is the same as earlier
```

Just a few months later, another improvement to the DQN algorithm was proposed.

Prioritized Experience Replay

Instead of sampling experiences *uniformly* from the replay buffer, why not sample important experiences more frequently? This idea is called *importance sampling* (IS) or *prioritized experience replay* (PER), and it was introduced in a 2015 paper[15] by DeepMind researchers (once again!).

More specifically, experiences are considered "important" if they are likely to lead to fast learning progress. But how can we estimate this? One reasonable approach is to measure the magnitude of the TD error $\delta = r + \gamma \cdot V(s') - V(s)$. A large TD error indicates that a transition (s, r, s') is very surprising, and thus probably worth learning

14 Hado van Hasselt et al., "Deep Reinforcement Learning with Double Q-Learning," *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.

15 Tom Schaul et al., "Prioritized Experience Replay," arXiv preprint arXiv:1511.05952 (2015).

from.[16] When an experience is recorded in the replay buffer, its priority is set to a very large value, to ensure that it gets sampled at least once. However, once it is sampled (and every time it is sampled), the TD error δ is computed, and this experience's priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a nonzero probability of being sampled). The probability P of sampling an experience with priority p is proportional to p^{ζ}, where ζ is a hyperparameter that controls how greedy we want importance sampling to be: when $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling. In the paper, the authors used $\zeta = 0.6$, but the optimal value will depend on the task.

There's one catch, though: since the samples will be biased toward important experiences, we must compensate for this bias during training by downweighting the experiences according to their importance, or else the model will just overfit the important experiences. To be clear, we want important experiences to be sampled more often, but this also means we must give them a lower weight during training. To do this, we define each experience's training weight as $w = (n\ P)^{-\beta}$, where n is the number of experiences in the replay buffer, and β is a hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, while 1 means entirely). In the paper, the authors used $\beta = 0.4$ at the beginning of training and linearly increased it to $\beta = 1$ by the end of training. Again, the optimal value will depend on the task, but if you increase one, you will usually want to increase the other as well.

Now let's look at one last important variant of the DQN algorithm.

Dueling DQN

The *Dueling DQN* algorithm (DDQN, not to be confused with Double DQN, although both techniques can easily be combined) was introduced in yet another 2015 paper[17] by DeepMind researchers. To understand how it works, we must first note that the Q-Value of a state-action pair (s, a) can be expressed as $Q(s, a) = V(s) + A(s, a)$, where $V(s)$ is the value of state s and $A(s, a)$ is the *advantage* of taking the action a in state s, compared to all other possible actions in that state. Moreover, the value of a state is equal to the Q-Value of the best action a^* for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s, a^*)$, which implies that $A(s, a^*) = 0$. In a Dueling DQN, the model estimates both the value of the state and the advantage of each possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all pre-

16 It could also just be that the rewards are noisy, in which case there are better methods for estimating an experience's importance (see the paper for some examples).

17 Ziyu Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning," arXiv preprint arXiv: 1511.06581 (2015).

dicted advantages. Here is a simple Dueling DQN model, implemented using the Functional API:

```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

The rest of the algorithm is just the same as earlier. In fact, you can build a Double Dueling DQN and combine it with prioritized experience replay! More generally, many RL techniques can be combined, as DeepMind demonstrated in a 2017 paper.[18] The paper's authors combined six different techniques into an agent called *Rainbow*, which largely outperformed the state of the art.

Unfortunately, implementing all of these techniques, debugging them, fine-tuning them, and of course training the models can require a huge amount of work. So instead of reinventing the wheel, it is often best to reuse scalable and well-tested libraries, such as TF-Agents.

The TF-Agents Library

The TF-Agents library is a Reinforcement Learning library based on TensorFlow, developed at Google and open sourced in 2018. Just like OpenAI Gym, it provides many off-the-shelf environments (including wrappers for all OpenAI Gym environments), plus it supports the PyBullet library (for 3D physics simulation), DeepMind's DM Control library (based on MuJoCo's physics engine), and Unity's ML-Agents library (simulating many 3D environments). It also implements many RL algorithms, including REINFORCE, DQN, and DDQN, as well as various RL components such as efficient replay buffers and metrics. It is fast, scalable, easy to use, and customizable: you can create your own environments and neural nets, and you can customize pretty much any component. In this section we will use TF-Agents to train an agent to play *Breakout*, the famous Atari game (see Figure 18-11[19]), using the DQN algorithm (you can easily switch to another algorithm if you prefer).

18 Matteo Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning," arXiv preprint arXiv:1710.02298 (2017): 3215–3222.

19 If you don't know this game, it's simple: a ball bounces around and breaks bricks when it touches them. You control a paddle near the bottom of the screen. The paddle can go left or right, and you must get the ball to break every brick, while preventing it from touching the bottom of the screen.

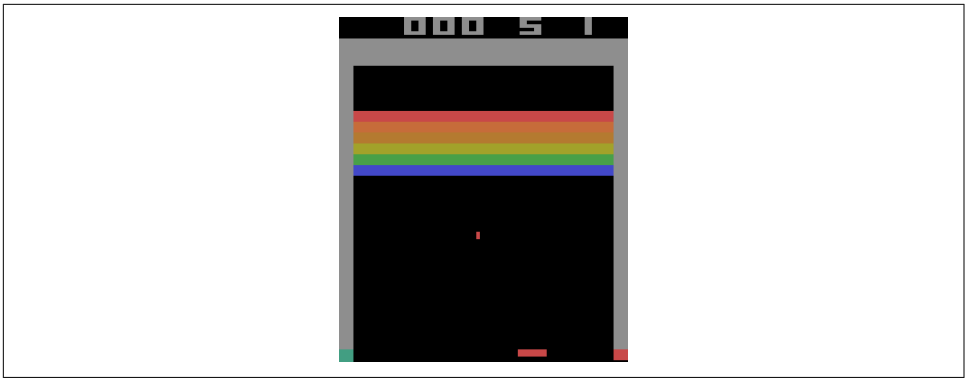


Figure 18-11. The famous Breakout game

Installing TF-Agents

Let's start by installing TF-Agents. This can be done using pip (as always, if you are using a virtual environment, make sure to activate it first; if not, you will need to use the `--user` option, or have administrator rights):

```
$ python3 -m pip install -U tf-agents
```

At the time of this writing, TF-Agents is still quite new and improving every day, so the API may change a bit by the time you read this—but the big picture should remain the same, as well as most of the code. If anything breaks, I will update the Jupyter notebook accordingly, so make sure to check it out.

Next, let's create a TF-Agents environment that will just wrap OpenAI GGym's Breakout environment. For this, you must first install OpenAI Gym's Atari dependencies:

```
$ python3 -m pip install -U 'gym[atari]'
```

Among other libraries, this command will install `atari-py`, which is a Python interface for the Arcade Learning Environment (ALE), a framework built on top of the Atari 2600 emulator Stella.

TF-Agents Environments

If everything went well, you should be able to import TF-Agents and create a Breakout environment:

```
>>> from tf_agents.environments import suite_gym
>>> env = suite_gym.load("Breakout-v4")
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

This is just a wrapper around an OpenAI Gym environment, which you can access through the `gym` attribute:

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

TF-Agents environments are very similar to OpenAI Gym environments, but there are a few differences. First, the `reset()` method does not return an observation; instead it returns a `TimeStep` object that wraps the observation, as well as some extra information:

```
>>> env.reset()
TimeStep(step_type=array(0, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.],...]]], dtype=float32))
```

The `step()` method returns a `TimeStep` object as well:

```
>>> env.step(1) # Fire
TimeStep(step_type=array(1, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[[0., 0., 0.], [0., 0., 0.],...]]], dtype=float32))
```

The `reward` and `observation` attributes are self-explanatory, and they are the same as for OpenAI Gym (except the `reward` is represented as a NumPy array). The `step_type` attribute is equal to 0 for the first time step in the episode, 1 for intermediate time steps, and 2 for the final time step. You can call the time step's `is_last()` method to check whether it is the final one or not. Lastly, the `discount` attribute indicates the discount factor to use at this time step. In this example it is equal to 1, so there will be no discount at all. You can define the discount factor by setting the `discount` parameter when loading the environment.

At any time, you can access the environment's current time step by calling its `current_time_step()` method.

Environment Specifications

Conveniently, a TF-Agents environment provides the specifications of the observations, actions, and time steps, including their shapes, data types, and names, as well as their minimum and maximum values:

```
>>> env.observation_spec()
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
                 minimum=[[[0. 0. 0.], [0. 0. 0.],...]],
                 maximum=[[[255., 255., 255.], [255., 255., 255.], ...]])
>>> env.action_spec()
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,
                 minimum=0, maximum=3)
>>> env.time_step_spec()
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'), name='step_type'),
         reward=ArraySpec(shape=(), dtype=dtype('float32'), name='reward'),
         discount=BoundedArraySpec(shape=(), ..., minimum=0.0, maximum=1.0),
         observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

As you can see, the observations are simply screenshots of the Atari screen, represented as NumPy arrays of shape [210, 160, 3]. To render an environment, you can call `env.render(mode="human")`, and if you want to get back the image in the form of a NumPy array, just call `env.render(mode="rgb_array")` (unlike in OpenAI Gym, this is the default mode).

There are four actions available. Gym's Atari environments have an extra method that you can call to know what each action corresponds to:

```
>>> env.gym.get_action_meanings()
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```

Specs can be instances of a specification class, nested lists, or dictionaries of specs. If the specification is nested, then the specified object must match the specification's nested structure. For example, if the observation spec is `{"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`, then a valid observation would be `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}`. The `tf.nest` package provides tools to handle such nested structures (a.k.a. *nests*).

The observations are quite large, so we will downsample them and also convert them to grayscale. This will speed up training and use less RAM. For this, we can use an *environment wrapper*.

Environment Wrappers and Atari Preprocessing

TF-Agents provides several environment wrappers in the `tf_agents.environments.wrappers` package. As their name suggests, they wrap an environment, forwarding every call to it, but also adding some extra functionality. Here are some of the available wrappers:

`ActionClipWrapper`
: Clips the actions to the action spec.

`ActionDiscretizeWrapper`
: Quantizes a continuous action space to a discrete action space. For example, if the original environment's action space is the continuous range from –1.0 to +1.0, but you want to use an algorithm that only supports discrete action spaces, such as a DQN, then you can wrap the environment using `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)`, and the new `discrete_env` will have a discrete action space with five possible actions: 0, 1, 2, 3, 4. These actions correspond to the actions –1.0, –0.5, 0.0, 0.5, and 1.0 in the original environment.

`ActionRepeat`
: Repeats each action over *n* steps, while accumulating the rewards. In many environments, this can speed up training significantly.

`RunStats`
: Records environment statistics such as the number of steps and the number of episodes.

`TimeLimit`
: Interrupts the environment if it runs for longer than a maximum number of steps.

`VideoWrapper`
: Records a video of the environment.

To create a wrapped environment, you must create a wrapper, passing the wrapped environment to the constructor. That's all! For example, the following code will wrap our environment in an `ActionRepeat` wrapper so that every action is repeated four times:

```
from tf_agents.environments.wrappers import ActionRepeat

repeating_env = ActionRepeat(env, times=4)
```

OpenAI Gym has some environment wrappers of its own in the `gym.wrappers` package. They are meant to wrap Gym environments, though, not TF-Agents environments, so to use them you must first wrap the Gym environment with a Gym wrapper, then wrap the resulting environment with a TF-Agents wrapper. The `suite_gym.wrap_env()` function will do this for you, provided you give it a Gym environment and a list of Gym wrappers and/or a list of TF-Agents wrappers. Alternatively, the `suite_gym.load()` function will both create the Gym environment and wrap it for you, if you give it some wrappers. Each wrapper will be created without any arguments, so if you want to set some arguments, you must pass a `lambda`. For example, the following code creates a Breakout environment that will run for a maximum of 10,000 steps during each episode, and each action will be repeated four times:

```
from gym.wrappers import TimeLimit

limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[lambda env: TimeLimit(env, max_episode_steps=10000)],
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

For Atari environments, some standard preprocessing steps are applied in most papers that use them, so TF-Agents provides a handy `AtariPreprocessing` wrapper that implements them. Here is the list of preprocessing steps it supports:

Grayscale and downsampling
: Observations are converted to grayscale and downsampled (by default to 84 × 84 pixels).

Max pooling
: The last two frames of the game are max-pooled using a 1 × 1 filter. This is to remove the flickering that occurs in some Atari games due to the limited number of sprites that the Atari 2600 could display in each frame.

Frame skipping
: The agent only gets to see every n frames of the game (by default $n = 4$), and its actions are repeated for each frame, collecting all the rewards. This effectively speeds up the game from the perspective of the agent, and it also speeds up training because rewards are less delayed.

End on life lost
: In some games, the rewards are just based on the score, so the agent gets no immediate penalty for losing a life. One solution is to end the game immediately whenever a life is lost. There is some debate over the actual benefits of this strategy, so it is off by default.

Since the default Atari environment already applies random frame skipping and max pooling, we will need to load the raw, nonskipping variant called `"BreakoutNoFrameskip-v4"`. Moreover, a single frame from the *Breakout* game is insufficient to know the direction and speed of the ball, which will make it very difficult for the agent to play the game properly (unless it is an RNN agent, which preserves some internal state between steps). One way to handle this is to use an environment wrapper that will output observations composed of multiple frames stacked on top of each other along the channels dimension. This strategy is implemented by the `FrameStack4` wrapper, which returns stacks of four frames. Let's create the wrapped Atari environment!

```
from tf_agents.environments import suite_atari
from tf_agents.environments.atari_preprocessing import AtariPreprocessing
from tf_agents.environments.atari_wrappers import FrameStack4

max_episode_steps = 27000 # <=> 108k ALE frames since 1 step = 4 frames
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

The result of all this preprocessing is shown in Figure 18-12. You can see that the resolution is much lower, but sufficient to play the game. Moreover, frames are stacked along the channels dimension, so red represents the frame from three steps ago, green is two steps ago, blue is the previous frame, and pink is the current frame.[20] From this single observation, the agent can see that the ball is going toward the lower-left corner, and that it should continue to move the paddle to the left (as it did in the previous steps).

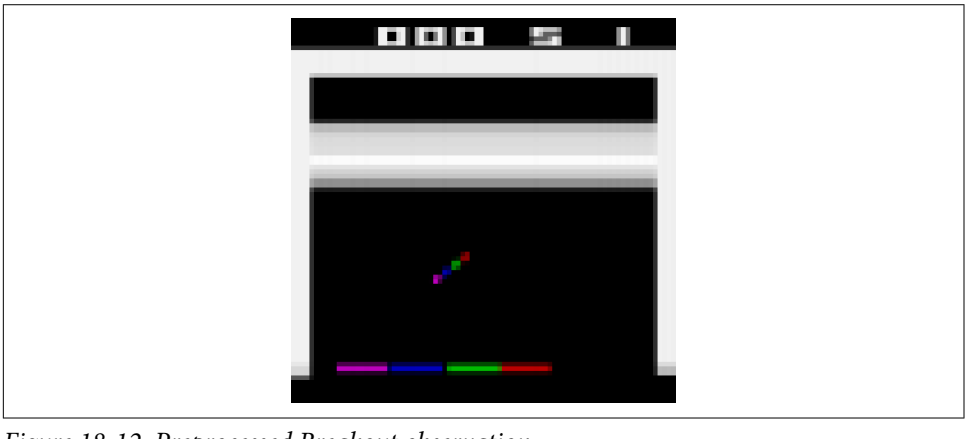


Figure 18-12. Preprocessed Breakout observation

Lastly, we can wrap the environment inside a `TFPyEnvironment`:

```
from tf_agents.environments.tf_py_environment import TFPyEnvironment

tf_env = TFPyEnvironment(env)
```

This will make the environment usable from within a TensorFlow graph (under the hood, this class relies on `tf.py_function()`, which allows a graph to call arbitrary

20 Since there are only three primary colors, you cannot just display an image with four color channels. For this reason, I combined the last channel with the first three to get the RGB image represented here. Pink is actually a mix of blue and red, but the agent sees four independent channels.

Python code). Thanks to the `TFPyEnvironment` class, TF-Agents supports both pure Python environments and TensorFlow-based environments. More generally, TF-Agents supports and provides both pure Python and TensorFlow-based components (agents, replay buffers, metrics, and so on).

Now that we have a nice Breakout environment, with all the appropriate preprocessing and TensorFlow support, we must create the DQN agent and the other components we will need to train it. Let's look at the architecture of the system we will build.

Training Architecture

A TF-Agents training program is usually split into two parts that run in parallel, as you can see in Figure 18-13: on the left, a *driver* explores the *environment* using a *collect policy* to choose actions, and it collects *trajectories* (i.e., experiences), sending them to an *observer*, which saves them to a *replay buffer*; on the right, an *agent* pulls batches of trajectories from the replay buffer and trains some *networks*, which the collect policy uses. In short, the left part explores the environment and collects trajectories, while the right part learns and updates the collect policy.

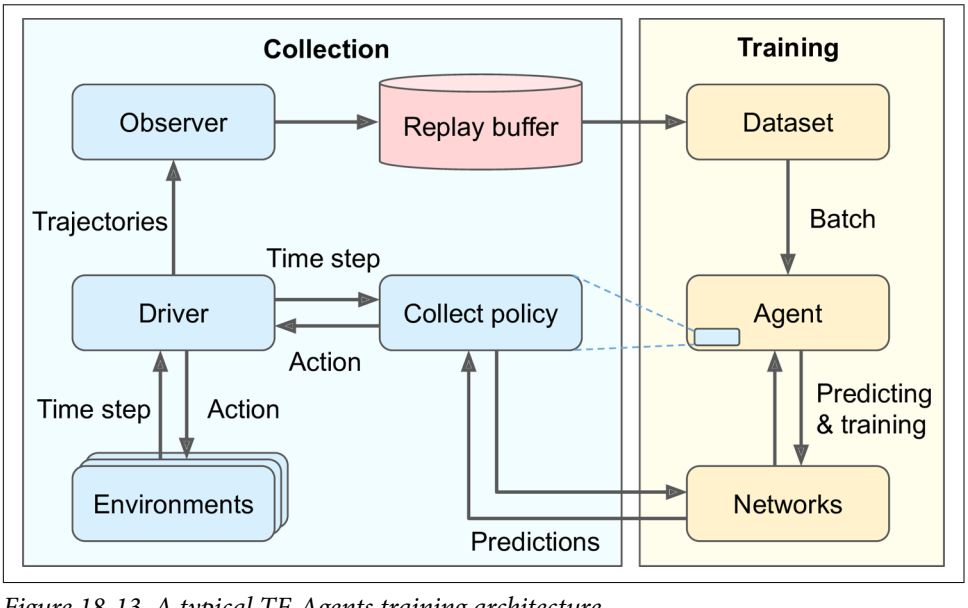


Figure 18-13. A typical TF-Agents training architecture

This figure begs a few questions, which I'll attempt to answer here:

- Why are there multiple environments? Instead of exploring a single environment, you generally want the driver to explore multiple copies of the environment in parallel, taking advantage of the power of all your CPU cores, keeping

the training GPUs busy, and providing less-correlated trajectories to the training algorithm.

- What is a *trajectory*? It is a concise representation of a *transition* from one time step to the next, or a sequence of consecutive transitions from time step n to time step $n + t$. The trajectories collected by the driver are passed to the observer, which saves them in the replay buffer, and they are later sampled by the agent and used for training.
- Why do we need an observer? Can't the driver save the trajectories directly? Indeed, it could, but this would make the architecture less flexible. For example, what if you don't want to use a replay buffer? What if you want to use the trajectories for something else, like computing metrics? In fact, an observer is just any function that takes a trajectory as an argument. You can use an observer to save the trajectories to a replay buffer, or to save them to a TFRecord file (see Chapter 13), or to compute metrics, or for anything else. Moreover, you can pass multiple observers to the driver, and it will broadcast the trajectories to all of them.

Although this architecture is the most common, you can customize it as you please, and even replace some components with your own. In fact, unless you are researching new RL algorithms, you will most likely want to use a custom environment for your task. For this, you just need to create a custom class that inherits from the `PyEnvironment` class in the `tf_agents.environments.py_environment` package and overrides the appropriate methods, such as `action_spec()`, `observation_spec()`, `_reset()`, and `_step()` (see the "Creating a Custom TF_Agents Environment" section of the notebook for an example).

Now we will create all these components: first the Deep Q-Network, then the DQN agent (which will take care of creating the collect policy), then the replay buffer and the observer to write to it, then a few training metrics, then the driver, and finally the dataset. Once we have all the components in place, we will populate the replay buffer with some initial trajectories, then we will run the main training loop. So, let's start by creating the Deep Q-Network.

Creating the Deep Q-Network

The TF-Agents library provides many networks in the `tf_agents.networks` package and its subpackages. We will use the `tf_agents.networks.q_network.QNetwork` class:

```
from tf_agents.networks.q_network import QNetwork

preprocessing_layer = keras.layers.Lambda(
                          lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]

q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)
```

This `QNetwork` takes an observation as input and outputs one Q-Value per action, so we must give it the specifications of the observations and the actions. It starts with a preprocessing layer: a simple `Lambda` layer that casts the observations to 32-bit floats and normalizes them (the values will range from 0.0 to 1.0). The observations contain unsigned bytes, which use 4 times less space than 32-bit floats, which is why we did not cast the observations to 32-bit floats earlier; we want to save RAM in the replay buffer. Next, the network applies three convolutional layers: the first has 32 8×8 filters and uses a stride of 4, the second has 64 4×4 filters and a stride of 2, and the third has 64 3×3 filters and a stride of 1. Lastly, it applies a dense layer with 512 units, followed by a dense output layer with 4 units, one per Q-Value to output (i.e., one per action). All convolutional layers and all dense layers except the output layer use the ReLU activation function by default (you can change this by setting the `activation_fn` argument). The output layer does not use any activation function.

Under the hood, a `QNetwork` is composed of two parts: an encoding network that processes the observations, followed by a dense output layer that outputs one Q-Value per action. TF-Agent's `EncodingNetwork` class implements a neural network architecture found in various agents (see Figure 18-14).

It may have one or more inputs. For example, if each observation is composed of some sensor data plus an image from a camera, you will have two inputs. Each input may require some preprocessing steps, in which case you can specify a list of Keras layers via the `preprocessing_layers` argument, with one preprocessing layer per input, and the network will apply each layer to the corresponding input (if an input requires multiple layers of preprocessing, you can pass a whole model, since a Keras model can always be used as a layer). If there are two inputs or more, you must also pass an extra layer via the `preprocessing_combiner` argument, to combine the outputs from the preprocessing layers into a single output.

Next, the encoding network will optionally apply a list of convolutions sequentially, provided you specify their parameters via the `conv_layer_params` argument. This must be a list composed of 3-tuples (one per convolutional layer) indicating the

number of filters, the kernel size, and the stride. After these convolutional layers, the encoding network will optionally apply a sequence of dense layers, if you set the `fc_layer_params` argument: it must be a list containing the number of neurons for each dense layer. Optionally, you can also pass a list of dropout rates (one per dense layer) via the `dropout_layer_params` argument if you want to apply dropout after each dense layer. The `QNetwork` takes the output of this encoding network and passes it to the dense output layer (with one unit per action).

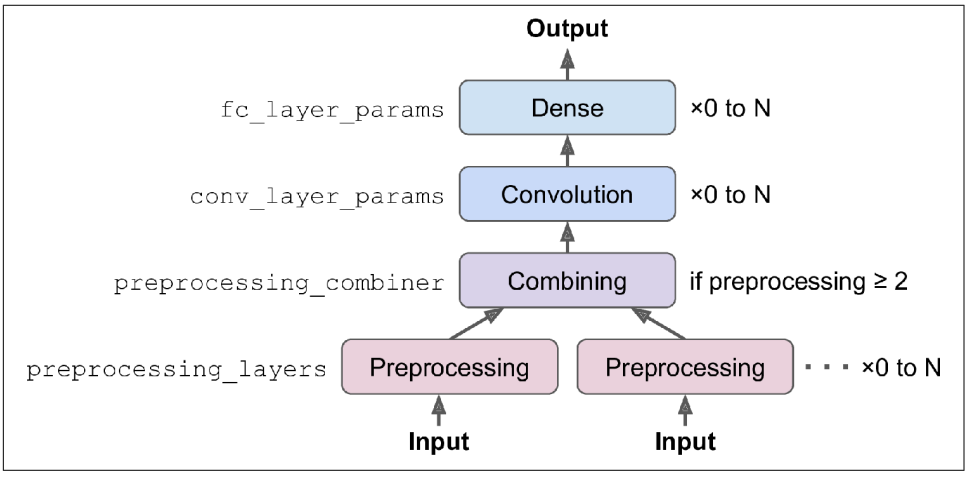


Figure 18-14. Architecture of an encoding network

The `QNetwork` class is flexible enough to build many different architectures, but you can always build your own network class if you need extra flexibility: extend the `tf_agents.networks.Network` class and implement it like a regular custom Keras layer. The `tf_agents.networks.Network` class is a subclass of the `keras.layers.Layer` class that adds some functionality required by some agents, such as the possibility to easily create shallow copies of the network (i.e., copying the network's architecture, but not its weights). For example, the `DQNAgent` uses this to create a copy of the online model.

Now that we have the DQN, we are ready to build the DQN agent.

Creating the DQN Agent

The TF-Agents library implements many types of agents, located in the `tf_agents.agents` package and its subpackages. We will use the `tf_agents.agents.dqn.dqn_agent.DqnAgent` class:

```
from tf_agents.agents.dqn.dqn_agent import DqnAgent

train_step = tf.Variable(0)
update_period = 4 # train the model every 4 steps
optimizer = keras.optimizers.RMSprop(lr=2.5e-4, rho=0.95, momentum=0.0,
                                     epsilon=0.00001, centered=True)
epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0, # initial ε
    decay_steps=250000 // update_period, # <=> 1,000,000 ALE frames
    end_learning_rate=0.01) # final ε
agent = DqnAgent(tf_env.time_step_spec(),
                 tf_env.action_spec(),
                 q_network=q_net,
                 optimizer=optimizer,
                 target_update_period=2000, # <=> 32,000 ALE frames
                 td_errors_loss_fn=keras.losses.Huber(reduction="none"),
                 gamma=0.99, # discount factor
                 train_step_counter=train_step,
                 epsilon_greedy=lambda: epsilon_fn(train_step))
agent.initialize()
```

Let's walk through this code:

- We first create a variable that will count the number of training steps.
- Then we build the optimizer, using the same hyperparameters as in the 2015 DQN paper.
- Next, we create a `PolynomialDecay` object that will compute the ε value for the ε-greedy collect policy, given the current training step (it is normally used to decay the learning rate, hence the names of the arguments, but it will work just fine to decay any other value). It will go from 1.0 down to 0.01 (the value used during in the 2015 DQN paper) in 1 million ALE frames, which corresponds to 250,000 steps, since we use frame skipping with a period of 4. Moreover, we will train the agent every 4 steps (i.e., 16 ALE frames), so ε will actually decay over 62,500 *training* steps.
- We then build the `DQNAgent`, passing it the time step and action specs, the `QNetwork` to train, the optimizer, the number of training steps between target model updates, the loss function to use, the discount factor, the `train_step` variable, and a function that returns the ε value (it must take no argument, which is why we need a lambda to pass the `train_step`).

  Note that the loss function must return an error per instance, not the mean error, which is why we set `reduction="none"`.
- Lastly, we initialize the agent.

Next, let's build the replay buffer and the observer that will write to it.

Creating the Replay Buffer and the Corresponding Observer

The TF-Agents library provides various replay buffer implementations in the `tf_agents.replay_buffers` package. Some are purely written in Python (their module names start with `py_`), and others are written based on TensorFlow (their module names start with `tf_`). We will use the `TFUniformReplayBuffer` class in the `tf_agents.replay_buffers.tf_uniform_replay_buffer` package. It provides a high-performance implementation of a replay buffer with uniform sampling:[21]

```
from tf_agents.replay_buffers import tf_uniform_replay_buffer

replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(
    data_spec=agent.collect_data_spec,
    batch_size=tf_env.batch_size,
    max_length=1000000)
```

Let's look at each of these arguments:

`data_spec`
: The specification of the data that will be saved in the replay buffer. The DQN agent knowns what the collected data will look like, and it makes the data spec available via its `collect_data_spec` attribute, so that's what we give the replay buffer.

`batch_size`
: The number of trajectories that will be added at each step. In our case, it will be one, since the driver will just execute one action per step and collect one trajectory. If the environment were a *batched environment*, meaning an environment that takes a batch of actions at each step and returns a batch of observations, then the driver would have to save a batch of trajectories at each step. Since we are using a TensorFlow replay buffer, it needs to know the size of the batches it will handle (to build the computation graph). An example of a batched environment is the `ParallelPyEnvironment` (from the `tf_agents.environments.parallel_py_environment` package): it runs multiple environments in parallel in separate processes (they can be different as long as they have the same action and observation specs), and at each step it takes a batch of actions and executes them in the environments (one action per environment), then it returns all the resulting observations.

21 At the time of this writing, there is no prioritized experience replay buffer yet, but one will likely be open sourced soon.

`max_length`
: The maximum size of the replay buffer. We created a large replay buffer that can store one million trajectories (as was done in the 2015 DQN paper). This will require a lot of RAM.

When we store two consecutive trajectories, they contain two consecutive observations with four frames each (since we used the `FrameStack4` wrapper), and unfortunately three of the four frames in the second observation are redundant (they are already present in the first observation). In other words, we are using about four times more RAM than necessary. To avoid this, you can instead use a `PyHashedReplayBuffer` from the `tf_agents.replay_buffers.py_hashed_replay_buffer` package: it deduplicates data in the stored trajectories along the last axis of the observations.

Now we can create the observer that will write the trajectories to the replay buffer. An observer is just a function (or a callable object) that takes a trajectory argument, so we can directly use the `add_method()` method (bound to the `replay_buffer` object) as our observer:

```
replay_buffer_observer = replay_buffer.add_batch
```

If you wanted to create your own observer, you could write any function with a `trajectory` argument. If it must have a state, you can write a class with a `__call__(self, trajectory)` method. For example, here is a simple observer that will increment a counter every time it is called (except when the trajectory represents a boundary between two episodes, which does not count as a step), and every 100 increments it displays the progress up to a given total (the carriage return `\r` along with `end=""` ensures that the displayed counter remains on the same line):

```
class ShowProgress:
    def __init__(self, total):
        self.counter = 0
        self.total = total
    def __call__(self, trajectory):
        if not trajectory.is_boundary():
            self.counter += 1
        if self.counter % 100 == 0:
            print("\r{}/{}".format(self.counter, self.total), end="")
```

Now let's create a few training metrics.

Creating Training Metrics

TF-Agents implements several RL metrics in the `tf_agents.metrics` package, some purely in Python and some based on TensorFlow. Let's create a few of them in order

to count the number of episodes, the number of steps taken, and most importantly the average return per episode and the average episode length:

```
from tf_agents.metrics import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```

Discounting the rewards makes sense for training or to implement a policy, as it makes it possible to balance the importance of immediate rewards with future rewards. However, once an episode is over, we can evaluate how good it was overalls by summing the *undiscounted* rewards. For this reason, the `AverageReturnMetric` computes the sum of undiscounted rewards for each episode, and it keeps track of the streaming mean of these sums over all the episodes it encounters.

At any time, you can get the value of each of these metrics by calling its `result()` method (e.g., `train_metrics[0].result()`). Alternatively, you can log all metrics by calling `log_metrics(train_metrics)` (this function is located in the `tf_agents.eval.metric_utils` package):

```
>>> from tf_agents.eval.metric_utils import log_metrics
>>> import logging
>>> logging.get_logger().set_level(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
AverageReturn = 0.0
AverageEpisodeLength = 0.0
```

Next, let's create the collect driver.

Creating the Collect Driver

As we explored in Figure 18-13, a driver is an object that explores an environment using a given policy, collects experiences, and broadcasts them to some observers. At each step, the following things happen:

- The driver passes the current time step to the collect policy, which uses this time step to choose an action and returns an *action step* object containing the action.

- The driver then passes the action to the environment, which returns the next time step.
- Finally, the driver creates a trajectory object to represent this transition and broadcasts it to all the observers.

Some policies, such as RNN policies, are stateful: they choose an action based on both the given time step and their own internal state. Stateful policies return their own state in the action step, along with the chosen action. The driver will then pass this state back to the policy at the next time step. Moreover, the driver saves the policy state to the trajectory (in the `policy_info` field), so it ends up in the replay buffer. This is essential when training a stateful policy: when the agent samples a trajectory, it must set the policy's state to the state it was in at the time of the sampled time step.

Also, as discussed earlier, the environment may be a batched environment, in which case the driver passes a *batched time step* to the policy (i.e., a time step object containing a batch of observations, a batch of step types, a batch of rewards, and a batch of discounts, all four batches of the same size). The driver also passes a batch of previous policy states. The policy then returns a *batched action step* containing a batch of actions and a batch of policy states. Finally, the driver creates a *batched trajectory* (i.e., a trajectory containing a batch of step types, a batch of observations, a batch of actions, a batch of rewards, and more generally a batch for each trajectory attribute, with all batches of the same size).

There are two main driver classes: `DynamicStepDriver` and `DynamicEpisodeDriver`. The first one collects experiences for a given number of steps, while the second collects experiences for a given number of episodes. We want to collect experiences for four steps for each training iteration (as was done in the 2015 DQN paper), so let's create a `DynamicStepDriver`:

```
from tf_agents.drivers.dynamic_step_driver import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period) # collect 4 steps for each training iteration
```

We give it the environment to play with, the agent's collect policy, a list of observers (including the replay buffer observer and the training metrics), and finally the number of steps to run (in this case, four). We could now run it by calling its `run()` method, but it's best to warm up the replay buffer with experiences collected using a purely random policy. For this, we can use the `RandomTFPolicy` class and create a second driver that will run this policy for 20,000 steps (which is equivalent to 80,000 simulator frames, as was done in the 2015 DQN paper). We can use our `ShowProgress` observer to display the progress:

```
from tf_agents.policies.random_tf_policy import RandomTFPolicy

initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                        tf_env.action_spec())
init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000) # <=> 80,000 ALE frames
final_time_step, final_policy_state = init_driver.run()
```

We're almost ready to run the training loop! We just need one last component: the dataset.

Creating the Dataset

To sample a batch of trajectories from the replay buffer, call its `get_next()` method. This returns the batch of trajectories plus a `BufferInfo` object that contains the sample identifiers and their sampling probabilities (this may be useful for some algorithms, such as PER). For example, the following code will sample a small batch of two trajectories (subepisodes), each containing three consecutive steps. These subepisodes are shown in Figure 18-15 (each row contains three consecutive steps from an episode):

```
>>> trajectories, buffer_info = replay_buffer.get_next(
...     sample_batch_size=2, num_steps=3)
...
>>> trajectories._fields
('step_type', 'observation', 'action', 'policy_info',
 'next_step_type', 'reward', 'discount')
>>> trajectories.observation.shape
TensorShape([2, 3, 84, 84, 4])
>>> trajectories.step_type.numpy()
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

The `trajectories` object is a named tuple, with seven fields. Each field contains a tensor whose first two dimensions are 2 and 3 (since there are two trajectories, each with three steps). This explains why the shape of the `observation` field is [2, 3, 84, 84, 4]: that's two trajectories, each with three steps, and each step's observation is 84 × 84 × 4. Similarly, the `step_type` tensor has a shape of [2, 3]: in this example, both trajectories contain three consecutive steps in the middle on an episode (types 1, 1, 1). In the second trajectory, you can barely see the ball at the lower left of the first observation, and it disappears in the next two observations, so the agent is about to lose a life, but the episode will not end immediately because it still has several lives left.

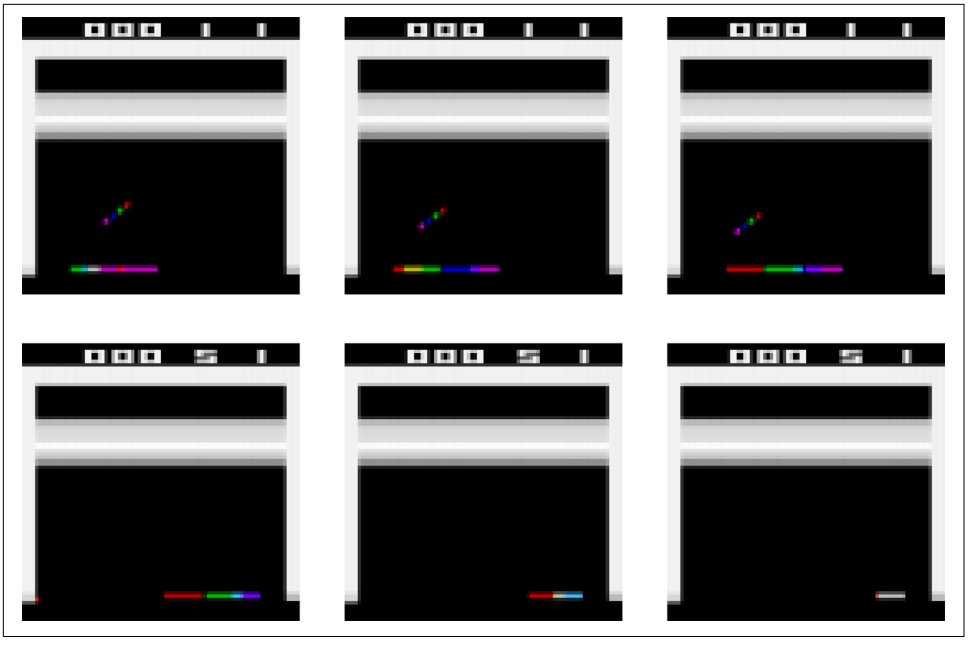


Figure 18-15. Two trajectories containing three consecutive steps each

Each trajectory is a concise representation of a sequence of consecutive time steps and action steps, designed to avoid redundancy. How so? Well, as you can see in Figure 18-16, transition n is composed of time step n, action step n, and time step $n + 1$, while transition $n + 1$ is composed of time step $n + 1$, action step $n + 1$, and time step $n + 2$. If we just stored these two transitions directly in the replay buffer, the time step $n + 1$ would be duplicated. To avoid this duplication, the n^{th} trajectory step includes only the type and observation from time step n (not its reward and discount), and it does not contain the observation from time step $n + 1$ (however, it does contain a copy of the next time step's type; that's the only duplication).

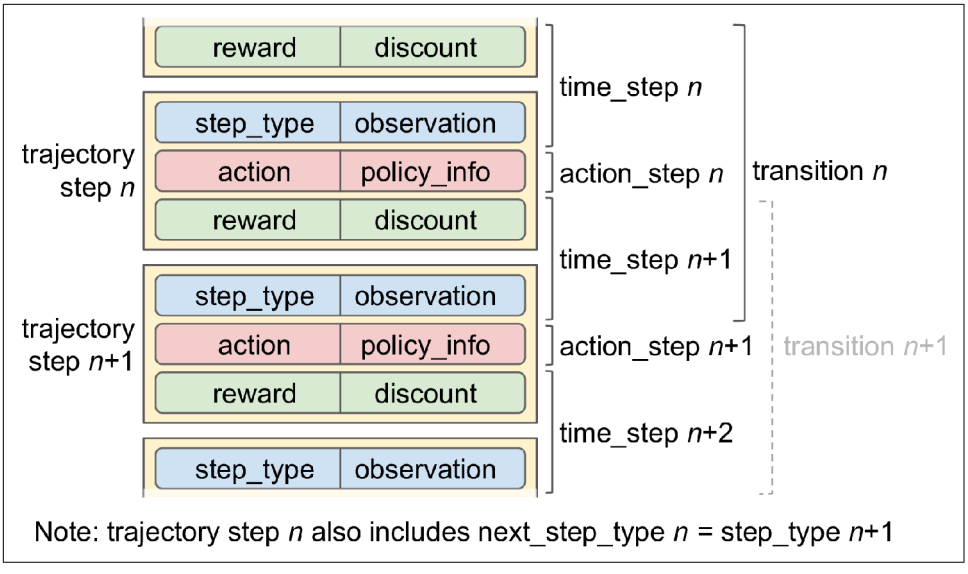


Figure 18-16. Trajectories, transitions, time steps, and action steps

So if you have a batch of trajectories where each trajectory has $t + 1$ steps (from time step n to time step $n + t$), then it contains all the data from time step n to time step $n + t$, except for the reward and discount from time step n (but it contains the reward and discount of time step $n + t + 1$). This represents t transitions (n to $n + 1$, $n + 1$ to $n + 2$, …, $n + t - 1$ to $n + t$).

The `to_transition()` function in the `tf_agents.trajectories.trajectory` module converts a batched trajectory into a list containing a batched `time_step`, a batched `action_step`, and a batched `next_time_step`. Notice that the second dimension is 2 instead of 3, since there are t transitions between $t + 1$ time steps (don't worry if you're a bit confused; you'll get the hang of it):

```
>>> from tf_agents.trajectories.trajectory import to_transition
>>> time_steps, action_steps, next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4]) # 3 time steps = 2 transitions
```

A sampled trajectory may actually overlap two (or more) episodes! In this case, it will contain *boundary transitions*, meaning transitions with a `step_type` equal to 2 (end) and a `next_step_type` equal to 0 (start). Of course, TF-Agents properly handles such trajectories (e.g., by resetting the policy state when encountering a boundary). The trajectory's `is_boundary()` method returns a tensor indicating whether each step is a boundary or not.

For our main training loop, instead of calling the `get_next()` method, we will use a `tf.data.Dataset`. This way, we can benefit from the power of the Data API (e.g., parallelism and prefetching). For this, we call the replay buffer's `as_dataset()` method:

```
dataset = replay_buffer.as_dataset(
    sample_batch_size=64,
    num_steps=2,
    num_parallel_calls=3).prefetch(3)
```

We will sample batches of 64 trajectories at each training step (as in the 2015 DQN paper), each with 2 steps (i.e., 2 steps = 1 full transition, including the next step's observation). This dataset will process three elements in parallel, and prefetch three batches.

For on-policy algorithms such as Policy Gradients, each experience should be sampled once, used from training, and then discarded. In this case, you can still use a replay buffer, but instead of using a `Dataset`, you would call the replay buffer's `gather_all()` method at each training iteration to get a tensor containing all the trajectories recorded so far, then use them to perform a training step, and finally clear the replay buffer by calling its `clear()` method.

Now that we have all the components in place, we are ready to train the model!

Creating the Training Loop

To speed up training, we will convert the main functions to TensorFlow Functions. For this we will use the `tf_agents.utils.common.function()` function, which wraps `tf.function()`, with some extra experimental options:

```
from tf_agents.utils.common import function

collect_driver.run = function(collect_driver.run)
agent.train = function(agent.train)
```

Let's create a small function that will run the main training loop for `n_iterations`:

```
def train_agent(n_iterations):
    time_step = None
    policy_state = agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state = collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
        print("\r{} loss:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)
```

The function first asks the collect policy for its initial state (given the environment batch size, which is 1 in this case). Since the policy is stateless, this returns an empty tuple (so we could have written `policy_state = ()`). Next, we create an iterator over the dataset, and we run the training loop. At each iteration, we call the driver's `run()` method, passing it the current time step (initially `None`) and the current policy state. It will run the collect policy and collect experience for four steps (as we configured earlier), broadcasting the collected trajectories to the replay buffer and the metrics. Next, we sample one batch of trajectories from the dataset, and we pass it to the agent's `train()` method. It returns a `train_loss` object which may vary depending on the type of agent. Next, we display the iteration number and the training loss, and every 1,000 iterations we log all the metrics. Now you can just call `train_agent()` for some number of iterations, and see the agent gradually learn to play *Breakout*!

```
train_agent(10000000)
```

This will take a lot of computing power and a lot of patience (it may take hours, or even days, depending on your hardware), plus you may need to run the algorithm several times with different random seeds to get good results, but once it's done, the agent will be superhuman (at least at *Breakout*). You can also try training this DQN agent on other Atari games: it can achieve superhuman skill at most action games, but it is not so good at games with long-running storylines.[22]

Overview of Some Popular RL Algorithms

Before we finish this chapter, let's take a quick look at a few popular RL algorithms:

Actor-Critic algorithms
: A family of RL algorithms that combine Policy Gradients with Deep Q-Networks. An Actor-Critic agent contains two neural networks: a policy net and a DQN. The DQN is trained normally, by learning from the agent's experiences. The policy net learns differently (and much faster) than in regular PG: instead of estimating the value of each action by going through multiple episodes, then summing the future discounted rewards for each action, and finally normalizing them, the agent (actor) relies on the action values estimated by the DQN (critic). It's a bit like an athlete (the agent) learning with the help of a coach (the DQN).

Asynchronous Advantage Actor-Critic[23] *(A3C)*
: An important Actor-Critic variant introduced by DeepMind researchers in 2016, where multiple agents learn in parallel, exploring different copies of the environ-

22 For a comparison of this algorithm's performance on various Atari games, see figure 3 in DeepMind's 2015 paper.

23 Volodymyr Mnih et al., "Asynchonous Methods for Deep Reinforcement Learning," *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.

ment. At regular intervals, but asynchronously (hence the name), each agent pushes some weight updates to a master network, then it pulls the latest weights from that network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Moreover, instead of estimating the Q-Values, the DQN estimates the advantage of each action (hence the second A in the name), which stabilizes training.

Advantage Actor-Critic (A2C)

A variant of the A3C algorithm that removes the asynchronicity. All model updates are synchronous, so gradient updates are performed over larger batches, which allows the model to better utilize the power of the GPU.

Soft Actor-Critic[24] *(SAC)*

An Actor-Critic variant proposed in 2018 by Tuomas Haarnoja and other UC Berkeley researchers. It learns not only rewards, but also to maximize the entropy of its actions. In other words, it tries to be as unpredictable as possible while still getting as many rewards as possible. This encourages the agent to explore the environment, which speeds up training, and makes it less likely to repeatedly execute the same action when the DQN produces imperfect estimates. This algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly). SAC is available in TF-Agents.

Proximal Policy Optimization (PPO)[25]

An algorithm based on A2C that clips the loss function to avoid excessively large weight updates (which often lead to training instabilities). PPO is a simplification of the previous *Trust Region Policy Optimization*[26] (TRPO) algorithm, also by John Schulman and other OpenAI researchers. OpenAI made the news in April 2019 with their AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*. PPO is also available in TF-Agents.

24 Tuomas Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.

25 John Schulman et al., "Proximal Policy Optimization Algorithms," arXiv preprint arXiv:1707.06347 (2017).

26 John Schulman et al., "Trust Region Policy Optimization," *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.

Curiosity-based exploration[27]

A recurring problem in RL is the sparsity of the rewards, which makes learning very slow and inefficient. Deepak Pathak and other UC Berkeley researchers have proposed an exciting way to tackle this issue: why not ignore the rewards, and just make the agent extremely curious to explore the environment? The rewards thus become intrinsic to the agent, rather than coming from the environment. Similarly, stimulating curiosity in a child is more likely to give good results than purely rewarding the child for getting good grades. How does this work? The agent continuously tries to predict the outcome of its actions, and it seeks situations where the outcome does not match its predictions. In other words, it wants to be surprised. If the outcome is predictable (boring), it goes elsewhere. However, if the outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while. With only curiosity, the authors succeeded in training an agent at many video games: even though the agent gets no penalty for losing, the game starts over, which is boring so it learns to avoid it.

We covered many topics in this chapter: Policy Gradients, Markov chains, Markov decision processes, Q-Learning, Approximate Q-Learning, and Deep Q-Learning and its main variants (fixed Q-Value targets, Double DQN, Dueling DQN, and prioritized experience replay). We discussed how to use TF-Agents to train agents at scale, and finally we took a quick look at a few other popular algorithms. Reinforcement Learning is a huge and exciting field, with new ideas and algorithms popping out every day, so I hope this chapter sparked your curiosity: there is a whole world to explore!

Exercises

1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
4. How do you measure the performance of a Reinforcement Learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay buffer?

27 Deepak Pathak et al., "Curiosity-Driven Exploration by Self-Supervised Prediction," *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.

7. What is an off-policy RL algorithm?
8. Use policy gradients to solve OpenAI Gym's LunarLander-v2 environment. You will need to install the Box2D dependencies (`python3 -m pip install -U gym[box2d]`).
9. Use TF-Agents to train an agent that can achieve a superhuman level at SpaceInvaders-v4 using any of the available algorithms.
10. If you have about $100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install TensorFlow on the Pi, and go wild! For an example, check out this fun post by Lukas Biewald, or take a look at GoPiGo or BrickPi. Start with simple goals, like making the robot turn around to find the brightest angle (if it has a light sensor) or the closest object (if it has a sonar sensor), and move in that direction. Then you can start using Deep Learning: for example, if the robot has a camera, you can try to implement an object detection algorithm so it detects people and moves toward them. You can also try to use RL to make the agent learn on its own how to use the motors to achieve that goal. Have fun!

Solutions to these exercises are available in Appendix A.

CHAPTER 19

Training and Deploying TensorFlow Models at Scale

Once you have a beautiful model that makes amazing predictions, what do you do with it? Well, you need to put it in production! This could be as simple as running the model on a batch of data and perhaps writing a script that runs this model every night. However, it is often much more involved. Various parts of your infrastructure may need to use this model on live data, in which case you probably want to wrap your model in a web service: this way, any part of your infrastructure can query your model at any time using a simple REST API (or some other protocol), as we discussed in Chapter 2. But as time passes, you need to regularly retrain your model on fresh data and push the updated version to production. You must handle model versioning, gracefully transition from one model to the next, possibly roll back to the previous model in case of problems, and perhaps run multiple different models in parallel to perform *A/B experiments*.[1] If your product becomes successful, your service may start to get plenty of *queries per second* (QPS), and it must scale up to support the load. A great solution to scale up your service, as we will see in this chapter, is to use TF Serving, either on your own hardware infrastructure or via a cloud service such as Google Cloud AI Platform. It will take care of efficiently serving your model, handle graceful model transitions, and more. If you use the cloud platform, you will also get many extra features, such as powerful monitoring tools.

Moreover, if you have a lot of training data, and compute-intensive models, then training time may be prohibitively long. If your product needs to adapt to changes quickly, then a long training time can be a showstopper (e.g., think of a news

1 An A/B experiment consists in testing two different versions of your product on different subsets of users in order to check which version works best and get other insights.

recommendation system promoting news from last week). Perhaps even more importantly, a long training time will prevent you from experimenting with new ideas. In Machine Learning (as in many other fields), it is hard to know in advance which ideas will work, so you should try out as many as possible, as fast as possible. One way to speed up training is to use hardware accelerators such as GPUs or TPUs. To go even faster, you can train a model across multiple machines, each equipped with multiple hardware accelerators. TensorFlow's simple yet powerful Distribution Strategies API makes this easy, as we will see.

In this chapter we will look at how to deploy models, first to TF Serving, then to Google Cloud AI Platform. We will also take a quick look at deploying models to mobile apps, embedded devices, and web apps. Lastly, we will discuss how to speed up computations using GPUs and how to train models across multiple devices and servers using the Distribution Strategies API. That's a lot of topics to discuss, so let's get started!

Serving a TensorFlow Model

Once you have trained a TensorFlow model, you can easily use it in any Python code: if it's a tf.keras model, just call its `predict()` method! But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API).[2] This decouples your model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all your software components rely on the same model versions. It also simplifies testing and development, and more. You could create your own microservice using any technology you want (e.g., using the Flask library), but why reinvent the wheel when you can just use TF Serving?

Using TensorFlow Serving

TF Serving is a very efficient, battle-tested model server that's written in C++. It can sustain a high load, serve multiple versions of your models and watch a model repository to automatically deploy the latest versions, and more (see Figure 19-1).

2 A REST (or RESTful) API is an API that uses standard HTTP verbs, such as GET, POST, PUT, and DELETE, and uses JSON inputs and outputs. The gRPC protocol is more complex but more efficient. Data is exchanged using protocol buffers (see Chapter 13).

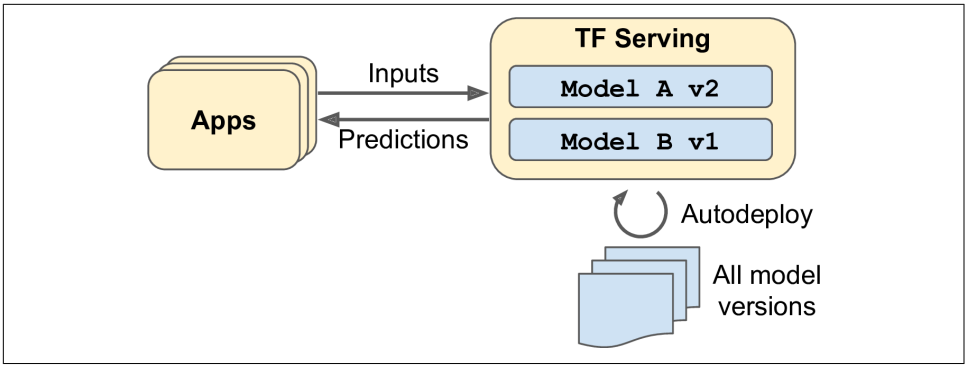


Figure 19-1. TF Serving can serve multiple models and automatically deploy the latest version of each model

So let's suppose you have trained an MNIST model using tf.keras, and you want to deploy it to TF Serving. The first thing you have to do is export this model to TensorFlow's *SavedModel format*.

Exporting SavedModels

TensorFlow provides a simple `tf.saved_model.save()` function to export models to the SavedModel format. All you need to do is give it the model, specifying its name and version number, and the function will save the model's computation graph and its weights:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

Alternatively, you can just use the model's `save()` method (`model.save(model_path)`): as long as the file's extension is not `.h5`, the model will be saved using the SavedModel format instead of the HDF5 format.

It's usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production. This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them later on and limits the risk of mismatch between a model and the preprocessing steps it requires.

Since a SavedModel saves the computation graph, it can only be used with models that are based exclusively on TensorFlow operations, excluding the `tf.py_function()` operation (which wraps arbitrary Python code). It also excludes dynamic tf.keras models (see Appendix G), since these models cannot be converted to computation graphs. Dynamic models need to be served using other tools (e.g., Flask).

A SavedModel represents a version of your model. It is stored as a directory containing a *saved_model.pb* file, which defines the computation graph (represented as a serialized protocol buffer), and a *variables* subdirectory containing the variable values. For models containing a large number of weights, these variable values may be split across multiple files. A SavedModel also includes an *assets* subdirectory that may contain additional data, such as vocabulary files, class names, or some example instances for this model. The directory structure is as follows (in this example, we don't use assets):

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

As you might expect, you can load a SavedModel using the `tf.saved_model.load()` function. However, the returned object is not a Keras model: it represents the SavedModel, including its computation graph and variable values. You can use it like a function, and it will make predictions (make sure to pass the inputs as tensors of the appropriate type):

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(tf.constant(X_new, dtype=tf.float32))
```

Alternatively, you can load this SavedModel directly to a Keras model using the `keras.models.load_model()` function:

```
model = keras.models.load_model(model_path)
y_pred = model.predict(tf.constant(X_new, dtype=tf.float32))
```

TensorFlow also comes with a small `saved_model_cli` command-line tool to inspect SavedModels:

```
$ export ML_PATH="$HOME/ml" # point to this project, wherever it is
$ cd $ML_PATH
$ saved_model_cli show --dir my_mnist_model/0001 --all
MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:
signature_def['__saved_model_init_op']:
  [...]
```

```
signature_def['serving_default']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['flatten_input'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 28, 28)
        name: serving_default_flatten_input:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['dense_1'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 10)
        name: StatefulPartitionedCall:0
  Method name is: tensorflow/serving/predict
```

A SavedModel contains one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions (including their input and output names, types, and shapes). Each metagraph is identified by a set of tags. For example, you may want to have a metagraph containing the full computation graph, including the training operations (this one may be tagged `"train"`, for example), and another metagraph containing a pruned computation graph with only the prediction operations, including some GPU-specific operations (this metagraph may be tagged `"serve", "gpu"`). However, when you pass a tf.keras model to the `tf.saved_model.save()` function, by default the function saves a much simpler SavedModel: it saves a single metagraph tagged `"serve"`, which contains two signature definitions, an initialization function (called `__saved_model_init_op`, which you do not need to worry about) and a default serving function (called `serving_default`). When saving a tf.keras model, the default serving function corresponds to the model's `call()` function, which of course makes predictions.

The `saved_model_cli` tool can also be used to make predictions (for testing, not really for production). Suppose you have a NumPy array (`X_new`) containing three images of handwritten digits that you want to make predictions for. You first need to export them to NumPy's `npy` format:

```
np.save("my_mnist_tests.npy", X_new)
```

Next, use the `saved_model_cli` command like this:

```
$ saved_model_cli run --dir my_mnist_model/0001 --tag_set serve \
                      --signature_def serving_default \
                      --inputs flatten_input=my_mnist_tests.npy
[...] Result for output key dense_1:
[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

The tool's output contains the 10 class probabilities of each of the 3 instances. Great! Now that you have a working SavedModel, the next step is to install TF Serving.

Installing TensorFlow Serving

There are many ways to install TF Serving: using a Docker image,[3] using the system's package manager, installing from source, and more. Let's use the Docker option, which is highly recommended by the TensorFlow team as it is simple to install, it will not mess with your system, and it offers high performance. You first need to install Docker. Then download the official TF Serving Docker image:

```
$ docker pull tensorflow/serving
```

Now you can create a Docker container to run this image:

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
             -v "$ML_PATH/my_mnist_model:/models/my_mnist_model" \
             -e MODEL_NAME=my_mnist_model \
             tensorflow/serving
[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model version: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
```

That's it! TF Serving is running. It loaded our MNIST model (version 1), and it is serving it through both gRPC (on port 8500) and REST (on port 8501). Here is what all the command-line options mean:

`-it`
: Makes the container interactive (so you can press Ctrl-C to stop it) and displays the server's output.

`--rm`
: Deletes the container when you stop it (no need to clutter your machine with interrupted containers). However, it does not delete the image.

`-p 8500:8500`
: Makes the Docker engine forward the host's TCP port 8500 to the container's TCP port 8500. By default, TF Serving uses this port to serve the gRPC API.

`-p 8501:8501`
: Forwards the host's TCP port 8501 to the container's TCP port 8501. By default, TF Serving uses this port to serve the REST API.

3 If you are not familiar with Docker, it allows you to easily download a set of applications packaged in a *Docker image* (including all their dependencies and usually some good default configuration) and then run them on your system using a *Docker engine*. When you run an image, the engine creates a *Docker container* that keeps the applications well isolated from your own system (but you can give it some limited access if you want). It is similar to a virtual machine, but much faster and more lightweight, as the container relies directly on the host's kernel. This means that the image does not need to include or run its own kernel.

`-v "$ML_PATH/my_mnist_model:/models/my_mnist_model"`
: Makes the host's `$ML_PATH/my_mnist_model` directory available to the container at the path */models/mnist_model*. On Windows, you may need to replace `/` with `\` in the host path (but not in the container path).

`-e MODEL_NAME=my_mnist_model`
: Sets the container's `MODEL_NAME` environment variable, so TF Serving knows which model to serve. By default, it will look for models in the */models* directory, and it will automatically serve the latest version it finds.

`tensorflow/serving`
: This is the name of the image to run.

Now let's go back to Python and query this server, first using the REST API, then the gRPC API.

Querying TF Serving through the REST API

Let's start by creating the query. It must contain the name of the function signature you want to call, and of course the input data:

```
import json

input_data_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Note that the JSON format is 100% text-based, so the `X_new` NumPy array had to be converted to a Python list and then formatted as JSON:

```
>>> input_data_json
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0, [...]
0.3294117647058824, 0.725490196078431, [...very long], 0.0, 0.0, 0.0, 0.0]]]}'
```

Now let's send the input data to TF Serving by sending an HTTP POST request. This can be done easily using the `requests` library (it is not part of Python's standard library, so you will need to install it first, e.g., using pip):

```
import requests

SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'
response = requests.post(SERVER_URL, data=input_data_json)
response.raise_for_status() # raise an exception in case of error
response = response.json()
```

The response is a dictionary containing a single `"predictions"` key. The corresponding value is the list of predictions. This list is a Python list, so let's convert it to a NumPy array and round the floats it contains to the second decimal:

```
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

Hurray, we have the predictions! The model is close to 100% confident that the first image is a 7, 99% confident that the second image is a 2, and 96% confident that the third image is a 1.

The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can make REST queries without additional dependencies, whereas other protocols are not always so readily available. However, it is based on JSON, which is text-based and fairly verbose. For example, we had to convert the NumPy array to a Python list, and every float ended up represented as a string. This is very inefficient, both in terms of serialization/deserialization time (to convert all the floats to strings and back) and in terms of payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth usage when transferring large NumPy arrays.[4] So let's use gRPC instead.

When transferring large amounts of data, it is much better to use the gRPC API (if the client supports it), as it is based on a compact binary format and an efficient communication protocol (based on HTTP/2 framing).

Querying TF Serving through the gRPC API

The gRPC API expects a serialized `PredictRequest` protocol buffer as input, and it outputs a serialized `PredictResponse` protocol buffer. These protobufs are part of the `tensorflow-serving-api` library, which you must install (e.g., using pip). First, let's create the request:

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

This code creates a `PredictRequest` protocol buffer and fills in the required fields, including the model name (defined earlier), the signature name of the function we

4 To be fair, this can be mitigated by serializing the data first and encoding it to Base64 before creating the REST request. Moreover, REST requests can be compressed using gzip, which reduces the payload size significantly.

want to call, and finally the input data, in the form of a `Tensor` protocol buffer. The `tf.make_tensor_proto()` function creates a `Tensor` protocol buffer based on the given tensor or NumPy array, in this case `X_new`.

Next, we'll send the request to the server and get its response (for this you will need the `grpcio` library, which you can install using pip):

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

The code is quite straightforward: after the imports, we create a gRPC communication channel to *localhost* on TCP port 8500, then we create a gRPC service over this channel and use it to send a request, with a 10-second timeout (not that the call is synchronous: it will block until it receives the response or the timeout period expires). In this example the channel is insecure (no encryption, no authentication), but gRPC and TensorFlow Serving also support secure channels over SSL/TLS.

Next, let's convert the `PredictResponse` protocol buffer to a tensor:

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

If you run this code and print `y_proba.numpy().round(2)`, you will get the exact same estimated class probabilities as earlier. And that's all there is to it: in just a few lines of code, you can now access your TensorFlow model remotely, using either REST or gRPC.

Deploying a new model version

Now let's create a new model version and export a SavedModel to the *my_mnist_model/0002* directory, just like earlier:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0002"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

At regular intervals (the delay is configurable), TensorFlow Serving checks for new model versions. If it finds one, it will automatically handle the transition gracefully: by default, it will answer pending requests (if any) with the previous model version,

while handling new requests with the new version.[5] As soon as every pending request has been answered, the previous model version is unloaded. You can see this at work in the TensorFlow Serving logs:

```
[...]
reserved resources to load servable {name: my_mnist_model version: 2}
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```

This approach offers a smooth transition, but it may use too much RAM (especially GPU RAM, which is generally the most limited). In this case, you can configure TF Serving so that it handles all pending requests with the previous model version and unloads it before loading and using the new model version. This configuration will avoid having two model versions loaded at the same time, but the service will be unavailable for a short period.

As you can see, TF Serving makes it quite simple to deploy new models. Moreover, if you discover that version 2 does not work as well as you expected, then rolling back to version 1 is as simple as removing the *my_mnist_model/0002* directory.

Another great feature of TF Serving is its automatic batching capability, which you can activate using the `--enable_batching` option upon startup. When TF Serving receives multiple requests within a short period of time (the delay is configurable), it will automatically batch them together before using the model. This offers a significant performance boost by leveraging the power of the GPU. Once the model returns the predictions, TF Serving dispatches each prediction to the right client. You can trade a bit of latency for a greater throughput by increasing the batching delay (see the `--batching_parameters_file` option).

If you expect to get many queries per second, you will want to deploy TF Serving on multiple servers and load-balance the queries (see Figure 19-2). This will require deploying and managing many TF Serving containers across these servers. One way to handle that is to use a tool such as Kubernetes, which is an open source system for simplifying container orchestration across many servers. If you do not want to pur-

5 If the SavedModel contains some example instances in the *assets/extra* directory, you can configure TF Serving to execute the model on these instances before starting to serve new requests with it. This is called *model warmup*: it will ensure that everything is properly loaded, avoiding long response times for the first requests.

chase, maintain, and upgrade all the hardware infrastructure, you will want to use virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud, or some other Platform-as-a-Service (PaaS). Managing all the virtual machines, handling container orchestration (even with the help of Kubernetes), taking care of TF Serving configuration, tuning and monitoring—all of this can be a full-time job. Fortunately, some service providers can take care of all this for you. In this chapter we will use Google Cloud AI Platform because it's the only platform with TPUs today, it supports TensorFlow 2, it offers a nice suite of AI services (e.g., AutoML, Vision API, Natural Language API), and it is the one I have the most experience with. But there are several other providers in this space, such as Amazon AWS SageMaker and Microsoft AI Platform, which are also capable of serving TensorFlow models.

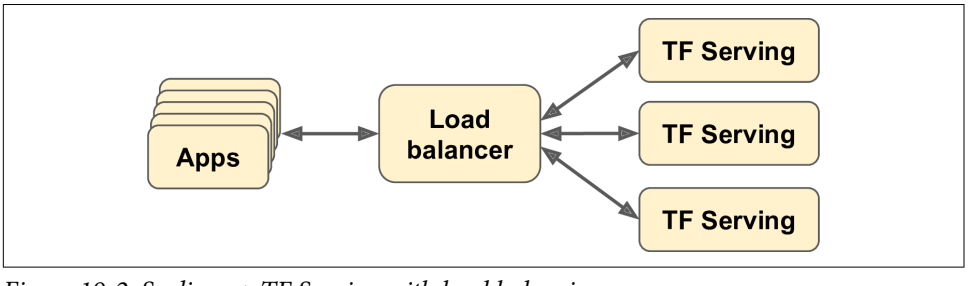


Figure 19-2. Scaling up TF Serving with load balancing

Now let's see how to serve our wonderful MNIST model on the cloud!

Creating a Prediction Service on GCP AI Platform

Before you can deploy a model, there's a little bit of setup to take care of:

1. Log in to your Google account, and then go to the Google Cloud Platform (GCP) console (see Figure 19-3). If you don't have a Google account, you'll have to create one.
2. If it is your first time using GCP, you will have to read and accept the terms and conditions. Click Tour Console if you want. At the time of this writing, new users are offered a free trial, including $300 worth of GCP credit that you can use over the course of 12 months. You will only need a small portion of that to pay for the services you will use in this chapter. Upon signing up for the free trial, you will still need to create a payment profile and enter your credit card number: it is used for verification purposes (probably to avoid people using the free trial multiple times), but you will not be billed. Activate and upgrade your account if requested.

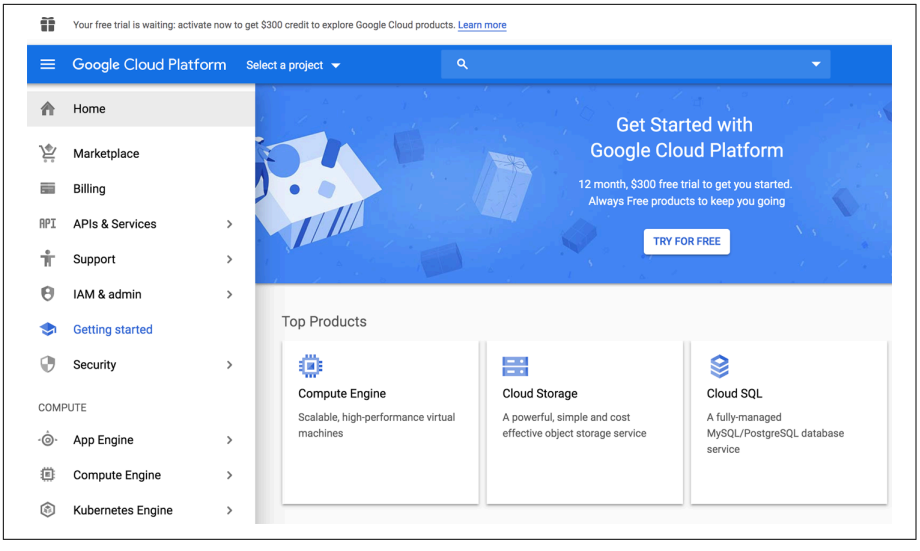


Figure 19-3. Google Cloud Platform console

3. If you have used GCP before and your free trial has expired, then the services you will use in this chapter will cost you some money. It should not be too much, especially if you remember to turn off the services when you do not need them anymore. Make sure you understand and agree to the pricing conditions before you run any service. I hereby decline any responsibility if services end up costing more than you expected! Also make sure your billing account is active. To check, open the navigation menu on the left and click Billing, and make sure you have set up a payment method and that the billing account is active.
4. Every resource in GCP belongs to a project. This includes all the virtual machines you may use, the files you store, and the training jobs you run. When you create an account, GCP automatically creates a project for you, called "My First Project." If you want, you can change its display name by going to the project settings: in the navigation menu (on the left of the screen), select IAM & admin → Settings, change the project's display name, and click Save. Note that the project also has a unique ID and number. You can choose the project ID when you create a project, but you cannot change it later. The project number is automatically generated and cannot be changed. If you want to create a new project, click the project name at the top of the page, then click New Project and enter the project ID. Make sure billing is active for this new project.

Always set an alarm to remind yourself to turn services off when you know you will only need them for a few hours, or else you might leave them running for days or months, incurring potentially significant costs.

5. Now that you have a GCP account with billing activated, you can start using the services. The first one you will need is Google Cloud Storage (GCS): this is where you will put the SavedModels, the training data, and more. In the navigation menu, scroll down to the Storage section, and click Storage → Browser. All your files will go in one or more *buckets*. Click Create Bucket and choose the bucket name (you may need to activate the Storage API first). GCS uses a single world-wide namespace for buckets, so simple names like "machine-learning" will most likely not be available. Make sure the bucket name conforms to DNS naming conventions, as it may be used in DNS records. Moreover, bucket names are public, so do not put anything private in there. It is common to use your domain name or your company name as a prefix to ensure uniqueness, or simply use a random number as part of the name. Choose the location where you want the bucket to be hosted, and the rest of the options should be fine by default. Then click Create.
6. Upload the *my_mnist_model* folder you created earlier (including one or more versions) to your bucket. To do this, just go to the GCS Browser, click the bucket, then drag and drop the *my_mnist_model* folder from your system to the bucket (see Figure 19-4). Alternatively, you can click "Upload folder" and select the *my_mnist_model* folder to upload. By default, the maximum size for a SavedModel is 250 MB, but it is possible to request a higher quota.

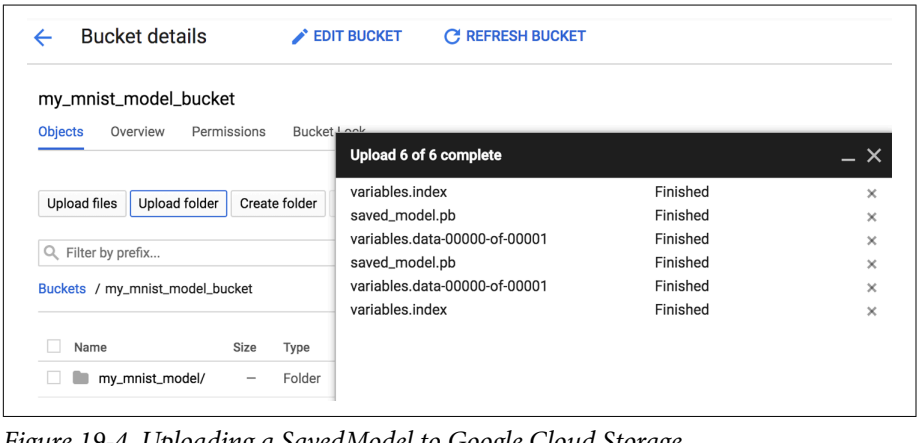


Figure 19-4. Uploading a SavedModel to Google Cloud Storage

7. Now you need to configure AI Platform (formerly known as ML Engine) so that it knows which models and versions you want to use. In the navigation menu, scroll down to the Artificial Intelligence section, and click AI Platform → Models. Click Activate API (it takes a few minutes), then click "Create model." Fill in the model details (see Figure 19-5) and click Create.

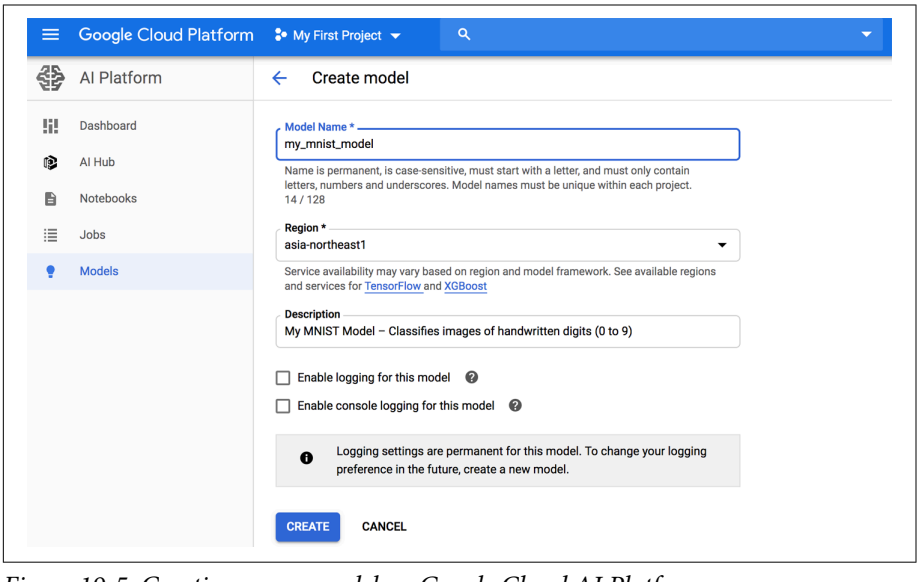


Figure 19-5. Creating a new model on Google Cloud AI Platform

8. Now that you have a model on AI Platform, you need to create a model version. In the list of models, click the model you just created, then click "Create version" and fill in the version details (see Figure 19-6): set the name, description, Python version (3.5 or above), framework (TensorFlow), framework version (2.0 if available, or 1.13),[6] ML runtime version (2.0, if available or 1.13), machine type (choose "Single core CPU" for now), model path on GCS (this is the full path to the actual version folder, e.g., *gs://my-mnist-model-bucket/my_mnist_model/0002/*), scaling (choose automatic), and minimum number of TF Serving containers to have running at all times (leave this field empty). Then click Save.

6 At the time of this writing, TensorFlow version 2 is not available yet on AI Platform, but that's OK: you can use 1.13, and it will run your TF 2 SavedModels just fine.

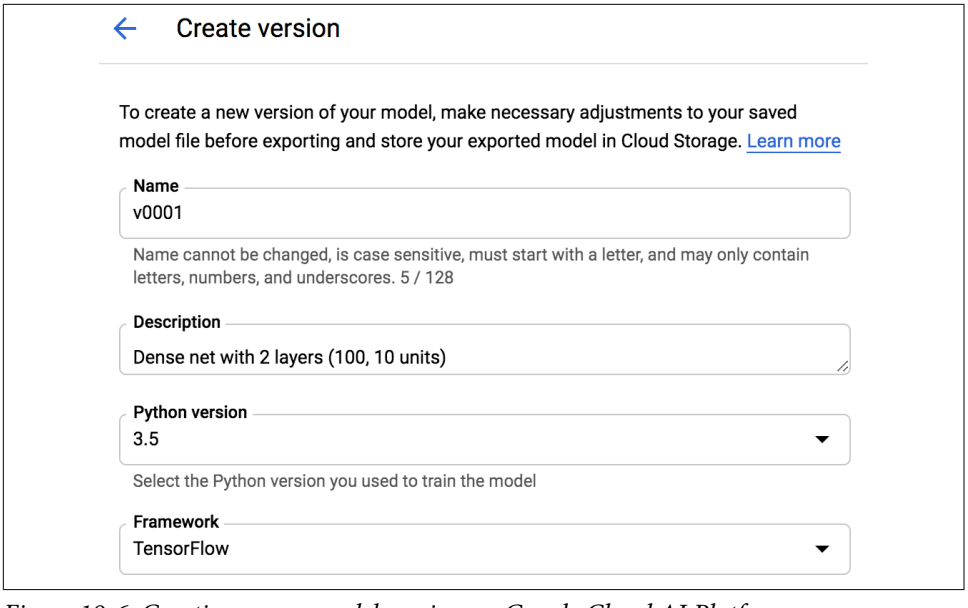


Figure 19-6. Creating a new model version on Google Cloud AI Platform

Congratulations, you have deployed your first model on the cloud! Because you selected automatic scaling, AI Platform will start more TF Serving containers when the number of queries per second increases, and it will load-balance the queries between them. If the QPS goes down, it will stop containers automatically. The cost is therefore directly linked to the QPS (as well as the type of machine you choose and the amount of data you store on GCS). This pricing model is particularly useful for occasional users and for services with important usage spikes, as well as for startups: the price remains low until the startup actually starts up.

If you do not use the prediction service, AI Platform will stop all containers. This means you will only pay for the amount of storage you use (a few cents per gigabyte per month). Note that when you query the service, AI Platform will need to start up a TF Serving container, which will take a few seconds. If this delay is unacceptable, you will have to set the minimum number of TF Serving containers to 1 when creating the model version. Of course, this means at least one machine will run constantly, so the monthly fee will be higher.

Now let's query this prediction service!

Using the Prediction Service

Under the hood, AI Platform just runs TF Serving, so in principle you could use the same code as earlier, if you knew which URL to query. There's just one problem: GCP also takes care of encryption and authentication. Encryption is based on SSL/TLS, and authentication is token-based: a secret authentication token must be sent to the server in every request. So before your code can use the prediction service (or any other GCP service), it must obtain a token. We will see how to do this shortly, but first you need to configure authentication and give your application the appropriate access rights on GCP. You have two options for authentication:

- Your application (i.e., the client code that will query the prediction service) could authenticate using user credentials with your own Google login and password. Using user credentials would give your application the exact same rights as on GCP, which is certainly way more than it needs. Moreover, you would have to deploy your credentials in your application, so anyone with access could steal your credentials and fully access your GCP account. In short, do not choose this option; it is only needed in very rare cases (e.g., when your application needs to access its user's GCP account).
- The client code can authenticate with a *service account*. This is an account that represents an application, not a user. It is generally given very restricted access rights: strictly what it needs, and no more. This is the recommended option.

So, let's create a service account for your application: in the navigation menu, go to IAM & admin → Service accounts, then click Create Service Account, fill in the form (service account name, ID, description), and click Create (see Figure 19-7). Next, you must give this account some access rights. Select the ML Engine Developer role: this will allow the service account to make predictions, and not much more. Optionally, you can grant some users access to the service account (this is useful when your GCP user account is part of an organization, and you wish to authorize other users in the organization to deploy applications that will be based on this service account or to manage the service account itself). Next, click Create Key to export the service account's private key, choose JSON, and click Create. This will download the private key in the form of a JSON file. Make sure to keep it private!

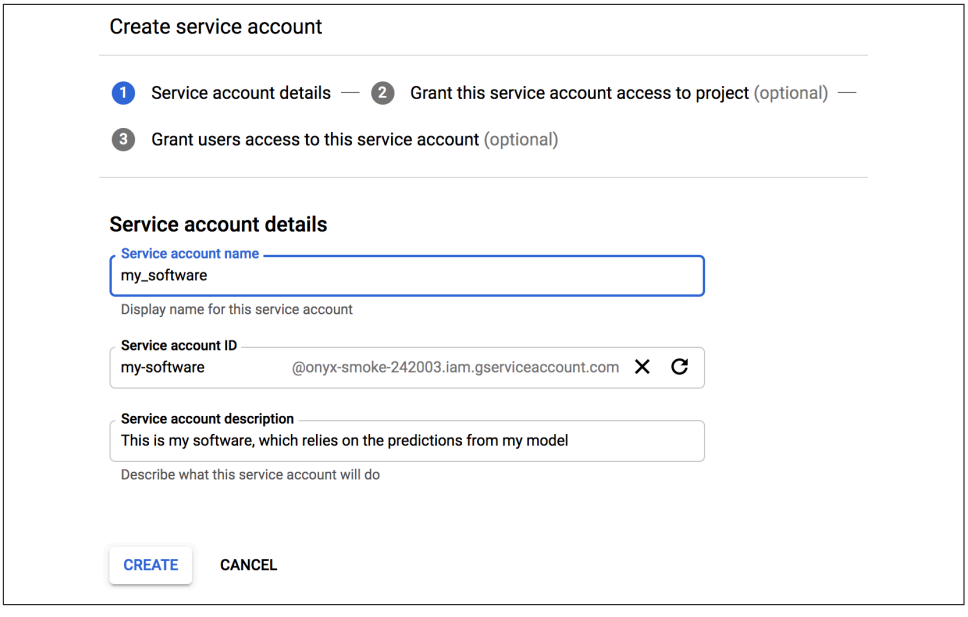


Figure 19-7. Creating a new service account in Google IAM

Great! Now let's write a small script that will query the prediction service. Google provides several libraries to simplify access to its services:

Google API Client Library
: This is a fairly thin layer on top of *OAuth 2.0* (for the authentication) and REST. You can use it with all GCP services, including AI Platform. You can install it using pip: the library is called `google-api-python-client`.

Google Cloud Client Libraries
: These are a bit more high-level: each one is dedicated to a particular service, such as GCS, Google BigQuery, Google Cloud Natural Language, and Google Cloud Vision. All these libraries can be installed using pip (e.g., the GCS Client Library is called `google-cloud-storage`). When a client library is available for a given service, it is recommended to use it rather than the Google API Client Library, as it implements all the best practices and will often use gRPC rather than REST, for better performance.

At the time of this writing there is no client library for AI Platform, so we will use the Google API Client Library. It will need to use the service account's private key; you can tell it where it is by setting the `GOOGLE_APPLICATION_CREDENTIALS` environment variable, either before starting the script or within the script like this:

```
import os

os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "my_service_account_key.json"
```

If you deploy your application to a virtual machine on Google Cloud Engine (GCE), or within a container using Google Cloud Kubernetes Engine, or as a web application on Google Cloud App Engine, or as a microservice on Google Cloud Functions, and if the `GOOGLE_APPLICATION_CREDENTIALS` environment variable is not set, then the library will use the default service account for the host service (e.g., the default GCE service account, if your application runs on GCE).

Next, you must create a resource object that wraps access to the prediction service:[7]

```
import googleapiclient.discovery

project_id = "onyx-smoke-242003" # change this to your project ID
model_id = "my_mnist_model"
model_path = "projects/{}/models/{}".format(project_id, model_id)
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

Note that you can append `/versions/0001` (or any other version number) to the `model_path` to specify the version you want to query: this can be useful for A/B testing or for testing a new version on a small group of users before releasing it widely (this is called a *canary*). Next, let's write a small function that will use the resource object to call the prediction service and get the predictions back:

```
def predict(X):
    input_data_json = {"signature_name": "serving_default",
                       "instances": X.tolist()}
    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()
    if "error" in response:
        raise RuntimeError(response["error"])
    return np.array([pred[output_name] for pred in response["predictions"]])
```

The function takes a NumPy array containing the input images and prepares a dictionary that the client library will convert to the JSON format (as we did earlier). Then it prepares a prediction request, and executes it; it raises an exception if the response contains an error, or else it extracts the predictions for each instance and bundles them in a NumPy array. Let's see if it works:

```
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.96, 0.01, 0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  ]])
```

7 If you get an error saying that module `google.appengine` was not found, set `cache_discovery=False` in the call to the `build()` method; see *https://stackoverflow.com/q/55561354*.

Yes! You now have a nice prediction service running on the cloud that can automatically scale up to any number of QPS, plus you can query it from anywhere securely. Moreover, it costs you close to nothing when you don't use it: you'll pay just a few cents per month per gigabyte used on GCS. And you can also get detailed logs and metrics using Google Stackdriver.

But what if you want to deploy your model to a mobile app? Or to an embedded device?

Deploying a Model to a Mobile or Embedded Device

If you need to deploy your model to a mobile or embedded device, a large model may simply take too long to download and use too much RAM and CPU, all of which will make your app unresponsive, heat the device, and drain its battery. To avoid this, you need to make a mobile-friendly, lightweight, and efficient model, without sacrificing too much of its accuracy. The TFLite library provides several tools[8] to help you deploy your models to mobile and embedded devices, with three main objectives:

- Reduce the model size, to shorten download time and reduce RAM usage.
- Reduce the amount of computations needed for each prediction, to reduce latency, battery usage, and heating.
- Adapt the model to device-specific constraints.

To reduce the model size, TFLite's model converter can take a SavedModel and compress it to a much lighter format based on FlatBuffers. This is an efficient cross-platform serialization library (a bit like protocol buffers) initially created by Google for gaming. It is designed so you can load FlatBuffers straight to RAM without any preprocessing: this reduces the loading time and memory footprint. Once the model is loaded into a mobile or embedded device, the TFLite interpreter will execute it to make predictions. Here is how you can convert a SavedModel to a FlatBuffer and save it to a *.tflite* file:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```

You can also save a tf.keras model directly to a FlatBuffer using `from_keras_model()`.

8 Also check out TensorFlow's Graph Transform Tools for modifying and optimizing computational graphs.

The converter also optimizes the model, both to shrink it and to reduce its latency. It prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes computations whenever possible; for example, $3 \times a + 4 \times a + 5 \times a$ will be converted to $(3 + 4 + 5) \times a$. It also tries to fuse operations whenever possible. For example, Batch Normalization layers end up folded into the previous layer's addition and multiplication operations, whenever possible. To get a good idea of how much TFLite can optimize a model, download one of the pretrained TFLite models, unzip the archive, then open the excellent Netron graph visualization tool and upload the *.pb* file to view the original model. It's a big, complex graph, right? Next, open the optimized *.tflite* model and marvel at its beauty!

Another way you can reduce the model size (other than simply using smaller neural network architectures) is by using smaller bit-widths: for example, if you use half-floats (16 bits) rather than regular floats (32 bits), the model size will shrink by a factor of 2, at the cost of a (generally small) accuracy drop. Moreover, training will be faster, and you will use roughly half the amount of GPU RAM.

TFLite's converter can go further than that, by quantizing the model weights down to fixed-point, 8-bit integers! This leads to a fourfold size reduction compared to using 32-bit floats. The simplest approach is called *post-training quantization*: it just quantizes the weights after training, using a fairly basic but efficient symmetrical quantization technique. It finds the maximum absolute weight value, m, then it maps the floating-point range $-m$ to $+m$ to the fixed-point (integer) range –127 to +127. For example (see Figure 19-8), if the weights range from –1.5 to +0.8, then the bytes –127, 0, and +127 will correspond to the floats –1.5, 0.0, and +1.5, respectively. Note that 0.0 always maps to 0 when using symmetrical quantization (also note that the byte values +68 to +127 will not be used, since they map to floats greater than +0.8).

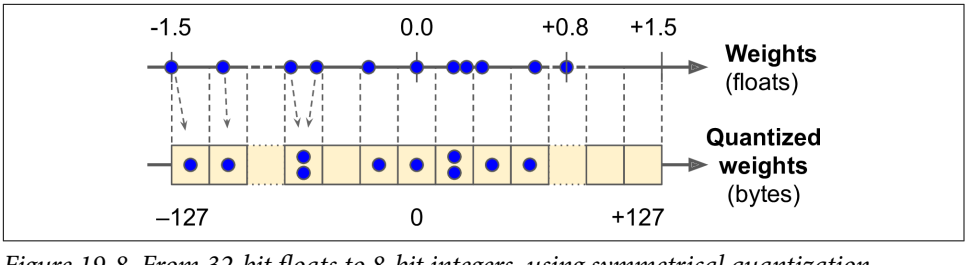


Figure 19-8. From 32-bit floats to 8-bit integers, using symmetrical quantization

To perform this post-training quantization, simply add `OPTIMIZE_FOR_SIZE` to the list of converter optimizations before calling the `convert()` method:

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

This technique dramatically reduces the model's size, so it's much faster to download and store. However, at runtime the quantized weights get converted back to floats before they are used (these recovered floats are not perfectly identical to the original

floats, but not too far off, so the accuracy loss is usually acceptable). To avoid recomputing them all the time, the recovered floats are cached, so there is no reduction of RAM usage. And there is no reduction either in compute speed.

The most effective way to reduce latency and power consumption is to also quantize the activations so that the computations can be done entirely with integers, without the need for any floating-point operations. Even when using the same bit-width (e.g., 32-bit integers instead of 32-bit floats), integer computations use less CPU cycles, consume less energy, and produce less heat. And if you also reduce the bit-width (e.g., down to 8-bit integers), you can get huge speedups. Moreover, some neural network accelerator devices (such as the Edge TPU) can only process integers, so full quantization of both weights and activations is compulsory. This can be done post-training; it requires a calibration step to find the maximum absolute value of the activations, so you need to provide a representative sample of training data to TFLite (it does not need to be huge), and it will process the data through the model and measure the activation statistics required for quantization (this step is typically fast).

The main problem with quantization is that it loses a bit of accuracy: it is equivalent to adding noise to the weights and activations. If the accuracy drop is too severe, then you may need to use *quantization-aware training*. This means adding fake quantization operations to the model so it can learn to ignore the quantization noise during training; the final weights will then be more robust to quantization. Moreover, the calibration step can be taken care of automatically during training, which simplifies the whole process.

I have explained the core concepts of TFLite, but going all the way to coding a mobile app or an embedded program would require a whole other book. Fortunately, one exists: if you want to learn more about building TensorFlow applications for mobile and embedded devices, check out the O'Reilly book *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers*, by Pete Warden (who leads the TFLite team) and Daniel Situnayake.

TensorFlow in the Browser

What if you want to use your model in a website, running directly in the user's browser? This can be useful in many scenarios, such as:

- When your web application is often used in situations where the user's connectivity is intermittent or slow (e.g., a website for hikers), so running the model directly on the client side is the only way to make your website reliable.
- When you need the model's responses to be as fast as possible (e.g., for an online game). Removing the need to query the server to make predictions will definitely reduce the latency and make the website much more responsive.
- When your web service makes predictions based on some private user data, and you want to protect the user's privacy by making the predictions on the client side so that the private data never has to leave the user's machine.[9]

For all these scenarios, you can export your model to a special format that can be loaded by the TensorFlow.js JavaScript library. This library can then use your model to make predictions directly in the user's browser. The TensorFlow.js project includes a `tensorflowjs_converter` tool that can convert a TensorFlow SavedModel or a Keras model file to the *TensorFlow.js Layers* format: this is a directory containing a set of sharded weight files in binary format and a *model.json* file that describes the model's architecture and links to the weight files. This format is optimized to be downloaded efficiently on the web. Users can then download the model and run predictions in the browser using the TensorFlow.js library. Here is a code snippet to give you an idea of what the JavaScript API looks like:

```
import * as tf from '@tensorflow/tfjs';
const model = await tf.loadLayersModel('https://example.com/tfjs/model.json');
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
```

Once again, doing justice to this topic would require a whole book. If you want to learn more about TensorFlow.js, check out the O'Reilly book *Practical Deep Learning for Cloud, Mobile, and Edge*, by Anirudh Koul, Siddha Ganju, and Meher Kasam.

Next, we will see how to use GPUs to speed up computations!

9 If you're interested in this topic, check out *federated learning*.

Using GPUs to Speed Up Computations

In Chapter 11 we discussed several techniques that can considerably speed up training: better weight initialization, Batch Normalization, sophisticated optimizers, and so on. But even with all of these techniques, training a large neural network on a single machine with a single CPU can take days or even weeks.

In this section we will look at how to speed up your models by using GPUs. We will also see how to split the computations across multiple devices, including the CPU and multiple GPU devices (see Figure 19-9). For now we will run everything on a single machine, but later in this chapter we will discuss how to distribute computations across multiple servers.

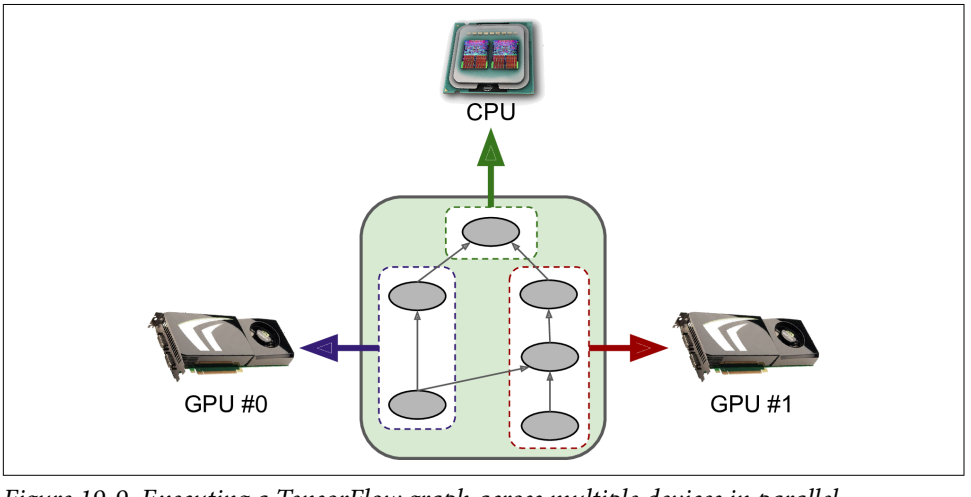


Figure 19-9. Executing a TensorFlow graph across multiple devices in parallel

Thanks to GPUs, instead of waiting for days or weeks for a training algorithm to complete, you may end up waiting for just a few minutes or hours. Not only does this save an enormous amount of time, but it also means that you can experiment with various models much more easily and frequently retrain your models on fresh data.

You can often get a major performance boost simply by adding GPU cards to a single machine. In fact, in many cases this will suffice; you won't need to use multiple machines at all. For example, you can typically train a neural network just as fast using four GPUs on a single machine rather than eight GPUs across multiple machines, due to the extra delay imposed by network communications in a distributed setup. Similarly, using a single powerful GPU is often preferable to using multiple slower GPUs.

The first step is to get your hands on a GPU. There are two options for this: you can either purchase your own GPU(s), or you can use GPU-equipped virtual machines on the cloud. Let's start with the first option.

Getting Your Own GPU

If you choose to purchase a GPU card, then take some time to make the right choice. Tim Dettmers wrote an excellent blog post to help you choose, and he updates it regularly: I encourage you to read it carefully. At the time of this writing, TensorFlow only supports Nvidia cards with CUDA Compute Capability 3.5+ (as well as Google's TPUs, of course), but it may extend its support to other manufacturers. Moreover, although TPUs are currently only available on GCP, it is highly likely that TPU-like cards will be available for sale in the near future, and TensorFlow may support them. In short, make sure to check TensorFlow's documentation to see what devices are supported at this point.

If you go for an Nvidia GPU card, you will need to install the appropriate Nvidia drivers and several Nvidia libraries.[10] These include the *Compute Unified Device Architecture* library (CUDA), which allows developers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration), and the *CUDA Deep Neural Network* library (cuDNN), a GPU-accelerated library of primitives for DNNs. cuDNN provides optimized implementations of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see Chapter 14). It is part of Nvidia's Deep Learning SDK (note that you'll need to create an Nvidia developer account in order to download it). TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see Figure 19-10).

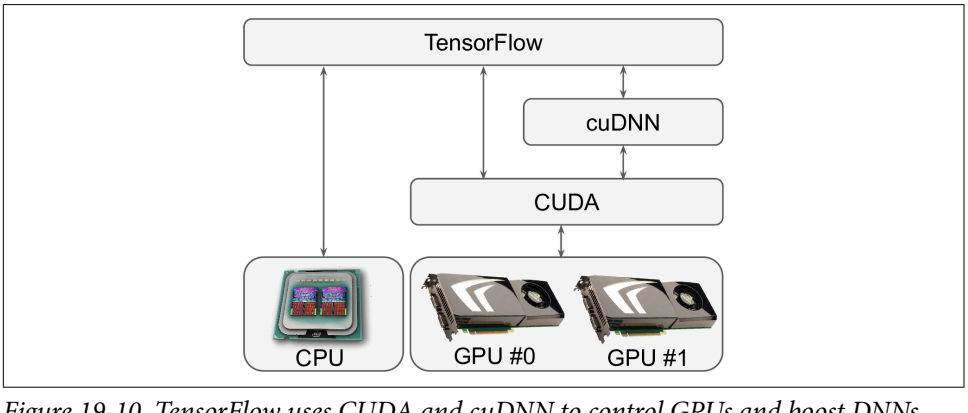


Figure 19-10. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

10 Please check the docs for detailed and up-to-date installation instructions, as they change quite often.

Once you have installed the GPU card(s) and all the required drivers and libraries, you can use the `nvidia-smi` command to check that CUDA is properly installed. It lists the available GPU cards, as well as processes running on each card:

```
$ nvidia-smi
Sun Jun  2 10:05:22 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.67       Driver Version: 410.79       CUDA Version: 10.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P8    17W /  70W |      0MiB / 15079MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

At the time of this writing, you'll also need to install the GPU version of TensorFlow (i.e., the `tensorflow-gpu` library); however, there is ongoing work to have a unified installation procedure for both CPU-only and GPU machines, so please check the installation documentation to see which library you should install. In any case, since installing every required library correctly is a bit long and tricky (and all hell breaks loose if you do not install the correct library versions), TensorFlow provides a Docker image with everything you need inside. However, in order for the Docker container to have access to the GPU, you will still need to install the Nvidia drivers on the host machine.

To check that TensorFlow actually sees the GPUs, run the following tests:

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

The `is_gpu_available()` function checks whether at least one GPU is available. The `gpu_device_name()` function gives the first GPU's name: by default, operations will

run on this GPU. The `list_physical_devices()` function returns the list of all available GPU devices (just one in this example).[11]

Now, what if you don't want to invest time and money in getting your own GPU card? Just use a GPU VM on the cloud!

Using a GPU-Equipped Virtual Machine

All major cloud platforms now offer GPU VMs, some preconfigured with all the drivers and libraries you need (including TensorFlow). Google Cloud Platform enforces various GPU quotas, both worldwide and per region: you cannot just create thousands of GPU VMs without prior authorization from Google.[12] By default, the worldwide GPU quota is zero, so you cannot use any GPU VMs. Therefore, the very first thing you need to do is to request a higher worldwide quota. In the GCP console, open the navigation menu and go to IAM & admin → Quotas. Click Metric, click None to uncheck all locations, then search for "GPU" and select "GPUs (all regions)" to see the corresponding quota. If this quota's value is zero (or just insufficient for your needs), then check the box next to it (it should be the only selected one) and click "Edit quotas." Fill in the requested information, then click "Submit request." It may take a few hours (or up to a few days) for your quota request to be processed and (generally) accepted. By default, there is also a quota of one GPU per region and per GPU type. You can request to increase these quotas too: click Metric, select None to uncheck all metrics, search for "GPU," and select the type of GPU you want (e.g., NVIDIA P4 GPUs). Then click the Location drop-down menu, click None to uncheck all metrics, and click the location you want; check the boxes next to the quota(s) you want to change, and click "Edit quotas" to file a request.

Once your GPU quota requests are approved, you can in no time create a VM equipped with one or more GPUs by using Google Cloud AI Platform's *Deep Learning VM Images*: go to *https://homl.info/dlvm*, click View Console, then click "Launch on Compute Engine" and fill in the VM configuration form. Note that some locations do not have all types of GPUs, and some have no GPUs at all (change the location to see the types of GPUs available, if any). Make sure to select TensorFlow 2.0 as the framework, and check "Install NVIDIA GPU driver automatically on first startup." It is also a good idea to check "Enable access to JupyterLab via URL instead of SSH": this will make it very easy to start a Jupyter notebook running on this GPU VM, powered by

11 Many code examples in this chapter use experimental APIs. They are very likely to be moved to the core API in future versions. So if an experimental function fails, try simply removing the word `experimental`, and hopefully it will work. If not, then perhaps the API has changed a bit; please check the Jupyter notebook, as I will ensure it contains the correct code.

12 Presumably, these quotas are meant to stop bad guys who might be tempted to use GCP with stolen credit cards to mine cryptocurrencies.

JupyterLab (this is an alternative web interface to run Jupyter notebooks). Once the VM is created, scroll down the navigation menu to the Artificial Intelligence section, then click AI Platform → Notebooks. Once the Notebook instance appears in the list (this may take a few minutes, so click Refresh once in a while until it appears), click its Open JupyterLab link. This will run JupyterLab on the VM and connect your browser to it. You can create notebooks and run any code you want on this VM, and benefit from its GPUs!

But if you just want to run some quick tests or easily share notebooks with your colleagues, then you should try Colaboratory.

Colaboratory

The simplest and cheapest way to access a GPU VM is to use *Colaboratory* (or *Colab*, for short). It's free! Just go to *https://colab.research.google.com/* and create a new Python 3 notebook: this will create a Jupyter notebook, stored on your Google Drive (alternatively, you can open any notebook on GitHub, or on Google Drive, or you can even upload your own notebooks). Colab's user interface is similar to Jupyter's, except you can share and use the notebooks like regular Google Docs, and there are a few other minor differences (e.g., you can create handy widgets using special comments in your code).

When you open a Colab notebook, it runs on a free Google VM dedicated to that notebook, called a *Colab Runtime* (see Figure 19-11). By default the Runtime is CPU-only, but you can change this by going to Runtime → "Change runtime type," selecting GPU in the "Hardware accelerator" drop-down menu, then clicking Save. In fact, you could even select TPU! (Yes, you can actually use a TPU for free; we will talk about TPUs later in this chapter, though, so for now just select GPU.)

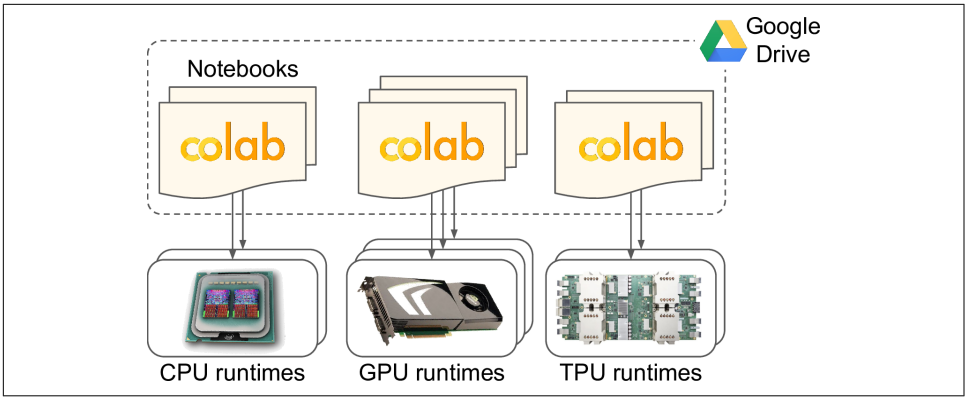


Figure 19-11. Colab Runtimes and notebooks

Colab does have some restrictions: first, there is a limit to the number of Colab notebooks you can run simultaneously (currently 5 per Runtime type). Moreover, as the FAQ states, "Colaboratory is intended for interactive use. Long-running background computations, particularly on GPUs, may be stopped. Please do not use Colaboratory for cryptocurrency mining." Also, the web interface will automatically disconnect from the Colab Runtime if you leave it unattended for a while (~30 minutes). When you reconnect to the Colab Runtime, it may have been reset, so make sure you always export any data you care about (e.g., download it or save it to Google Drive). Even if you never disconnect, the Colab Runtime will automatically shut down after 12 hours, as it is not meant for long-running computations. Despite these limitations, it's a fantastic tool to run tests easily, get quick results, and collaborate with your colleagues.

Managing the GPU RAM

By default TensorFlow automatically grabs all the RAM in all available GPUs the first time you run a computation. It does this to limit GPU RAM fragmentation. This means that if you try to start a second TensorFlow program (or any program that requires the GPU), it will quickly run out of RAM. This does not happen as often as you might think, as you will most often have a single TensorFlow program running on a machine: usually a training script, a TF Serving node, or a Jupyter notebook. If you need to run multiple programs for some reason (e.g., to train two different models in parallel on the same machine), then you will need to split the GPU RAM between these processes more evenly.

If you have multiple GPU cards on your machine, a simple solution is to assign each of them to a single process. To do this, you can set the `CUDA_VISIBLE_DEVICES` environment variable so that each process only sees the appropriate GPU card(s). Also set the `CUDA_DEVICE_ORDER` environment variable to `PCI_BUS_ID` to ensure that

each ID always refers to the same GPU card. For example, if you have four GPU cards, you could start two programs, assigning two GPUs to each of them, by executing commands like the following in two separate terminal windows:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program 1 will then only see GPU cards 0 and 1, named `/gpu:0` and `/gpu:1` respectively, and program 2 will only see GPU cards 2 and 3, named `/gpu:1` and `/gpu:0` respectively (note the order). Everything will work fine (see Figure 19-12). Of course, you can also define these environment variables in Python by setting `os.environ["CUDA_DEVICE_ORDER"]` and `os.environ["CUDA_VISIBLE_DEVICES"]`, as long as you do so before using TensorFlow.

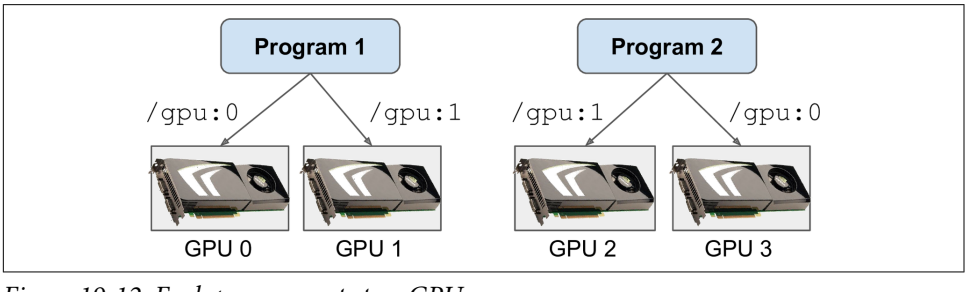


Figure 19-12. Each program gets two GPUs

Another option is to tell TensorFlow to grab only a specific amount of GPU RAM. This must be done immediately after importing TensorFlow. For example, to make TensorFlow grab only 2 GiB of RAM on each GPU, you must create a *virtual GPU device* (also called a *logical GPU device*) for each physical GPU device and set its memory limit to 2 GiB (i.e., 2,048 MiB):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_virtual_device_configuration(
        gpu,
        [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

Now (supposing you have four GPUs, each with at least 4 GiB of RAM) two programs like this one can run in parallel, each using all four GPU cards (see Figure 19-13).

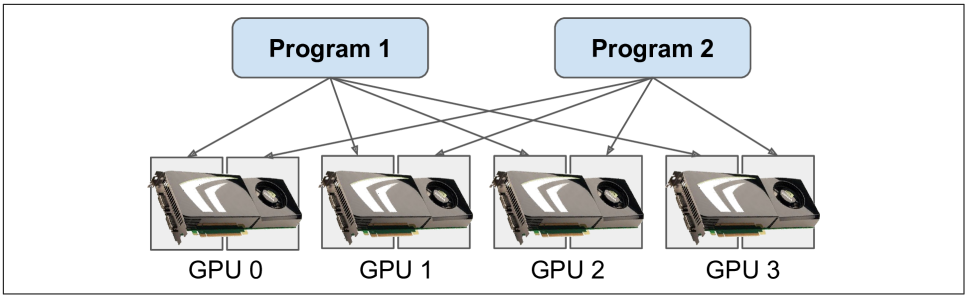


Figure 19-13. Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

If you run the `nvidia-smi` command while both programs are running, you should see that each process holds 2 GiB of RAM on each card:

```
$ nvidia-smi
[...]
+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      2373      C   /usr/bin/python3                          2241MiB |
|    0      2533      C   /usr/bin/python3                          2241MiB |
|    1      2373      C   /usr/bin/python3                          2241MiB |
|    1      2533      C   /usr/bin/python3                          2241MiB |
[...]
```

Yet another option is to tell TensorFlow to grab memory only when it needs it (this also must be done immediately after importing TensorFlow):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):
    tf.config.experimental.set_memory_growth(gpu, True)
```

Another way to do this is to set the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable to `true`. With this option, TensorFlow will never release memory once it has grabbed it (again, to avoid memory fragmentation), except of course when the program ends. It can be harder to guarantee deterministic behavior using this option (e.g., one program may crash because another program's memory usage went through the roof), so in production you'll probably want to stick with one of the previous options. However, there are some cases where it is very useful: for example, when you use a machine to run multiple Jupyter notebooks, several of which use TensorFlow. This is why the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable is set to `true` in Colab Runtimes.

Lastly, in some cases you may want to split a GPU into two or more *virtual GPUs*—for example, if you want to test a distribution algorithm (this is a handy way to try out the code examples in the rest of this chapter even if you have a single GPU, such

as in a Colab Runtime). The following code splits the first GPU into two virtual devices, with 2 GiB of RAM each (again, this must be done immediately after importing TensorFlow):

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")
tf.config.experimental.set_virtual_device_configuration(
    physical_gpus[0],
    [tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048),
     tf.config.experimental.VirtualDeviceConfiguration(memory_limit=2048)])
```

These two virtual devices will then be called `/gpu:0` and `/gpu:1`, and you can place operations and variables on each of them as if they were really two independent GPUs. Now let's see how TensorFlow decides which devices it should place variables and execute operations on.

Placing Operations and Variables on Devices

The TensorFlow whitepaper[13] presents a friendly *dynamic placer* algorithm that automagically distributes operations across all available devices, taking into account things like the measured computation time in previous runs of the graph, estimations of the size of the input and output tensors for each operation, the amount of RAM available in each device, communication delay when transferring data into and out of devices, and hints and constraints from the user. In practice this algorithm turned out to be less efficient than a small set of placement rules specified by the user, so the TensorFlow team ended up dropping the dynamic placer.

That said, tf.keras and tf.data generally do a good job of placing operations and variables where they belong (e.g., heavy computations on the GPU, and data preprocessing on the CPU). But you can also place operations and variables manually on each device, if you want more control:

- As just mentioned, you generally want to place the data preprocessing operations on the CPU, and place the neural network operations on the GPUs.
- GPUs usually have a fairly limited communication bandwidth, so it is important to avoid unnecessary data transfers in and out of the GPUs.
- Adding more CPU RAM to a machine is simple and fairly cheap, so there's usually plenty of it, whereas the GPU RAM is baked into the GPU: it is an expensive and thus limited resource, so if a variable is not needed in the next few training steps, it should probably be placed on the CPU (e.g., datasets generally belong on the CPU).

13 Martín Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems" Google Research whitepaper (2015).

By default, all variables and all operations will be placed on the first GPU (named `/gpu:0`), except for variables and operations that don't have a GPU kernel:[14] these are placed on the CPU (named `/cpu:0`). A tensor or variable's `device` attribute tells you which device it was placed on:[15]

```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

You can safely ignore the prefix `/job:localhost/replica:0/task:0` for now (it allows you to place operations on other machines when using a TensorFlow cluster; we will talk about jobs, replicas, and tasks later in this chapter). As you can see, the first variable was placed on GPU 0, which is the default device. However, the second variable was placed on the CPU: this is because there are no GPU kernels for integer variables (or for operations involving integer tensors), so TensorFlow fell back to the CPU.

If you want to place an operation on a different device than the default one, use a `tf.device()` context:

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

The CPU is always treated as a single device (`/cpu:0`), even if your machine has multiple CPU cores. Any operation placed on the CPU may run in parallel across multiple cores if it has a multi-threaded kernel.

If you explicitly try to place an operation or variable on a device that does not exist or for which there is no kernel, then you will get an exception. However, in some cases you may prefer to fall back to the CPU; for example, if your program may run both on CPU-only machines and on GPU machines, you may want TensorFlow to ignore your `tf.device("/gpu:*")` on CPU-only machines. To do this, you can call `tf.config.set_soft_device_placement(True)` just after importing TensorFlow: when a

14 As we saw in Chapter 12, a kernel is a variable or operation's implementation for a specific data type and device type. For example, there is a GPU kernel for the `float32` `tf.matmul()` operation, but there is no GPU kernel for `int32` `tf.matmul()` (only a CPU kernel).

15 You can also use `tf.debugging.set_log_device_placement(True)` to log all device placements.

placement request fails, TensorFlow will fall back to its default placement rules (i.e., GPU 0 by default if it exists and there is a GPU kernel, and CPU 0 otherwise).

Now how exactly will TensorFlow execute all these operations across multiple devices?

Parallel Execution Across Multiple Devices

As we saw in Chapter 12, one of the benefits of using TF Functions is parallelism. Let's look at this a bit more closely. When TensorFlow runs a TF Function, it starts by analyzing its graph to find the list of operations that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then adds each operation with zero dependencies (i.e., each source operation) to the evaluation queue of this operation's device (see Figure 19-14). Once an operation has been evaluated, the dependency counter of each operation that depends on it is decremented. Once an operation's dependency counter reaches zero, it is pushed to the evaluation queue of its device. And once all the nodes that TensorFlow needs have been evaluated, it returns their outputs.

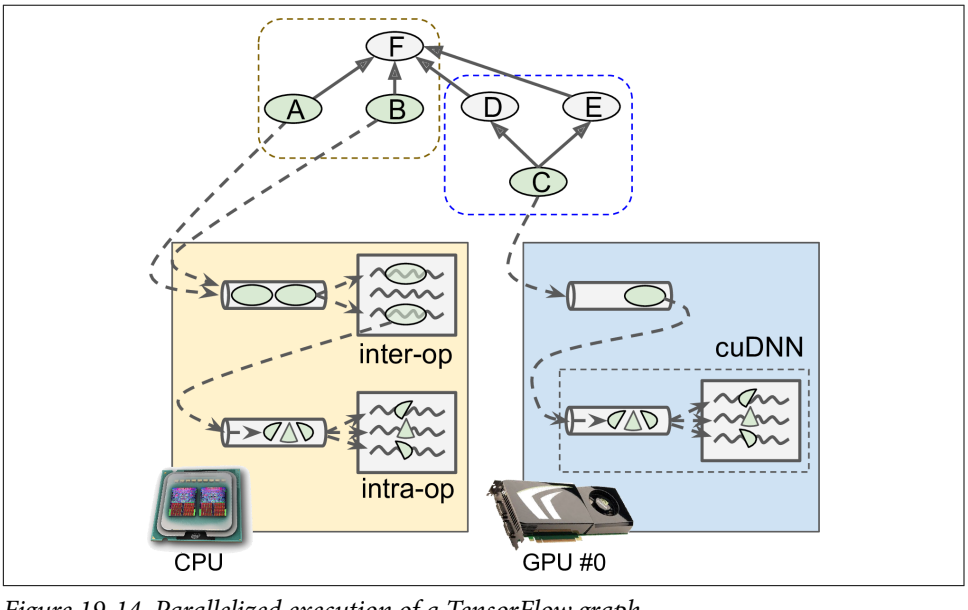


Figure 19-14. Parallelized execution of a TensorFlow graph

Operations in the CPU's evaluation queue are dispatched to a thread pool called the *inter-op thread pool*. If the CPU has multiple cores, then these operations will effectively be evaluated in parallel. Some operations have multithreaded CPU kernels: these kernels split their tasks into multiple suboperations, which are placed in another evaluation queue and dispatched to a second thread pool called the *intra-op*

thread pool (shared by all multithreaded CPU kernels). In short, multiple operations and suboperations may be evaluated in parallel on different CPU cores.

For the GPU, things are a bit simpler. Operations in a GPU's evaluation queue are evaluated sequentially. However, most operations have multithreaded GPU kernels, typically implemented by libraries that TensorFlow depends on, such as CUDA and cuDNN. These implementations have their own thread pools, and they typically exploit as many GPU threads as they can (which is the reason why there is no need for an inter-op thread pool in GPUs: each operation already floods most GPU threads).

For example, in Figure 19-14, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on the CPU, so they are sent to the CPU's evaluation queue, then they are dispatched to the inter-op thread pool and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split into three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU 0's evaluation queue, and in this example its GPU kernel happens to use cuDNN, which manages its own intra-op thread pool and runs the operation across many GPU threads in parallel. Suppose C finishes first. The dependency counters of D and E are decremented and they reach zero, so both operations are pushed to GPU 0's evaluation queue, and they are executed sequentially. Note that C only gets evaluated once, even though both D and E depend on it. Suppose B finishes next. Then F's dependency counter is decremented from 4 to 3, and since that's not 0, it does not run yet. Once A, D, and E are finished, then F's dependency counter reaches 0, and it is pushed to the CPU's evaluation queue and evaluated. Finally, TensorFlow returns the requested outputs.

An extra bit of magic that TensorFlow performs is when the TF Function modifies a stateful resource, such as a variable: it ensures that the order of execution matches the order in the code, even if there is no explicit dependency between the statements. For example, if your TF Function contains `v.assign_add(1)` followed by `v.assign(v * 2)`, TensorFlow will ensure that these operations are executed in that order.

You can control the number of threads in the inter-op thread pool by calling `tf.config.threading.set_inter_op_parallelism_threads()`. To set the number of intra-op threads, use `tf.config.threading.set_intra_op_parallelism_threads()`. This is useful if you want do not want TensorFlow to use all the CPU cores or if you want it to be single-threaded.[16]

16 This can be useful if you want to guarantee perfect reproducibility, as I explain in this video, based on TF 1.

With that, you have all you need to run any operation on any device, and exploit the power of your GPUs! Here are some of the things you could do:

- You could train several models in parallel, each on its own GPU: just write a training script for each model and run them in parallel, setting `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` so that each script only sees a single GPU device. This is great for hyperparameter tuning, as you can train in parallel multiple models with different hyperparameters. If you have a single machine with two GPUs, and it takes one hour to train one model on one GPU, then training two models in parallel, each on its own dedicated GPU, will take just one hour. Simple!
- You could train a model on a single GPU and perform all the preprocessing in parallel on the CPU, using the dataset's `prefetch()` method[17] to prepare the next few batches in advance so that they are ready when the GPU needs them (see Chapter 13).
- If your model takes two images as input and processes them using two CNNs before joining their outputs, then it will probably run much faster if you place each CNN on a different GPU.
- You can create an efficient ensemble: just place a different trained model on each GPU so that you can get all the predictions much faster to produce the ensemble's final prediction.

But what if you want to *train* a single model across multiple GPUs?

Training Models Across Multiple Devices

There are two main approaches to training a single model across multiple devices: *model parallelism*, where the model is split across the devices, and *data parallelism*, where the model is replicated across every device, and each replica is trained on a subset of the data. Let's look at these two options closely before we train a model on multiple GPUs.

Model Parallelism

So far we have trained each neural network on a single device. What if we want to train a single neural network across multiple devices? This requires chopping the model into separate chunks and running each chunk on a different device.

17 At the time of this writing it only prefetches the data to the CPU RAM, but you can use `tf.data.experimental.prefetch_to_device()` to make it prefetch the data and push it to the device of your choice so that the GPU does not waste time waiting for the data to be transferred.

Unfortunately, such model parallelism turns out to be pretty tricky, and it really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see Figure 19-15). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work because each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (represented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (especially when the devices are located on different machines).

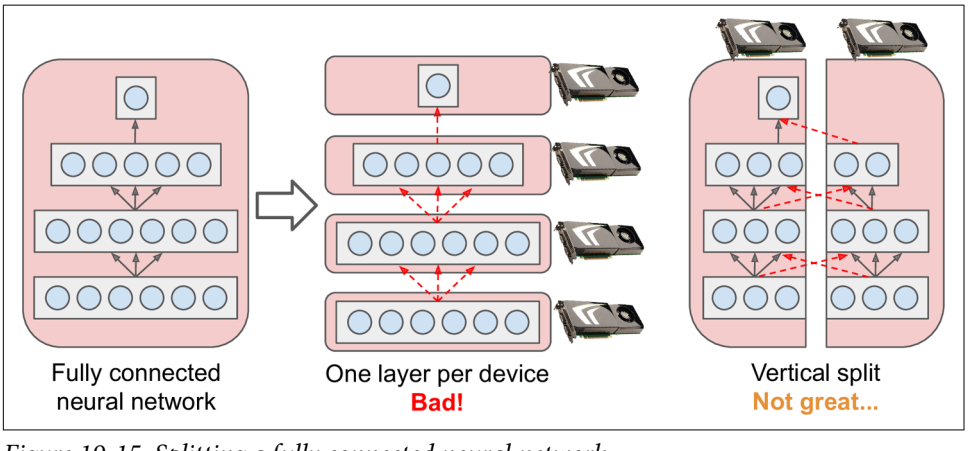


Figure 19-15. Splitting a fully connected neural network

Some neural network architectures, such as convolutional neural networks (see Chapter 14), contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way (Figure 19-16).

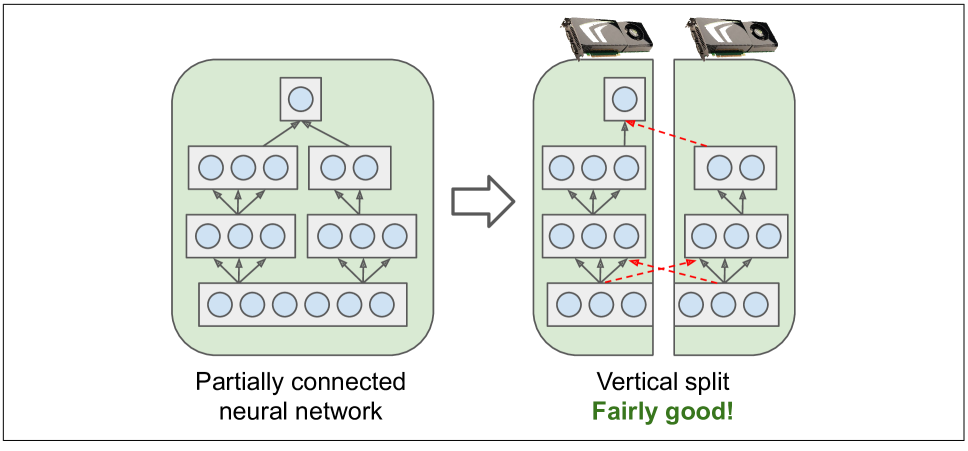


Figure 19-16. Splitting a partially connected neural network

Deep recurrent neural networks (see Chapter 15) can be split a bit more efficiently across multiple GPUs. If you split the network horizontally by placing each layer on a different device, and you feed the network with an input sequence to process, then at the first time step only one device will be active (working on the sequence's first value), at the second step two will be active (the second layer will be handling the output of the first layer for the first value, while the first layer will be handling the second value), and by the time the signal propagates to the output layer, all devices will be active simultaneously (Figure 19-17). There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel may (in theory) outweigh the communication penalty. However, in practice a regular stack of `LSTM` layers running on a single GPU actually runs much faster.

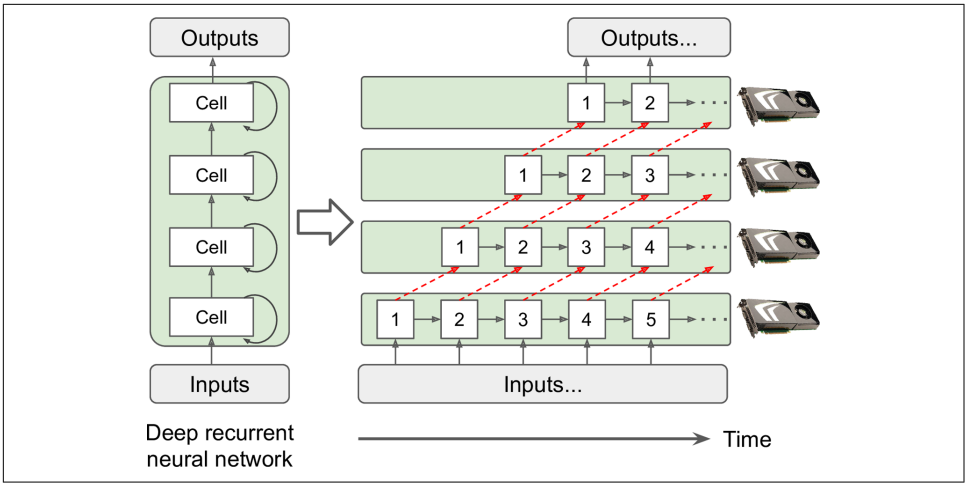


Figure 19-17. Splitting a deep recurrent neural network

In short, model parallelism may speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.[18] Let's look at a much simpler and generally more efficient option: data parallelism.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on every device and run each training step simultaneously on all replicas, using a different mini-batch for each. The gradients computed by each replica are then averaged, and the result is used to update the model parameters. This is called *data parallelism*. There are many variants of this idea, so let's look at the most important ones.

Data parallelism using the mirrored strategy

Arguably the simplest approach is to completely mirror all the model parameters across all the GPUs and always apply the exact same parameter updates on every GPU. This way, all replicas always remain perfectly identical. This is called the *mirrored strategy*, and it turns out to be quite efficient, especially when using a single machine (see Figure 19-18).

18 If you are interested in going further with model parallelism, check out Mesh TensorFlow.

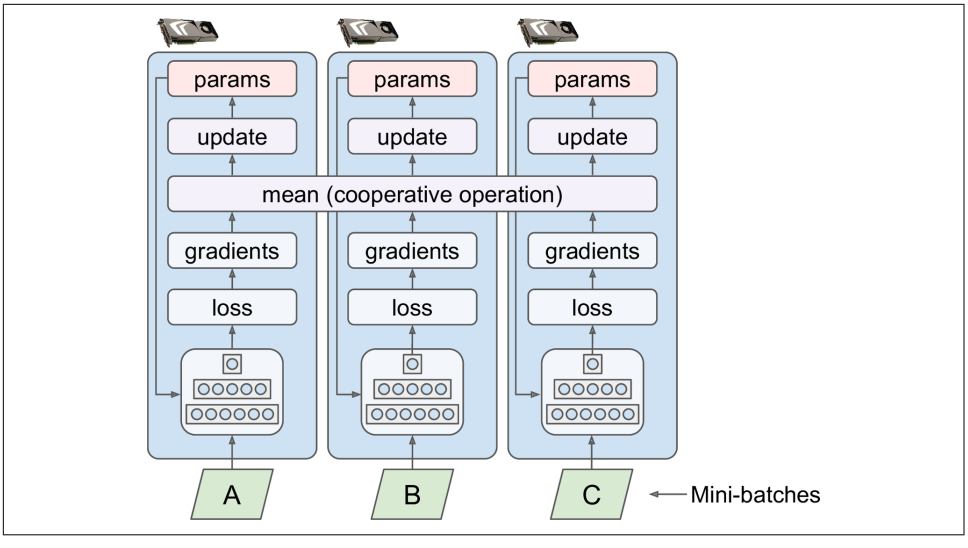


Figure 19-18. Data parallelism using the mirrored strategy

The tricky part when using this approach is to efficiently compute the mean of all the gradients from all the GPUs and distribute the result across all the GPUs. This can be done using an *AllReduce* algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a reduce operation (such as computing the mean, sum, and max), while ensuring that all nodes obtain the same final result. Fortunately, there are off-the-shelf implementations of such algorithms, as we will see.

Data parallelism with centralized parameters

Another approach is to store the model parameters outside of the GPU devices performing the computations (called *workers*), for example on the CPU (see Figure 19-19). In a distributed setup, you may place all the parameters on one or more CPU-only servers called *parameter servers*, whose only role is to host and update the parameters.

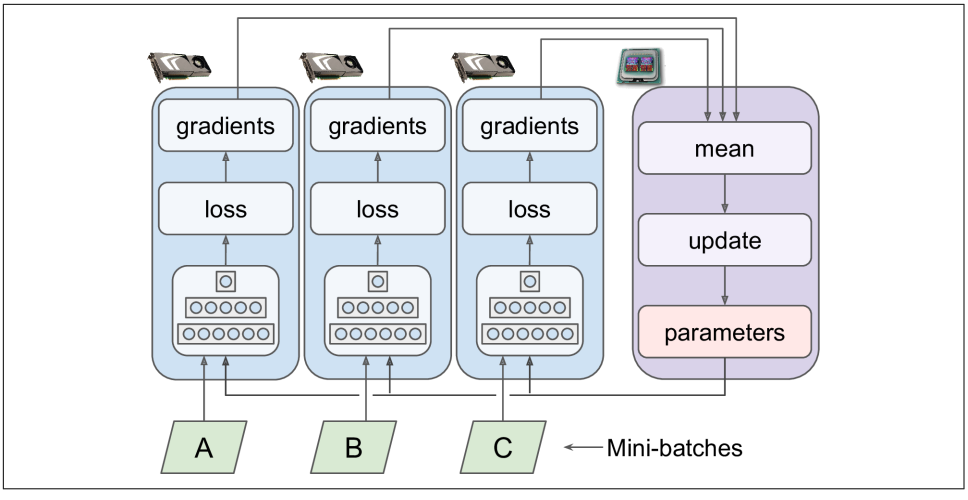


Figure 19-19. Data parallelism with centralized parameters

Whereas the mirrored strategy imposes synchronous weight updates across all GPUs, this centralized approach allows either synchronous or asynchronous updates. Let's see the pros and cons of both options.

Synchronous updates. With *synchronous updates*, the aggregator waits until all gradients are available before it computes the average gradients and passes them to the optimizer, which will update the model parameters. Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so all other devices will have to wait for them at every step. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically ~10%). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.[19]

19 This name is slightly confusing because it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless some devices are really slower than others). However, it does mean that if a server crashes, training will continue just fine.

Asynchronous updates. With asynchronous updates, whenever a replica has finished computing the gradients, it immediately uses them to update the model parameters. There is no aggregation (it removes the "mean" step in Figure 19-19) and no synchronization. Replicas work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica, so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice because of its simplicity, the absence of synchronization delay, and a better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times, if there are N replicas), and there is no guarantee that the computed gradients will still be pointing in the right direction (see Figure 19-20). When gradients are severely out-of-date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge.

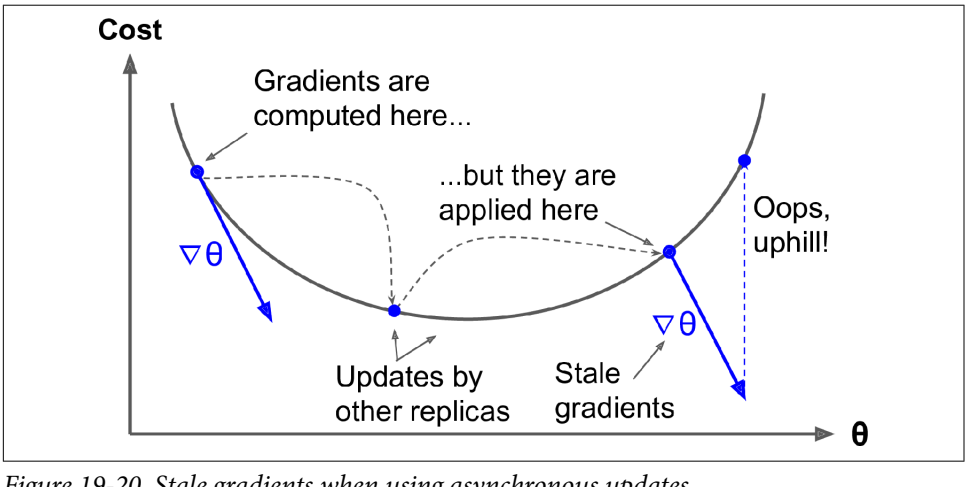


Figure 19-20. Stale gradients when using asynchronous updates

There are a few ways you can reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.

- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in 2016[20] benchmarked various approaches and found that using synchronous updates with a few spare replicas was more efficient than using asynchronous updates, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates just yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism with centralized parameters still requires communicating the model parameters from the parameter servers to every replica at the beginning of each training step, and the gradients in the other direction at the end of each training step. Similarly, when using the mirrored strategy, the gradients produced by each GPU will need to be shared with every other GPU. Unfortunately, there always comes a point where adding an extra GPU will not improve performance at all because the time spent moving the data into and out of GPU RAM (and across the network in a distributed setup) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just worsen the bandwidth saturation and actually slow down training.

For some models, typically relatively small and trained on a very large training set, you are often better off training the model on a single machine with a single powerful GPU with a large memory bandwidth.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is limited) and for large sparse models, where the gradients are typically mostly zeros and so can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, reported typical speedups of 25–40× when distributing computations across 50 GPUs for dense models, and a 300× speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

20 Jianmin Chen et al., "Revisiting Distributed Synchronous SGD," arXiv preprint arXiv:1604.00981 (2016).

- Neural machine translation: 6× speedup on 8 GPUs
- Inception/ImageNet: 32× speedup on 50 GPUs
- RankBrain: 300× speedup on 500 GPUs

Beyond a few dozen GPUs for a dense model or few hundred GPUs for a sparse model, saturation kicks in and performance degrades. There is plenty of research going on to solve this problem (exploring peer-to-peer architectures rather than centralized parameter servers, using lossy model compression, optimizing when and what the replicas need to communicate, and so on), so there will likely be a lot of progress in parallelizing neural networks in the next few years.

In the meantime, to reduce the saturation problem, you probably want to use a few powerful GPUs rather than plenty of weak GPUs, and you should also group your GPUs on few and very well interconnected servers. You can also try dropping the float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, often without much impact on the convergence rate or the model's performance. Lastly, if you are using centralized parameters, you can shard (split) the parameters across multiple parameter servers: adding more parameter servers will reduce the network load on each server and limit the risk of bandwidth saturation.

OK, now let's train a model across multiple GPUs!

Training at Scale Using the Distribution Strategies API

Many models can be trained quite well on a single GPU, or even on a CPU. But if training is too slow, you can try distributing it across multiple GPUs on the same machine. If that's still too slow, try using more powerful GPUs, or add more GPUs to the machine. If your model performs heavy computations (such as large matrix multiplications), then it will run much faster on powerful GPUs, and you could even try to use TPUs on Google Cloud AI Platform, which will usually run even faster for such models. But if you can't fit any more GPUs on the same machine, and if TPUs aren't for you (e.g., perhaps your model doesn't benefit much from TPUs, or perhaps you want to use your own hardware infrastructure), then you can try training it across several servers, each with multiple GPUs (if this is still not enough, as a last resort you can try adding some model parallelism, but this requires a lot more effort). In this section we will see how to train models at scale, starting with multiple GPUs on the same machine (or TPUs) and then moving on to multiple GPUs across multiple machines.

Luckily, TensorFlow comes with a very simple API that takes care of all the complexity for you: the *Distribution Strategies API*. To train a Keras model across all available GPUs (on a single machine, for now) using data parallelism with the mirrored

strategy, create a `MirroredStrategy` object, call its `scope()` method to get a distribution context, and wrap the creation and compilation of your model inside that context. Then call the model's `fit()` method normally:

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Under the hood, tf.keras is distribution-aware, so in this `MirroredStrategy` context it knows that it must replicate all variables and operations across all available GPU devices. Note that the `fit()` method will automatically split each training batch across all the replicas, so it's important that the batch size be divisible by the number of replicas. And that's all! Training will generally be significantly faster than using a single device, and the code change was really minimal.

Once you have finished training your model, you can use it to make predictions efficiently: call the `predict()` method, and it will automatically split the batch across all replicas, making predictions in parallel (again, the batch size must be divisible by the number of replicas). If you call the model's `save()` method, it will be saved as a regular model, *not* as a mirrored model with multiple replicas. So when you load it, it will run like a regular model, on a single device (by default GPU 0, or the CPU if there are no GPUs). If you want to load a model and run it on all available devices, you must call `keras.models.load_model()` within a distribution context:

```
with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")
```

If you only want to use a subset of all the available GPU devices, you can pass the list to the `MirroredStrategy`'s constructor:

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

By default, the `MirroredStrategy` class uses the *NVIDIA Collective Communications Library* (NCCL) for the AllReduce mean operation, but you can change it by setting the `cross_device_ops` argument to an instance of the `tf.distribute.HierarchicalCopyAllReduce` class, or an instance of the `tf.distribute.ReductionToOneDevice` class. The default NCCL option is based on the `tf.distribute.NcclAllReduce` class, which is usually faster, but this depends on the number and types of GPUs, so you may want to give the alternatives a try.[21]

21 For more details on AllReduce algorithms, read this great post by Yuichiro Ueno, and this page on scaling with NCCL.

If you want to try using data parallelism with centralized parameters, replace the `MirroredStrategy` with the `CentralStorageStrategy`:

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

You can optionally set the `compute_devices` argument to specify the list of devices you want to use as workers (by default it will use all available GPUs), and you can optionally set the `parameter_device` argument to specify the device you want to store the parameters on (by default it will use the CPU, or the GPU if there is just one).

Now let's see how to train a model across a cluster of TensorFlow servers!

Training a Model on a TensorFlow Cluster

A *TensorFlow cluster* is a group of TensorFlow processes running in parallel, usually on different machines, and talking to each other to complete some work—for example, training or executing a neural network. Each TF process in the cluster is called a *task*, or a *TF server*. It has an IP address, a port, and a type (also called its *role* or its *job*). The type can be either `"worker"`, `"chief"`, `"ps"` (parameter server), or `"evaluator"`:

- Each *worker* performs computations, usually on a machine with one or more GPUs.
- The *chief* performs computations as well (it is a worker), but it also handles extra work such as writing TensorBoard logs or saving checkpoints. There is a single chief in a cluster. If no chief is specified, then the first worker is the chief.
- A *parameter server* only keeps track of variable values, and it is usually on a CPU-only machine. This type of task is only used with the `ParameterServerStrategy`.
- An *evaluator* obviously takes care of evaluation.

To start a TensorFlow cluster, you must first specify it. This means defining each task's IP address, TCP port, and type. For example, the following *cluster specification* defines a cluster with three tasks (two workers and one parameter server; see Figure 19-21). The cluster spec is a dictionary with one key per job, and the values are lists of task addresses (*IP:port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",  # /job:worker/task:0
        "machine-b.example.com:2222"   # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0
}
```

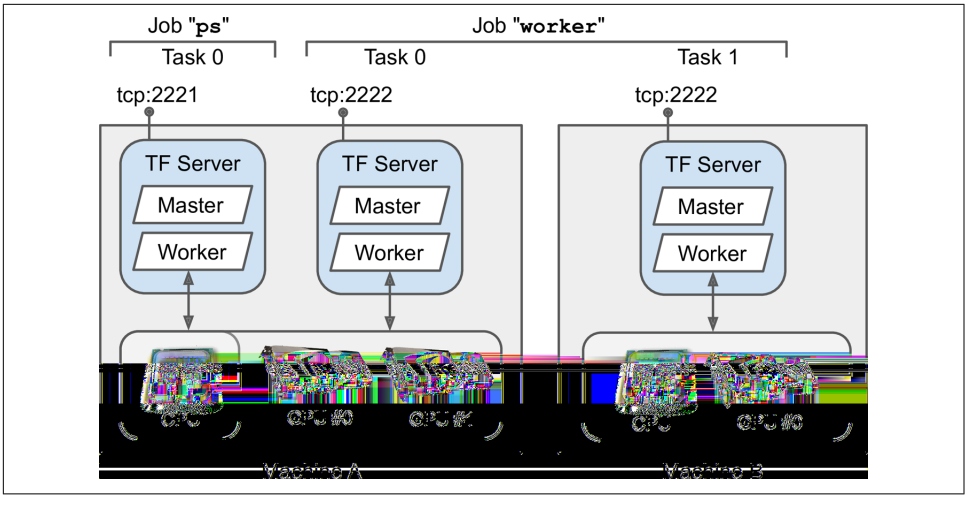


Figure 19-21. TensorFlow cluster

In general there will be a single task per machine, but as this example shows, you can configure multiple tasks on the same machine if you want (if they share the same GPUs, make sure the RAM is split appropriately, as discussed earlier).

By default, every task in the cluster may communicate with every other task, so make sure to configure your firewall to authorize all communications between these machines on these ports (it's usually simpler if you use the same port on every machine).

When you start a task, you must give it the cluster spec, and you must also tell it what its type and index are (e.g., worker 0). The simplest way to specify everything at once (both the cluster spec and the current task's type and index) is to set the `TF_CONFIG` environment variable before starting TensorFlow. It must be a JSON-encoded dictionary containing a cluster specification (under the `"cluster"` key) and the type and index of the current task (under the `"task"` key). For example, the following `TF_CONFIG` environment variable uses the cluster we just defined and specifies that the task to start is the first worker:

```
import os
import json

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
    "task": {"type": "worker", "index": 0}
})
```

In general you want to define the `TF_CONFIG` environment variable outside of Python, so the code does not need to include the current task's type and index (this makes it possible to use the same code across all workers).

Now let's train a model on a cluster! We will start with the mirrored strategy—it's surprisingly simple! First, you need to set the `TF_CONFIG` environment variable appropriately for each task. There should be no parameter server (remove the "ps" key in the cluster spec), and in general you will want a single worker per machine. Make extra sure you set a different task index for each task. Finally, run the following training code on every worker:

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()

with distribution.scope():
    mirrored_model = keras.models.Sequential([...])
    mirrored_model.compile([...])

batch_size = 100 # must be divisible by the number of replicas
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Yes, that's exactly the same code we used earlier, except this time we are using the `MultiWorkerMirroredStrategy` (in future versions, the `MirroredStrategy` will probably handle both the single machine and multimachine cases). When you start this script on the first workers, they will remain blocked at the AllReduce step, but as soon as the last worker starts up training will begin, and you will see them all advancing at exactly the same rate (since they synchronize at each step).

You can choose from two AllReduce implementations for this distribution strategy: a ring AllReduce algorithm based on gRPC for the network communications, and NCCL's implementation. The best algorithm to use depends on the number of workers, the number and types of GPUs, and the network. By default, TensorFlow will apply some heuristics to select the right algorithm for you, but if you want to force one algorithm, pass `CollectiveCommunication.RING` or `CollectiveCommunication.NCCL` (from `tf.distribute.experimental`) to the strategy's constructor.

If you prefer to implement asynchronous data parallelism with parameter servers, change the strategy to `ParameterServerStrategy`, add one or more parameter servers, and configure `TF_CONFIG` appropriately for each task. Note that although the workers will work asynchronously, the replicas on each worker will work synchronously.

Lastly, if you have access to TPUs on Google Cloud, you can create a `TPUStrategy` like this (then use it like the other strategies):

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

If you are a researcher, you may be eligible to use TPUs for free; see *https://tensorflow.org/tfrc* for more details.

You can now train models across multiple GPUs and multiple servers: give yourself a pat on the back! If you want to train a large model, you will need many GPUs, across many servers, which will require either buying a lot of hardware or managing a lot of cloud VMs. In many cases, it's going to be less hassle and less expensive to use a cloud service that takes care of provisioning and managing all this infrastructure for you, just when you need it. Let's see how to do that on GCP.

Running Large Training Jobs on Google Cloud AI Platform

If you decide to use Google AI Platform, you can deploy a training job with the same training code as you would run on your own TF cluster, and the platform will take care of provisioning and configuring as many GPU VMs as you desire (within your quotas).

To start the job, you will need the `gcloud` command-line tool, which is part of the Google Cloud SDK. You can either install the SDK on your own machine, or just use the Google Cloud Shell on GCP. This is a terminal you can use directly in your web browser; it runs on a free Linux VM (Debian), with the SDK already installed and preconfigured for you. The Cloud Shell is available anywhere in GCP: just click the Activate Cloud Shell icon at the top right of the page (see Figure 19-22).

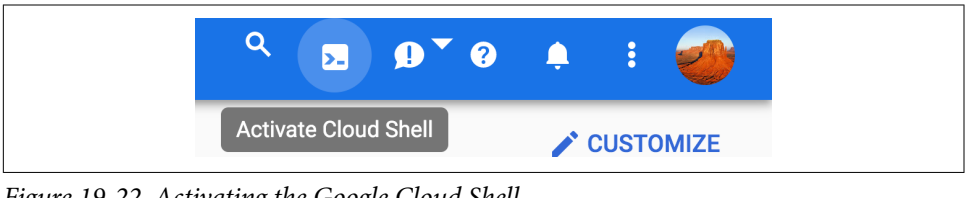


Figure 19-22. Activating the Google Cloud Shell

If you prefer to install the SDK on your machine, once you have installed it, you need to initialize it by running `gcloud init`: you will need to log in to GCP and grant access to your GCP resources, then select the GCP project you want to use (if you have more than one), as well as the region where you want the job to run. The `gcloud` command gives you access to every GCP feature, including the ones we used earlier. You don't have to go through the web interface every time; you can write scripts that start or stop VMs for you, deploy models, or perform any other GCP action.

Before you can run the training job, you need to write the training code, exactly like you did earlier for a distributed setup (e.g., using the `ParameterServerStrategy`). AI Platform will take care of setting `TF_CONFIG` for you on each VM. Once that's done, you can deploy it and run it on a TF cluster with a command line like this:

```
$ gcloud ai-platform jobs submit training my_job_20190531_164700 \
    --region asia-southeast1 \
    --scale-tier PREMIUM_1 \
    --runtime-version 2.0 \
    --python-version 3.5 \
    --package-path /my_project/src/trainer \
    --module-name trainer.task \
    --staging-bucket gs://my-staging-bucket \
    --job-dir gs://my-mnist-model-bucket/trained_model \
    --
    --my-extra-argument1 foo --my-extra-argument2 bar
```

Let's go through these options. The command will start a training job named `my_job_20190531_164700`, in the `asia-southeast1` region, using a `PREMIUM_1` *scale tier*: this corresponds to 20 workers (including a chief) and 11 parameter servers (check out the other available scale tiers). All these VMs will be based on AI Platform's 2.0 runtime (a VM configuration that includes TensorFlow 2.0 and many other packages)[22] and Python 3.5. The training code is located in the */my_project/src/trainer* directory, and the `gcloud` command will automatically bundle it into a pip package and upload it to GCS at *gs://my-staging-bucket*. Next, AI Platform will start several VMs, deploy the package to them, and run the `trainer.task` module. Lastly, the `--job-dir` argument and the extra arguments (i.e., all the arguments located after the `--` separator) will be passed to the training program: the chief task will usually use the `--job-dir` argument to find out where to save the final model on GCS, in this case at *gs://my-mnist-model-bucket/trained_model*. And that's it! In the GCP console, you can then open the navigation menu, scroll down to the Artificial Intelligence section, and open AI Platform → Jobs. You should see your job running, and if you click it you will see graphs showing the CPU, GPU, and RAM utilization for every task. You can click View Logs to access the detailed logs using Stackdriver.

If you place the training data on GCS, you can create a `tf.data.TextLineDataset` or `tf.data.TFRecordDataset` to access it: just use the GCS paths as the filenames (e.g., *gs://my-data-bucket/my_data_001.csv*). These datasets rely on the `tf.io.gfile` package to access files: it supports both local files and GCS files (but make sure the service account you use has access to GCS).

22 At the time of this writing, the 2.0 runtime is not yet available, but it should be ready by the time you read this. Check out the list of available runtimes.

If you want to explore a few hyperparameter values, you can simply run multiple jobs and specify the hyperparameter values using the extra arguments for your tasks. However, if you want to explore many hyperparameters efficiently, it's a good idea to use AI Platform's hyperparameter tuning service instead.

Black Box Hyperparameter Tuning on AI Platform

AI Platform provides a powerful Bayesian optimization hyperparameter tuning service called Google Vizier.[23] To use it, you need to pass a YAML configuration file when creating the job (`--config tuning.yaml`). For example, it may look like this:

```
trainingInput:
  hyperparameters:
    goal: MAXIMIZE
    hyperparameterMetricTag: accuracy
    maxTrials: 10
    maxParallelTrials: 2
    params:
      - parameterName: n_layers
        type: INTEGER
        minValue: 10
        maxValue: 100
        scaleType: UNIT_LINEAR_SCALE
      - parameterName: momentum
        type: DOUBLE
        minValue: 0.1
        maxValue: 1.0
        scaleType: UNIT_LOG_SCALE
```

This tells AI Platform that we want to maximize the metric named `"accuracy"`, the job will run a maximum of 10 trials (each trial will run our training code to train the model from scratch), and it will run a maximum of 2 trials in parallel. We want it to tune two hyperparameters: the `n_layers` hyperparameter (an integer between 10 and 100) and the `momentum` hyperparameter (a float between 0.1 and 1.0). The `scaleType` argument specifies the prior for the hyperparameter value: `UNIT_LINEAR_SCALE` means a flat prior (i.e., no a priori preference), while `UNIT_LOG_SCALE` says we have a prior belief that the optimal value lies closer to the max value (the other possible prior is `UNIT_REVERSE_LOG_SCALE`, when we believe the optimal value to be close to the min value).

The `n_layers` and `momentum` arguments will be passed as command-line arguments to the training code, and of course it is expected to use them. The question is, how will the training code communicate the metric back to the AI Platform so that it can

23 Daniel Golovin et al., "Google Vizier: A Service for Black-Box Optimization," *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017): 1487–1495.

decide which hyperparameter values to use during the next trial? Well, AI Platform just monitors the output directory (specified via `--job-dir`) for any event file (introduced in Chapter 10) containing summaries for a metric named `"accuracy"` (or whatever metric name is specified as the `hyperparameterMetricTag`), and it reads those values. So your training code simply has to use the `TensorBoard()` callback (which you will want to do anyway for monitoring), and you're good to go!

Once the job is finished, all the hyperparameter values used in each trial and the resulting accuracy will be available in the job's output (available via the AI Platform → Jobs page).

AI Platform jobs can also be used to efficiently execute your model on large amounts of data: each worker can read part of the data from GCS, make predictions, and save them to GCS.

Now you have all the tools and knowledge you need to create state-of-the-art neural net architectures and train them at scale using various distribution strategies, on your own infrastructure or on the cloud—and you can even perform powerful Bayesian optimization to fine-tune the hyperparameters!

Exercises

1. What does a SavedModel contain? How do you inspect its content?
2. When should you use TF Serving? What are its main features? What are some tools you can use to deploy it?
3. How do you deploy a model across multiple TF Serving instances?
4. When should you use the gRPC API rather than the REST API to query a model served by TF Serving?
5. What are the different ways TFLite reduces a model's size to make it run on a mobile or embedded device?
6. What is quantization-aware training, and why would you need it?
7. What are model parallelism and data parallelism? Why is the latter generally recommended?
8. When training a model across multiple servers, what distribution strategies can you use? How do you choose which one to use?
9. Train a model (any model you like) and deploy it to TF Serving or Google Cloud AI Platform. Write the client code to query it using the REST API or the gRPC

API. Update the model and deploy the new version. Your client code will now query the new version. Roll back to the first version.

10. Train any model across multiple GPUs on the same machine using the `MirroredStrategy` (if you do not have access to GPUs, you can use Colaboratory with a GPU Runtime and create two virtual GPUs). Train the model again using the `CentralStorageStrategy` and compare the training time.
11. Train a small model on Google Cloud AI Platform, using black box hyperparameter tuning.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, through the *ageron/handson-ml2* GitHub project, or on Twitter at @aureliengeron.

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the Jupyter notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. It also helps tremendously to have a concrete project to work on, whether it is for work or for fun (ideally for both), so if there's anything you have always dreamt of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy to build, it will be immensely rewarding.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

—Aurélien Géron, June 17, 2019

APPENDIX A

Exercise Solutions

Solutions to the coding exercises are available in the online Jupyter notebooks at *https://github.com/ageron/handson-ml2*.

Chapter 1: The Machine Learning Landscape

1. Machine Learning is about building systems that can learn from data. Learning means getting better at some task, given some performance measure.
2. Machine Learning is great for complex problems for which we have no algorithmic solution, to replace long lists of hand-tuned rules, to build systems that adapt to fluctuating environments, and finally to help humans learn (e.g., data mining).
3. A labeled training set is a training set that contains the desired solution (a.k.a. a label) for each instance.
4. The two most common supervised tasks are regression and classification.
5. Common unsupervised tasks include clustering, visualization, dimensionality reduction, and association rule learning.
6. Reinforcement Learning is likely to perform best if we want a robot to learn to walk in various unknown terrains, since this is typically the type of problem that Reinforcement Learning tackles. It might be possible to express the problem as a supervised or semisupervised learning problem, but it would be less natural.
7. If you don't know how to define the groups, then you can use a clustering algorithm (unsupervised learning) to segment your customers into clusters of similar customers. However, if you know what groups you would like to have, then you

can feed many examples of each group to a classification algorithm (supervised learning), and it will classify all your customers into these groups.

8. Spam detection is a typical supervised learning problem: the algorithm is fed many emails along with their labels (spam or not spam).
9. An online learning system can learn incrementally, as opposed to a batch learning system. This makes it capable of adapting rapidly to both changing data and autonomous systems, and of training on very large quantities of data.
10. Out-of-core algorithms can handle vast quantities of data that cannot fit in a computer's main memory. An out-of-core learning algorithm chops the data into mini-batches and uses online learning techniques to learn from these mini-batches.
11. An instance-based learning system learns the training data by heart; then, when given a new instance, it uses a similarity measure to find the most similar learned instances and uses them to make predictions.
12. A model has one or more model parameters that determine what it will predict given a new instance (e.g., the slope of a linear model). A learning algorithm tries to find optimal values for these parameters such that the model generalizes well to new instances. A hyperparameter is a parameter of the learning algorithm itself, not of the model (e.g., the amount of regularization to apply).
13. Model-based learning algorithms search for an optimal value for the model parameters such that the model will generalize well to new instances. We usually train such systems by minimizing a cost function that measures how bad the system is at making predictions on the training data, plus a penalty for model complexity if the model is regularized. To make predictions, we feed the new instance's features into the model's prediction function, using the parameter values found by the learning algorithm.
14. Some of the main challenges in Machine Learning are the lack of data, poor data quality, nonrepresentative data, uninformative features, excessively simple models that underfit the training data, and excessively complex models that overfit the data.
15. If a model performs great on the training data but generalizes poorly to new instances, the model is likely overfitting the training data (or we got extremely lucky on the training data). Possible solutions to overfitting are getting more data, simplifying the model (selecting a simpler algorithm, reducing the number of parameters or features used, or regularizing the model), or reducing the noise in the training data.
16. A test set is used to estimate the generalization error that a model will make on new instances, before the model is launched in production.

17. A validation set is used to compare models. It makes it possible to select the best model and tune the hyperparameters.
18. The train-dev set is used when there is a risk of mismatch between the training data and the data used in the validation and test datasets (which should always be as close as possible to the data used once the model is in production). The train-dev set is a part of the training set that's held out (the model is not trained on it). The model is trained on the rest of the training set, and evaluated on both the train-dev set and the validation set. If the model performs well on the training set but not on the train-dev set, then the model is likely overfitting the training set. If it performs well on both the training set and the train-dev set, but not on the validation set, then there is probably a significant data mismatch between the training data and the validation + test data, and you should try to improve the training data to make it look more like the validation + test data.
19. If you tune hyperparameters using the test set, you risk overfitting the test set, and the generalization error you measure will be optimistic (you may launch a model that performs worse than you expect).

Chapter 2: End-to-End Machine Learning Project

See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 3: Classification

See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 4: Training Models

1. If you have a training set with millions of features you can use Stochastic Gradient Descent or Mini-batch Gradient Descent, and perhaps Batch Gradient Descent if the training set fits in memory. But you cannot use the Normal Equation or the SVD approach because the computational complexity grows quickly (more than quadratically) with the number of features.
2. If the features in your training set have very different scales, the cost function will have the shape of an elongated bowl, so the Gradient Descent algorithms will take a long time to converge. To solve this you should scale the data before training the model. Note that the Normal Equation or SVD approach will work just fine without scaling. Moreover, regularized models may converge to a suboptimal solution if the features are not scaled: since regularization penalizes large weights, features with smaller values will tend to be ignored compared to features with larger values.

3. Gradient Descent cannot get stuck in a local minimum when training a Logistic Regression model because the cost function is convex.[1]
4. If the optimization problem is convex (such as Linear Regression or Logistic Regression), and assuming the learning rate is not too high, then all Gradient Descent algorithms will approach the global optimum and end up producing fairly similar models. However, unless you gradually reduce the learning rate, Stochastic GD and Mini-batch GD will never truly converge; instead, they will keep jumping back and forth around the global optimum. This means that even if you let them run for a very long time, these Gradient Descent algorithms will produce slightly different models.
5. If the validation error consistently goes up after every epoch, then one possibility is that the learning rate is too high and the algorithm is diverging. If the training error also goes up, then this is clearly the problem and you should reduce the learning rate. However, if the training error is not going up, then your model is overfitting the training set and you should stop training.
6. Due to their random nature, neither Stochastic Gradient Descent nor Mini-batch Gradient Descent is guaranteed to make progress at every single training iteration. So if you immediately stop training when the validation error goes up, you may stop much too early, before the optimum is reached. A better option is to save the model at regular intervals; then, when it has not improved for a long time (meaning it will probably never beat the record), you can revert to the best saved model.
7. Stochastic Gradient Descent has the fastest training iteration since it considers only one training instance at a time, so it is generally the first to reach the vicinity of the global optimum (or Mini-batch GD with a very small mini-batch size). However, only Batch Gradient Descent will actually converge, given enough training time. As mentioned, Stochastic GD and Mini-batch GD will bounce around the optimum, unless you gradually reduce the learning rate.
8. If the validation error is much higher than the training error, this is likely because your model is overfitting the training set. One way to try to fix this is to reduce the polynomial degree: a model with fewer degrees of freedom is less likely to overfit. Another thing you can try is to regularize the model—for example, by adding an ℓ_2 penalty (Ridge) or an ℓ_1 penalty (Lasso) to the cost function. This will also reduce the degrees of freedom of the model. Lastly, you can try to increase the size of the training set.

1 If you draw a straight line between any two points on the curve, the line never crosses the curve.

9. If both the training error and the validation error are almost equal and fairly high, the model is likely underfitting the training set, which means it has a high bias. You should try reducing the regularization hyperparameter α.
10. Let's see:
    - A model with some regularization typically performs better than a model without any regularization, so you should generally prefer Ridge Regression over plain Linear Regression.
    - Lasso Regression uses an ℓ_1 penalty, which tends to push the weights down to exactly zero. This leads to sparse models, where all weights are zero except for the most important weights. This is a way to perform feature selection automatically, which is good if you suspect that only a few features actually matter. When you are not sure, you should prefer Ridge Regression.
    - Elastic Net is generally preferred over Lasso since Lasso may behave erratically in some cases (when several features are strongly correlated or when there are more features than training instances). However, it does add an extra hyperparameter to tune. If you want Lasso without the erratic behavior, you can just use Elastic Net with an `l1_ratio` close to 1.
11. If you want to classify pictures as outdoor/indoor and daytime/nighttime, since these are not exclusive classes (i.e., all four combinations are possible) you should train two Logistic Regression classifiers.
12. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 5: Support Vector Machines

1. The fundamental idea behind Support Vector Machines is to fit the widest possible "street" between the classes. In other words, the goal is to have the largest possible margin between the decision boundary that separates the two classes and the training instances. When performing soft margin classification, the SVM searches for a compromise between perfectly separating the two classes and having the widest possible street (i.e., a few instances may end up on the street). Another key idea is to use kernels when training on nonlinear datasets.
2. After training an SVM, a *support vector* is any instance located on the "street" (see the previous answer), including its border. The decision boundary is entirely determined by the support vectors. Any instance that is *not* a support vector (i.e., is off the street) has no influence whatsoever; you could remove them, add more instances, or move them around, and as long as they stay off the street they won't affect the decision boundary. Computing the predictions only involves the support vectors, not the whole training set.

3. SVMs try to fit the largest possible "street" between the classes (see the first answer), so if the training set is not scaled, the SVM will tend to neglect small features (see Figure 5-2).
4. An SVM classifier can output the distance between the test instance and the decision boundary, and you can use this as a confidence score. However, this score cannot be directly converted into an estimation of the class probability. If you set `probability=True` when creating an SVM in Scikit-Learn, then after training it will calibrate the probabilities using Logistic Regression on the SVM's scores (trained by an additional five-fold cross-validation on the training data). This will add the `predict_proba()` and `predict_log_proba()` methods to the SVM.
5. This question applies only to linear SVMs since kernelized SVMs can only use the dual form. The computational complexity of the primal form of the SVM problem is proportional to the number of training instances m, while the computational complexity of the dual form is proportional to a number between m^2 and m^3. So if there are millions of instances, you should definitely use the primal form, because the dual form will be much too slow.
6. If an SVM classifier trained with an RBF kernel underfits the training set, there might be too much regularization. To decrease it, you need to increase `gamma` or `C` (or both).
7. Let's call the QP parameters for the hard margin problem H′, f′, A′, and b′ (see "Quadratic Programming" on page 167). The QP parameters for the soft margin problem have m additional parameters ($n_p = n + 1 + m$) and m additional constraints ($n_c = 2m$). They can be defined like so:
   - H is equal to H′, plus m columns of 0s on the right and m rows of 0s at the bottom: $\mathrm{H} = \begin{pmatrix} \mathrm{H}' & 0 & \cdots \\ 0 & 0 & \\ \vdots & & \ddots \end{pmatrix}$
   - f is equal to f′ with m additional elements, all equal to the value of the hyperparameter C.
   - b is equal to b′ with m additional elements, all equal to 0.
   - A is equal to A′, with an extra $m \times m$ identity matrix I_m appended to the right, $-\mathrm{I}_m$ just below it, and the rest filled with 0s: $\mathrm{A} = \begin{pmatrix} \mathrm{A}' & \mathrm{I}_m \\ 0 & -\mathrm{I}_m \end{pmatrix}$

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 6: Decision Trees

1. The depth of a well-balanced binary tree containing m leaves is equal to $\log_2(m)$,[2] rounded up. A binary Decision Tree (one that makes only binary decisions, as is the case with all trees in Scikit-Learn) will end up more or less well balanced at the end of training, with one leaf per training instance if it is trained without restrictions. Thus, if the training set contains one million instances, the Decision Tree will have a depth of $\log_2(10^6) \approx 20$ (actually a bit more since the tree will generally not be perfectly well balanced).
2. A node's Gini impurity is generally lower than its parent's. This is due to the CART training algorithm's cost function, which splits each node in a way that minimizes the weighted sum of its children's Gini impurities. However, it is possible for a node to have a higher Gini impurity than its parent, as long as this increase is more than compensated for by a decrease in the other child's impurity. For example, consider a node containing four instances of class A and one of class B. Its Gini impurity is $1 - (1/5)^2 - (4/5)^2 = 0.32$. Now suppose the dataset is one-dimensional and the instances are lined up in the following order: A, B, A, A, A. You can verify that the algorithm will split this node after the second instance, producing one child node with instances A, B, and the other child node with instances A, A, A. The first child node's Gini impurity is $1 - (1/2)^2 - (1/2)^2 = 0.5$, which is higher than its parent's. This is compensated for by the fact that the other node is pure, so its overall weighted Gini impurity is $2/5 \times 0.5 + 3/5 \times 0 = 0.2$, which is lower than the parent's Gini impurity.
3. If a Decision Tree is overfitting the training set, it may be a good idea to decrease `max_depth`, since this will constrain the model, regularizing it.
4. Decision Trees don't care whether or not the training data is scaled or centered; that's one of the nice things about them. So if a Decision Tree underfits the training set, scaling the input features will just be a waste of time.
5. The computational complexity of training a Decision Tree is $O(n \times m \log(m))$. So if you multiply the training set size by 10, the training time will be multiplied by $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. If $m = 10^6$, then $K \approx 11.7$, so you can expect the training time to be roughly 11.7 hours.
6. Presorting the training set speeds up training only if the dataset is smaller than a few thousand instances. If it contains 100,000 instances, setting `presort=True` will considerably slow down training.

For the solutions to exercises 7 and 8, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

2 $\log_2$ is the binary log; $\log_2(m) = \log(m) / \log(2)$.

Chapter 7: Ensemble Learning and Random Forests

1. If you have trained five different models and they all achieve 95% precision, you can try combining them into a voting ensemble, which will often give you even better results. It works better if the models are very different (e.g., an SVM classifier, a Decision Tree classifier, a Logistic Regression classifier, and so on). It is even better if they are trained on different training instances (that's the whole point of bagging and pasting ensembles), but if not this will still be effective as long as the models are very different.
2. A hard voting classifier just counts the votes of each classifier in the ensemble and picks the class that gets the most votes. A soft voting classifier computes the average estimated class probability for each class and picks the class with the highest probability. This gives high-confidence votes more weight and often performs better, but it works only if every classifier is able to estimate class probabilities (e.g., for the SVM classifiers in Scikit-Learn you must set `probability=True`).
3. It is quite possible to speed up training of a bagging ensemble by distributing it across multiple servers, since each predictor in the ensemble is independent of the others. The same goes for pasting ensembles and Random Forests, for the same reason. However, each predictor in a boosting ensemble is built based on the previous predictor, so training is necessarily sequential, and you will not gain anything by distributing training across multiple servers. Regarding stacking ensembles, all the predictors in a given layer are independent of each other, so they can be trained in parallel on multiple servers. However, the predictors in one layer can only be trained after the predictors in the previous layer have all been trained.
4. With out-of-bag evaluation, each predictor in a bagging ensemble is evaluated using instances that it was not trained on (they were held out). This makes it possible to have a fairly unbiased evaluation of the ensemble without the need for an additional validation set. Thus, you have more instances available for training, and your ensemble can perform slightly better.
5. When you are growing a tree in a Random Forest, only a random subset of the features is considered for splitting at each node. This is true as well for Extra-Trees, but they go one step further: rather than searching for the best possible thresholds, like regular Decision Trees do, they use random thresholds for each feature. This extra randomness acts like a form of regularization: if a Random Forest overfits the training data, Extra-Trees might perform better. Moreover, since Extra-Trees don't search for the best possible thresholds, they are much faster to train than Random Forests. However, they are neither faster nor slower than Random Forests when making predictions.

6. If your AdaBoost ensemble underfits the training data, you can try increasing the number of estimators or reducing the regularization hyperparameters of the base estimator. You may also try slightly increasing the learning rate.
7. If your Gradient Boosting ensemble overfits the training set, you should try decreasing the learning rate. You could also use early stopping to find the right number of predictors (you probably have too many).

For the solutions to exercises 8 and 9, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 8: Dimensionality Reduction

1. The main motivations for dimensionality reduction are:

   - To speed up a subsequent training algorithm (in some cases it may even remove noise and redundant features, making the training algorithm perform better)
   - To visualize the data and gain insights on the most important features
   - To save space (compression)

   The main drawbacks are:

   - Some information is lost, possibly degrading the performance of subsequent training algorithms.
   - It can be computationally intensive.
   - It adds some complexity to your Machine Learning pipelines.
   - Transformed features are often hard to interpret.

2. The curse of dimensionality refers to the fact that many problems that do not exist in low-dimensional space arise in high-dimensional space. In Machine Learning, one common manifestation is the fact that randomly sampled high-dimensional vectors are generally very sparse, increasing the risk of overfitting and making it very difficult to identify patterns in the data without having plenty of training data.
3. Once a dataset's dimensionality has been reduced using one of the algorithms we discussed, it is almost always impossible to perfectly reverse the operation, because some information gets lost during dimensionality reduction. Moreover, while some algorithms (such as PCA) have a simple reverse transformation procedure that can reconstruct a dataset relatively similar to the original, other algorithms (such as T-SNE) do not.

4. PCA can be used to significantly reduce the dimensionality of most datasets, even if they are highly nonlinear, because it can at least get rid of useless dimensions. However, if there are no useless dimensions—as in a Swiss roll dataset—then reducing dimensionality with PCA will lose too much information. You want to unroll the Swiss roll, not squash it.
5. That's a trick question: it depends on the dataset. Let's look at two extreme examples. First, suppose the dataset is composed of points that are almost perfectly aligned. In this case, PCA can reduce the dataset down to just one dimension while still preserving 95% of the variance. Now imagine that the dataset is composed of perfectly random points, scattered all around the 1,000 dimensions. In this case roughly 950 dimensions are required to preserve 95% of the variance. So the answer is, it depends on the dataset, and it could be any number between 1 and 950. Plotting the explained variance as a function of the number of dimensions is one way to get a rough idea of the dataset's intrinsic dimensionality.
6. Regular PCA is the default, but it works only if the dataset fits in memory. Incremental PCA is useful for large datasets that don't fit in memory, but it is slower than regular PCA, so if the dataset fits in memory you should prefer regular PCA. Incremental PCA is also useful for online tasks, when you need to apply PCA on the fly, every time a new instance arrives. Randomized PCA is useful when you want to considerably reduce dimensionality and the dataset fits in memory; in this case, it is much faster than regular PCA. Finally, Kernel PCA is useful for nonlinear datasets.
7. Intuitively, a dimensionality reduction algorithm performs well if it eliminates a lot of dimensions from the dataset without losing too much information. One way to measure this is to apply the reverse transformation and measure the reconstruction error. However, not all dimensionality reduction algorithms provide a reverse transformation. Alternatively, if you are using dimensionality reduction as a preprocessing step before another Machine Learning algorithm (e.g., a Random Forest classifier), then you can simply measure the performance of that second algorithm; if dimensionality reduction did not lose too much information, then the algorithm should perform just as well as when using the original dataset.
8. It can absolutely make sense to chain two different dimensionality reduction algorithms. A common example is using PCA to quickly get rid of a large number of useless dimensions, then applying another much slower dimensionality reduction algorithm, such as LLE. This two-step approach will likely yield the same performance as using LLE only, but in a fraction of the time.

For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 9: Unsupervised Learning Techniques

1. In Machine Learning, clustering is the unsupervised task of grouping similar instances together. The notion of similarity depends on the task at hand: for example, in some cases two nearby instances will be considered similar, while in others similar instances may be far apart as long as they belong to the same densely packed group. Popular clustering algorithms include K-Means, DBSCAN, agglomerative clustering, BIRCH, Mean-Shift, affinity propagation, and spectral clustering.
2. The main applications of clustering algorithms include data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, anomaly detection, and novelty detection.
3. The elbow rule is a simple technique to select the number of clusters when using K-Means: just plot the inertia (the mean squared distance from each instance to its nearest centroid) as a function of the number of clusters, and find the point in the curve where the inertia stops dropping fast (the "elbow"). This is generally close to the optimal number of clusters. Another approach is to plot the silhouette score as a function of the number of clusters. There will often be a peak, and the optimal number of clusters is generally nearby. The silhouette score is the mean silhouette coefficient over all instances. This coefficient varies from +1 for instances that are well inside their cluster and far from other clusters, to –1 for instances that are very close to another cluster. You may also plot the silhouette diagrams and perform a more thorough analysis.
4. Labeling a dataset is costly and time-consuming. Therefore, it is common to have plenty of unlabeled instances, but few labeled instances. Label propagation is a technique that consists in copying some (or all) of the labels from the labeled instances to similar unlabeled instances. This can greatly extend the number of labeled instances, and thereby allow a supervised algorithm to reach better performance (this is a form of semi-supervised learning). One approach is to use a clustering algorithm such as K-Means on all the instances, then for each cluster find the most common label or the label of the most representative instance (i.e., the one closest to the centroid) and propagate it to the unlabeled instances in the same cluster.
5. K-Means and BIRCH scale well to large datasets. DBSCAN and Mean-Shift look for regions of high density.
6. Active learning is useful whenever you have plenty of unlabeled instances but labeling is costly. In this case (which is very common), rather than randomly selecting instances to label, it is often preferable to perform active learning, where human experts interact with the learning algorithm, providing labels for

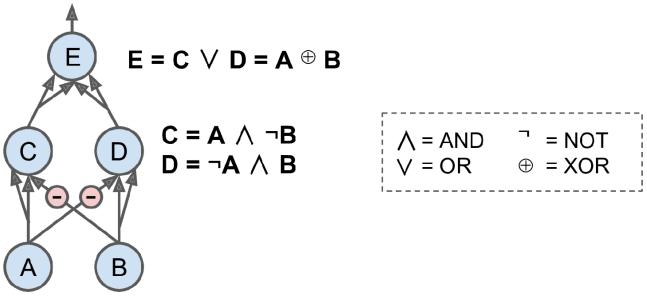
specific instances when the algorithm requests them. A common approach is uncertainty sampling (see the description in "Active Learning" on page 255).

7. Many people use the terms *anomaly detection* and *novelty detection* interchangeably, but they are not exactly the same. In anomaly detection, the algorithm is trained on a dataset that may contain outliers, and the goal is typically to identify these outliers (within the training set), as well as outliers among new instances. In novelty detection, the algorithm is trained on a dataset that is presumed to be "clean," and the objective is to detect novelties strictly among new instances. Some algorithms work best for anomaly detection (e.g., Isolation Forest), while others are better suited for novelty detection (e.g., one-class SVM).
8. A Gaussian mixture model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. In other words, the assumption is that the data is grouped into a finite number of clusters, each with an ellipsoidal shape (but the clusters may have different ellipsoidal shapes, sizes, orientations, and densities), and we don't know which cluster each instance belongs to. This model is useful for density estimation, clustering, and anomaly detection.
9. One way to find the right number of clusters when using a Gaussian mixture model is to plot the Bayesian information criterion (BIC) or the Akaike information criterion (AIC) as a function of the number of clusters, then choose the number of clusters that minimizes the BIC or AIC. Another technique is to use a Bayesian Gaussian mixture model, which automatically selects the number of clusters.

For the solutions to exercises 10 to 13, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 10: Introduction to Artificial Neural Networks with Keras

1. Visit the TensorFlow Playground and play around with it, as described in this exercise.
2. Here is a neural network based on the original artificial neurons that computes $A \oplus B$ (where $\oplus$ represents the exclusive OR), using the fact that $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. There are other solutions—for example, using the fact that $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$, or the fact that $A \oplus B = (A \vee B) \wedge (\neg A \vee \wedge B)$, and so on.



3. A classical Perceptron will converge only if the dataset is linearly separable, and it won't be able to estimate class probabilities. In contrast, a Logistic Regression classifier will converge to a good solution even if the dataset is not linearly separable, and it will output class probabilities. If you change the Perceptron's activation function to the logistic activation function (or the softmax activation function if there are multiple neurons), and if you train it using Gradient Descent (or some other optimization algorithm minimizing the cost function, typically cross entropy), then it becomes equivalent to a Logistic Regression classifier.
4. The logistic activation function was a key ingredient in training the first MLPs because its derivative is always nonzero, so Gradient Descent can always roll down the slope. When the activation function is a step function, Gradient Descent cannot move, as there is no slope at all.
5. Popular activation functions include the step function, the logistic (sigmoid) function, the hyperbolic tangent (tanh) function, and the Rectified Linear Unit (ReLU) function (see Figure 10-8). See Chapter 11 for other examples, such as ELU and variants of the ReLU function.
6. Considering the MLP described in the question, composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons, where all artificial neurons use the ReLU activation function: ..The shape of the input matrix X is $m \times 10$, where m represents the training batch size.
   a. The shape of the hidden layer's weight vector W_h is 10×50, and the length of its bias vector b_h is 50.
   b. The shape of the output layer's weight vector W_o is 50×3, and the length of its bias vector b_o is 3.
   c. The shape of the network's output matrix Y is $m \times 3$.
   d. $Y^* = \text{ReLU}(\text{ReLU}(X\ W_h + b_h)\ W_o + b_o)$. Recall that the ReLU function just sets every negative number in the matrix to zero. Also note that when you are adding a bias vector to a matrix, it is added to every single row in the matrix, which is called *broadcasting*.

7. To classify email into spam or ham, you just need one neuron in the output layer of a neural network—for example, indicating the probability that the email is spam. You would typically use the logistic activation function in the output layer when estimating a probability. If instead you want to tackle MNIST, you need 10 neurons in the output layer, and you must replace the logistic function with the softmax activation function, which can handle multiple classes, outputting one probability per class. If you want your neural network to predict housing prices like in Chapter 2, then you need one output neuron, using no activation function at all in the output layer.[3]
8. Backpropagation is a technique used to train artificial neural networks. It first computes the gradients of the cost function with regard to every model parameter (all the weights and biases), then it performs a Gradient Descent step using these gradients. This backpropagation step is typically performed thousands or millions of times, using many training batches, until the model parameters converge to values that (hopefully) minimize the cost function. To compute the gradients, backpropagation uses reverse-mode autodiff (although it wasn't called that when backpropagation was invented, and it has been reinvented several times). Reverse-mode autodiff performs a forward pass through a computation graph, computing every node's value for the current training batch, and then it performs a reverse pass, computing all the gradients at once (see Appendix D for more details). So what's the difference? Well, backpropagation refers to the whole process of training an artificial neural network using multiple backpropagation steps, each of which computes gradients and uses them to perform a Gradient Descent step. In contrast, reverse-mode autodiff is just a technique to compute gradients efficiently, and it happens to be used by backpropagation.
9. Here is a list of all the hyperparameters you can tweak in a basic MLP: the number of hidden layers, the number of neurons in each hidden layer, and the activation function used in each hidden layer and in the output layer.[4] In general, the ReLU activation function (or one of its variants; see Chapter 11) is a good default for the hidden layers. For the output layer, in general you will want the logistic activation function for binary classification, the softmax activation function for multiclass classification, or no activation function for regression.

3 When the values to predict can vary by many orders of magnitude, you may want to predict the logarithm of the target value rather than the target value directly. Simply computing the exponential of the neural network's output will give you the estimated value (since $\exp(\log v) = v$).

4 In Chapter 11 we discuss many techniques that introduce additional hyperparameters: type of weight initialization, activation function hyperparameters (e.g., the amount of leak in leaky ReLU), Gradient Clipping threshold, type of optimizer and its hyperparameters (e.g., the momentum hyperparameter when using a `MomentumOptimizer`), type of regularization for each layer and regularization hyperparameters (e.g., dropout rate when using dropout), and so on.

If the MLP overfits the training data, you can try reducing the number of hidden layers and reducing the number of neurons per hidden layer.

10. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 11: Training Deep Neural Networks

1. No, all weights should be sampled independently; they should not all have the same initial value. One important goal of sampling weights randomly is to break symmetry: if all the weights have the same initial value, even if that value is not zero, then symmetry is not broken (i.e., all neurons in a given layer are equivalent), and backpropagation will be unable to break it. Concretely, this means that all the neurons in any given layer will always have the same weights. It's like having just one neuron per layer, and much slower. It is virtually impossible for such a configuration to converge to a good solution.
2. It is perfectly fine to initialize the bias terms to zero. Some people like to initialize them just like weights, and that's okay too; it does not make much difference.
3. A few advantages of the SELU function over the ReLU function are:

   - It can take on negative values, so the average output of the neurons in any given layer is typically closer to zero than when using the ReLU activation function (which never outputs negative values). This helps alleviate the vanishing gradients problem.
   - It always has a nonzero derivative, which avoids the dying units issue that can affect ReLU units.
   - When the conditions are right (i.e., if the model is sequential, and the weights are initialized using LeCun initialization, and the inputs are standardized, and there's no incompatible layer or regularization, such as dropout or ℓ_1 regularization), then the SELU activation function ensures the model is self-normalized, which solves the exploding/vanishing gradients problems.

4. The SELU activation function is a good default. If you need the neural network to be as fast as possible, you can use one of the leaky ReLU variants instead (e.g., a simple leaky ReLU using the default hyperparameter value). The simplicity of the ReLU activation function makes it many people's preferred option, despite the fact that it is generally outperformed by SELU and leaky ReLU. However, the ReLU activation function's ability to output precisely zero can be useful in some cases (e.g., see Chapter 17). Moreover, it can sometimes benefit from optimized implementation as well as from hardware acceleration. The hyperbolic tangent (tanh) can be useful in the output layer if you need to output a number between –1 and 1, but nowadays it is not used much in hidden layers (except in recurrent

nets). The logistic activation function is also useful in the output layer when you need to estimate a probability (e.g., for binary classification), but is rarely used in hidden layers (there are exceptions—for example, for the coding layer of variational autoencoders; see Chapter 17). Finally, the softmax activation function is useful in the output layer to output probabilities for mutually exclusive classes, but it is rarely (if ever) used in hidden layers.

5. If you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an `SGD` optimizer, then the algorithm will likely pick up a lot of speed, hopefully moving roughly toward the global minimum, but its momentum will carry it right past the minimum. Then it will slow down and come back, accelerate again, overshoot again, and so on. It may oscillate this way many times before converging, so overall it will take much longer to converge than with a smaller `momentum` value.
6. One way to produce a sparse model (i.e., with most weights equal to zero) is to train the model normally, then zero out tiny weights. For more sparsity, you can apply ℓ_1 regularization during training, which pushes the optimizer toward sparsity. A third option is to use the TensorFlow Model Optimization Toolkit.
7. Yes, dropout does slow down training, in general roughly by a factor of two. However, it has no impact on inference speed since it is only turned on during training. MC Dropout is exactly like dropout during training, but it is still active during inference, so each inference is slowed down slightly. More importantly, when using MC Dropout you generally want to run inference 10 times or more to get better predictions. This means that making predictions is slowed down by a factor of 10 or more.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 12: Custom Models and Training with TensorFlow

1. TensorFlow is an open-source library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning. Its core is similar to NumPy, but it also features GPU support, support for distributed computing, computation graph analysis and optimization capabilities (with a portable graph format that allows you to train a TensorFlow model in one environment and run it in another), an optimization API based on reverse-mode autodiff, and several powerful APIs such as tf.keras, tf.data, tf.image, tf.signal, and more. Other popular Deep Learning libraries include PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2, and Chainer.
2. Although TensorFlow offers most of the functionalities provided by NumPy, it is not a drop-in replacement, for a few reasons. First, the names of the functions are

not always the same (for example, `tf.reduce_sum()` versus `np.sum()`). Second, some functions do not behave in exactly the same way (for example, `tf.transpose()` creates a transposed copy of a tensor, while NumPy's `T` attribute creates a transposed view, without actually copying any data). Lastly, NumPy arrays are mutable, while TensorFlow tensors are not (but you can use a `tf.Variable` if you need a mutable object).

3. Both `tf.range(10)` and `tf.constant(np.arange(10))` return a one-dimensional tensor containing the integers 0 to 9. However, the former uses 32-bit integers while the latter uses 64-bit integers. Indeed, TensorFlow defaults to 32 bits, while NumPy defaults to 64 bits.
4. Beyond regular tensors, TensorFlow offers several other data structures, including sparse tensors, tensor arrays, ragged tensors, queues, string tensors, and sets. The last two are actually represented as regular tensors, but TensorFlow provides special functions to manipulate them (in `tf.strings` and `tf.sets`).
5. When you want to define a custom loss function, in general you can just implement it as a regular Python function. However, if your custom loss function must support some hyperparameters (or any other state), then you should subclass the `keras.losses.Loss` class and implement the `__init__()` and `call()` methods. If you want the loss function's hyperparameters to be saved along with the model, then you must also implement the `get_config()` method.
6. Much like custom loss functions, most metrics can be defined as regular Python functions. But if you want your custom metric to support some hyperparameters (or any other state), then you should subclass the `keras.metrics.Metric` class. Moreover, if computing the metric over a whole epoch is not equivalent to computing the mean metric over all batches in that epoch (e.g., as for the precision and recall metrics), then you should subclass the `keras.metrics.Metric` class and implement the `__init__()`, `update_state()`, and `result()` methods to keep track of a running metric during each epoch. You should also implement the `reset_states()` method unless all it needs to do is reset all variables to 0.0. If you want the state to be saved along with the model, then you should implement the `get_config()` method as well.
7. You should distinguish the internal components of your model (i.e., layers or reusable blocks of layers) from the model itself (i.e., the object you will train). The former should subclass the `keras.layers.Layer` class, while the latter should subclass the `keras.models.Model` class.
8. Writing your own custom training loop is fairly advanced, so you should only do it if you really need to. Keras provides several tools to customize training without having to write a custom training loop: callbacks, custom regularizers, custom constraints, custom losses, and so on. You should use these instead of writing a custom training loop whenever possible: writing a custom training loop is more

error-prone, and it will be harder to reuse the custom code you write. However, in some cases writing a custom training loop is necessary—for example, if you want to use different optimizers for different parts of your neural network, like in the Wide & Deep paper. A custom training loop can also be useful when debugging, or when trying to understand exactly how training works.

9. Custom Keras components should be convertible to TF Functions, which means they should stick to TF operations as much as possible and respect all the rules listed in "TF Function Rules" on page 409. If you absolutely need to include arbitrary Python code in a custom component, you can either wrap it in a `tf.py_function()` operation (but this will reduce performance and limit your model's portability) or set `dynamic=True` when creating the custom layer or model (or set `run_eagerly=True` when calling the model's `compile()` method).
10. Please refer to "TF Function Rules" on page 409 for the list of rules to respect when creating a TF Function.
11. Creating a dynamic Keras model can be useful for debugging, as it will not compile any custom component to a TF Function, and you can use any Python debugger to debug your code. It can also be useful if you want to include arbitrary Python code in your model (or in your training code), including calls to external libraries. To make a model dynamic, you must set `dynamic=True` when creating it. Alternatively, you can set `run_eagerly=True` when calling the model's `compile()` method. Making a model dynamic prevents Keras from using any of TensorFlow's graph features, so it will slow down training and inference, and you will not have the possibility to export the computation graph, which will limit your model's portability.

For the solutions to exercises 12 and 13, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 13: Loading and Preprocessing Data with TensorFlow

1. Ingesting a large dataset and preprocessing it efficiently can be a complex engineering challenge. The Data API makes it fairly simple. It offers many features, including loading data from various sources (such as text or binary files), reading data in parallel from multiple sources, transforming it, interleaving the records, shuffling the data, batching it, and prefetching it.
2. Splitting a large dataset into multiple files makes it possible to shuffle it at a coarse level before shuffling it at a finer level using a shuffling buffer. It also makes it possible to handle huge datasets that do not fit on a single machine. It's also simpler to manipulate thousands of small files rather than one huge file; for

example, it's easier to split the data into multiple subsets. Lastly, if the data is split across multiple files spread across multiple servers, it is possible to download several files from different servers simultaneously, which improves the bandwidth usage.

3. You can use TensorBoard to visualize profiling data: if the GPU is not fully utilized then your input pipeline is likely to be the bottleneck. You can fix it by making sure it reads and preprocesses the data in multiple threads in parallel, and ensuring it prefetches a few batches. If this is insufficient to get your GPU to 100% usage during training, make sure your preprocessing code is optimized. You can also try saving the dataset into multiple TFRecord files, and if necessary perform some of the preprocessing ahead of time so that it does not need to be done on the fly during training (TF Transform can help with this). If necessary, use a machine with more CPU and RAM, and ensure that the GPU bandwidth is large enough.
4. A TFRecord file is composed of a sequence of arbitrary binary records: you can store absolutely any binary data you want in each record. However, in practice most TFRecord files contain sequences of serialized protocol buffers. This makes it possible to benefit from the advantages of protocol buffers, such as the fact that they can be read easily across multiple platforms and languages and their definition can be updated later in a backward-compatible way.
5. The `Example` protobuf format has the advantage that TensorFlow provides some operations to parse it (the `tf.io.parse*example()` functions) without you having to define your own format. It is sufficiently flexible to represent instances in most datasets. However, if it does not cover your use case, you can define your own protocol buffer, compile it using `protoc` (setting the `--descriptor_set_out` and `--include_imports` arguments to export the protobuf descriptor), and use the `tf.io.decode_proto()` function to parse the serialized protobufs (see the "Custom protobuf" section of the notebook for an example). It's more complicated, and it requires deploying the descriptor along with the model, but it can be done.
6. When using TFRecords, you will generally want to activate compression if the TFRecord files will need to be downloaded by the training script, as compression will make files smaller and thus reduce download time. But if the files are located on the same machine as the training script, it's usually preferable to leave compression off, to avoid wasting CPU for decompression.
7. Let's look at the pros and cons of each preprocessing option:

    - If you preprocess the data when creating the data files, the training script will run faster, since it will not have to perform preprocessing on the fly. In some cases, the preprocessed data will also be much smaller than the original data, so you can save some space and speed up downloads. It may also be helpful to

materialize the preprocessed data, for example to inspect it or archive it. However, this approach has a few cons. First, it's not easy to experiment with various preprocessing logics if you need to generate a preprocessed dataset for each variant. Second, if you want to perform data augmentation, you have to materialize many variants of your dataset, which will use a large amount of disk space and take a lot of time to generate. Lastly, the trained model will expect preprocessed data, so you will have to add preprocessing code in your application before it calls the model.

- If the data is preprocessed with the tf.data pipeline, it's much easier to tweak the preprocessing logic and apply data augmentation. Also, tf.data makes it easy to build highly efficient preprocessing pipelines (e.g., with multithreading and prefetching). However, preprocessing the data this way will slow down training. Moreover, each training instance will be preprocessed once per epoch rather than just once if the data was preprocessed when creating the data files. Lastly, the trained model will still expect preprocessed data.
- If you add preprocessing layers to your model, you will only have to write the preprocessing code once for both training and inference. If your model needs to be deployed to many different platforms, you will not need to write the preprocessing code multiple times. Plus, you will not run the risk of using the wrong preprocessing logic for your model, since it will be part of the model. On the downside, preprocessing the data will slow down training, and each training instance will be preprocessed once per epoch. Moreover, by default the preprocessing operations will run on the GPU for the current batch (you will not benefit from parallel preprocessing on the CPU, and prefetching). Fortunately, the upcoming Keras preprocessing layers should be able to lift the preprocessing operations from the preprocessing layers and run them as part of the tf.data pipeline, so you will benefit from multithreaded execution on the CPU and prefetching.
- Lastly, using TF Transform for preprocessing gives you many of the benefits from the previous options: the preprocessed data is materialized, each instance is preprocessed just once (speeding up training), and preprocessing layers get generated automatically so you only need to write the preprocessing code once. The main drawback is the fact that you need to learn how to use this tool.

8. Let's look at how to encode categorical features and text:

   - To encode a categorical feature that has a natural order, such as a movie rating (e.g., "bad," "average," "good"), the simplest option is to use ordinal encoding: sort the categories in their natural order and map each category to its rank (e.g., "bad" maps to 0, "average" maps to 1, and "good" maps to 2). However, most categorical features don't have such a natural order. For example, there's

no natural order for professions or countries. In this case, you can use one-hot encoding or, if there are many categories, embeddings.

- For text, one option is to use a bag-of-words representation: a sentence is represented by a vector counting the counts of each possible word. Since common words are usually not very important, you'll want to use TF-IDF to reduce their weight. Instead of counting words, it is also common to count *n*-grams, which are sequences of *n* consecutive words—nice and simple. Alternatively, you can encode each word using word embeddings, possibly pretrained. Rather than encoding words, it is also possible to encode each letter, or subword tokens (e.g., splitting "smartest" into "smart" and "est"). These last two options are discussed in Chapter 16.

For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 14: Deep Computer Vision Using Convolutional Neural Networks

1. These are the main advantages of a CNN over a fully connected DNN for image classification:

   - Because consecutive layers are only partially connected and because it heavily reuses its weights, a CNN has many fewer parameters than a fully connected DNN, which makes it much faster to train, reduces the risk of overfitting, and requires much less training data.
   - When a CNN has learned a kernel that can detect a particular feature, it can detect that feature anywhere in the image. In contrast, when a DNN learns a feature in one location, it can detect it only in that particular location. Since images typically have very repetitive features, CNNs are able to generalize much better than DNNs for image processing tasks such as classification, using fewer training examples.
   - Finally, a DNN has no prior knowledge of how pixels are organized; it does not know that nearby pixels are close. A CNN's architecture embeds this prior knowledge. Lower layers typically identify features in small areas of the images, while higher layers combine the lower-level features into larger features. This works well with most natural images, giving CNNs a decisive head start compared to DNNs.

2. Let's compute how many parameters the CNN has. Since its first convolutional layer has 3×3 kernels, and the input has three channels (red, green, and blue), each feature map has $3 \times 3 \times 3$ weights, plus a bias term. That's 28 parameters per

feature map. Since this first convolutional layer has 100 feature maps, it has a total of 2,800 parameters. The second convolutional layer has 3 × 3 kernels and its input is the set of 100 feature maps of the previous layer, so each feature map has 3 × 3 × 100 = 900 weights, plus a bias term. Since it has 200 feature maps, this layer has 901 × 200 = 180,200 parameters. Finally, the third and last convolutional layer also has 3 × 3 kernels, and its input is the set of 200 feature maps of the previous layers, so each feature map has 3 × 3 × 200 = 1,800 weights, plus a bias term. Since it has 400 feature maps, this layer has a total of 1,801 × 400 = 720,400 parameters. All in all, the CNN has 2,800 + 180,200 + 720,400 = 903,400 parameters.

Now let's compute how much RAM this neural network will require (at least) when making a prediction for a single instance. First let's compute the feature map size for each layer. Since we are using a stride of 2 and `"same"` padding, the horizontal and vertical dimensions of the feature maps are divided by 2 at each layer (rounding up if necessary). So, as the input channels are 200 × 300 pixels, the first layer's feature maps are 100 × 150, the second layer's feature maps are 50 × 75, and the third layer's feature maps are 25 × 38. Since 32 bits is 4 bytes and the first convolutional layer has 100 feature maps, this first layer takes up 4 × 100 × 150 × 100 = 6 million bytes (6 MB). The second layer takes up 4 × 50 × 75 × 200 = 3 million bytes (3 MB). Finally, the third layer takes up 4 × 25 × 38 × 400 = 1,520,000 bytes (about 1.5 MB). However, once a layer has been computed, the memory occupied by the previous layer can be released, so if everything is well optimized, only 6 + 3 = 9 million bytes (9 MB) of RAM will be required (when the second layer has just been computed, but the memory occupied by the first layer has not been released yet). But wait, you also need to add the memory occupied by the CNN's parameters! We computed earlier that it has 903,400 parameters, each using up 4 bytes, so this adds 3,613,600 bytes (about 3.6 MB). The total RAM required is therefore (at least) 12,613,600 bytes (about 12.6 MB).

Lastly, let's compute the minimum amount of RAM required when training the CNN on a mini-batch of 50 images. During training TensorFlow uses backpropagation, which requires keeping all values computed during the forward pass until the reverse pass begins. So we must compute the total RAM required by all layers for a single instance and multiply that by 50. At this point, let's start counting in megabytes rather than bytes. We computed before that the three layers require respectively 6, 3, and 1.5 MB for each instance. That's a total of 10.5 MB per instance, so for 50 instances the total RAM required is 525 MB. Add to that the RAM required by the input images, which is 50 × 4 × 200 × 300 × 3 = 36 million bytes (36 MB), plus the RAM required for the model parameters, which is about 3.6 MB (computed earlier), plus some RAM for the gradients (we will neglect this since it can be released gradually as backpropagation goes down the layers during the reverse pass). We are up to a total of roughly 525 + 36 + 3.6 = 564.6 MB, and that's really an optimistic bare minimum.

3. If your GPU runs out of memory while training a CNN, here are five things you could try to solve the problem (other than purchasing a GPU with more RAM):
   - Reduce the mini-batch size.
   - Reduce dimensionality using a larger stride in one or more layers.
   - Remove one or more layers.
   - Use 16-bit floats instead of 32-bit floats.
   - Distribute the CNN across multiple devices.
4. A max pooling layer has no parameters at all, whereas a convolutional layer has quite a few (see the previous questions).
5. A local response normalization layer makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps, which encourages different feature maps to specialize and pushes them apart, forcing them to explore a wider range of features. It is typically used in the lower layers to have a larger pool of low-level features that the upper layers can build upon.
6. The main innovations in AlexNet compared to LeNet-5 are that it is much larger and deeper, and it stacks convolutional layers directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer. The main innovation in GoogLeNet is the introduction of *inception modules*, which make it possible to have a much deeper net than previous CNN architectures, with fewer parameters. ResNet's main innovation is the introduction of skip connections, which make it possible to go well beyond 100 layers. Arguably, its simplicity and consistency are also rather innovative. SENet's main innovation was the idea of using an SE block (a two-layer dense network) after every inception module in an inception network or every residual unit in a ResNet to recalibrate the relative importance of feature maps. Finally, Xception's main innovation was the use of depthwise separable convolutional layers, which look at spatial patterns and depthwise patterns separately.
7. Fully convolutional networks are neural networks composed exclusively of convolutional and pooling layers. FCNs can efficiently process images of any width and height (at least above the minimum size). They are most useful for object detection and semantic segmentation because they only need to look at the image once (instead of having to run a CNN multiple times on different parts of the image). If you have a CNN with some dense layers on top, you can convert these dense layers to convolutional layers to create an FCN: just replace the lowest dense layer with a convolutional layer with a kernel size equal to the layer's input size, with one filter per neuron in the dense layer, and using `"valid"` padding. Generally the stride should be 1, but you can set it to a higher value if you want. The activation function should be the same as the dense layer's. The other dense layers should be converted the same way, but using 1×1 filters. It is actually pos-

sible to convert a trained CNN this way by appropriately reshaping the dense layers' weight matrices.

8. The main technical difficulty of semantic segmentation is the fact that a lot of the spatial information gets lost in a CNN as the signal flows through each layer, especially in pooling layers and layers with a stride greater than 1. This spatial information needs to be restored somehow to accurately predict the class of each pixel.

For the solutions to exercises 9 to 12, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 15: Processing Sequences Using RNNs and CNNs

1. Here are a few RNN applications:

   - For a sequence-to-sequence RNN: predicting the weather (or any other time series), machine translation (using an Encoder–Decoder architecture), video captioning, speech to text, music generation (or other sequence generation), identifying the chords of a song
   - For a sequence-to-vector RNN: classifying music samples by music genre, analyzing the sentiment of a book review, predicting what word an aphasic patient is thinking of based on readings from brain implants, predicting the probability that a user will want to watch a movie based on their watch history (this is one of many possible implementations of *collaborative filtering* for a recommender system)
   - For a vector-to-sequence RNN: image captioning, creating a music playlist based on an embedding of the current artist, generating a melody based on a set of parameters, locating pedestrians in a picture (e.g., a video frame from a self-driving car's camera)

2. An RNN layer must have three-dimensional inputs: the first dimension is the batch dimension (its size is the batch size), the second dimension represents the time (its size is the number of time steps), and the third dimension holds the inputs at each time step (its size is the number of input features per time step). For example, if you want to process a batch containing 5 time series of 10 time steps each, with 2 values per time step (e.g., the temperature and the wind speed), the shape will be [5, 10, 2]. The outputs are also three-dimensional, with the same first two dimensions, but the last dimension is equal to the number of neurons. For example, if an RNN layer with 32 neurons processes the batch we just discussed, the output will have a shape of [5, 10, 32].

3. To build a deep sequence-to-sequence RNN using Keras, you must set `return_sequences=True` for all RNN layers. To build a sequence-to-vector RNN, you must set `return_sequences=True` for all RNN layers except for the top RNN layer, which must have `return_sequences=False` (or do not set this argument at all, since `False` is the default).
4. If you have a daily univariate time series, and you want to forecast the next seven days, the simplest RNN architecture you can use is a stack of RNN layers (all with `return_sequences=True` except for the top RNN layer), using seven neurons in the output RNN layer. You can then train this model using random windows from the time series (e.g., sequences of 30 consecutive days as the inputs, and a vector containing the values of the next 7 days as the target). This is a sequence-to-vector RNN. Alternatively, you could set `return_sequences=True` for all RNN layers to create a sequence-to-sequence RNN. You can train this model using random windows from the time series, with sequences of the same length as the inputs as the targets. Each target sequence should have seven values per time step (e.g., for time step t, the target should be a vector containing the values at time steps $t + 1$ to $t + 7$).
5. The two main difficulties when training RNNs are unstable gradients (exploding or vanishing) and a very limited short-term memory. These problems both get worse when dealing with long sequences. To alleviate the unstable gradients problem, you can use a smaller learning rate, use a saturating activation function such as the hyperbolic tangent (which is the default), and possibly use gradient clipping, Layer Normalization, or dropout at each time step. To tackle the limited short-term memory problem, you can use `LSTM` or `GRU` layers (this also helps with the unstable gradients problem).
6. An LSTM cell's architecture looks complicated, but it's actually not too hard if you understand the underlying logic. The cell has a short-term state vector and a long-term state vector. At each time step, the inputs and the previous short-term state are fed to a simple RNN cell and three gates: the forget gate decides what to remove from the long-term state, the input gate decides which part of the output of the simple RNN cell should be added to the long-term state, and the output gate decides which part of the long-term state should be output at this time step (after going through the tanh activation function). The new short-term state is equal to the output of the cell. See Figure 15-9.
7. An RNN layer is fundamentally sequential: in order to compute the outputs at time step t, it has to first compute the outputs at all earlier time steps. This makes it impossible to parallelize. On the other hand, a 1D convolutional layer lends itself well to parallelization since it does not hold a state between time steps. In other words, it has no memory: the output at any time step can be computed based only on a small window of values from the inputs without having to know all the past values. Moreover, since a 1D convolutional layer is not recurrent, it

suffers less from unstable gradients. One or more 1D convolutional layers can be useful in an RNN to efficiently preprocess the inputs, for example to reduce their temporal resolution (downsampling) and thereby help the RNN layers detect long-term patterns. In fact, it is possible to use only convolutional layers, for example by building a WaveNet architecture.

8. To classify videos based on their visual content, one possible architecture could be to take (say) one frame per second, then run every frame through the same convolutional neural network (e.g., a pretrained Xception model, possibly frozen if your dataset is not large), feed the sequence of outputs from the CNN to a sequence-to-vector RNN, and finally run its output through a softmax layer, giving you all the class probabilities. For training you would use cross entropy as the cost function. If you wanted to use the audio for classification as well, you could use a stack of strided 1D convolutional layers to reduce the temporal resolution from thousands of audio frames per second to just one per second (to match the number of images per second), and concatenate the output sequence to the inputs of the sequence-to-vector RNN (along the last dimension).

For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 16: Natural Language Processing with RNNs and Attention

1. Stateless RNNs can only capture patterns whose length is less than, or equal to, the size of the windows the RNN is trained on. Conversely, stateful RNNs can capture longer-term patterns. However, implementing a stateful RNN is much harder—especially preparing the dataset properly. Moreover, stateful RNNs do not always work better, in part because consecutive batches are not independent and identically distributed (IID). Gradient Descent is not fond of non-IID datasets.
2. In general, if you translate a sentence one word at a time, the result will be terrible. For example, the French sentence "Je vous en prie" means "You are welcome," but if you translate it one word at a time, you get "I you in pray." Huh? It is much better to read the whole sentence first and then translate it. A plain sequence-to-sequence RNN would start translating a sentence immediately after reading the first word, while an Encoder–Decoder RNN will first read the whole sentence and then translate it. That said, one could imagine a plain sequence-to-sequence RNN that would output silence whenever it is unsure about what to say next (just like human translators do when they must translate a live broadcast).
3. Variable-length input sequences can be handled by padding the shorter sequences so that all sequences in a batch have the same length, and using masking to

ensure the RNN ignores the padding token. For better performance, you may also want to create batches containing sequences of similar sizes. Ragged tensors can hold sequences of variable lengths, and tf.keras will likely support them eventually, which will greatly simplify handling variable-length input sequences (at the time of this writing, it is not the case yet). Regarding variable-length output sequences, if the length of the output sequence is known in advance (e.g., if you know that it is the same as the input sequence), then you just need to configure the loss function so that it ignores tokens that come after the end of the sequence. Similarly, the code that will use the model should ignore tokens beyond the end of the sequence. But generally the length of the output sequence is not known ahead of time, so the solution is to train the model so that it outputs an end-of-sequence token at the end of each sequence.

4. Beam search is a technique used to improve the performance of a trained Encoder–Decoder model, for example in a neural machine translation system. The algorithm keeps track of a short list of the *k* most promising output sentences (say, the top three), and at each decoder step it tries to extend them by one word; then it keeps only the *k* most likely sentences. The parameter *k* is called the *beam width*: the larger it is, the more CPU and RAM will be used, but also the more accurate the system will be. Instead of greedily choosing the most likely next word at each step to extend a single sentence, this technique allows the system to explore several promising sentences simultaneously. Moreover, this technique lends itself well to parallelization. You can implement beam search fairly easily using TensorFlow Addons.
5. An attention mechanism is a technique initially used in Encoder–Decoder models to give the decoder more direct access to the input sequence, allowing it to deal with longer input sequences. At each decoder time step, the current decoder's state and the full output of the encoder are processed by an alignment model that outputs an alignment score for each input time step. This score indicates which part of the input is most relevant to the current decoder time step. The weighted sum of the encoder output (weighted by their alignment score) is then fed to the decoder, which produces the next decoder state and the output for this time step. The main benefit of using an attention mechanism is the fact that the Encoder–Decoder model can successfully process longer input sequences. Another benefit is that the alignment scores makes the model easier to debug and interpret: for example, if the model makes a mistake, you can look at which part of the input it was paying attention to, and this can help diagnose the issue. An attention mechanism is also at the core of the Transformer architecture, in the Multi-Head Attention layers. See the next answer.
6. The most important layer in the Transformer architecture is the Multi-Head Attention layer (the original Transformer architecture contains 18 of them, including 6 Masked Multi-Head Attention layers). It is at the core of language

models such as BERT and GPT-2. Its purpose is to allow the model to identify which words are most aligned with each other, and then improve each word's representation using these contextual clues.

7. Sampled softmax is used when training a classification model when there are many classes (e.g., thousands). It computes an approximation of the cross-entropy loss based on the logit predicted by the model for the correct class, and the predicted logits for a sample of incorrect words. This speeds up training considerably compared to computing the softmax over all logits and then estimating the cross-entropy loss. After training, the model can be used normally, using the regular softmax function to compute all the class probabilities based on all the logits.

For the solutions to exercises 8 to 11, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 17: Representation Learning and Generative Learning Using Autoencoders and GANs

1. Here are some of the main tasks that autoencoders are used for:

   - Feature extraction
   - Unsupervised pretraining
   - Dimensionality reduction
   - Generative models
   - Anomaly detection (an autoencoder is generally bad at reconstructing outliers)

2. If you want to train a classifier and you have plenty of unlabeled training data but only a few thousand labeled instances, then you could first train a deep autoencoder on the full dataset (labeled + unlabeled), then reuse its lower half for the classifier (i.e., reuse the layers up to the codings layer, included) and train the classifier using the labeled data. If you have little labeled data, you probably want to freeze the reused layers when training the classifier.
3. The fact that an autoencoder perfectly reconstructs its inputs does not necessarily mean that it is a good autoencoder; perhaps it is simply an overcomplete autoencoder that learned to copy its inputs to the codings layer and then to the outputs. In fact, even if the codings layer contained a single neuron, it would be possible for a very deep autoencoder to learn to map each training instance to a different coding (e.g., the first instance could be mapped to 0.001, the second to 0.002, the third to 0.003, and so on), and it could learn "by heart" to reconstruct the right training instance for each coding. It would perfectly reconstruct its inputs

without really learning any useful pattern in the data. In practice such a mapping is unlikely to happen, but it illustrates the fact that perfect reconstructions are not a guarantee that the autoencoder learned anything useful. However, if it produces very bad reconstructions, then it is almost guaranteed to be a bad autoencoder. To evaluate the performance of an autoencoder, one option is to measure the reconstruction loss (e.g., compute the MSE, or the mean square of the outputs minus the inputs). Again, a high reconstruction loss is a good sign that the autoencoder is bad, but a low reconstruction loss is not a guarantee that it is good. You should also evaluate the autoencoder according to what it will be used for. For example, if you are using it for unsupervised pretraining of a classifier, then you should also evaluate the classifier's performance.

4. An undercomplete autoencoder is one whose codings layer is smaller than the input and output layers. If it is larger, then it is an overcomplete autoencoder. The main risk of an excessively undercomplete autoencoder is that it may fail to reconstruct the inputs. The main risk of an overcomplete autoencoder is that it may just copy the inputs to the outputs, without learning any useful features.
5. To tie the weights of an encoder layer and its corresponding decoder layer, you simply make the decoder weights equal to the transpose of the encoder weights. This reduces the number of parameters in the model by half, often making training converge faster with less training data and reducing the risk of overfitting the training set.
6. A generative model is a model capable of randomly generating outputs that resemble the training instances. For example, once trained successfully on the MNIST dataset, a generative model can be used to randomly generate realistic images of digits. The output distribution is typically similar to the training data. For example, since MNIST contains many images of each digit, the generative model would output roughly the same number of images of each digit. Some generative models can be parametrized—for example, to generate only some kinds of outputs. An example of a generative autoencoder is the variational autoencoder.
7. A generative adversarial network is a neural network architecture composed of two parts, the generator and the discriminator, which have opposing objectives. The generator's goal is to generate instances similar to those in the training set, to fool the discriminator. The discriminator must distinguish the real instances from the generated ones. At each training iteration, the discriminator is trained like a normal binary classifier, then the generator is trained to maximize the discriminator's error. GANs are used for advanced image processing tasks such as super resolution, colorization, image editing (replacing objects with realistic background), turning a simple sketch into a photorealistic image, or predicting the next frames in a video. They are also used to augment a dataset (to train other

models), to generate other types of data (such as text, audio, and time series), and to identify the weaknesses in other models and strengthen them.

8. Training GANs is notoriously difficult, because of the complex dynamics between the generator and the discriminator. The biggest difficulty is mode collapse, where the generator produces outputs with very little diversity. Moreover, training can be terribly unstable: it may start out fine and then suddenly start oscillating or diverging, without any apparent reason. GANs are also very sensitive to the choice of hyperparameters.

For the solutions to exercises 9, 10, and 11, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 18: Reinforcement Learning

1. Reinforcement Learning is an area of Machine Learning aimed at creating agents capable of taking actions in an environment in a way that maximizes rewards over time. There are many differences between RL and regular supervised and unsupervised learning. Here are a few:

   - In supervised and unsupervised learning, the goal is generally to find patterns in the data and use them to make predictions. In Reinforcement Learning, the goal is to find a good policy.
   - Unlike in supervised learning, the agent is not explicitly given the "right" answer. It must learn by trial and error.
   - Unlike in unsupervised learning, there is a form of supervision, through rewards. We do not tell the agent how to perform the task, but we do tell it when it is making progress or when it is failing.
   - A Reinforcement Learning agent needs to find the right balance between exploring the environment, looking for new ways of getting rewards, and exploiting sources of rewards that it already knows. In contrast, supervised and unsupervised learning systems generally don't need to worry about exploration; they just feed on the training data they are given.
   - In supervised and unsupervised learning, training instances are typically independent (in fact, they are generally shuffled). In Reinforcement Learning, consecutive observations are generally *not* independent. An agent may remain in the same region of the environment for a while before it moves on, so consecutive observations will be very correlated. In some cases a replay memory (buffer) is used to ensure that the training algorithm gets fairly independent observations.

2. Here are a few possible applications of Reinforcement Learning, other than those mentioned in Chapter 18:

   Music personalization
   : The environment is a user's personalized web radio. The agent is the software deciding what song to play next for that user. Its possible actions are to play any song in the catalog (it must try to choose a song the user will enjoy) or to play an advertisement (it must try to choose an ad that the user will be interested in). It gets a small reward every time the user listens to a song, a larger reward every time the user listens to an ad, a negative reward when the user skips a song or an ad, and a very negative reward if the user leaves.

   Marketing
   : The environment is your company's marketing department. The agent is the software that defines which customers a mailing campaign should be sent to, given their profile and purchase history (for each customer it has two possible actions: send or don't send). It gets a negative reward for the cost of the mailing campaign, and a positive reward for estimated revenue generated from this campaign.

   Product delivery
   : Let the agent control a fleet of delivery trucks, deciding what they should pick up at the depots, where they should go, what they should drop off, and so on. It will get positive rewards for each product delivered on time, and negative rewards for late deliveries.

3. When estimating the value of an action, Reinforcement Learning algorithms typically sum all the rewards that this action led to, giving more weight to immediate rewards and less weight to later rewards (considering that an action has more influence on the near future than on the distant future). To model this, a discount factor is typically applied at each time step. For example, with a discount factor of 0.9, a reward of 100 that is received two time steps later is counted as only $0.9^2 \times 100 = 81$ when you are estimating the value of the action. You can think of the discount factor as a measure of how much the future is valued relative to the present: if it is very close to 1, then the future is valued almost as much as the present; if it is close to 0, then only immediate rewards matter. Of course, this impacts the optimal policy tremendously: if you value the future, you may be willing to put up with a lot of immediate pain for the prospect of eventual rewards, while if you don't value the future, you will just grab any immediate reward you can find, never investing in the future.
4. To measure the performance of a Reinforcement Learning agent, you can simply sum up the rewards it gets. In a simulated environment, you can run many episodes and look at the total rewards it gets on average (and possibly look at the min, max, standard deviation, and so on).

5. The credit assignment problem is the fact that when a Reinforcement Learning agent receives a reward, it has no direct way of knowing which of its previous actions contributed to this reward. It typically occurs when there is a large delay between an action and the resulting reward (e.g., during a game of Atari's *Pong*, there may be a few dozen time steps between the moment the agent hits the ball and the moment it wins the point). One way to alleviate it is to provide the agent with shorter-term rewards, when possible. This usually requires prior knowledge about the task. For example, if we want to build an agent that will learn to play chess, instead of giving it a reward only when it wins the game, we could give it a reward every time it captures one of the opponent's pieces.
6. An agent can often remain in the same region of its environment for a while, so all of its experiences will be very similar for that period of time. This can introduce some bias in the learning algorithm. It may tune its policy for this region of the environment, but it will not perform well as soon as it moves out of this region. To solve this problem, you can use a replay memory; instead of using only the most immediate experiences for learning, the agent will learn based on a buffer of its past experiences, recent and not so recent (perhaps this is why we dream at night: to replay our experiences of the day and better learn from them?).
7. An off-policy RL algorithm learns the value of the optimal policy (i.e., the sum of discounted rewards that can be expected for each state if the agent acts optimally) while the agent follows a different policy. Q-Learning is a good example of such an algorithm. In contrast, an on-policy algorithm learns the value of the policy that the agent actually executes, including both exploration and exploitation.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

Chapter 19: Training and Deploying TensorFlow Models at Scale

1. A SavedModel contains a TensorFlow model, including its architecture (a computation graph) and its weights. It is stored as a directory containing a *saved_model.pb* file, which defines the computation graph (represented as a serialized protocol buffer), and a *variables* subdirectory containing the variable values. For models containing a large number of weights, these variable values may be split across multiple files. A SavedModel also includes an *assets* subdirectory that may contain additional data, such as vocabulary files, class names, or some example instances for this model. To be more accurate, a SavedModel can contain one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions (including their input and output names, types, and shapes). Each metagraph is identified by a set of tags. To inspect a SavedMo-

del, you can use the command-line tool `saved_model_cli` or just load it using `tf.saved_model.load()` and inspect it in Python.

2. TF Serving allows you to deploy multiple TensorFlow models (or multiple versions of the same model) and make them accessible to all your applications easily via a REST API or a gRPC API. Using your models directly in your applications would make it harder to deploy a new version of a model across all applications. Implementing your own microservice to wrap a TF model would require extra work, and it would be hard to match TF Serving's features. TF Serving has many features: it can monitor a directory and autodeploy the models that are placed there, and you won't have to change or even restart any of your applications to benefit from the new model versions; it's fast, well tested, and scales very well; and it supports A/B testing of experimental models and deploying a new model version to just a subset of your users (in this case the model is called a *canary*). TF Serving is also capable of grouping individual requests into batches to run them jointly on the GPU. To deploy TF Serving, you can install it from source, but it is much simpler to install it using a Docker image. To deploy a cluster of TF Serving Docker images, you can use an orchestration tool such as Kubernetes, or use a fully hosted solution such as Google Cloud AI Platform.
3. To deploy a model across multiple TF Serving instances, all you need to do is configure these TF Serving instances to monitor the same *models* directory, and then export your new model as a SavedModel into a subdirectory.
4. The gRPC API is more efficient than the REST API. However, its client libraries are not as widely available, and if you activate compression when using the REST API, you can get almost the same performance. So, the gRPC API is most useful when you need the highest possible performance and the clients are not limited to the REST API.
5. To reduce a model's size so it can run on a mobile or embedded device, TFLite uses several techniques:

    - It provides a converter which can optimize a SavedModel: it shrinks the model and reduces its latency. To do this, it prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes and fuses operations whenever possible.
    - The converter can also perform post-training quantization: this technique dramatically reduces the model's size, so it's much faster to download and store.
    - It saves the optimized model using the FlatBuffer format, which can be loaded to RAM directly, without parsing. This reduces the loading time and memory footprint.

6. Quantization-aware training consists in adding fake quantization operations to the model during training. This allows the model to learn to ignore the quantization noise; the final weights will be more robust to quantization.
7. Model parallelism means chopping your model into multiple parts and running them in parallel across multiple devices, hopefully speeding up the model during training or inference. Data parallelism means creating multiple exact replicas of your model and deploying them across multiple devices. At each iteration during training, each replica is given a different batch of data, and it computes the gradients of the loss with regard to the model parameters. In synchronous data parallelism, the gradients from all replicas are then aggregated and the optimizer performs a Gradient Descent step. The parameters may be centralized (e.g., on parameter servers) or replicated across all replicas and kept in sync using AllReduce. In asynchronous data parallelism, the parameters are centralized and the replicas run independently from each other, each updating the central parameters directly at the end of each training iteration, without having to wait for the other replicas. To speed up training, data parallelism turns out to work better than model parallelism, in general. This is mostly because it requires less communication across devices. Moreover, it is much easier to implement, and it works the same way for any model, whereas model parallelism requires analyzing the model to determine the best way to chop it into pieces.
8. When training a model across multiple servers, you can use the following distribution strategies:

    - The `MultiWorkerMirroredStrategy` performs mirrored data parallelism. The model is replicated across all available servers and devices, and each replica gets a different batch of data at each training iteration and computes its own gradients. The mean of the gradients is computed and shared across all replicas using a distributed AllReduce implementation (NCCL by default), and all replicas perform the same Gradient Descent step. This strategy is the simplest to use since all servers and devices are treated in exactly the same way, and it performs fairly well. In general, you should use this strategy. Its main limitation is that it requires the model to fit in RAM on every replica.
    - The `ParameterServerStrategy` performs asynchronous data parallelism. The model is replicated across all devices on all workers, and the parameters are sharded across all parameter servers. Each worker has its own training loop, running asynchronously with the other workers; at each training iteration, each worker gets its own batch of data and fetches the latest version of the model parameters from the parameter servers, then it computes the gradients of the loss with regard to these parameters, and it sends them to the parameter servers. Lastly, the parameter servers perform a Gradient Descent step using these gradients. This strategy is generally slower than the previous strategy,

and a bit harder to deploy, since it requires managing parameter servers. However, it is useful to train huge models that don't fit in GPU RAM.

For the solutions to exercises 9, 10, and 11, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml2*.

APPENDIX B

Machine Learning Project Checklist

This checklist can guide you through your Machine Learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

Frame the Problem and Look at the Big Picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions/workarounds (if any)?
4. How should you frame this problem (supervised/unsupervised, online/offline, etc.)?
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?

7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

Get the Data

Note: automate as much as possible so you can easily get fresh data.

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

Explore the Data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
    - Name
    - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)

- % of missing values
- Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
- Usefulness for the task
- Type of distribution (Gaussian, uniform, logarithmic, etc.)

4. For supervised learning tasks, identify the target attribute(s).
5. Visualize the data.
6. Study the correlations between attributes.
7. Study how you would solve the problem manually.
8. Identify the promising transformations you may want to apply.
9. Identify extra data that would be useful (go back to "Get the Data" on page 756).
10. Document what you have learned.

Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
- Write functions for all data transformations you apply, for five reasons:
  - So you can easily prepare the data the next time you get a fresh dataset
  - So you can apply these transformations in future projects
  - To clean and prepare the test set
  - To clean and prepare new data instances once your solution is live
  - To make it easy to treat your preparation choices as hyperparameters

1. Data cleaning:
   - Fix or remove outliers (optional).
   - Fill in missing values (e.g., with zero, mean, median…) or drop their rows (or columns).
2. Feature selection (optional):
   - Drop the attributes that provide no useful information for the task.
3. Feature engineering, where appropriate:
   - Discretize continuous features.

- Decompose features (e.g., categorical, date/time, etc.).
- Add promising transformations of features (e.g., log(x), sqrt(x), x^2, etc.).
- Aggregate features into promising new features.

4. Feature scaling:
   - Standardize or normalize features.

Shortlist Promising Models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
- Once again, try to automate these steps as much as possible.

1. Train many quick-and-dirty models from different categories (e.g., linear, naive Bayes, SVM, Random Forest, neural net, etc.) using standard parameters.
2. Measure and compare their performance.
   - For each model, use N-fold cross-validation and compute the mean and standard deviation of the performance measure on the N folds.
3. Analyze the most significant variables for each algorithm.
4. Analyze the types of errors the models make.
   - What data would a human have used to avoid these errors?
5. Perform a quick round of feature selection and engineering.
6. Perform one or two more quick iterations of the five previous steps.
7. Shortlist the top three to five most promising models, preferring models that make different types of errors.

Fine-Tune the System

Notes:

- You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.

- As always, automate what you can.

1. Fine-tune the hyperparameters using cross-validation:
   - Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g., if you're not sure whether to replace missing values with zeros or with the median value, or to just drop the rows).
   - Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, as described by Jasper Snoek et al.).[1]
2. Try Ensemble methods. Combining your best models will often produce better performance than running them individually.
3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.

Don't tweak your model after measuring the generalization error: you would just start overfitting the test set.

Present Your Solution

1. Document what you have done.
2. Create a nice presentation.
   - Make sure you highlight the big picture first.
3. Explain why your solution achieves the business objective.
4. Don't forget to present interesting points you noticed along the way.
   - Describe what worked and what did not.
   - List your assumptions and your system's limitations.

1 Jasper Snoek et al., "Practical Bayesian Optimization of Machine Learning Algorithms," *Proceedings of the 25th International Conference on Neural Information Processing Systems* 2 (2012): 2951–2959.

5. Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements (e.g., "the median income is the number-one predictor of housing prices").

Launch!

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
   - Beware of slow degradation: models tend to "rot" as data evolves.
   - Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
   - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

APPENDIX C
SVM Dual Problem

To understand *duality*, you first need to understand the *Lagrange multipliers* method. The general idea is to transform a constrained optimization objective into an unconstrained one, by moving the constraints into the objective function. Let's look at a simple example. Suppose you want to find the values of x and y that minimize the function $f(x, y) = x^2 + 2y$, subject to an *equality constraint*: $3x + 2y + 1 = 0$. Using the Lagrange multipliers method, we start by defining a new function called the *Lagrangian* (or *Lagrange function*): $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$. Each constraint (in this case just one) is subtracted from the original objective, multiplied by a new variable called a Lagrange multiplier.

Joseph-Louis Lagrange showed that if $(\hat{x}, \hat{y})$ is a solution to the constrained optimization problem, then there must exist an $\hat{\alpha}$ such that $(\hat{x}, \hat{y}, \hat{\alpha})$ is a *stationary point* of the Lagrangian (a stationary point is a point where all partial derivatives are equal to zero). In other words, we can compute the partial derivatives of $g(x, y, \alpha)$ with regard to x, y, and α; we can find the points where these derivatives are all equal to zero; and the solutions to the constrained optimization problem (if they exist) must be among these stationary points.

In this example the partial derivatives are: $\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$

When all these partial derivatives are equal to 0, we find that $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$, from which we can easily find that $\hat{x} = \frac{3}{2}$, $\hat{y} = -\frac{11}{4}$, and $\hat{\alpha} = 1$. This is the only stationary point, and as it respects the constraint, it must be the solution to the constrained optimization problem.

However, this method applies only to equality constraints. Fortunately, under some regularity conditions (which are respected by the SVM objectives), this method can be generalized to *inequality constraints* as well (e.g., $3x + 2y + 1 \geq 0$). The *generalized Lagrangian* for the hard margin problem is given by Equation C-1, where the $\alpha^{(i)}$ variables are called the *Karush–Kuhn–Tucker* (KKT) multipliers, and they must be greater or equal to zero.

Equation C-1. Generalized Lagrangian for the hard margin problem

$$\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)}\left(t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) - 1\right)$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Just like with the Lagrange multipliers method, you can compute the partial derivatives and locate the stationary points. If there is a solution, it will necessarily be among the stationary points $\left(\widehat{\mathbf{w}}, \hat{b}, \hat{\boldsymbol{\alpha}}\right)$ that respect the *KKT conditions*:

- Respect the problem's constraints: $t^{(i)}\left(\widehat{\mathbf{w}}^\intercal\mathbf{x}^{(i)} + \hat{b}\right) \geq 1 \quad \text{for } i = 1, 2, \ldots, m$.
- Verify $\hat{\alpha}^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$.
- Either $\hat{\alpha}^{(i)} = 0$ or the i^{th} constraint must be an *active constraint*, meaning it must hold by equality: $t^{(i)}\left(\widehat{\mathbf{w}}^\intercal\mathbf{x}^{(i)} + \hat{b}\right) = 1$. This condition is called the *complementary slackness* condition. It implies that either $\hat{\alpha}^{(i)} = 0$ or the i^{th} instance lies on the boundary (it is a support vector).

Note that the KKT conditions are necessary conditions for a stationary point to be a solution of the constrained optimization problem. Under some conditions, they are also sufficient conditions. Luckily, the SVM optimization problem happens to meet these conditions, so any stationary point that meets the KKT conditions is guaranteed to be a solution to the constrained optimization problem.

We can compute the partial derivatives of the generalized Lagrangian with regard to $\mathbf{w}$ and b with Equation C-2.

Equation C-2. Partial derivatives of the generalized Lagrangian

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = \mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\frac{\partial}{\partial b}\mathcal{L}(\mathbf{w}, b, \boldsymbol{\alpha}) = -\sum_{i=1}^{m} \alpha^{(i)} t^{(i)}$$

When these partial derivatives are equal to zero, we have Equation C-3.

Equation C-3. Properties of the stationary points

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \widehat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^{m} \widehat{\alpha}^{(i)} t^{(i)} = 0$$

If we plug these results into the definition of the generalized Lagrangian, some terms disappear and we find Equation C-4.

Equation C-4. Dual form of the SVM problem

$$\mathcal{L}\left(\widehat{\mathbf{w}}, \hat{b}, \alpha\right) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\intercal} \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The goal is now to find the vector $\widehat{\alpha}$ that minimizes this function, with $\widehat{\alpha}^{(i)} \geq 0$ for all instances. This constrained optimization problem is the dual problem we were looking for.

Once you find the optimal $\widehat{\alpha}$, you can compute $\widehat{\mathbf{w}}$ using the first line of Equation C-3. To compute $\hat{b}$, you can use the fact that a support vector must verify $t^{(i)}(\widehat{\mathbf{w}}^{\intercal} \mathbf{x}^{(i)} + \hat{b}) = 1$, so if the k^{th} instance is a support vector (i.e., $\widehat{\alpha}^{(k)} > 0$), you can use it to compute $\hat{b} = t^{(k)} - \widehat{\mathbf{w}}^{\intercal} \mathbf{x}^{(k)}$. However, it is often preferred to compute the average over all support vectors to get a more stable and precise value, as in Equation C-5.

Equation C-5. Bias term estimation using the dual form

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^{m} \left[t^{(i)} - \widehat{\mathbf{w}}^{\intercal} \mathbf{x}^{(i)} \right]$$

APPENDIX D
Autodiff

This appendix explains how TensorFlow's autodifferentiation (autodiff) feature works, and how it compares to other solutions.

Suppose you define a function $f(x, y) = x^2y + y + 2$, and you need its partial derivatives $\partial f/\partial x$ and $\partial f/\partial y$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are manual differentiation, finite difference approximation, forward-mode autodiff, and reverse-mode autodiff. TensorFlow implements reverse-mode autodiff, but to understand it, it's useful to look at the other options first. So let's go through each of them, starting with manual differentiation.

Manual Differentiation

The first approach to compute derivatives is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the appropriate equation. For the function $f(x, y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda - 1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions' derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive Equation D-1.

Equation D-1. Partial derivatives of f(x, y)

$$\frac{\partial f}{\partial x} = \frac{\partial\left(x^2 y\right)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y\frac{\partial\left(x^2\right)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial\left(x^2 y\right)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. Fortunately, there are other options. Let's look at finite difference approximation now.

Finite Difference Approximation

Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point. More precisely, the derivative is defined as the limit of the slope of a straight line going through this point x_0 and another point x on the function, as x gets infinitely close to x_0 (see Equation D-2).

Equation D-2. Definition of the derivative of a function h(x) at point x_0

$$h'(x_0) = \lim_{x \to x_0} \frac{h(x) - h(x_0)}{x - x_0}$$

$$= \lim_{\varepsilon \to 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon}$$

So, if we wanted to calculate the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$, we could compute $f(3 + \varepsilon, 4) - f(3, 4)$ and divide the result by ε, using a very small value for ε. This type of numerical approximation of the derivative is called a *finite difference approximation*, and this specific equation is called *Newton's difference quotient*. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complicated functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call `f()` at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When you are dealing with large neural networks, this makes finite difference approximation way too inefficient.

However, this method is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

So far, we have considered two ways to compute gradients: using manual differentiation and using finite difference approximation. Unfortunately, both were fatally flawed to train a large-scale neural network. So let's turn to autodiff, starting with forward mode.

Forward-Mode Autodiff

Figure D-1 shows how forward-mode autodiff works on an even simpler function, $g(x, y) = 5 + xy$. The graph for that function is represented on the left. After forward-mode autodiff, we get the graph on the right, which represents the partial derivative $\partial g/\partial x = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regard to y).

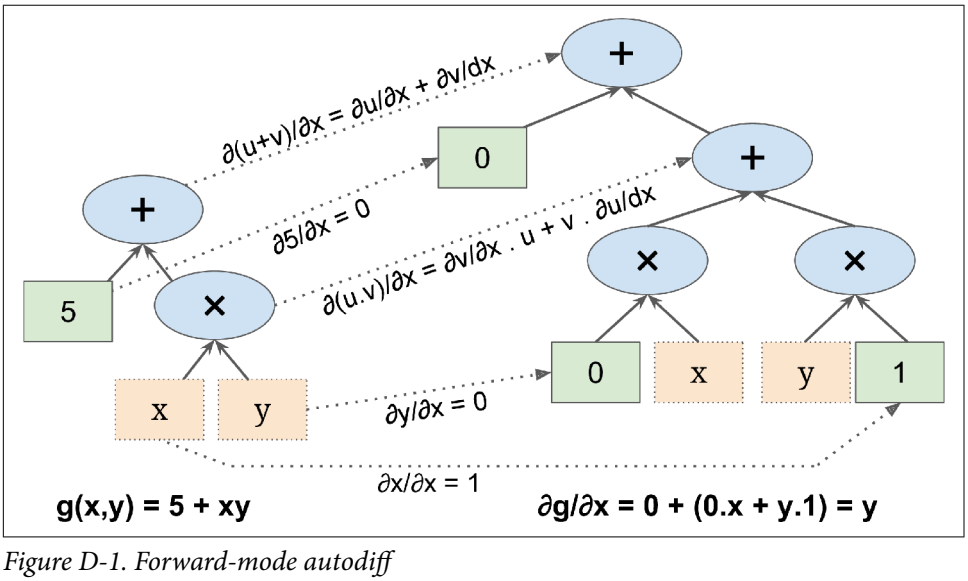


Figure D-1. Forward-mode autodiff

The algorithm will go through the computation graph from the inputs to the outputs (hence the name "forward mode"). It starts by getting the partial derivatives of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable x returns the constant 1 since $\partial x/\partial x = 1$, and the variable y returns the constant 0 since $\partial y/\partial x = 0$ (if we were looking for the partial derivative with regard to y, it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g. Calculus tells us that the derivative of the product of two functions u and v is $\partial(u \times v)/\partial x = \partial v/\partial x \times u + v \times \partial u/\partial x$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g. As mentioned, the derivative of a sum of functions is the sum of these functions' derivatives. So we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\partial g/\partial x = 0 + (0 \times x + y \times 1)$.

However, this equation can be simplified (a lot). A few pruning steps can be applied to the computation graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\partial g/\partial x = y$. In this case simplification is fairly easy, but for a more complex function forward-mode autodiff can produce a huge graph that may be tough to simplify and lead to suboptimal performance.

Note that we started with a computation graph, and forward-mode autodiff produced another computation graph. This is called *symbolic differentiation*, and it has two nice features: first, once the computation graph of the derivative has been produced, we can use it as many times as we want to compute the derivatives of the given function for any value of x and y; second, we can run forward-mode autodiff again on the resulting graph to get second-order derivatives if we ever need to (i.e., derivatives of derivatives). We could even compute third-order derivatives, and so on.

But it is also possible to run forward-mode autodiff without constructing a graph (i.e., numerically, not symbolically), just by computing intermediate results on the fly. One way to do this is to use *dual numbers*, which are weird but fascinating numbers of the form $a + b\varepsilon$, where a and b are real numbers and ε is an infinitesimal number such that $\varepsilon^2 = 0$ (but $\varepsilon \neq 0$). You can think of the dual number $42 + 24\varepsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, $42 + 24\varepsilon$ is represented by the pair (42.0, 24.0).

Dual numbers can be added, multiplied, and so on, as shown in Equation D-3.

Equation D-3. A few operations with dual numbers

$$\lambda(a + b\varepsilon) = \lambda a + \lambda b\varepsilon$$
$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$
$$(a + b\varepsilon) \times (c + d\varepsilon) = ac + (ad + bc)\varepsilon + (bd)\varepsilon^2 = ac + (ad + bc)\varepsilon$$

Most importantly, it can be shown that $h(a + b\varepsilon) = h(a) + b \times h'(a)\varepsilon$, so computing $h(a + \varepsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. Figure D-2 shows that the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$ (which we will write $\partial f/\partial x$ (3, 4)) can be computed using dual numbers. All we need to do is compute $f(3 + \varepsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\partial f/\partial x$ (3, 4).

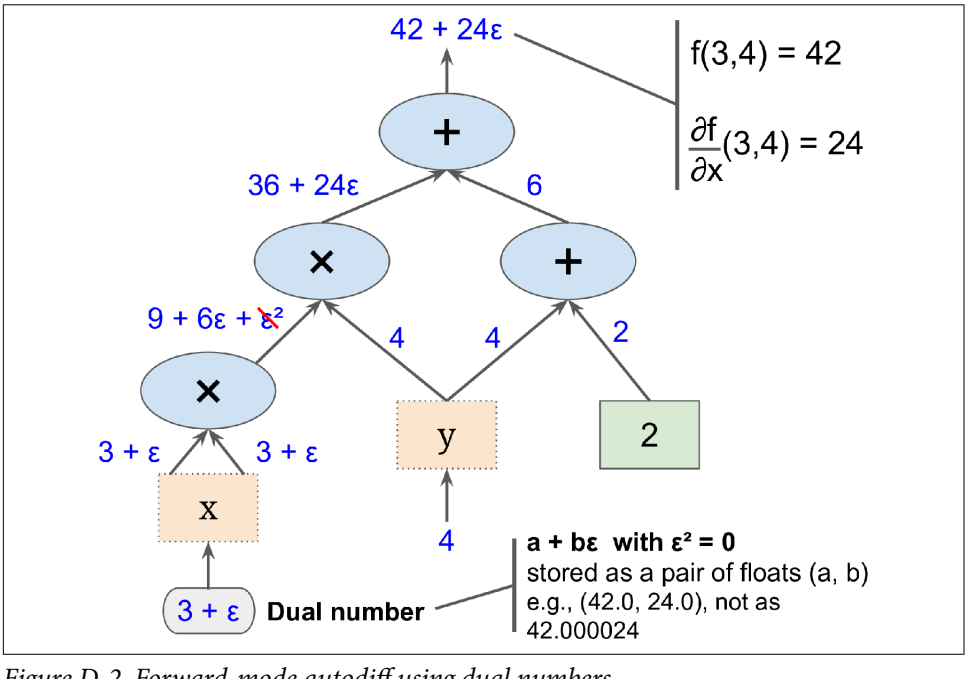


Figure D-2. Forward-mode autodiff using dual numbers

To compute $\partial f/\partial x$ (3, 4) we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \varepsilon$.

So forward-mode autodiff is much more accurate than finite difference approximation, but it suffers from the same major flaw, at least when there are many inputs and few outputs (as is the case when dealing with neural networks): if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial

derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph. Let's see how.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs), to compute all the partial derivatives. The name "reverse mode" comes from this second pass through the graph, where gradients flow in the reverse direction. Figure D-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7: $f(3, 4) = n_7 = 42$.

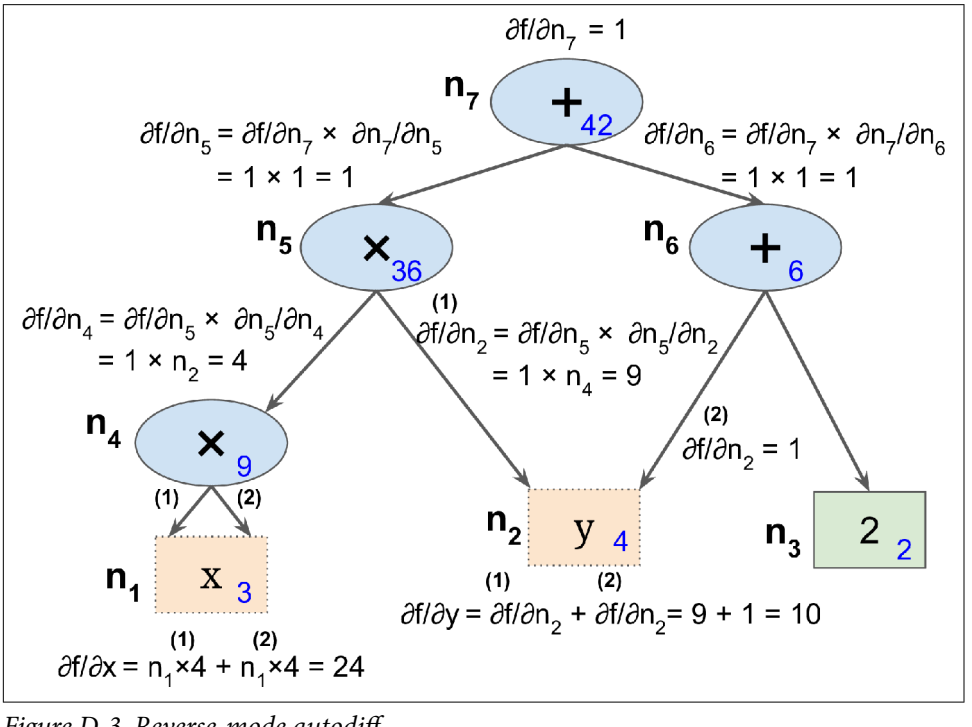


Figure D-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of $f(x, y)$ with regard to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown in Equation D-4.

Equation D-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$ so $\partial f/\partial n_7 = 1$.

Let's continue down the graph to n_5: how much does f vary when n_5 varies? The answer is $\partial f/\partial n_5 = \partial f/\partial n_7 \times \partial n_7/\partial n_5$. We already know that $\partial f/\partial n_7 = 1$, so all we need is $\partial n_7/\partial n_5$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\partial n_7/\partial n_5 = 1$, so $\partial f/\partial n_5 = 1 \times 1 = 1$.

Now we can proceed to node n_4: how much does f vary when n_4 varies? The answer is $\partial f/\partial n_4 = \partial f/\partial n_5 \times \partial n_5/\partial n_4$. Since $n_5 = n_4 \times n_2$, we find that $\partial n_5/\partial n_4 = n_2$, so $\partial f/\partial n_4 = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x, y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\partial f/\partial x = 24$ and $\partial f/\partial y = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regard to all the inputs. When training neural networks, we generally want to minimize the loss, so there is a single output (the loss), and hence only two passes through the graph are needed to compute the gradients. Reverse-mode autodiff can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that are differentiable.

In Figure D-3, the numerical results are computed on the fly, at each node. However, that's not exactly what TensorFlow does: instead, it creates a new computation graph. In other words, it implements *symbolic* reverse-mode autodiff. This way, the computation graph to compute the gradients of the loss with regard to all the parameters in the neural network only needs to be generated once, and then it can be executed over and over again, whenever the optimizer needs to compute the gradients. Moreover, this makes it possible to compute higher-order derivatives if needed.

If you ever want to implement a new type of low-level TensorFlow operation in C++, and you want to make it compatible with autodiff, then you will need to provide a function that returns the partial derivatives of the function's outputs with regard to its inputs. For example, suppose you implement a function that computes the square of its input: $f(x) = x^2$. In that case you would need to provide the corresponding derivative function: $f'(x) = 2x$.

APPENDIX E
Other Popular ANN Architectures

In this appendix I will give a quick overview of a few historically important neural network architectures that are much less used today than deep Multilayer Perceptrons (Chapter 10), convolutional neural networks (Chapter 14), recurrent neural networks (Chapter 15), or autoencoders (Chapter 17). They are often mentioned in the literature, and some are still used in a range of applications, so it is worth knowing about them. Additionally, we will discuss *deep belief nets*, which were the state of the art in Deep Learning until the early 2010s. They are still the subject of very active research, so they may well come back with a vengeance in the future.

Hopfield Networks

Hopfield networks were first introduced by W. A. Little in 1974, then popularized by J. Hopfield in 1982. They are *associative memory* networks: you first teach them some patterns, and then when they see a new pattern they (hopefully) output the closest learned pattern. This made them useful for character recognition, in particular, before they were outperformed by other approaches: you first train the network by showing it examples of character images (each binary pixel maps to one neuron), and then when you show it a new character image, after a few iterations it outputs the closest learned character.

Hopfield networks are fully connected graphs (see Figure E-1); that is, every neuron is connected to every other neuron. Note that in the diagram the images are 6×6 pixels, so the neural network on the left should contain 36 neurons (and 630 connections), but for visual clarity a much smaller network is represented.

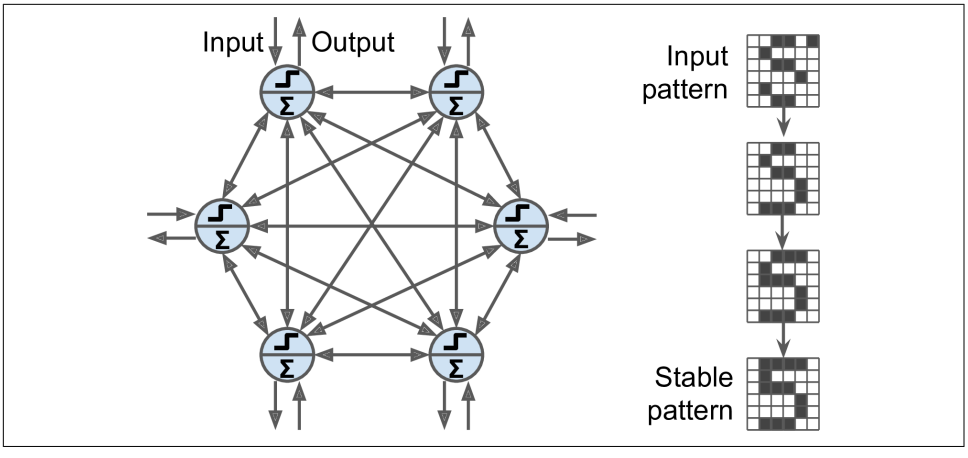


Figure E-1. Hopfield network

The training algorithm works by using Hebb's rule (see "The Perceptron" on page 284): for each training image, the weight between two neurons is increased if the corresponding pixels are both on or both off, but decreased if one pixel is on and the other is off.

To show a new image to the network, you just activate the neurons that correspond to active pixels. The network then computes the output of every neuron, and this gives you a new image. You can then take this new image and repeat the whole process. After a while, the network reaches a stable state. Generally, this corresponds to the training image that most resembles the input image.

A so-called *energy function* is associated with Hopfield nets. At each iteration, the energy decreases, so the network is guaranteed to eventually stabilize to a low-energy state. The training algorithm tweaks the weights in a way that decreases the energy level of the training patterns, so the network is likely to stabilize in one of these low-energy configurations. Unfortunately, some patterns that were not in the training set also end up with low energy, so the network sometimes stabilizes in a configuration that was not learned. These are called *spurious patterns*.

Another major flaw with Hopfield nets is that they don't scale very well—their memory capacity is roughly equal to 14% of the number of neurons. For example, to classify 28 × 28–pixel images, you would need a Hopfield net with 784 fully connected neurons and 306,936 weights. Such a network would only be able to learn about 110 different characters (14% of 784). That's a lot of parameters for such a small memory.

Boltzmann Machines

Boltzmann machines were invented in 1985 by Geoffrey Hinton and Terrence Sejnowski. Just like Hopfield nets, they are fully connected ANNs, but they are based on *stochastic neurons*: instead of using a deterministic step function to decide what value to output, these neurons output 1 with some probability, and 0 otherwise. The probability function that these ANNs use is based on the Boltzmann distribution (used in statistical mechanics), hence their name. Equation E-1 gives the probability that a particular neuron will output 1.

Equation E-1. Probability that the i^{th} neuron will output 1

$$p\left(s_i^{(\text{next step})} = 1\right) = \sigma\left(\frac{\sum_{j=1}^{N} w_{i,j} s_j + b_i}{T}\right)$$

- s_j is the j^{th} neuron's state (0 or 1).
- $w_{i,j}$ is the connection weight between the i^{th} and j^{th} neurons. Note that $w_{i,i} = 0$.
- b_i is the i^{th} neuron's bias term. We can implement this term by adding a bias neuron to the network.
- N is the number of neurons in the network.
- T is a number called the network's *temperature*; the higher the temperature, the more random the output is (i.e., the more the probability approaches 50%).
- σ is the logistic function.

Neurons in Boltzmann machines are separated into two groups: *visible units* and *hidden units* (see Figure E-2). All neurons work in the same stochastic way, but the visible units are the ones that receive the inputs and from which outputs are read.

Because of its stochastic nature, a Boltzmann machine will never stabilize into a fixed configuration; instead, it will keep switching between many configurations. If it is left running for a sufficiently long time, the probability of observing a particular configuration will only be a function of the connection weights and bias terms, not of the original configuration (similarly, after you shuffle a deck of cards for long enough, the configuration of the deck does not depend on the initial state). When the network reaches this state where the original configuration is "forgotten," it is said to be in *thermal equilibrium* (although its configuration keeps changing all the time). By setting the network parameters appropriately, letting the network reach thermal equilibrium, and then observing its state, we can simulate a wide range of probability distributions. This is called a *generative model*.

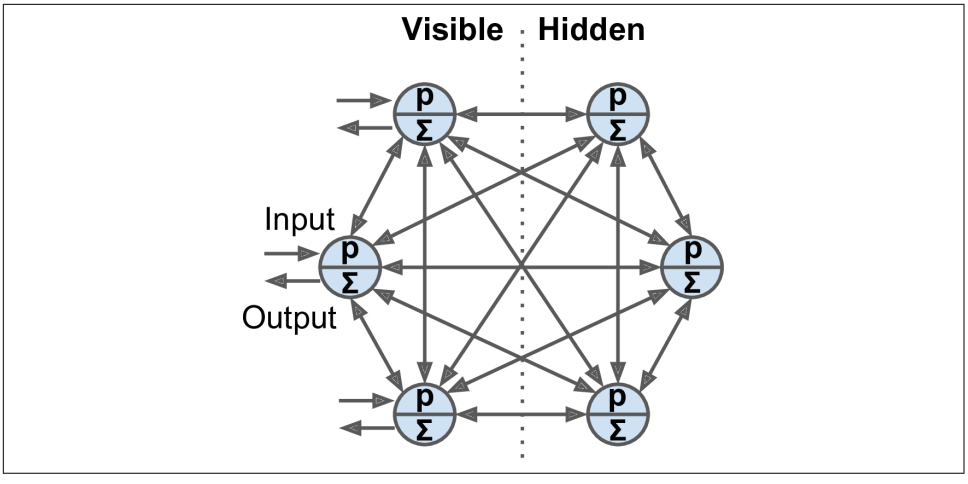


Figure E-2. Boltzmann machine

Training a Boltzmann machine means finding the parameters that will make the network approximate the training set's probability distribution. For example, if there are three visible neurons and the training set contains 75% (0, 1, 1) triplets, 10% (0, 0, 1) triplets, and 15% (1, 1, 1) triplets, then after training a Boltzmann machine, you could use it to generate random binary triplets with about the same probability distribution. For example, about 75% of the time it would output the (0, 1, 1) triplet.

Such a generative model can be used in a variety of ways. For example, if it is trained on images, and you provide an incomplete or noisy image to the network, it will automatically "repair" the image in a reasonable way. You can also use a generative model for classification. Just add a few visible neurons to encode the training image's class (e.g., add 10 visible neurons and turn on only the fifth neuron when the training image represents a 5). Then, when given a new image, the network will automatically turn on the appropriate visible neurons, indicating the image's class (e.g., it will turn on the fifth visible neuron if the image represents a 5).

Unfortunately, there is no efficient technique to train Boltzmann machines. However, fairly efficient algorithms have been developed to train *restricted Boltzmann machines* (RBMs).

Restricted Boltzmann Machines

An RBM is simply a Boltzmann machine in which there are no connections between visible units or between hidden units, only between visible and hidden units. For example, Figure E-3 represents an RBM with three visible units and four hidden units.

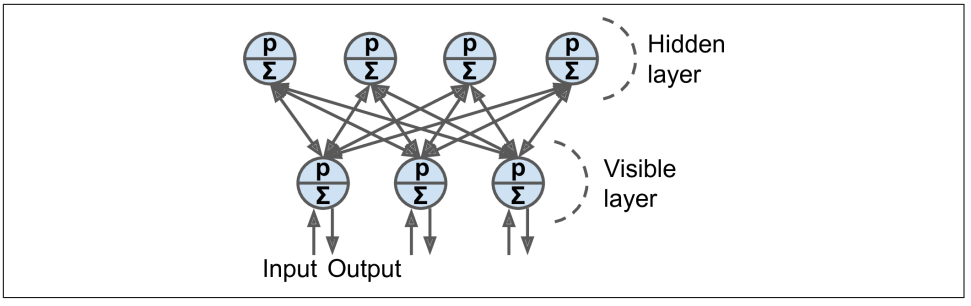


Figure E-3. Restricted Boltzmann machine

A very efficient training algorithm called *Contrastive Divergence* was introduced in 2005 by Miguel Á. Carreira-Perpiñán and Geoffrey Hinton.[1] Here is how it works: for each training instance x, the algorithm starts by feeding it to the network by setting the state of the visible units to $x_1, x_2, \cdots, x_n$. Then you compute the state of the hidden units by applying the stochastic equation described before (Equation E-1). This gives you a hidden vector h (where h_i is equal to the state of the i^{th} unit). Next you compute the state of the visible units, by applying the same stochastic equation. This gives you a vector x′. Then once again you compute the state of the hidden units, which gives you a vector h′. Now you can update each connection weight by applying the rule in Equation E-2, where η is the learning rate.

Equation E-2. Contrastive divergence weight update

$$w_{i,j} \leftarrow w_{i,j} + \eta\left(\mathrm{x}\mathrm{h}^{\intercal} - \mathrm{x}'\mathrm{h}'^{\intercal}\right)$$

The great benefit of this algorithm is that it does not require waiting for the network to reach thermal equilibrium: it just goes forward, backward, and forward again, and that's it. This makes it incomparably more efficient than previous algorithms, and it was a key ingredient to the first success of Deep Learning based on multiple stacked RBMs.

Deep Belief Nets

Several layers of RBMs can be stacked; the hidden units of the first-level RBM serve as the visible units for the second-layer RBM, and so on. Such an RBM stack is called a *deep belief net* (DBN).

1 Miguel Á. Carreira-Perpiñán and Geoffrey E. Hinton, "On Contrastive Divergence Learning," *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics* (2005): 59–66.

Yee-Whye Teh, one of Geoffrey Hinton's students, observed that it was possible to train DBNs one layer at a time using Contrastive Divergence, starting with the lower layers and then gradually moving up to the top layers. This led to the groundbreaking article that kickstarted the Deep Learning tsunami in 2006.[2]

Just like RBMs, DBNs learn to reproduce the probability distribution of their inputs, without any supervision. However, they are much better at it, for the same reason that deep neural networks are more powerful than shallow ones: real-world data is often organized in hierarchical patterns, and DBNs take advantage of that. Their lower layers learn low-level features in the input data, while higher layers learn high-level features.

Just like RBMs, DBNs are fundamentally unsupervised, but you can also train them in a supervised manner by adding some visible units to represent the labels. Moreover, one great feature of DBNs is that they can be trained in a semisupervised fashion. Figure E-4 represents such a DBN configured for semisupervised learning.

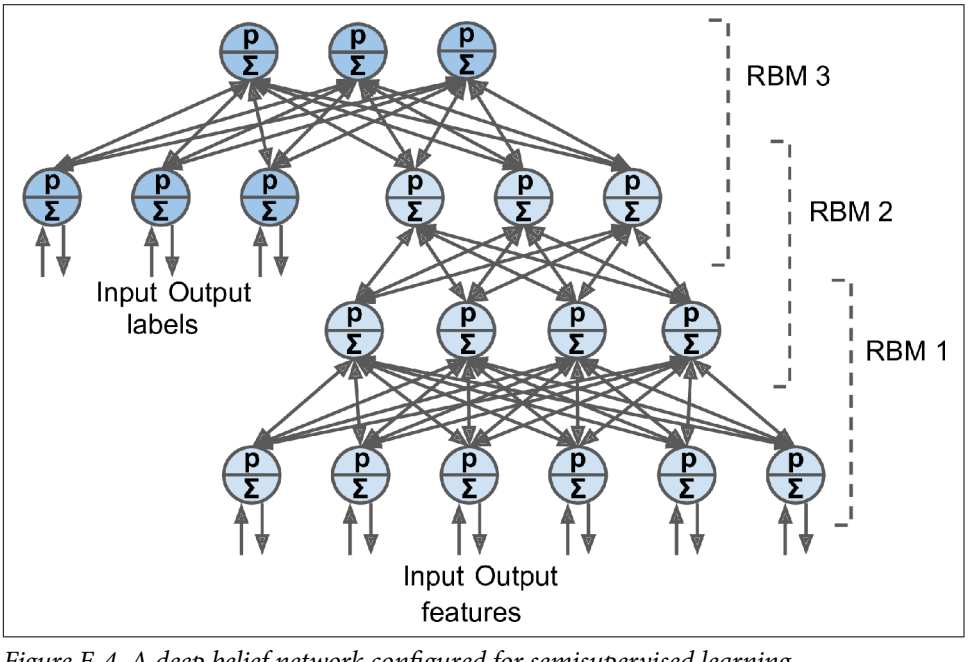


Figure E-4. A deep belief network configured for semisupervised learning

First, RBM 1 is trained without supervision. It learns low-level features in the training data. Then RBM 2 is trained with RBM 1's hidden units as inputs, again without

2 Geoffrey E. Hinton et al., "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation* 18 (2006): 1527–1554.

supervision: it learns higher-level features (note that RBM 2's hidden units include only the three rightmost units, not the label units). Several more RBMs could be stacked this way, but you get the idea. So far, training was 100% unsupervised. Lastly, RBM 3 is trained using RBM 2's hidden units as inputs, as well as extra visible units used to represent the target labels (e.g., a one-hot vector representing the instance class). It learns to associate high-level features with training labels. This is the supervised step.

At the end of training, if you feed RBM 1 a new instance, the signal will propagate up to RBM 2, then up to the top of RBM 3, and then back down to the label units; hopefully, the appropriate label will light up. This is how a DBN can be used for classification.

One great benefit of this semisupervised approach is that you don't need much labeled training data. If the unsupervised RBMs do a good enough job, then only a small amount of labeled training instances per class will be necessary. Similarly, a baby learns to recognize objects without supervision, so when you point to a chair and say "chair," the baby can associate the word "chair" with the class of objects it has already learned to recognize on its own. You don't need to point to every single chair and say "chair"; only a few examples will suffice (just enough so the baby can be sure that you are indeed referring to the chair, not to its color or one of the chair's parts).

Quite amazingly, DBNs can also work in reverse. If you activate one of the label units, the signal will propagate up to the hidden units of RBM 3, then down to RBM 2, and then RBM 1, and a new instance will be output by the visible units of RBM 1. This new instance will usually look like a regular instance of the class whose label unit you activated. This generative capability of DBNs is quite powerful. For example, it has been used to automatically generate captions for images, and vice versa: first a DBN is trained (without supervision) to learn features in images, and another DBN is trained (again without supervision) to learn features in sets of captions (e.g., "car" often comes with "automobile"). Then an RBM is stacked on top of both DBNs and trained with a set of images along with their captions; it learns to associate high-level features in images with high-level features in captions. Next, if you feed the image DBN an image of a car, the signal will propagate through the network, up to the top-level RBM, and back down to the bottom of the caption DBN, producing a caption. Due to the stochastic nature of RBMs and DBNs, the caption will keep changing randomly, but it will generally be appropriate for the image. If you generate a few hundred captions, the most frequently generated ones will likely be a good description of the image.[3]

3 See this video by Geoffrey Hinton for more details and a demo: *https://homl.info/137*.

Self-Organizing Maps

Self-organizing maps (SOMs) are quite different from all the other types of neural networks we have discussed so far. They are used to produce a low-dimensional representation of a high-dimensional dataset, generally for visualization, clustering, or classification. The neurons are spread across a map (typically 2D for visualization, but it can be any number of dimensions you want), as shown in Figure E-5, and each neuron has a weighted connection to every input (note that the diagram shows just two inputs, but there are typically a very large number, since the whole point of SOMs is to reduce dimensionality).

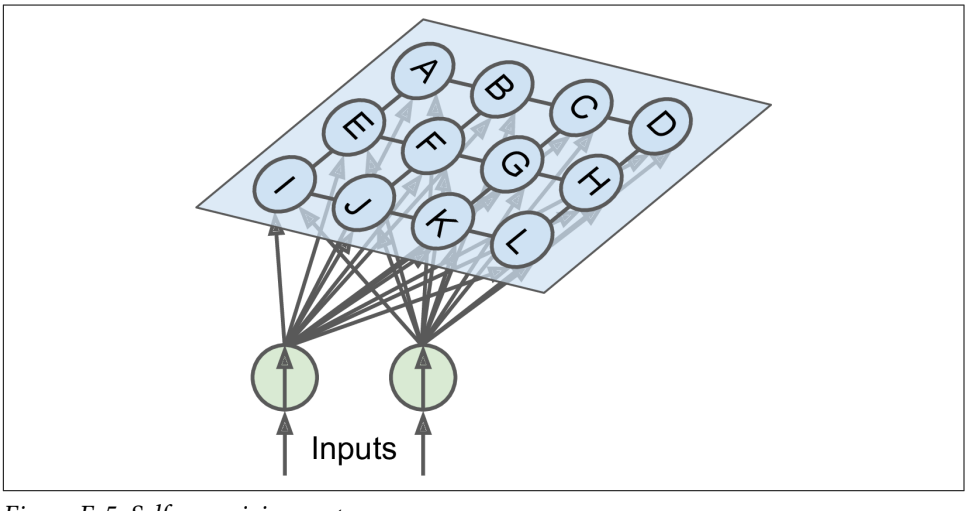


Figure E-5. Self-organizing map

Once the network is trained, you can feed it a new instance and this will activate only one neuron (i.e., one point on the map): the neuron whose weight vector is closest to the input vector. In general, instances that are nearby in the original input space will activate neurons that are nearby on the map. This makes SOMs useful not only for visualization (in particular, you can easily identify clusters on the map), but also for applications like speech recognition. For example, if each instance represents an audio recording of a person pronouncing a vowel, then different pronunciations of the vowel "a" will activate neurons in the same area of the map, while instances of the vowel "e" will activate neurons in another area, and intermediate sounds will generally activate intermediate neurons on the map.

One important difference from the other dimensionality reduction techniques discussed in Chapter 8 is that all instances get mapped to a discrete number of points in the low-dimensional space (one point per neuron). When there are very few neurons, this technique is better described as clustering rather than dimensionality reduction.

The training algorithm is unsupervised. It works by having all the neurons compete against each other. First, all the weights are initialized randomly. Then a training instance is picked randomly and fed to the network. All neurons compute the distance between their weight vector and the input vector (this is very different from the artificial neurons we have seen so far). The neuron that measures the smallest distance wins and tweaks its weight vector to be slightly closer to the input vector, making it more likely to win future competitions for other inputs similar to this one. It also recruits its neighboring neurons, and they too update their weight vectors to be slightly closer to the input vector (but they don't update their weights as much as the winning neuron). Then the algorithm picks another training instance and repeats the process, again and again. This algorithm tends to make nearby neurons gradually specialize in similar inputs.[4]

4 You can imagine a class of young children with roughly similar skills. One child happens to be slightly better at basketball. This motivates them to practice more, especially with their friends. After a while, this group of friends gets so good at basketball that other kids cannot compete. But that's okay, because the other kids specialize in other areas. After a while, the class is full of little specialized groups.

APPENDIX F
Special Data Structures

In this appendix we will take a very quick look at the data structures supported by TensorFlow, beyond regular float or integer tensors. This includes strings, ragged tensors, sparse tensors, tensor arrays, sets, and queues.

Strings

Tensors can hold byte strings, which is useful in particular for natural language processing (see Chapter 16):

```
>>> tf.constant(b"hello world")
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'hello world'>
```

If you try to build a tensor with a Unicode string, TensorFlow automatically encodes it to UTF-8:

```
>>> tf.constant("café")
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

It is also possible to create tensors representing Unicode strings. Just create an array of 32-bit integers, each representing a single Unicode code point:[1]

```
>>> tf.constant([ord(c) for c in "café"])
<tf.Tensor: id=211, shape=(4,), dtype=int32,
            numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

1 If you are not familiar with Unicode code points, please check out *https://homl.info/unicode*.

In tensors of type tf.string, the string length is not part of the tensor's shape. In other words, strings are considered as atomic values. However, in a Unicode string tensor (i.e., an int32 tensor), the length of the string *is* part of the tensor's shape.

The `tf.strings` package contains several functions to manipulate string tensors, such as `length()` to count the number of bytes in a byte string (or the number of code points if you set `unit="UTF8_CHAR"`), `unicode_encode()` to convert a Unicode string tensor (i.e., int32 tensor) to a byte string tensor, and `unicode_decode()` to do the reverse:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
            numpy=array([ 99,  97, 102, 233], dtype=int32)>
```

You can also manipulate tensors containing multiple strings:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "    "])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
            numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[   67    97   102   233    67   111   102   102   101   101    99    97
   102   102   232 21654 21857], shape=(17,), dtype=int32),
   row_splits=tf.Tensor([ 0  4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
                  [99, 97, 102, 102, 232], [21654, 21857]]>
```

Notice that the decoded strings are stored in a `RaggedTensor`. What is that?

Ragged Tensors

A ragged tensor is a special kind of tensor that represents a list of arrays of different sizes. More generally, it is a tensor with one or more *ragged dimensions*, meaning dimensions whose slices may have different lengths. In the ragged tensor `r`, the second dimension is a ragged dimension. In all ragged tensors, the first dimension is always a regular dimension (also called a *uniform dimension*).

All the elements of the ragged tensor `r` are regular tensors. For example, let's look at the second element of the ragged tensor:

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

The `tf.ragged` package contains several functions to create and manipulate ragged tensors. Let's create a second ragged tensor using `tf.ragged.constant()` and concatenate it with the first ragged tensor, along axis 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101], [99, 97,
102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

The result is not too surprising: the tensors in `r2` were appended after the tensors in `r` along axis 0. But what if we concatenate `r` and another ragged tensor along axis 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102, 102, 101, 101,
71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

This time, notice that the i^{th} tensor in `r` and the i^{th} tensor in `r3` were concatenated. Now that's more unusual, since all of these tensors can have different lengths.

If you call the `to_tensor()` method, it gets converted to a regular tensor, padding shorter tensors with zeros to get tensors of equal lengths (you can change the default value by setting the `default_value` argument):

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[   67,    97,   102,   233,     0,     0],
       [   67,   111,   102,   102,   101,   101],
       [   99,    97,   102,   102,   232,     0],
       [21654, 21857,     0,     0,     0,     0]], dtype=int32)>
```

Many TF operations support ragged tensors. For the full list, see the documentation of the `tf.RaggedTensor` class.

Sparse Tensors

TensorFlow can also efficiently represent *sparse tensors* (i.e., tensors containing mostly zeros). Just create a `tf.SparseTensor`, specifying the indices and values of the nonzero elements and the tensor's shape. The indices must be listed in "reading order" (from left to right, and top to bottom). If you are unsure, just use `tf.sparse.reorder()`. You can convert a sparse tensor to a dense tensor (i.e., a regular tensor) using `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                        values=[1., 2., 3.],
                        dense_shape=[3, 4])
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Note that sparse tensors do not support as many operations as dense tensors. For example, you can multiply a sparse tensor by any scalar value, and you get a new sparse tensor, but you cannot add a scalar value to a sparse tensor, as this would not return a sparse tensor:

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +: 'SparseTensor' and 'float'
```

Tensor Arrays

A `tf.TensorArray` represents a list of tensors. This can be handy in dynamic models containing loops, to accumulate results and later compute some statistics. You can read or write tensors at any location in the array:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1) # => returns (and pops!) tf.constant([3., 10.])
```

Notice that reading an item pops it from the array, replacing it with a tensor of the same shape, full of zeros.

When you write to the array, you must assign the output back to the array, as shown in this code example. If you don't, although your code will work fine in eager mode, it will break in graph mode (these modes were presented in Chapter 12).

When creating a `TensorArray`, you must provide its `size`, except in graph mode. Alternatively, you can leave the `size` unset and instead set `dynamic_size=True`, but this will hinder performance, so if you know the `size` in advance, you should set it. You must also specify the `dtype`, and all elements must have the same shape as the first one written to the array.

You can stack all the items into a regular tensor by calling the `stack()` method:

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

Sets

TensorFlow supports sets of integers or strings (but not floats). It represents them using regular tensors. For example, the set `{1, 5, 9}` is just represented as the tensor `[[1, 5, 9]]`. Note that the tensor must have at least two dimensions, and the sets must be in the last dimension. For example, `[[1, 5, 9], [2, 5, 11]]` is a tensor holding two independent sets: `{1, 5, 9}` and `{2, 5, 11}`. If some sets are shorter than others, you must pad them with a padding value (0 by default, but you can use any other value you prefer).

The `tf.sets` package contains several functions to manipulate sets. For example, let's create two sets and compute their union (the result is a sparse tensor, so we call `to_dense()` to display it):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

You can also compute the union of multiple pairs of sets simultaneously:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13,  0,  0]], dtype=int32)>
```

If you prefer to use a different padding value, you must set `default_value` when calling `to_dense()`:

```
>>> tf.sparse.to_dense(u, default_value=-1)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13, -1, -1]], dtype=int32)>
```

The default `default_value` is 0, so when dealing with string sets, you must set the `default_value` (e.g., to an empty string).

Other functions available in `tf.sets` include `difference()`, `intersection()`, and `size()`, which are self-explanatory. If you want to check whether or not a set contains some given values, you can compute the intersection of that set and the values. If you want to add some values to a set, you can compute the union of the set and the values.

Queues

A queue is a data structure to which you can push data records, and later pull them out. TensorFlow implements several types of queues in the `tf.queue` package. They used to be very important when implementing efficient data loading and preprocessing pipelines, but the tf.data API has essentially rendered them useless (except perhaps in some rare cases) because it is much simpler to use and provides all the tools you need to build efficient pipelines. For the sake of completeness, though, let's take a quick look at them.

The simplest kind of queue is the first-in, first-out (FIFO) queue. To build it, you need to specify the maximum number of records it can contain. Moreover, each record is a tuple of tensors, so you must specify the type of each tensor, and optionally their shapes. For example, the following code example creates a FIFO queue with maximum three records, each containing a tuple with a 32-bit integer and a string. Then it pushes two records to it, looks at the size (which is 2 at this point), and pulls a record out:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

It is also possible to enqueue and dequeue multiple records at once (the latter requires specifying the shapes when creating the queue):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'rainy']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'rainy'], dtype=object)>]
```

Other queue types include:

`PaddingFIFOQueue`
: Same as `FIFOQueue`, but its `dequeue_many()` method supports dequeueing multiple records of different shapes. It automatically pads the shortest records to ensure all the records in the batch have the same shape.

`PriorityQueue`
: A queue that dequeues records in a prioritized order. The priority must be a 64-bit integer included as the first element of each record. Surprisingly, records with a lower priority will be dequeued first. Records with the same priority will be dequeued in FIFO order.

`RandomShuffleQueue`
: A queue whose records are dequeued in random order. This was useful to implement a shuffle buffer before tf.data existed.

If a queue is already full and you try to enqueue another record, the `enqueue*()` method will freeze until a record is dequeued by another thread. Similarly, if a queue is empty and you try to dequeue a record, the `dequeue*()` method will freeze until records are pushed to the queue by another thread.

APPENDIX G

TensorFlow Graphs

In this appendix, we will explore the graphs generated by TF Functions (see Chapter 12).

TF Functions and Concrete Functions

TF Functions are polymorphic, meaning they support inputs of different types (and shapes). For example, consider the following `tf_cube()` function:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Every time you call a TF Function with a new combination of input types or shapes, it generates a new *concrete function*, with its own graph specialized for this particular combination. Such a combination of argument types and shapes is called an *input signature*. If you call the TF Function with an input signature it has already seen before, it will reuse the concrete function it generated earlier. For example, if you call `tf_cube(tf.constant(3.0))`, the TF Function will reuse the same concrete function it used for `tf_cube(tf.constant(2.0))` (for float32 scalar tensors). But it will generate a new concrete function if you call `tf_cube(tf.constant([2.0]))` or `tf_cube(tf.constant([3.0]))` (for float32 tensors of shape [1]), and yet another for `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (for float32 tensors of shape [2, 2]). You can get the concrete function for a particular combination of inputs by calling the TF Function's `get_concrete_function()` method. It can then be called like a regular function, but it will only support one input signature (in this example, float32 scalar tensors):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

Figure G-1 shows the `tf_cube()` TF Function, after we called `tf_cube(2)` and `tf_cube(tf.constant(2.0))`: two concrete functions were generated, one for each signature, each with its own optimized *function graph* (`FuncGraph`), and its own *function definition* (`FunctionDef`). A function definition points to the parts of the graph that correspond to the function's inputs and outputs. In each `FuncGraph`, the nodes (ovals) represent operations (e.g., power, constants, or placeholders for arguments like `x`), while the edges (the solid arrows between the operations) represent the tensors that will flow through the graph. The concrete function on the left is specialized for `x = 2`, so TensorFlow managed to simplify it to just output 8 all the time (note that the function definition does not even have an input). The concrete function on the right is specialized for float32 scalar tensors, and it could not be simplified. If we call `tf_cube(tf.constant(5.0))`, the second concrete function will be called, the placeholder operation for `x` will output 5.0, then the power operation will compute `5.0 ** 3`, so the output will be 125.0.

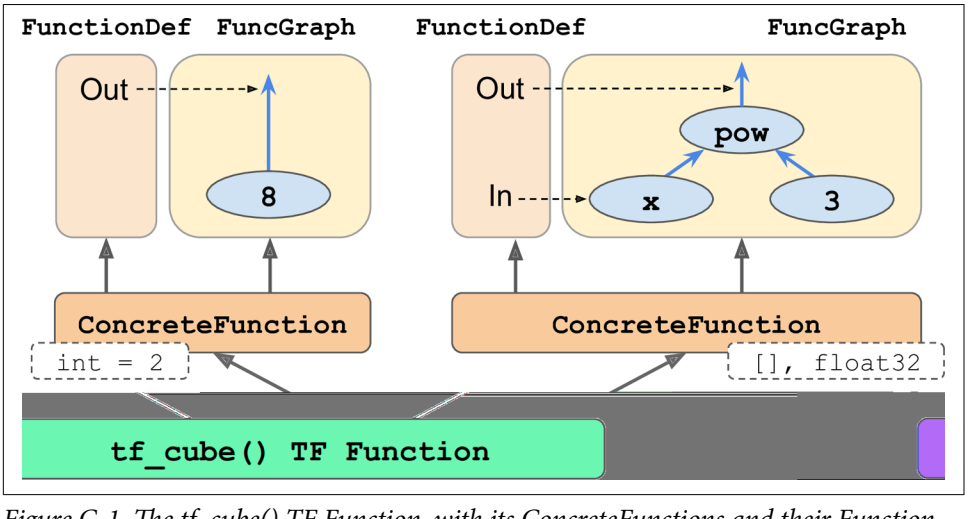


Figure G-1. The tf_cube() TF Function, with its ConcreteFunctions and their FunctionGraphs

The tensors in these graphs are *symbolic tensors*, meaning they don't have an actual value, just a data type, a shape, and a name. They represent the future tensors that will flow through the graph once an actual value is fed to the placeholder `x` and the graph is executed. Symbolic tensors make it possible to specify ahead of time how to

connect operations, and they also allow TensorFlow to recursively infer the data types and shapes of all tensors, given the data types and shapes of their inputs.

Now let's continue to peek under the hood, and see how to access function definitions and function graphs and how to explore a graph's operations and tensors.

Exploring Function Definitions and Graphs

You can access a concrete function's computation graph using the `graph` attribute, and get the list of its operations by calling the graph's `get_operations()` method:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

In this example, the first operation represents the input argument `x` (it is called a *placeholder*), the second "operation" represents the constant 3, the third operation represents the power operation (`**`), and the final operation represents the output of this function (it is an identity operation, meaning it will do nothing more than copy the output of the addition operation[1]). Each operation has a list of input and output tensors that you can easily access using the operation's `inputs` and `outputs` attributes. For example, let's get the list of inputs and outputs of the power operation:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

This computation graph is represented in Figure G-2.

1 You can safely ignore it—it is only here for technical reasons, to ensure that TF Functions don't leak internal structures.

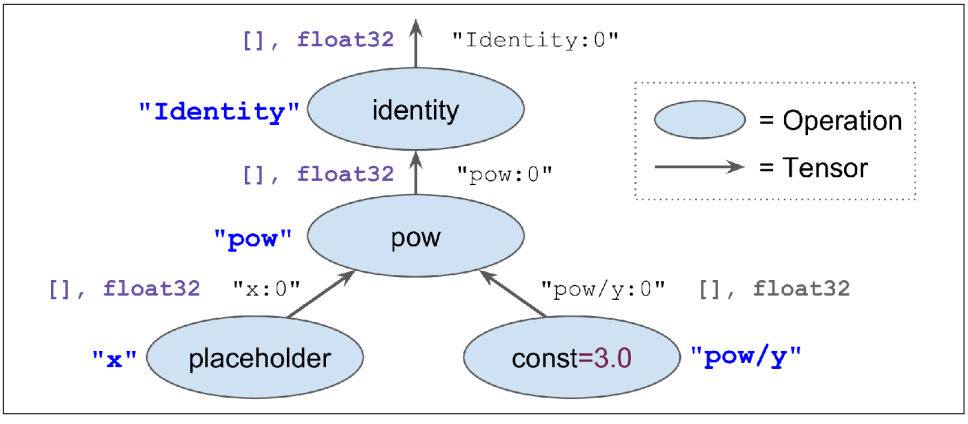


Figure G-2. Example of a computation graph

Note that each operation has a name. It defaults to the name of the operation (e.g., `"pow"`), but you can define it manually when calling the operation (e.g., `tf.pow(x, 3, name="other_name")`). If a name already exists, TensorFlow automatically adds a unique index (e.g., `"pow_1"`, `"pow_2"`, etc.). Each tensor also has a unique name: it is always the name of the operation that outputs this tensor, plus `:0` if it is the operation's first output, or `:1` if it is the second output, and so on. You can fetch an operation or a tensor by name using the graph's `get_operation_by_name()` or `get_tensor_by_name()` methods:

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

The concrete function also contains the function definition (represented as a protocol buffer[2]), which includes the function's signature. This signature allows the concrete function to know which placeholders to feed with the input values, and which tensors to return:

```
>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
  name: "x"
  type: DT_FLOAT
}
output_arg {
  name: "identity"
  type: DT_FLOAT
}
```

2 A popular binary format discussed in Chapter 13.

Now let's look more closely at tracing.

A Closer Look at Tracing

Let's tweak the `tf_cube()` function to print its input:

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

Now let's call it:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

The `result` looks good, but look at what was printed: `x` is a symbolic tensor! It has a shape and a data type, but no value. Plus it has a name (`"x:0"`). This is because the `print()` function is not a TensorFlow operation, so it will only run when the Python function is traced, which happens in graph mode, with arguments replaced with symbolic tensors (same type and shape, but no value). Since the `print()` function was not captured into the graph, the next times we call `tf_cube()` with float32 scalar tensors, nothing is printed:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

But if we call `tf_cube()` with a tensor of a different type or shape, or with a new Python value, the function will be traced again, so the `print()` function will be called:

```
>>> result = tf_cube(2) # new Python value: trace!
x = 2
>>> result = tf_cube(3) # new Python value: trace!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]])) # New shape: trace!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # New shape: trace!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # Same shape: no trace
```

If your function has Python side effects (e.g., it saves some logs to disk), be aware that this code will only run when the function is traced (i.e., every time the TF Function is called with a new input signature). It best to assume that the function may be traced (or not) any time the TF Function is called.

In some cases, you may want to restrict a TF Function to a specific input signature. For example, suppose you know that you will only ever call a TF Function with batches of 28 × 28–pixel images, but the batches will have very different sizes. You may not want TensorFlow to generate a different concrete function for each batch size, or count on it to figure out on its own when to use `None`. In this case, you can specify the input signature like this:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # drop half the rows and columns
```

This TF Function will accept any float32 tensor of shape [*, 28, 28], and it will reuse the same concrete function every time:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
preprocessed_images = shrink(img_batch_1) # Works fine. Traces the function.
preprocessed_images = shrink(img_batch_2) # Works fine. Same concrete function.
```

However, if you try to call this TF Function with a Python value, or a tensor of an unexpected data type or shape, you will get an exception:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])
preprocessed_images = shrink(img_batch_3)  # ValueError! Unexpected signature.
```

Using AutoGraph to Capture Control Flow

If your function contains a simple `for` loop, what do you expect will happen? For example, let's write a function that will add 10 to its input, by just adding 1 10 times:

```
@tf.function
def add_10(x):
    for i in range(10):
        x += 1
    return x
```

It works fine, but when we look at its graph, we find that it does not contain a loop: it just contains 10 addition operations!

```
>>> add_10(tf.constant(0))
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'add' type=Add>, [...],
 <tf.Operation 'add_1' type=Add>, [...],
 <tf.Operation 'add_2' type=Add>, [...],
 [...]
 <tf.Operation 'add_9' type=Add>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

This actually makes sense: when the function got traced, the loop ran 10 times, so the `x += 1` operation was run 10 times, and since it was in graph mode, it recorded this operation 10 times in the graph. You can think of this `for` loop as a "static" loop that gets unrolled when the graph is created.

If you want the graph to contain a "dynamic" loop instead (i.e., one that runs when the graph is executed), you can create one manually using the `tf.while_loop()` operation, but it is not very intuitive (see the "Using AutoGraph to Capture Control Flow" section of the Chapter 12 notebook for an example). Instead, it is much simpler to use TensorFlow's *AutoGraph* feature, discussed in Chapter 12. AutoGraph is actually activated by default (if you ever need to turn it off, you can pass `autograph=False` to `tf.function()`). So if it is on, why didn't it capture the `for` loop in the `add_10()` function? Well, it only captures `for` loops that iterate over `tf.range()`, not `range()`. This is to give you the choice:

- If you use `range()`, the `for` loop will be static, meaning it will only be executed when the function is traced. The loop will be "unrolled" into a set of operations for each iteration, as we saw.
- If you use `tf.range()`, the loop will be dynamic, meaning that it will be included in the graph itself (but it will not run during tracing).

Let's look at the graph that gets generated if you just replace `range()` with `tf.range()` in the `add_10()` function:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()
[<tf.Operation 'x' type=Placeholder>, [...],
 <tf.Operation 'range' type=Range>, [...],
 <tf.Operation 'while' type=While>, [...],
 <tf.Operation 'Identity' type=Identity>]
```

As you can see, the graph now contains a `While` loop operation, as if you had called the `tf.while_loop()` function.

Handling Variables and Other Resources in TF Functions

In TensorFlow, variables and other stateful objects, such as queues or datasets, are called *resources*. TF Functions treat them with special care: any operation that reads or updates a resource is considered stateful, and TF Functions ensure that stateful operations are executed in the order they appear (as opposed to stateless operations, which may be run in parallel, so their order of execution is not guaranteed). Moreover, when you pass a resource as an argument to a TF Function, it gets passed by reference, so the function may modify it. For example:

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter) # counter is now equal to 1
increment(counter) # counter is now equal to 2
```

If you peek at the function definition, the first argument is marked as a resource:

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE
```

It is also possible to use a `tf.Variable` defined outside of the function, without explicitly passing it as an argument:

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

The TF Function will treat this as an implicit first argument, so it will actually end up with the same signature (except for the name of the argument). However, using global variables can quickly become messy, so you should generally wrap variables (and other resources) inside classes. The good news is `@tf.function` works fine with methods too:

```
class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

    @tf.function
    def increment(self, c=1):
        return self.counter.assign_add(c)
```

Do not use `=`, `+=`, `-=`, or any other Python assignment operator with TF variables. Instead, you must use the `assign()`, `assign_add()`, or `assign_sub()` methods. If you try to use a Python assignment operator, you will get an exception when you call the method.

A good example of this object-oriented approach is, of course, tf.keras. Let's see how to use TF Functions with tf.keras.

Using TF Functions with tf.keras (or Not)

By default, any custom function, layer, or model you use with tf.keras will automatically be converted to a TF Function; you do not need to do anything at all! However, in some cases you may want to deactivate this automatic conversion—for example, if your custom code cannot be turned into a TF Function, or if you just want to debug your code, which is much easier in eager mode. To do this, you can simply pass `dynamic=True` when creating the model or any of its layers:

```
model = MyModel(dynamic=True)
```

If your custom model or layer will always be dynamic, you can instead call the base class's constructor with `dynamic=True`:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

Alternatively, you can pass `run_eagerly=True` when calling the `compile()` method:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
              run_eagerly=True)
```

Now you know how TF Functions handle polymorphism (with multiple concrete functions), how graphs are automatically generated using AutoGraph and tracing, what graphs look like, how to explore their symbolic operations and tensors, how to handle variables and resources, and how to use TF Functions with tf.keras.

Index

Symbols

A

B

C

D

E

F

G

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

About the Author

Aurélien Géron is a Machine Learning consultant and lecturer. A former Googler, he led YouTube's video classification team from 2013 to 2016. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Kiwisoft, a consulting firm focused on Machine Learning and data privacy.

Before all that he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured about computer science at a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* is the fire salamander (*Salamandra salamandra*), an amphibian found across most of Europe. Its black, glossy skin features large yellow spots on the head and back, signaling the presence of alkaloid toxins. This is a possible source of this amphibian's common name: contact with these toxins (which they can also spray short distances) causes convulsions and hyperventilation. Either the painful poisons or the moistness of the salamander's skin (or both) led to a misguided belief that these creatures not only could survive being placed in fire but could extinguish it as well.

Fire salamanders live in shaded forests, hiding in moist crevices and under logs near the pools or other freshwater bodies that facilitate their breeding. Though they spend most of their lives on land, they give birth to their young in water. They subsist mostly on a diet of insects, spiders, slugs, and worms. Fire salamanders can grow up to a foot in length, and in captivity may live as long as 50 years.

The fire salamander's numbers have been reduced by destruction of their forest habitat and capture for the pet trade, but the greatest threat they face is the susceptibility of their moisture-permeable skin to pollutants and microbes. Since 2014, they have become extinct in parts of the Netherlands and Belgium due to an introduced fungus.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. The cover illustration is by Karen Montgomery, based on an engraving from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.