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

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^\intercal\mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)}\left(t^{(i)}\left(\mathbf{w}^\intercal\mathbf{x}^{(i)} + b\right) - 1\right)$$
$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

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 $\left(\widehat{\mathbf{w}}, \hat{b}, \hat{\alpha}\right)$ 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

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$
$$\frac{\partial}{\partial b}\mathcal{L}(\mathbf{w}, b, \alpha) = -\sum_{i=1}^{m} \alpha^{(i)} t^{(i)}$$

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 $\widehat{\boldsymbol{\alpha}}$ 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 $\widehat{\boldsymbol{\alpha}}$, 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).

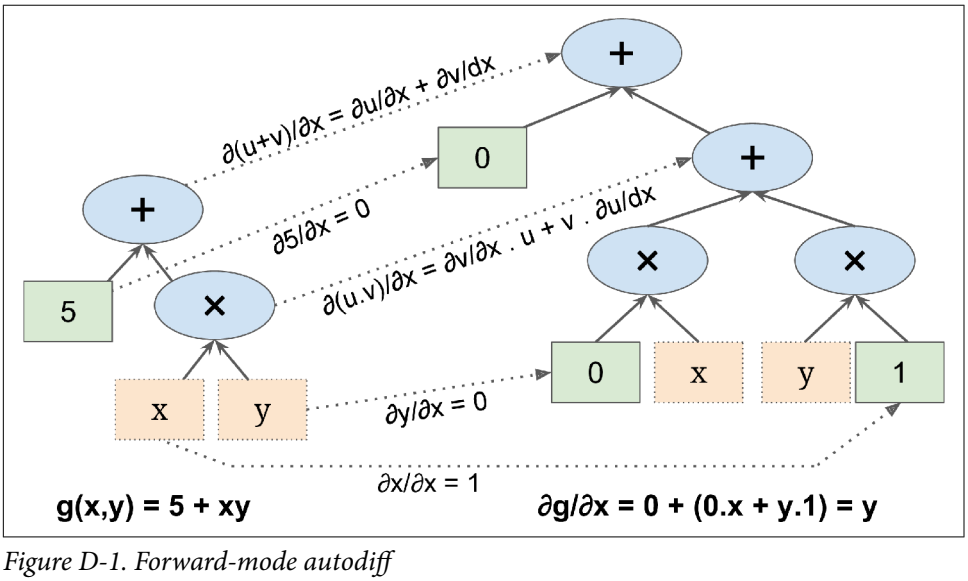

Figure D-1. Forward-mode autodiff

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).

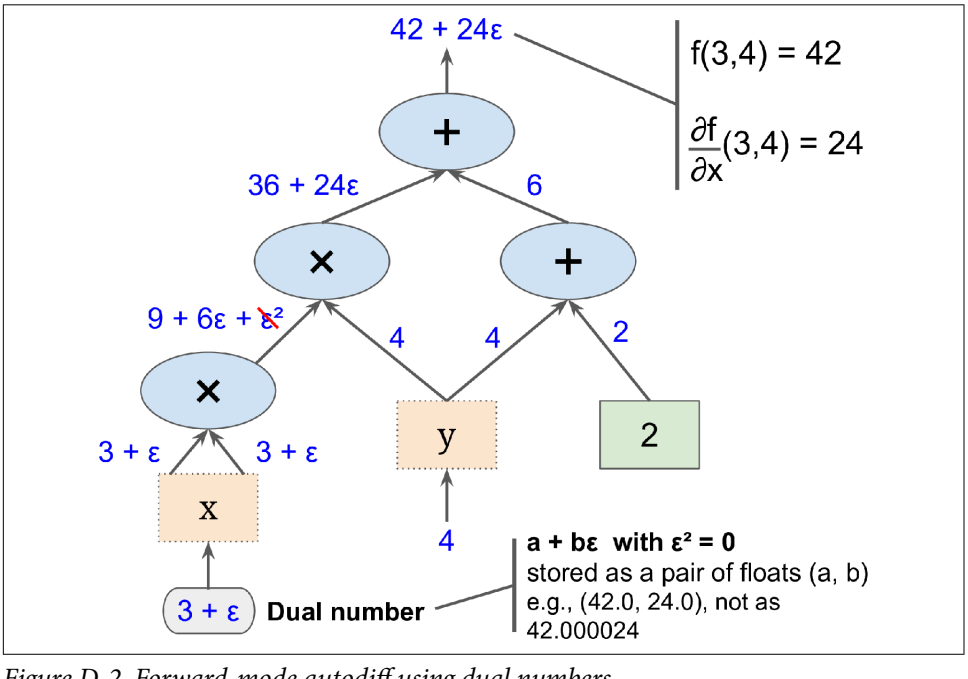

Figure D-2. Forward-mode autodiff using dual numbers

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: $f(3, 4) = n_7 = 42$.

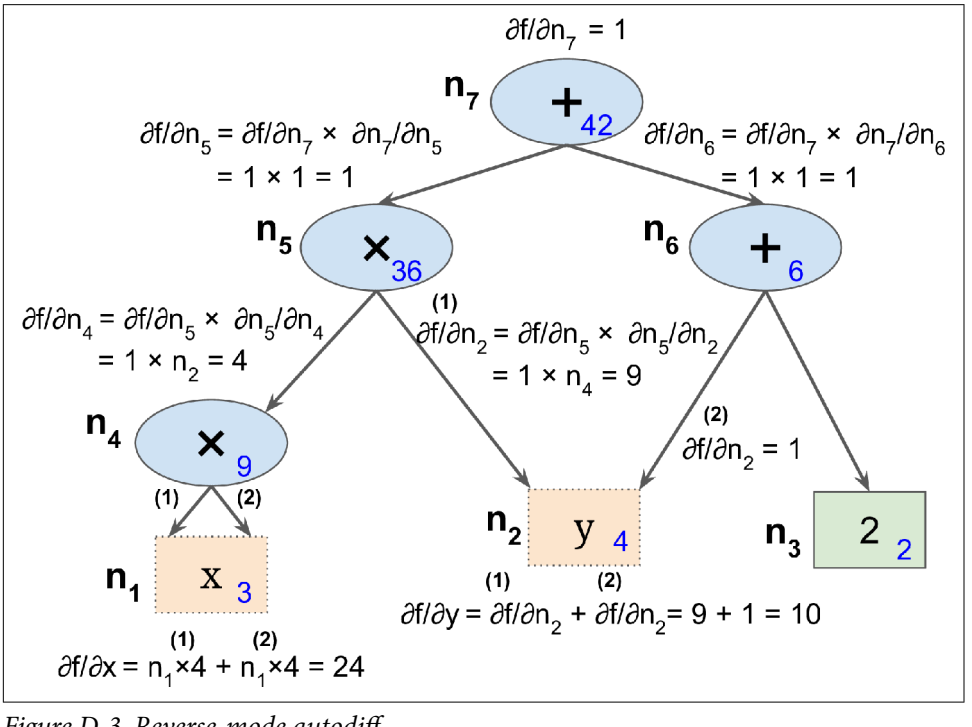

Figure D-3. Reverse-mode autodiff

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.

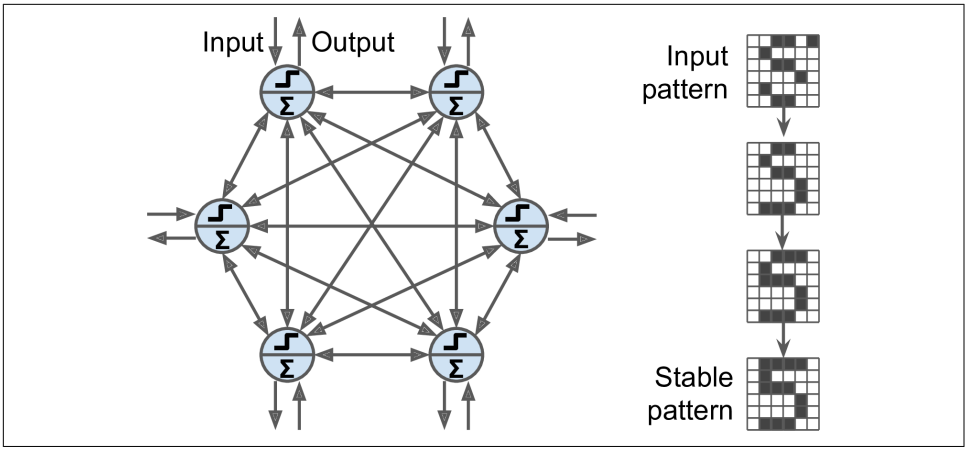

Figure E-1. Hopfield network

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*.

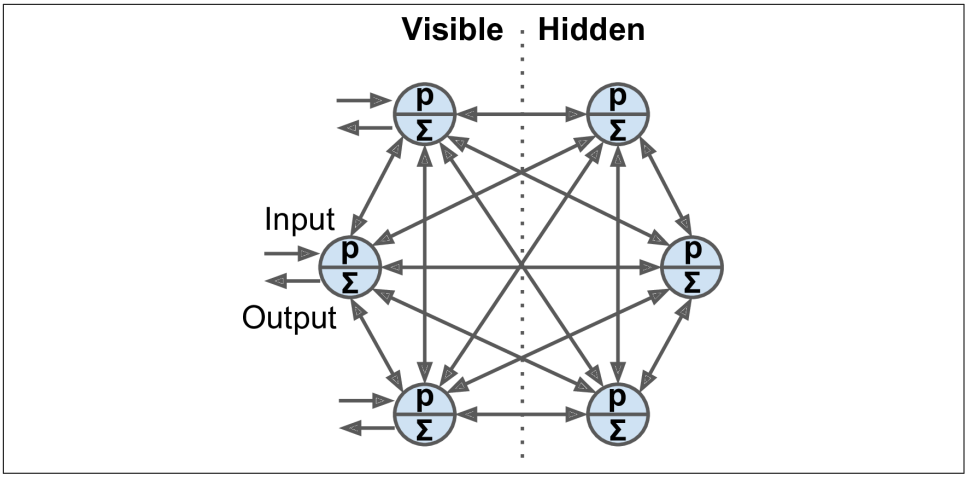

Figure E-2. Boltzmann machine

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.

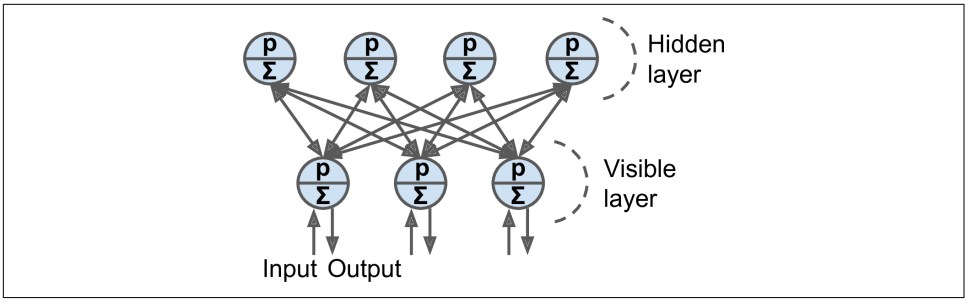

Figure E-3. Restricted Boltzmann machine

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

$$w_{i,j} \leftarrow w_{i,j} + \eta\left(\mathbf{x}\mathbf{h}^{\intercal} - \mathbf{x}'\mathbf{h}'^{\intercal}\right)$$

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.

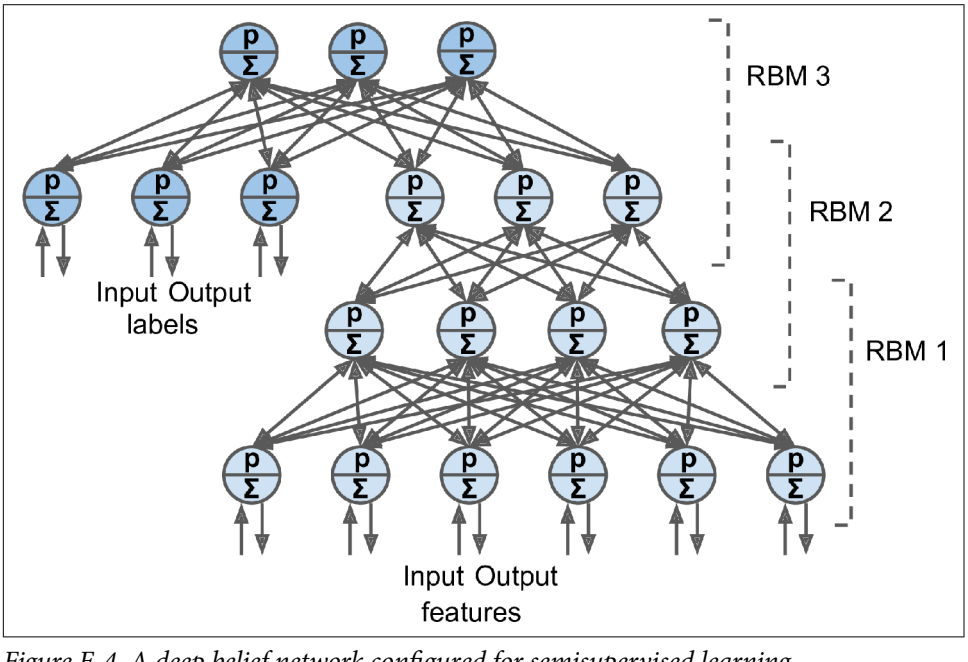

Figure E-4. A deep belief network configured for semisupervised learning

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).

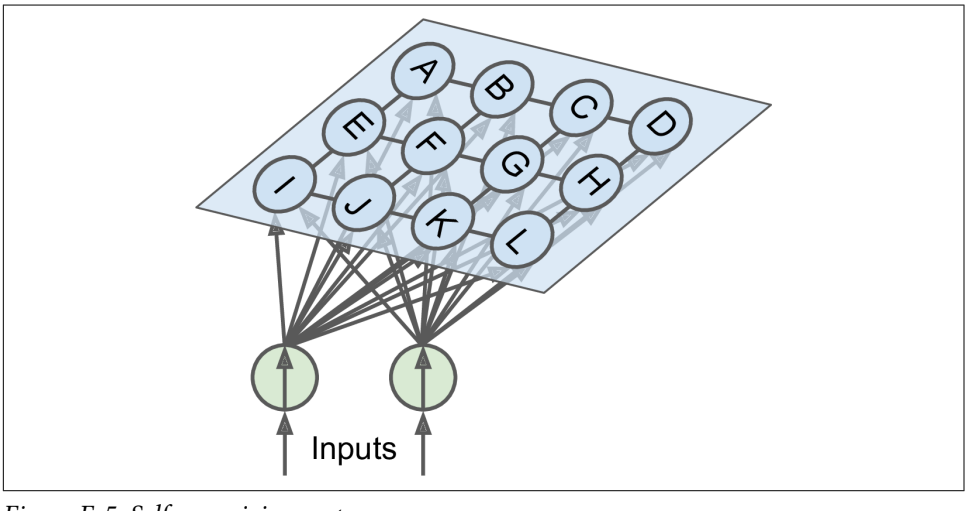

Figure E-5. Self-organizing map

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*.