List of Figures

Chapter 1. A machine-learning odyssey

Figure 1.1. Each pair of integers, when summed, results in an even or odd number. The input and output correspondences listed are called the ground-truth dataset.

Figure 1.2. This table reveals the inner logic behind how the output response corresponds to the input pairs.

Figure 1.3. An ML approach to solving problems can be thought of as tuning the parameters of a black box until it produces satisfactory results.

Figure 1.4. The learning approach generally follows a structured recipe. First, the dataset needs to be transformed into a representation, most often a list of features, which can be used by the learning algorithm. The learning algorithm chooses a model and efficiently searches for the model's parameters.

Figure 1.5. The inference approach generally uses a model that has already been either learned or given. After converting data into a usable representation, such as a feature vector, it uses the model to produce intended output.

Figure 1.6. Feature engineering is the process of selecting relevant features for the task.

Figure 1.7. The L1 distance is also called the Manhattan distance (also referred to as the taxicab metric), because it resembles the route of a car in a grid-like neighborhood such as Manhattan. If a car is traveling from point (0,1) to point (1,0), the shortest route requires a length of 2 units.

Figure 1.8. The L2 norm between points (0,1) and (1,0) is the length of a single straight-line segment between both points.

Figure 1.9. Example of TensorBoard in action

Figure 1.10. This chapter introduced fundamental machine-learning concepts, and the next chapter begins your journey into TensorFlow. Other tools to apply machine-learning algorithms (such as Caffe, Theano, and Torch) are available, but you'll see in chapter 2 why TensorFlow is the way to go.

Chapter 2. TensorFlow essentials

Figure 2.1. The matrix in the lower half of the diagram is a visualization from its compact code notation in the upper half of the diagram. This form of notation is a common paradigm in most scientific computing libraries.

Figure 2.2. This tensor can be thought of as multiple matrices stacked on top of each other. To specify an element, you must indicate the row and column, as well as which matrix is being accessed. Therefore, the rank of this tensor is 3.

Figure 2.3. The graph represents the operations needed to produce a Gaussian distribution. The links between the nodes represent how data flows from one operation to the next. The operations themselves are simple, but the complexity arises from the way they intertwine.

Figure 2.4. The session dictates how the hardware will be used to process the graph most efficiently. When the session starts, it assigns the CPU and GPU devices to each of the nodes. After processing, the session outputs data in a usable format, such as a NumPy array. A session optionally may be fed placeholders, variables, and constants.

Figure 2.5. Running the Jupyter Notebook will launch an interactive notebook on http://localhost:8888.

Figure 2.6. The drop-down menu changes the type of cell in the notebook. The Code cell is for Python code, whereas the Markdown code is for text descriptions.

Figure 2.7. An interactive Python notebook presents both code and comments grouped for readability.

Figure 2.8. The summary display in TensorBoard created in listing 2.16. TensorBoard provides a user-friendly interface to visualize data produced in TensorFlow.

Chapter 3. Linear regression and beyond

Figure 3.1. Different values of the parameter w result in different linear equations. The set of all these linear equations is what constitutes the linear model M.

Figure 3.2. A regression algorithm is meant to produce continuous output. The input is allowed to be discrete or continuous. This distinction is important because discrete-valued outputs are handled better by classification, which is discussed in the next chapter.

Figure 3.3. Ideally, the best-fit curve fits well on both the training data and the test data. If we witness it fitting poorly with the test data and the training data, there’s a chance that our model is underfitting. On the other hand, if it performs poorly on the test data but well on the training data, we know the model is overfitting.

Figure 3.4. Examples of underfitting and overfitting the data

Figure 3.5. Scatter plot of y = x + (noise)

Figure 3.6. Whichever parameter w minimizes, the cost is optimal. Cost is defined as the norm of the error between the ideal value with the model response. And, lastly, the response value is calculated from the function in the model set.

Figure 3.7. The cost is the norm of the point-wise difference between the model response and the true value.

Figure 3.8. Linear regression estimate shown by running listing 3.2

Figure 3.9. The learning algorithm updates the model's parameters to minimize the given cost function.

Figure 3.10. Data points like this aren't suitable for a linear model.

Figure 3.11. The best-fit curve smoothly aligns with the nonlinear data.

Figure 3.12. When the model is too flexible, a best-fit curve can look awkwardly complicated or unintuitive. We need to use regularization to improve the fit, so that the learned model performs well against test data.

Figure 3.13. As you increase the regularization parameter to some extent, the cost decreases. This implies that the model was originally overfitting the data, and regularization helped add structure.

Chapter 4. A gentle introduction to classification

Figure 4.1. A classifier produces discrete outputs but may take either continuous or discrete inputs.

Figure 4.2. There are two types of discrete sets: those with values that can be ordered (ordinal) and those with values that can't (nominal).

Figure 4.3. If the values of a variable are nominal, they might need to be preprocessed. One solution is to treat each nominal value as a Boolean variable, as shown on the right: banana, apple, and orange are three newly added variables, each having a value of 0 or 1. The original fruit variable is removed.

Figure 4.4. You can compare predicted results to actual results by using a matrix of positive (green check mark) and negative (red forbidden) labels.

Figure 4.5. An example of a confusion matrix for evaluating the performance of a classification algorithm

Figure 4.6. The principled way to compare algorithms is by examining their ROC curves. When the true-positive rate is greater than the false-positive rate in every situation, it's straightforward to declare that one algorithm is dominant in terms of its performance. If the true-positive rate is less than the false-positive rate, the plot dips below the baseline shown by the dotted line.

Figure 4.7. A visualization of a binary classification training dataset. The values are divided into two classes: all points where y = 1, and all points where y = 0.

Figure 4.8. The diagonal line is the best-fit line on a classification dataset. Clearly, the line doesn't fit the data well, but it provides an imprecise approach for classifying new data.

Figure 4.9. A new training element of value 20 greatly influences the best-fit line. The line is too sensitive to outlying data, and therefore linear regression is a sloppy classifier.

Figure 4.10. A visualization of the sigmoid function

Figure 4.11. Here's a visualization of how the two cost functions penalize values at 0 and 1. Notice that the left function heavily penalizes 0 but has no cost at 1. The right cost function displays the opposite phenomena.

Figure 4.12. Here's a best-fit sigmoid curve for a binary classification dataset. Notice that the curve resides within y = 0 and y = 1. That way, this curve isn't that sensitive to outliers.

Figure 4.13. The x-axis and y-axis represent the two independent variables. The dependent variable holds two possible labels, represented by the shape and color of the plotted points.

Figure 4.14. The diagonal dotted line represents when the probability between the two decisions is split equally. The confidence of making a decision increases as data lies farther away from the line.

Figure 4.15. The independent variable is two-dimensional, indicated by the x-axis and y-axis. The dependent variable can be one of three labels, shown by the color and shape of the data points.

Figure 4.16. One-versus-all is a multiclass classifier approach that requires a detector for each class.

Figure 4.17. In one-versus-one multiclass classification, there's a detector for each pair of classes.

Figure 4.18. 2D training data for multi-output classification

Chapter 5. Automatically clustering data

Figure 5.1. You can use a queue in TensorFlow to read files. The queue is built into the TensorFlow framework, and you can use the reader.read(...) function to access (and dequeue) it.

Figure 5.2. The chromagram matrix, where the x-axis represents time, and the y-axis represents pitch class. The green parallelograms indicate the presence of that pitch at that time.

Figure 5.3. The most influential pitch at every time interval is highlighted. You can think of it as the loudest pitch at each time interval.

Figure 5.4. You count the frequency of loudest pitches heard at each interval to generate this histogram, which acts as your feature vector.

Figure 5.5. Four examples of audio files. As you can see, the two on the right appear to have similar histograms. The two on the left also have similar histograms. Your clustering algorithms will be able to group these sounds together.

Figure 5.6. One iteration of the k-means algorithm. Let’s say you’re clustering colors into three buckets (an informal way to say category). You can start with an initial guess of red, green, and blue and begin the assignment step. Then you update the bucket colors by averaging the colors that belong to each bucket. You keep repeating until the buckets no longer substantially change color, arriving at the color representing the centroid of each cluster.

Figure 5.7. Audio segmentation is the process of automatically labeling segments.

Figure 5.8. In the real world, we see groups of people in clusters all the time. Applying k-means requires knowing the number of clusters ahead of time. A more flexible tool is a self-organizing map, which has no preconceptions about the number of clusters.

Figure 5.9. One iteration of the SOM algorithm. The first step is to identify the best matching unit (BMU), and the second step is to update the neighboring nodes. You keep iterating these two steps with training data until certain convergence criteria are reached.

Figure 5.10. The SOM places all three-dimensional data points into a two-dimensional grid. From it, you can pick the cluster centroids (automatically or manually) and achieve clustering in an intuitive lower-dimensional space.

Chapter 6. Hidden Markov models

Figure 6.1. Weather conditions (states) represented as nodes in a graph

Figure 6.2. Transition probabilities between weather conditions are represented as directed edges.

Figure 6.3. A trellis representation of the Markov system changing states over time

Figure 6.4. A transition matrix conveys the probabilities of a state from the left (rows) transitioning to a state at the top (columns).

Figure 6.5. A hidden Markov model trellis showing how weather conditions might produce temperature readings

Figure 6.6. Screenshot of HMM example scenario from Wikipedia

Chapter 7. A peek into autoencoders

Figure 7.1. A graphical representation of the linear equation f(x) = w × x + b. The nodes are represented as circles, and edges are represented as arrows. The values on the edges are often called weights, and they act as a multiplication on the input. When two arrows lead to the same node, they act as a summation of the inputs.

Figure 7.2. Use nonlinear functions such as sig, tan, and ReLU to introduce nonlinearity to your models.

Figure 7.3. A nonlinear function, such as sigmoid, is applied to the output of a node.

Figure 7.4. A two-input network will have three parameters (w1, w2, and b). Remember, multiple lines leading to the same node indicate summation.

Figure 7.5. dot producttwo-input networkThe input dimension can be arbitrarily long. For example, each pixel in a grayscale image can have a corresponding input xi. This neural network uses all inputs to generate a single output number, which you might use for regression or classification. The notation wT means you’re transposing w, which is an n × 1 vector, into a 1 × n vector. That way, you can properly multiply it with x (which has the dimensions n × 1). Such a matrix multiplication is also called a dot product, and it yields a scalar (one-dimensional) value.

Figure 7.6. Nodes that don’t interface to both the input and the output are called hidden neurons. A hidden layer is a collection of hidden units that aren’t connected to each other.

Figure 7.7. If you want to create a network where the input equals the output, you can connect the corresponding nodes and set each parameter’s weight to 1.

Figure 7.8. Here, you introduce a restriction to a network that tries to reconstruct its input. Data will pass through a narrow channel, as illustrated by the hidden layer. In this example, there’s only one node in the hidden layer. This network is trying to encode (and decode) an n-dimensional input signal into just one dimension, which will likely be difficult in practice.

Figure 7.9. A colored image is composed of pixels, and each pixel contains values for red, green, and blue.

Figure 7.10. An image can be represented in row-major order. That way, you can represent a two-dimensional structure as a onedimensional structure.

Chapter 8. Reinforcement learning

Figure 8.1. A person navigating to reach a destination in the midst of traffic and unexpected situations is a problem setup for reinforcement learning.

Figure 8.2. Actions are represented by arrows, and states are represented by circles. Performing an action on a state produces a reward. If you start at state s1, you can perform action a1 to obtain a reward r(s1, a1).

Figure 8.3. A policy suggests which action to take, given a state.

Figure 8.4. Given a state and the action taken, applying a utility function Q predicts the expected and the total rewards: the immediate reward (next state) plus rewards gained later by following an optimal policy.

Figure 8.5. Ideally, our algorithm should buy low and sell high. Doing so just once, as shown here, might yield a reward of around $160. But the real profit rolls in when you buy and sell more frequently. Ever heard the term high-frequency trading? It’s about buying low and selling high as frequently as possible to maximize profits within a period of time.

Figure 8.6. This chart summarizes the opening stock prices of Microsoft (MSFT) from 7/22/1992 to 7/22/2016. Wouldn’t it have been nice to buy around day 3000 and sell around day 5000? Let’s see if our code can learn to buy, sell, and hold to make optimal gain.

Figure 8.7. Most reinforcement-learning algorithms boil down to just three main steps: infer, do, and learn. During the first step, the algorithm selects the best action (a), given a state (s), using the knowledge it has so far. Next, it does the action to find out the reward (r) as well as the next state (s'). Then it improves its understanding of the world by using the newly acquired knowledge (s, r, a, s').

Figure 8.8. A rolling window of a certain size iterates through the stock prices, as shown by the chart segmented to form states S1, S2, and S3. The policy suggests an action to take: you may either choose to exploit it or randomly explore another action. As you get rewards for performing an action, you can update the policy function over time.

Figure 8.9. The input is the state space vector, with three outputs: one for each output's Q-value.

Figure 8.10. The algorithm learns a good policy to trade Microsoft stocks.

Chapter 9. Convolutional neural networks

Figure 9.1. In a fully connected network, each pixel of an image is treated as an input. For a grayscale image of size 256 × 256, that's 256 × 256 neurons! Connecting each neuron to 10 outputs yields 256 × 256 × 10 = 655,360 weights.

Figure 9.2. Convolving a 5 × 5 patch over an image, as shown on the left, produces another image, as shown on the right. In this case, the produced image is the same size as the original. Converting an original image to a convolved image requires only 5 × 5 = 25 parameters!

Figure 9.3. Images from the CIFAR-10 dataset. Because they're only 32 × 32 in size, they're a bit difficult to see, but you can generally recognize some of the objects.

Figure 9.4. These are 32 randomly initialized matrices, each of size 5 × 5. They represent the filters you'll use to convolve an input image.

Figure 9.5. An example 24 × 24 image from the CIFAR-10 dataset

Figure 9.6. Resulting images from convolving the random filters on an image of a car

Figure 9.7. After you add a bias term and an activation function, the resulting convolutions can capture more-powerful patterns within images.

Figure 9.8. After running maxpool, the convolved outputs are halved in size, making the algorithm computationally faster without losing too much information.

Figure 9.9. An input image is convolved by multiple 5 × 5 filters. The convolution layer includes an added bias term with an activation function, resulting in 5 × 5 + 5 = 30 parameters. Next, a max-pooling layer reduces the dimensionality of the data (which requires no extra parameters).

Chapter 10. Recurrent neural networks

Figure 10.1. A neural network with the input and output layers labeled as X(t) and Y(t), respectively

Figure 10.2. The hidden layer of a neural network can be thought of as a hidden representation of the data, which is encoded by the input weights and decoded by the output weights.

Figure 10.3. Often you end up running the same neural network multiple times, without using knowledge about the hidden states of the previous runs.

Figure 10.4. A recurrent neural network architecture can use the previous states of the network to its advantage.

Figure 10.5. Raw data showing the number of international airline passengers throughout the years

Figure 10.6. The predictions match trends fairly well when tested against ground-truth data.

Figure 10.7. If the algorithm uses previously predicted results to make further predictions, then the general trend matches well, but not specific bumps.

Chapter 11. Sequence-to-sequence models for chatbots

Figure 11.1. Here's a high-level view of your neural network model. The input ayy is passed into the encoder RNN, and the decoder RNN is expected to respond with lmao. These are just toy examples for your chatbot, but you could imagine more-complicated pairs of sentences for the input and output.

Figure 11.2. The input, output, and states of an RNN. You can ignore the intricacies of exactly how an RNN is implemented. All that matters is the formatting of your input and output.

Figure 11.3. You can stack RNN cells to form a more complicated architecture.

Figure 11.4. TensorFlow lets you stack as many RNN cells as you want.

Figure 11.5. You can use the last states of the first cell as the next cell's initial state. This model can learn mapping from an input sequence to an output sequence. The model is called seq2seq.

Figure 11.6. A mapping from symbols to scalars

Figure 11.7. A mapping from symbols to vectors

Figure 11.8. A mapping from symbols to tensors

Figure 11.9. The seq2seq model learns a transformation between an input sequence to an output sequence by using an encoder RNN and a decoder RNN.

Figure 11.10. The RNNs accept only sequences of numeric values as input or output, so you'll convert your symbols to vectors. In this case, the symbols are words, such as the, fight, wind, and like. Their corresponding vectors are associated in the embedding matrix.

Figure 11.11. The decoder's input is prefixed with a special <GO> symbol, whereas the output is suffixed by a special <EOS> symbol.

Chapter 12. Utility landscape

Figure 12.1. Wrinkled clothes in a less favorable state than well-folded clothes. This diagram shows how you might score each state of a piece of cloth; higher scores represent a more favorable state.

Figure 12.2. This is a possible set of pairwise rankings between objects. Specifically, you have four food items, and you want to rank them by fanciness, so you employ two pairwise ranking decisions: steak is a fancier meal than a hotdog, and shrimp cocktail is a fancier meal than a burger.

Figure 12.3. Example data that you'll work with. The circles represent morefavorable states, whereas the crosses represent less-favorable states. You have an equal number of circles and crosses because the data comes in pairs; each pair is a ranking, as in figure 12.2.

Figure 12.4. The landscape of scores learned by the ranking neural network

Figure 12.5. Images can be embedded into much lower dimensions, such as 2D as shown here. Notice that points representing similar states of a shirt occur in nearby clusters. Embedding images allows you to use the ranking neural network to learn a preference between the states of a piece of cloth.

Figure 12.6. The VGG16 architecture is a deep convolutional neural network used for classifying images. This particular diagram is from www.cs.toronto.edu/~frossard/post/vgg16/.

Figure 12.7. A small segment of the computation graph shown in TensorBoard for the VGG16 neural network. The topmost node is the softmax operator used for classification. The three fully connected layers are labeled fc1, fc2, and fc3.

Figure 12.8. Videos of folding a shirt reveal how the cloth changes form through time. You can extract the first state and the last state of the shirt as your training data to learn a utility function to rank states. Final states of shirt in each video should be ranked with a higher utility than those shirts near the beginning of the video.

Figure 12.9. The utility increases over time, indicating the goal is being accomplished. The utility of the cloth near the beginning of the video is near 0, but it dramatically increases to 120,000 units by the end.

Appendix Installation

Figure A.1. Ensure that your 64-bit computer has virtualization enabled.

Figure A.2. Running the official TensorFlow container

Figure A.3. Docker's IP address can be found using the docker-machine ip command or can be found in the intro text under the ASCII whale.

Figure A.4. You can interact with TensorFlow through a Python interface called Jupyter.

Figure A.5. A possible error message from running the TensorFlow container

Figure A.6. Listing and killing a Docker container to get rid of the error message in figure A.5

Recommended / Playlists / History / Topics / Tutorials / Settings / Get the App / Sign Out

PREV
Index

NEXT
List of Tables