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 short-list 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.)
- Possibly useful 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 498).
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.

## Short-List 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 Forests, 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. Have a quick round of feature selection and engineering.
6. Have one or two more quick iterations of the five previous steps.
7. Short-list 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., should I replace missing values with zero or with the median value? Or 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, Hugo Larochelle, and Ryan Adams).[1]
2. Try Ensemble methods. Combining your best models will often perform better 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.
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").

1 "Practical Bayesian Optimization of Machine Learning Algorithms," J. Snoek, H. Larochelle, R. Adams (2012).

## 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 too: 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 regards to $x$, $y$, and $\alpha$; 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*

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)}\left(t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) - 1\right)$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

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: $t^{(i)}\left((\widehat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$,
- 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: $t^{(i)}\left((\widehat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}\right) = 1$. 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 regards 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 0, we have Equation C-3.

*Equation C-3. Properties of the stationary points*

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} = 0$$

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*

$$\mathcal{L}\left(\widehat{\mathbf{w}}, \hat{b}, \alpha\right) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The goal is now to find the vector $\hat{\alpha}$ that minimizes this function, with $\hat{\alpha}^{(i)} \geq 0$ for all instances. This constrained optimization problem is the dual problem we were looking for.

Once you find the optimal $\hat{\alpha}$, 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 verifies $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) = 1$, so if the k$^{\text{th}}$ instance is a support vector (i.e., $\alpha_k > 0$), you can use it to compute $\hat{b} = 1 - t^{(k)}\left(\widehat{\mathbf{w}}^T \cdot \mathbf{x}^{(k)}\right)$. However, it is often prefered 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*

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left[1 - t^{(i)}\left(\widehat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}\right)\right]$$

# APPENDIX D
# Autodiff

This appendix explains how TensorFlow's 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 $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are manual differentiation, symbolic differentiation, numerical differentiation, forward-mode autodiff, and finally reverse-mode autodiff. TensorFlow implements this last option. Let's go through each of these options.

## Manual Differentiation

The first approach is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the partial derivatives manually. 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 $\lambda x$ is $\lambda$ (where $\lambda$ is a constant).
- The derivative of $x^\lambda$ 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 $\lambda$ times a function is $\lambda$ 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. The good news is that deriving the mathematical equations for the partial derivatives like we just did can be automated, through a process called *symbolic differentiation*.

# Symbolic Differentiation

Figure D-1 shows how symbolic differentiation works on an even simpler function, $g(x,y) = 5 + xy$. The graph for that function is represented on the left. After symbolic differentiation, we get the graph on the right, which represents the partial derivative $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regards to $y$).

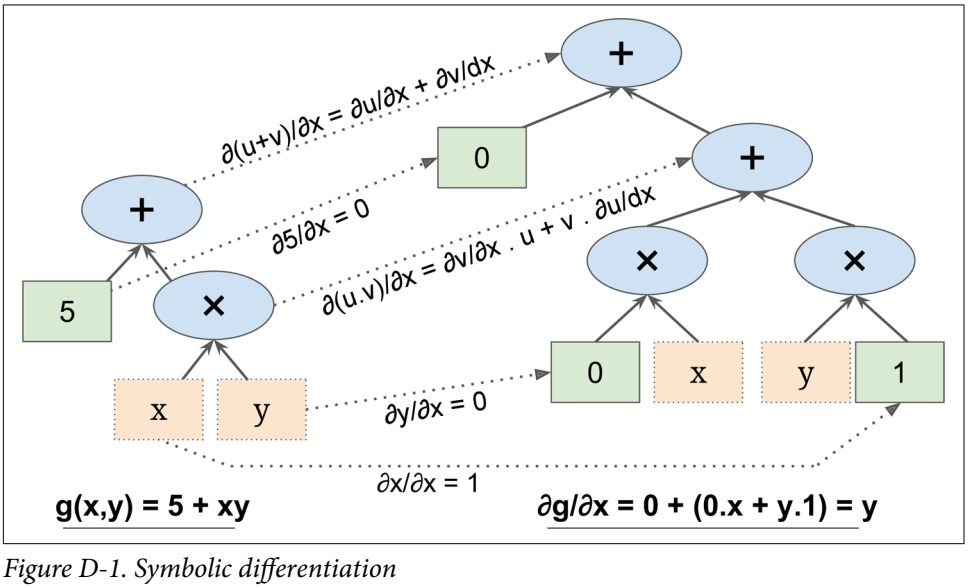


*Figure D-1. Symbolic differentiation*

The algorithm starts by getting the partial derivative 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 $\frac{\partial x}{\partial x} = 1$, and the variable $y$ returns the constant 0 since $\frac{\partial y}{\partial x} = 0$ (if we were looking for the partial derivative with regards 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 $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + \frac{\partial u}{\partial x} \times u$. 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: $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$.

However, it can be simplified (a lot). A few trivial pruning steps can be applied to this graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\frac{\partial g}{\partial x} = y$.

In this case, simplification is fairly easy, but for a more complex function, symbolic differentiation can produce a huge graph that may be tough to simplify and lead to suboptimal performance. Most importantly, symbolic differentiation cannot deal with functions defined with arbitrary code—for example, the following function discussed in Chapter 9:

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

# Numerical Differentiation

The simplest solution is to compute an approximation of the derivatives, numerically. 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, or more precisely Equation D-2.

*Equation D-2. Derivative of a function h(x) at point $x_0$*

$$h'(x) = \lim_{x \to x_0} \frac{h(x) - h(x_0)}{x - x_0}$$

$$= \lim_{\epsilon \to 0} \frac{h(x_0 + \epsilon) - h(x_0)}{\epsilon}$$

So if we want to calculate the partial derivative of $f(x,y)$ with regards to $x$, at $x = 3$ and $y = 4$, we can simply compute $f(3 + \epsilon, 4) - f(3, 4)$ and divide the result by $\epsilon$, using a very small value for $\epsilon$. 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 complex 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 numerical differentiation way too inefficient.

However, numerical differentiation 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.

# Forward-Mode Autodiff

*Forward-mode autodiff* is neither numerical differentiation nor symbolic differentiation, but in some ways it is their love child. It relies on *dual numbers*, which are (weird but fascinating) numbers of the form $a + b\epsilon$ where $a$ and $b$ are real numbers and $\epsilon$ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$). You can think of the dual number $42 + 24\epsilon$ 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, $42 + 24\epsilon$ 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*

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$
$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$
$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Most importantly, it can be shown that $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, so computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. Figure D-2 shows how forward-mode autodiff computes the partial derivative of $f(x,y)$ with regards to $x$ at $x = 3$ and $y = 4$. All we need to do is compute $f(3 + \epsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\frac{\partial f}{\partial x}(3, 4)$.

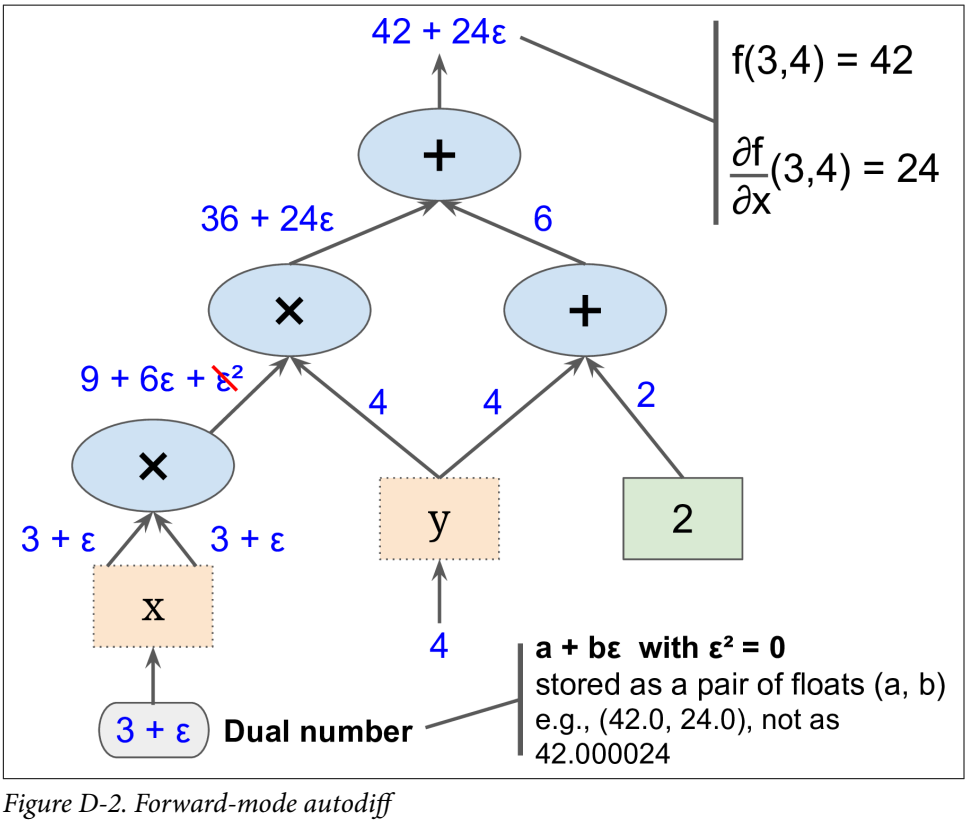


*Figure D-2. Forward-mode autodiff*

To compute $\frac{\partial f}{\partial y}(3, 4)$ we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \epsilon$.

So forward-mode autodiff is much more accurate than numerical differentiation, but it suffers from the same major flaw: 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.

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

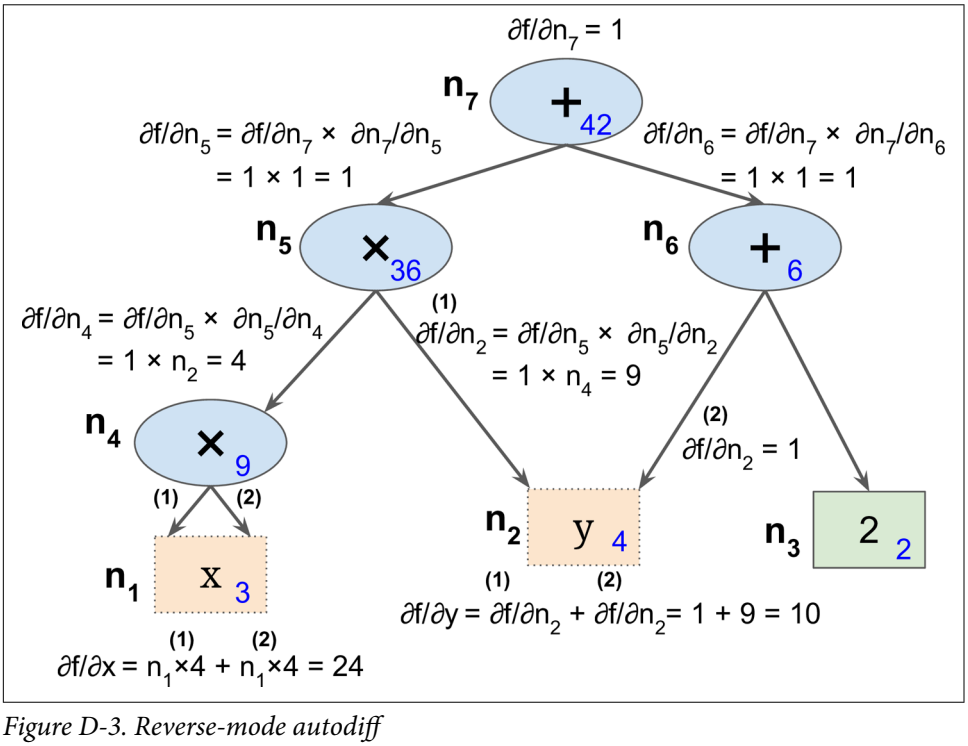


*Figure D-3. Reverse-mode autodiff*

The idea is to gradually go down the graph, computing the partial derivative of $f(x,y)$ with regards 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 trivially $\frac{\partial f}{\partial n_7} = 1$.

Let's continue down the graph to $n_5$: how much does $f$ vary when $n_5$ varies? The answer is $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. We already know that $\frac{\partial f}{\partial n_7} = 1$, so all we need is $\frac{\partial n_7}{\partial n_5}$. Since $n_7$ simply performs the sum $n_5 + n_6$, we find that $\frac{\partial n_7}{\partial n_5} = 1$, so $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Now we can proceed to node $n_4$: how much does $f$ vary when $n_4$ varies? The answer is $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Since $n_5 = n_4 \times n_2$, we find that $\frac{\partial n_5}{\partial n_4} = n_2$, so $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$.

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 $\frac{\partial f}{\partial x} = 24$ and $\frac{\partial f}{\partial y} = 10$. 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 regards to all the inputs. Most importantly, it can deal with functions defined by arbitrary code. It 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.

If you implement a new type of operation in TensorFlow and you want to make it compatible with autodiff, then you need to provide a function that builds a subgraph to compute its partial derivatives with regards 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 $f'(x) = 2x$. Note that this function does not compute a numerical result, but instead builds a subgraph that will (later) compute the result. This is very useful because it means that you can compute gradients of gradients (to compute second-order derivatives, or even higher-order derivatives).

APPENDIX E

# Other Popular ANN Architectures

In this appendix we will give a quick overview of a few historically important neural network architectures that are much less used today than deep Multi-Layer Perceptrons (Chapter 10), convolutional neural networks (Chapter 13), recurrent neural networks (Chapter 14), or autoencoders (Chapter 15). They are often mentioned in the literature, and some are still used in many applications, so it is worth knowing about them. Moreover, we will discuss *deep belief nets* (DBNs), 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 near 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 has made them useful in particular for character recognition 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.

They are fully connected graphs (see Figure E-1); that is, every neuron is connected to every other neuron. Note that on the diagram the images are $6 \times 6$ pixels, so the neural network on the left should contain 36 neurons (and 648 connections), but for visual clarity a much smaller network is represented.

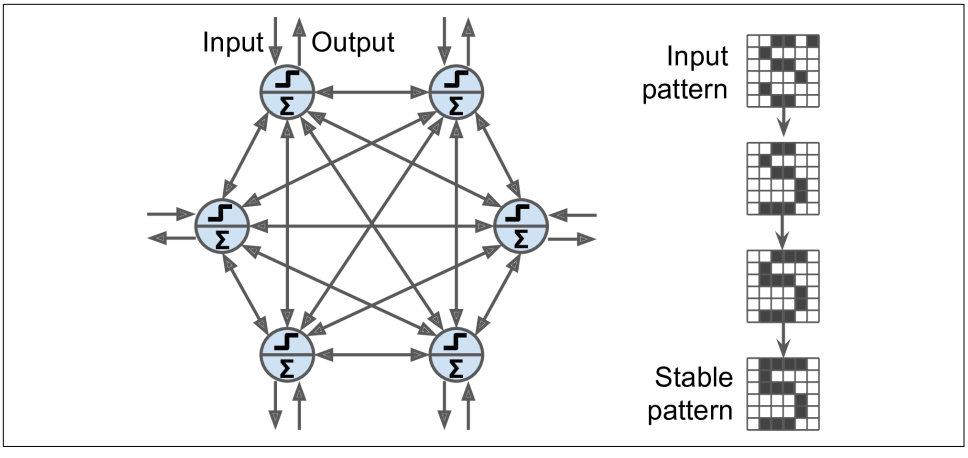


*Figure E-1. Hopfield network*

The training algorithm works by using Hebb's rule: 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 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 *sto-*

*chastic 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 a 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%).
- $\sigma$ 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.

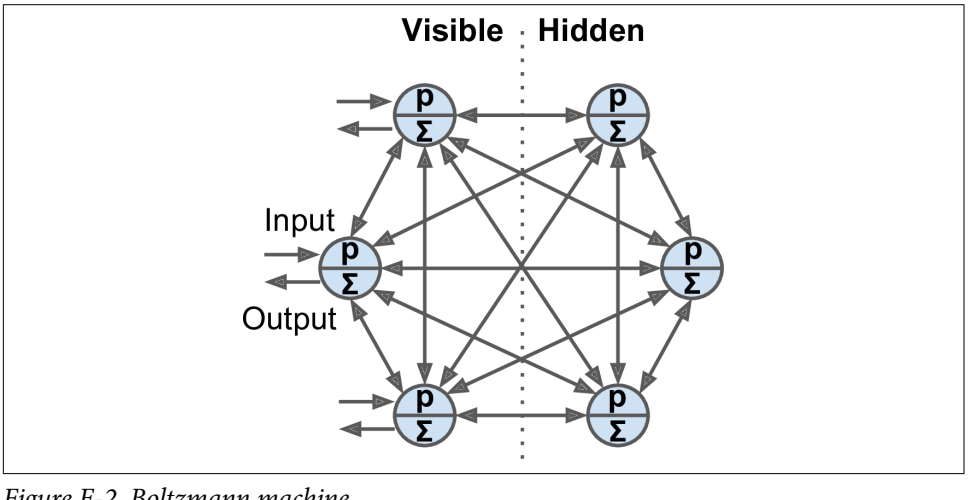


*Figure E-2. Boltzmann machine*

Because of its stochastic nature, a Boltzmann machine will never stabilize into a fixed configuration, but 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*.

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* (RBM).

## 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.

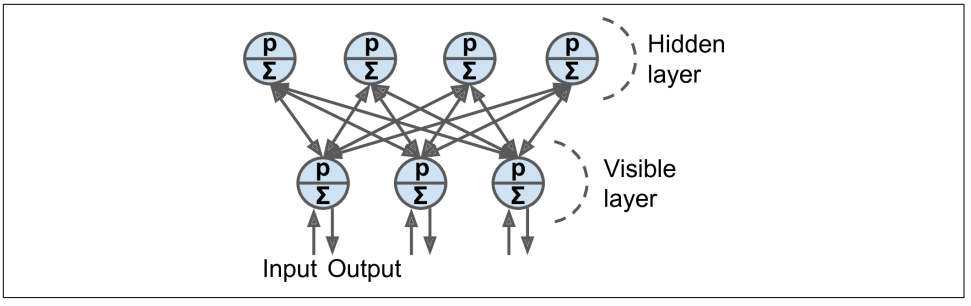


*Figure E-3. Restricted Boltzmann machine*

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 $\dot{\mathbf{x}}$. Then once again you compute the state of the hidden units, which gives you a vector $\dot{\mathbf{h}}$. Now you can update each connection weight by applying the rule in Equation E-2.

*Equation E-2. Contrastive divergence weight update*

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\mathbf{x}\mathbf{h}^T - \dot{\mathbf{x}}\dot{\mathbf{h}}^T\right)$$

The great benefit of this algorithm it 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 serves as the visible units for the second-layer RBM, and so on. Such an RBM stack is called a *deep belief net* (DBN).

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

1 "On Contrastive Divergence Learning," M. Á. Carreira-Perpiñán and G. Hinton (2005).

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.

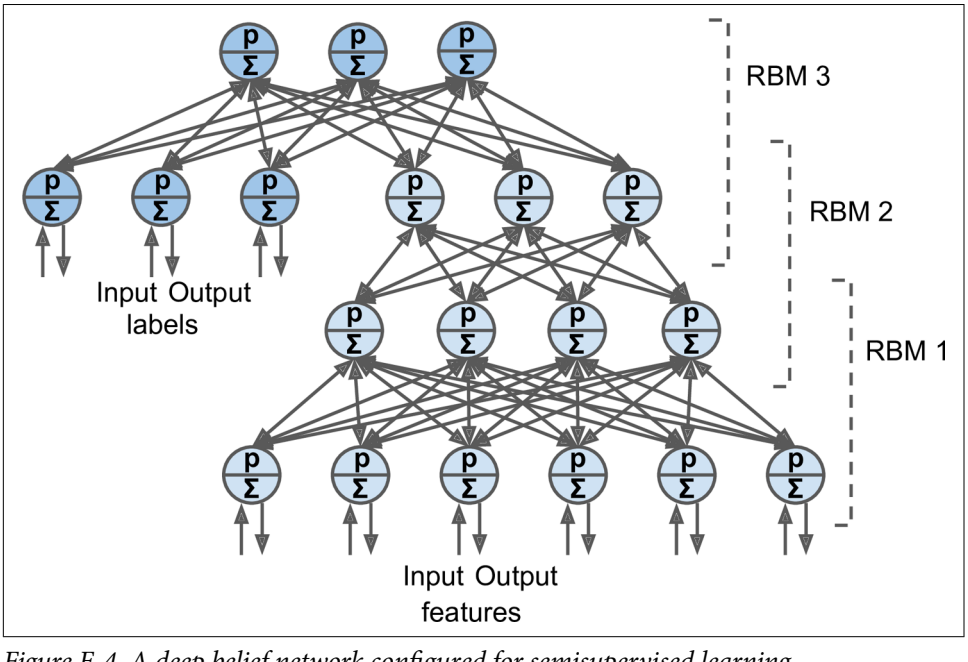


*Figure E-4. A deep belief network configured for semisupervised learning*

First, the 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 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.

2 "A Fast Learning Algorithm for Deep Belief Nets," G. Hinton, S. Osindero, Y. Teh (2006).

Lastly, RBM 3 is trained using both 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]

# Self-Organizing Maps

*Self-organizing maps* (SOM) are quite different from all the other types of neural networks we have discussed so far. They are used to produce a low-dimensional repre-

3 See this video by Geoffrey Hinton for more details and a demo: *http://goo.gl/7Z5QiS*.

sentation 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).

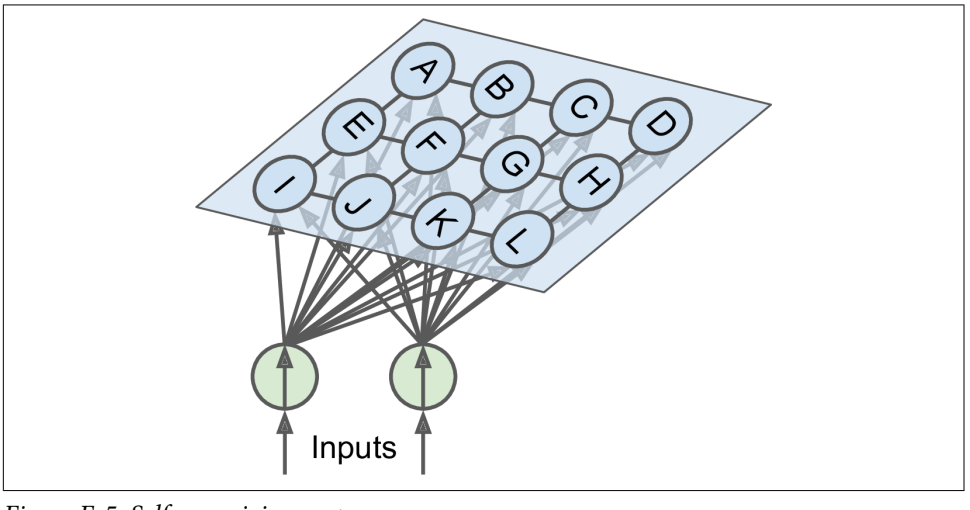


*Figure E-5. Self-organizing maps*

Once the network is trained, you can feed it a new instance and this will activate only one neuron (i.e., hence 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 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 the 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 with 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 even 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 vector to be slightly closer to the input vector (but they don't update their weights as much as the winner 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 her to practice more, especially with her 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 topics. After a while, the class is full of little specialized groups.

# Index

## Symbols


__call__(), 385
ε-greedy policy, 459, 464
ε-insensitive, 155
χ 2 test (see chi square test)
ℓ 0 norm, 39
ℓ 1 and ℓ 2 regularization, 303-304
ℓ 1 norm, 39, 130, 139, 300, 303
ℓ 2 norm, 39, 128-130, 139, 142, 303, 307
ℓ k norm, 39
ℓ ∞ norm, 39


## A


accuracy, 4, 83-84
actions, evaluating, 447-448
activation functions, 262-264
active constraints, 504
actors, 463
actual class, 85
AdaBoost, 192-195
Adagrad, 296-298
Adam optimization, 293, 298-300
adaptive learning rate, 297
adaptive moment optimization, 298
agents, 438
AlexNet architecture, 367-368
algorithms
    preparing data for, 59-68
AlphaGo, 14, 253, 437, 453
Anaconda, 41
anomaly detection, 12
Apple's Siri, 253
apply_gradients(), 286, 450
area under the curve (AUC), 92
arg_scope(), 285
array_split(), 217
artificial neural networks (ANNs), 253-274
    Boltzmann Machines, 516-518
    deep belief networks (DBNs), 519-521
    evolution of, 254
    Hopfield Networks, 515-516
    hyperparameter fine-tuning, 270-272
    overview, 253-255
    Perceptrons, 257-264
    self-organizing maps, 521-523
    training a DNN with TensorFlow, 265-270
artificial neuron, 256
    (see also artificial neural network (ANN))
assign(), 237
association rule learning, 12
associative memory networks, 515
assumptions, checking, 40
asynchronous updates, 348-349
asynchrous communication, 329-334
atrous_conv2d(), 376
attention mechanism, 409
attributes, 9, 45-48
    (see also data structure)
    combinations of, 58-59
    preprocessed, 48
    target, 48
autodiff, 238-239, 507-513
    forward-mode, 510-512
    manual differentiation, 507
    numerical differentiation, 509
    reverse-mode, 512-513
    symbolic differentiation, 508-509
autoencoders, 411-435

adversarial, 433
contractive, 432
denoising, 424-425
efficient data representations, 412
generative stochastic network (GSN), 433
overcomplete, 424
PCA with undercomplete linear autoencoder, 413
reconstructions, 413
sparse, 426-428
stacked, 415-424
stacked convolutional, 433
undercomplete, 413
variational, 428-432
visualizing features, 421-422
winner-take-all (WTA), 433
automatic differentiating, 231
autonomous driving systems, 379
Average Absolute Deviation, 39
average pooling layer, 364
avg_pool(), 364

## B

backpropagation, 261-262, 275, 291, 422
backpropagation through time (BPTT), 389
bagging and pasting, 185-188
out-of-bag evaluation, 187-188
in Scikit-Learn, 186-187
bandwidth saturation, 349-351
BasicLSTMCell, 401
BasicRNNCell, 397-398
Batch Gradient Descent, 114-117, 130
batch learning, 14-15
Batch Normalization, 282-286, 374
operation summary, 282
with TensorFlow, 284-286
batch(), 341
batch_join(), 341
batch_norm(), 284-285
Bellman Optimality Equation, 455
between-graph replication, 344
bias neurons, 258
bias term, 106
bias/variance tradeoff, 126
biases, 267
binary classifiers, 82, 134
biological neurons, 254-256
black box models, 170
blending, 200-203
Boltzmann Machines, 516-518
(see also restricted Boltzman machines (RBMs))
boosting, 191-200
AdaBoost, 192-195
Gradient Boosting, 195-200
bootstrap aggregation (see bagging)
bootstrapping, 72, 185, 442, 469
bottleneck layers, 369
brew, 202

## C

Caffe model zoo, 291
call__(), 398
CART (Classification and Regression Tree) algorithm, 170-171, 176
categorical attributes, 62-64
cell wrapper, 392
chi square test, 174
classification versus regression, 8, 101
classifiers
binary, 82
error analysis, 96-99
evaluating, 96
MNIST dataset, 79-81
multiclass, 93-96
multilabel, 100-101
multioutput, 101-102
performance measures, 82-93
precision of, 85
voting, 181-184
clip_by_value(), 286
closed-form equation, 105, 128, 136
cluster specification, 324
clustering algorithms, 10
clusters, 323
coding space, 429
codings, 411
complementary slackness condition, 504
components_, 214
computational complexity, 110, 153, 172
compute_gradients(), 286, 449
concat(), 369
config.gpu_options, 318
ConfigProto, 317
confusion matrix, 84-86, 96-99
connectionism, 260
constrained optimization, 158, 503
Contrastive Divergence, 519

control dependencies, 323
conv1d(), 376
conv2d_transpose(), 376
conv3d(), 376
convergence rate, 117
convex function, 113
convolution kernels, 357, 365, 370
convolutional neural networks (CNNs), 353-378
  architectures, 365-376
    AlexNet, 367-368
    GoogleNet, 368-372
    LeNet5, 366-367
    ResNet, 372-375
  convolutional layer, 355-363, 370, 376
    feature maps, 358-360
    filters, 357
    memory requirement, 362-363
  evolution of, 354
  pooling layer, 363-365
  TensorFlow implementation, 360-362
Coordinator class, 338-340
correlation coefficient, 55-58
correlations, finding, 55-58
cost function, 20, 39
  in AdaBoost, 193
  in adagrad, 297
  in artificial neural networks, 264, 267-268
  in autodiff, 238
  in batch normalization, 285
  cross entropy, 367
  deep Q-Learning, 465
  in Elastic Net, 132
  in Gradient Descent, 105, 111-112, 114, 117-119, 200, 275
  in Logistic Regression, 135-136
  in PG algorithms, 449
  in variational autoencoders, 430
  in Lasso Regression, 130-131
  in Linear Regression, 108, 113
  in Momentum optimization, 294-295
  in pretrained layers reuse, 293
  in ridge regression, 127-129
  in RNNs, 389, 393
  stale gradients and, 349
creative sequences, 396
credit assignment problem, 447-448
critics, 463
cross entropy, 140-141, 264, 428, 449
cross-validation, 30, 69-71, 83-84
CUDA library, 315
cuDNN library, 315
curse of dimensionality, 205-207
  (see also dimensionality reduction)
custom transformers, 64-65

## D

data, 30
  (see also test data; training data)
  creating workspace for, 40-43
  downloading, 43-45
  finding correlations in, 55-58
  making assumptions about, 30
  preparing for Machine Learning algorithms, 59-68
  test-set creation, 49-53
  working with real data, 33
data augmentation, 309-310
data cleaning, 60-62
data mining, 6
data parallelism, 347-351
  asynchronous updates, 348-349
  bandwidth saturation, 349-351
  synchronous updates, 348
  TensorFlow implementation, 351
data pipeline, 36
data snooping bias, 49
data structure, 45-48
data visualization, 53-55
DataFrame, 60
dataquest, xvi
decay, 284
decision boundaries, 136-139, 142, 170
decision function, 87, 156-157
Decision Stumps, 195
decision threshold, 87
Decision Trees, 69-70, 167-179, 181
  binary trees, 170
  class probability estimates, 171
  computational complexity, 172
  decision boundaries, 170
  GINI impurity, 172
  instability with, 177-178
  numbers of children, 170
  predictions, 169-171
  Random Forests (see Random Forests)
  regression tasks, 175-176
  regularization hyperparameters, 173-174

training and visualizing, 167-169
decoder, 412
deconvolutional layer, 376
deep autoencoders (see stacked autoencoders)
deep belief networks (DBNs), 13, 519-521
Deep Learning, 437
(see also Reinforcement Learning; TensorFlow)
about, xiii, xvi
libraries, 230-231
deep neural networks (DNNs), 261, 275-312
(see also Multi-Layer Perceptrons (MLP))
faster optimizers for, 293-302
regularization, 302-310
reusing pretrained layers, 286-293
training guidelines overview, 310
training with TensorFlow, 265-270
training with TF.Learn, 264
unstable gradients, 276
vanishing and exploding gradients, 275-286
Deep Q-Learning, 460-469
Ms. Pac Man example, 460-469
deep Q-network, 460
deep RNNs, 396-400
applying dropout, 399
distributing across multiple GPUs, 397
long sequence difficulties, 400
truncated backpropagation through time, 400
DeepMind, 14, 253, 437, 460
degrees of freedom, 27, 126
denoising autoencoders, 424-425
depth concat layer, 369
depth radius, 368
depthwise_conv2d(), 376
dequeue(), 332
dequeue_many(), 332, 334
dequeue_up_to(), 333-334
dequeuing data, 331
describe(), 46
device blocks, 327
device(), 319
dimensionality reduction, 12, 205-225, 411
approaches to
Manifold Learning, 210
projection, 207-209
choosing the right number of dimensions, 215
curse of dimensionality, 205-207
and data visualization, 205
Isomap, 224
LLE (Locally Linear Embedding), 221-223
Multidimensional Scaling, 223-224
PCA (Principal Component Analysis), 211-218
t-Distributed Stochastic Neighbor Embedding (t-SNE), 224
discount rate, 447
distributed computing, 229
distributed sessions, 328-329
DNNClassifier, 264
drop(), 60
dropconnect, 307
dropna(), 60
dropout, 272, 399
dropout rate, 304
dropout(), 306
DropoutWrapper, 399
DRY (Don't Repeat Yourself), 247
Dual Averaging, 300
dual numbers, 510
dual problem, 160
duality, 503
dying ReLUs, 279
dynamic placements, 320
dynamic placer, 318
Dynamic Programming, 456
dynamic unrolling through time, 387
dynamic_rnn(), 387, 398, 409

## E

early stopping, 133-134, 198, 272, 303
Elastic Net, 132
embedded device blocks, 327
Embedded Reber grammars, 410
embeddings, 405-407
embedding_lookup(), 406
encoder, 412
Encoder–Decoder, 383
end-of-sequence (EOS) token, 388
energy functions, 516
enqueuing data, 330
Ensemble Learning, 70, 74, 181-203
bagging and pasting, 185-188
boosting, 191-200
in-graph versus between-graph replication, 343-345
Random Forests, 189-191

(see also Random Forests)
random patches and random subspaces, 188
stacking, 200-202
entropy impurity measure, 172
environments, in reinforcement learning, 438-447, 459, 464
episodes (in RL), 444, 448-449, 451-452, 469
epochs, 118
ε-insensitive, 155
equality contraints, 504
error analysis, 96-99
estimators, 61
Euclidian norm, 39
eval(), 240
evaluating models, 29-31
explained variance, 215
explained variance ratio, 214
exploding gradients, 276
(see also gradients, vanishing and exploding)
exploration policies, 459
exponential decay, 284
exponential linear unit (ELU), 280-281
exponential scheduling, 301
Extra-Trees, 190

## F

F-1 score, 86-87
face-recognition, 100
fake X server, 443
false positive rate (FPR), 91-93
fan-in, 277, 279
fan-out, 277, 279
feature detection, 411
feature engineering, 25
feature extraction, 12
feature importance, 190-191
feature maps, 220, 357-360, 374
feature scaling, 65
feature selection, 26, 74, 130, 191, 499
feature space, 218, 220
feature vector, 39, 107, 156, 237
features, 9
FeatureUnion, 66
feedforward neural network (FNN), 263
feed_dict, 240
FIFOQueue, 330, 333
fillna(), 60
first-in first-out (FIFO) queues, 330
first-order partial derivatives (Jacobians), 300
fit(), 61, 66, 217
fitness function, 20
fit_inverse_transform=, 221
fit_transform(), 61, 66
folds, 69, 81, 83-84
Follow The Regularized Leader (FTRL), 300
forget gate, 402
forward-mode autodiff, 510-512
framing a problem, 35-37
frozen layers, 289-290
fully_connected(), 267, 278, 284-285, 417

## G

game play (see reinforcement learning)
gamma value, 152
gate controllers, 402
Gaussian distribution, 37, 429, 431
Gaussian RBF, 151
Gaussian RBF kernel, 152-153, 163
generalization error, 29
generalized Lagrangian, 504-505
generative autoencoders, 428
generative models, 411, 518
genetic algorithms, 440
geodesic distance, 224
get_variable(), 249-250
GINI impurity, 169, 172
global average pooling, 372
global_step, 466
global_variables(), 308
global_variables_initializer(), 233
Glorot initialization, 276-279
Google, 230
Google Images, 253
Google Photos, 13
GoogleNet architecture, 368-372
gpu_options.per_process_gpu_memory_fraction, 317
gradient ascent, 441
Gradient Boosted Regression Trees (GBRT), 195
Gradient Boosting, 195-200
Gradient Descent (GD), 105, 111-121, 164, 275, 294, 296
algorithm comparisons, 119-121
automatically computing gradients, 238-239
Batch GD, 114-117, 130
defining, 111

local minimum versus global minimum, 112
manually computing gradients, 237
Mini-batch GD, 119-121, 239-241
optimizer, 239
Stochastic GD, 117-119, 148
with TensorFlow, 237-239
Gradient Tree Boosting, 195
GradientDescentOptimizer, 268
gradients(), 238
gradients, vanishing and exploding, 275-286, 400
Batch Normalization, 282-286
Glorot and He initialization, 276-279
gradient clipping, 286
nonsaturating activation functions, 279-281
graphviz, 168
greedy algorithm, 172
grid search, 71-74, 151
group(), 464
GRU (Gated Recurrent Unit) cell, 404-405

## H

hailstone sequence, 412
hard margin classification, 146-147
hard voting classifiers, 181-184
harmonic mean, 86
He initialization, 276-279
Heaviside step function, 257
Hebb's rule, 258, 516
Hebbian learning, 259
hidden layers, 261
hierarchical clustering, 10
hinge loss function, 164
histograms, 47-48
hold-out sets, 200
(see also blenders)
Hopfield Networks, 515-516
hyperbolic tangent (htan activation function), 262, 272, 276, 278, 381
hyperparameters, 28, 65, 72-74, 76, 111, 151, 154, 270
(see also neural network hyperparameters)
hyperplane, 157, 210-211, 213, 224
hypothesis, 39
manifold, 210
hypothesis boosting (see boosting)
hypothesis function, 107
hypothesis, null, 174

## I

identity matrix, 128, 160
ILSVRC ImageNet challenge, 365
image classification, 365
impurity measures, 169, 172
in-graph replication, 343
inception modules, 369
Inception-v4, 375
incremental learning, 16, 217
inequality constraints, 504
inference, 22, 311, 363, 408
info(), 45
information gain, 173
information theory, 172
init node, 241
input gate, 402
input neurons, 258
input_put_keep_prob, 399
instance-based learning, 17, 21
InteractiveSession, 233
intercept term, 106
Internal Covariate Shift problem, 282
inter_op_parallelism_threads, 322
intra_op_parallelism_threads, 322
inverse_transform(), 221
in_top_k(), 268
irreducible error, 127
isolated environment, 41-42
Isomap, 224
is_training, 284-285, 399

## J

jobs, 323
join(), 325, 339
Jupyter, 40, 42, 48

## K

K-fold cross-validation, 69-71, 83
k-Nearest Neighbors, 21, 100
Karush–Kuhn–Tucker (KKT) conditions, 504
keep probability, 306
Keras, 231
Kernel PCA (kPCA), 218-221
kernel trick, 150, 152, 161-164, 218
kernelized SVM, 161-164
kernels, 150-153, 321
Kullback–Leibler divergence, 141, 426

## L


l1_l2_regularizer(), 303
LabelBinarizer, 66
labels, 8, 37
Lagrange function, 504-505
Lagrange multiplier, 503
landmarks, 151-152
large margin classification, 145-146
Lasso Regression, 130-132
latent loss, 430
latent space, 429
law of large numbers, 183
leaky ReLU, 279
learning rate, 16, 111, 115-118
learning rate scheduling, 118, 300-302
LeNet-5 architecture, 355, 366-367
Levenshtein distance, 153
liblinear library, 153
libsvm library, 154
Linear Discriminant Analysis (LDA), 224
linear models
  early stopping, 133-134
  Elastic Net, 132
  Lasso Regression, 130-132
  Linear Regression (see Linear Regression)
  regression (see Linear Regression)
  Ridge Regression, 127-129, 132
  SVM, 145-148
Linear Regression, 20, 68, 105-121, 132
  computational complexity, 110
  Gradient Descent in, 111-121
  learning curves in, 123-127
  Normal Equation, 108-110
  regularizing models (see regularization)
  using Stochastic Gradient Descent (SGD), 119
  with TensorFlow, 235-236
linear SVM classification, 145-148
linear threshold units (LTUs), 257
Lipschitz continuous, 113
LLE (Locally Linear Embedding), 221-223
load_sample_images(), 360
local receptive field, 354
local response normalization, 368
local sessions, 328
location invariance, 363
log loss, 136
logging placements, 320-320
logistic function, 134
Logistic Regression, 9, 134-142
  decision boundaries, 136-139
  estimating probablities, 134-135
  Softmax Regression model, 139-142
  training and cost function, 135-136
log_device_placement, 320
LSTM (Long Short-Term Memory) cell, 401-405


## M


machine control (see reinforcement learning)
Machine Learning
  large-scale projects (see TensorFlow)
  notations, 38-39
  process example, 33-77
  project checklist, 35, 497-502
  resources on, xvi-xvii
  uses for, xiii-xiv
Machine Learning basics
  attributes, 9
  challenges, 22-29
    algorithm problems, 26-28
    training data problems, 25
  definition, 4
  features, 9
  overview, 3
  reasons for using, 4-7
  spam filter example, 4-6
  summary, 28
  testing and validating, 29-31
  types of systems, 7-22
    batch and online learning, 14-17
    instance-based versus model-based learning, 17-22
    supervised/unsupervised learning, 8-14
  workflow example, 18-22
machine translation (see natural language processing (NLP))
make(), 442
Manhattan norm, 39
manifold assumption/hypothesis, 210
Manifold Learning, 210, 221
  (see also LLE (Locally Linear Embedding)
MapReduce, 37
margin violations, 147
Markov chains, 453
Markov decision processes, 453-457
master service, 325
Matplotlib, 40, 48, 91, 97

max margin learning, 293
max pooling layer, 363
max-norm regularization, 307-308
max_norm(), 308
max_norm_regularizer(), 308
max_pool(), 364
Mean Absolute Error (MAE), 39-40
mean coding, 429
Mean Square Error (MSE), 107, 237, 426
measure of similarity, 17
memmap, 217
memory cells, 346, 382
Mercer's theorem, 163
meta learner (see blending)
min-max scaling, 65
Mini-batch Gradient Descent, 119-121, 136, 239-241
mini-batches, 15
minimize(), 286, 289, 449, 466
min_after_dequeue, 333
MNIST dataset, 79-81
model parallelism, 345-347
model parameters, 114, 116, 133, 156, 159, 234, 268, 389
  defining, 19
model selection, 19
model zoos, 291
model-based learning, 18-22
models
  analyzing, 74-75
  evaluating on test set, 75-76
moments, 298
Momentum optimization, 294-295
Monte Carlo tree search, 453
Multi-Layer Perceptrons (MLP), 253, 260-263, 446
  training with TF.Learn, 264
multiclass classifiers, 93-96
Multidimensional Scaling (MDS), 223
multilabel classifiers, 100-101
Multinomial Logistic Regression (see Softmax Regression)
multinomial(), 446
multioutput classifiers, 101-102
MultiRNNCell, 398
multithreaded readers, 338-340
multivariate regression, 37

## N

naive Bayes classifiers, 94
name scopes, 245
natural language processing (NLP), 379, 405-410
  encoder-decoder network for machine translation, 407-410
  TensorFlow tutorials, 405, 408
  word embeddings, 405-407
Nesterov Accelerated Gradient (NAG), 295-296
Nesterov momentum optimization, 295-296
network topology, 270
neural network hyperparameters, 270-272
  activation functions, 272
  neurons per hidden layer, 272
  number of hidden layers, 270-271
neural network policies, 444-447
neurons
  biological, 254-256
  logical computations with, 256
neuron_layer(), 267
next_batch(), 269
No Free Lunch theorem, 30
node edges, 244
nonlinear dimensionality reduction (NLDR), 221
  (see also Kernel PCA; LLE (Locally Linear Embedding))
nonlinear SVM classification, 149-154
  computational complexity, 153
  Gaussian RBF kernel, 152-153
  with polynomial features, 149-150
  polynomial kernel, 150-151
  similarity features, adding, 151-152
nonparametric models, 173
nonresponse bias, 25
nonsaturating activation functions, 279-281
normal distribution (see Gaussian distribution)
Normal Equation, 108-110
normalization, 65
normalized exponential, 139
norms, 39
notations, 38-39
NP-Complete problems, 172
null hypothesis, 174
numerical differentiation, 509
NumPy, 40
NumPy arrays, 63
NVidia Compute Capability, 314

nvidia-smi, 318
n_components, 215

## O

observation space, 446
off-policy algorithm, 459
offline learning, 14
one-hot encoding, 63
one-versus-all (OvA) strategy, 94, 141, 165
one-versus-one (OvO) strategy, 94
online learning, 15-17
online SVMs, 164-165
OpenAI Gym, 441-444
operation_timeout_in_ms, 345
Optical Character Recognition (OCR), 3
optimal state value, 455
optimizers, 293-302
  AdaGrad, 296-298
  Adam optimization, 293, 298-300
  Gradient Descent (see Gradient Descent optimizer)
  learning rate scheduling, 300-302
  Momentum optimization, 294-295
  Nesterov Accelerated Gradient (NAG), 295-296
  RMSProp, 298
out-of-bag evaluation, 187-188
out-of-core learning, 16
out-of-memory (OOM) errors, 386
out-of-sample error, 29
OutOfRangeError, 337, 339
output gate, 402
output layer, 261
OutputProjectionWrapper, 392-395
output_put_keep_prob, 399
overcomplete autoencoder, 424
overfitting, 26-28, 49, 147, 152, 173, 176, 272
  avoiding through regularization, 302-310

## P

p-value, 174
PaddingFIFOQueue, 334
Pandas, 40, 44
  scatter_matrix, 56-57
parallel distributed computing, 313-352
  data parallelism, 347-351
  in-graph versus between-graph replication, 343-345
  model parallelism, 345-347
  multiple devices across multiple servers, 323-342
    asynchronous communication using queues, 329-334
    loading training data, 335-342
    master and worker services, 325
    opening a session, 325
    pinning operations across tasks, 326
    sharding variables, 327
    sharing state across sessions, 328-329
  multiple devices on a single machine, 314-323
    control dependencies, 323
    installation, 314-316
    managing the GPU RAM, 317-318
    parallel execution, 321-322
    placing operations on devices, 318-321
  one neural network per device, 342-343
parameter efficiency, 271
parameter matrix, 139
parameter server (ps), 324
parameter space, 114
parameter vector, 107, 111, 135, 139
parametric models, 173
partial derivative, 114
partial_fit(), 217
Pearson's r, 55
peephole connections, 403
penalties (see rewards, in RL)
percentiles, 46
Perceptron convergence theorem, 259
Perceptrons, 257-264
  versus Logistic Regression, 260
  training, 258-259
performance measures, 37-40
  confusion matrix, 84-86
  cross-validation, 83-84
  precision and recall, 86-90
  ROC (receiver operating characteristic) curve, 91-93
performance scheduling, 301
permutation(), 49
PG algorithms, 448
photo-hosting services, 13
pinning operations, 326
pip, 41
Pipeline constructor, 66-68
pipelines, 36
placeholder nodes, 239

placers (see simple placer; dynamic placer)
policy, 440
policy gradients, 441 (see PG algorithms)
policy space, 440
polynomial features, adding, 149-150
polynomial kernel, 150-151, 162
Polynomial Regression, 106, 121-123
  learning curves in, 123-127
pooling kernel, 363
pooling layer, 363-365
power scheduling, 301
precision, 85
precision and recall, 86-90
  F-1 score, 86-87
  precision/recall (PR) curve, 92
  precision/recall tradeoff, 87-90
predetermined piecewise constant learning rate, 301
predict(), 62
predicted class, 85
predictions, 84-86, 156-157, 169-171
predictors, 8, 62
preloading training data, 335
PReLU (parametric leaky ReLU), 279
preprocessed attributes, 48
pretrained layers reuse, 286-293
  auxiliary task, 292-293
  caching frozen layers, 290
  freezing lower layers, 289
  model zoos, 291
  other frameworks, 288
  TensorFlow model, 287-288
  unsupervised pretraining, 291-292
  upper layers, 290
Pretty Tensor, 231
primal problem, 160
principal component, 212
Principal Component Analysis (PCA), 211-218
  explained variance ratios, 214
  finding principal components, 212-213
  for compression, 216-217
  Incremental PCA, 217-218
  Kernel PCA (kPCA), 218-221
  projecting down to d dimensions, 213
  Randomized PCA, 218
  Scikit Learn for, 214
  variance, preserving, 211-212
probabilistic autoencoders, 428
probabilities, estimating, 134-135, 171
producer functions, 341
projection, 207-209
propositional logic, 254
pruning, 174, 509
Python
  isolated environment in, 41-42
  notebooks in, 42-43
  pickle, 71
  pip, 41

## Q

Q-Learning algorithm, 458-469
  approximate Q-Learning, 460
  deep Q-Learning, 460-469
Q-Value Iteration Algorithm, 456
Q-Values, 456
Quadratic Programming (QP) Problems, 159-160
quantizing, 351
queries per second (QPS), 343
QueueRunner, 338-340
queues, 329-334
  closing, 333
  dequeuing data, 331
  enqueuing data, 330
  first-in first-out (FIFO), 330
  of tuples, 332
  PaddingFIFOQueue, 334
  RandomShuffleQueue, 333
q_network(), 463

## R

Radial Basis Function (RBF), 151
Random Forests, 70-72, 94, 167, 178, 181, 189-191
  Extra-Trees, 190
  feature importance, 190-191
random initialization, 111, 116, 118, 276
Random Patches and Random Subspaces, 188
randomized leaky ReLU (RReLU), 279
Randomized PCA, 218
randomized search, 74, 270
RandomShuffleQueue, 333, 337
random_uniform(), 237
reader operations, 335
recall, 85
recognition network, 412
reconstruction error, 216
reconstruction loss, 413, 428, 430

reconstruction pre-image, 220
reconstructions, 413
recurrent neural networks (RNNs), 379-410
  deep RNNs, 396-400
  exploration policies, 459
  GRU cell, 404-405
  input and output sequences, 382-383
  LSTM cell, 401-405
  natural language processing (NLP), 405-410
  in TensorFlow, 384-388
    dynamic unrolling through time, 387
    static unrolling through time, 385-386
    variable length input sequences, 387
    variable length output sequences, 388
  training, 389-396
    backpropagation through time (BPTT), 389
    creative sequences, 396
    sequence classifiers, 389-391
    time series predictions, 392-396
recurrent neurons, 380-383
  memory cells, 382
reduce_mean(), 268
reduce_sum(), 427-428, 430, 466
regression, 8
  Decision Trees, 175-176
regression models
  linear, 68
regression versus classification, 101
regularization, 27-28, 30, 127-134
  data augmentation, 309-310
  Decision Trees, 173-174
  dropout, 304-307
  early stopping, 133-134, 303
  Elastic Net, 132
  Lasso Regression, 130-132
  max-norm, 307-308
  Ridge Regression, 127-129
  shrinkage, 197
  ℓ 1 and ℓ 2 regularization, 303-304
REINFORCE algorithms, 448
Reinforcement Learning (RL), 13-14, 437-470
  actions, 447-448
  credit assignment problem, 447-448
  discount rate, 447
  examples of, 438
  Markov decision processes, 453-457
  neural network policies, 444-447
  OpenAI gym, 441-444
  PG algorithms, 448-453
  policy search, 440-441
  Q-Learning algorithm, 458-469
  rewards, learning to optimize, 438-439
  Temporal Difference (TD) Learning, 457-458
ReLU (rectified linear units), 246-248
ReLU activation, 374
ReLU function, 262, 272, 278-281
relu(z), 266
render(), 442
replay memory, 464
replica_device_setter(), 327
request_stop(), 339
reset(), 442
reset_default_graph(), 234
reshape(), 395
residual errors, 195-196
residual learning, 372
residual network (ResNet), 291, 372-375
residual units, 373
ResNet, 372-375
resource containers, 328-329
restore(), 241
restricted Boltzmann machines (RBMs), 13, 291, 518
reuse_variables(), 249
reverse-mode autodiff, 512-513
rewards, in RL, 438-439
rgb_array, 443
Ridge Regression, 127-129, 132
RMSProp, 298
ROC (receiver operating characteristic) curve, 91-93
Root Mean Square Error (RMSE), 37-40, 107
RReLU (randomized leaky ReLU), 279
run(), 233, 345

## S

Sampled Softmax, 409
sampling bias, 24-25, 51
sampling noise, 24
save(), 241
Saver node, 241
Scikit Flow, 231
Scikit-Learn, 40
  about, xiv
  bagging and pasting in, 186-187
  CART algorithm, 170-171, 176

cross-validation, 69-71
design principles, 61-62
imputer, 60-62
LinearSVR class, 156
MinMaxScaler, 65
min_ and max_ hyperparameters, 173
PCA implementation, 214
Perceptron class, 259
Pipeline constructor, 66-68, 149
Randomized PCA, 218
Ridge Regression with, 129
SAMME, 195
SGDClassifier, 82, 87-88, 94
SGDRegressor, 119
sklearn.base.BaseEstimator, 64, 67, 84
sklearn.base.clone(), 83, 133
sklearn.base.TransformerMixin, 64, 67
sklearn.datasets.fetch_california_housing(), 236
sklearn.datasets.fetch_mldata(), 79
sklearn.datasets.load_iris(), 137, 148, 167, 190, 259
sklearn.datasets.load_sample_images(), 360-361
sklearn.datasets.make_moons(), 149, 178
sklearn.decomposition.IncrementalPCA, 217
sklearn.decomposition.KernelPCA, 218-219, 221
sklearn.decomposition.PCA, 214
sklearn.ensemble.AdaBoostClassifier, 195
sklearn.ensemble.BaggingClassifier, 186-189
sklearn.ensemble.GradientBoostingRegressor, 196, 198-199
sklearn.ensemble.RandomForestClassifier, 92, 95, 184
sklearn.ensemble.RandomForestRegressor, 70, 72-74, 189-190, 196
sklearn.ensemble.VotingClassifier, 184
sklearn.externals.joblib, 71
sklearn.linear_model.ElasticNet, 132
sklearn.linear_model.Lasso, 132
sklearn.linear_model.LinearRegression, 20-21, 62, 68, 110, 120, 122, 124-125
sklearn.linear_model.LogisticRegression, 137, 139, 141, 184, 219
sklearn.linear_model.Perceptron, 259
sklearn.linear_model.Ridge, 129
sklearn.linear_model.SGDClassifier, 82
sklearn.linear_model.SGDRegressor, 119-120, 129, 132-133
sklearn.manifold.LocallyLinearEmbedding, 221-222
sklearn.metrics.accuracy_score(), 184, 188, 264
sklearn.metrics.confusion_matrix(), 85, 96
sklearn.metrics.f1_score(), 87, 100
sklearn.metrics.mean_squared_error(), 68-69, 76, 124, 133, 198-199, 221
sklearn.metrics.precision_recall_curve(), 88
sklearn.metrics.precision_score(), 86, 90
sklearn.metrics.recall_score(), 86, 90
sklearn.metrics.roc_auc_score(), 92-93
sklearn.metrics.roc_curve(), 91-92
sklearn.model_selection.cross_val_predict(), 84, 88, 92, 96, 100
sklearn.model_selection.cross_val_score(), 69-70, 83-84
sklearn.model_selection.GridSearchCV, 72-74, 77, 96, 179, 219
sklearn.model_selection.StratifiedKFold, 83
sklearn.model_selection.StratifiedShuffleSplit, 52
sklearn.model_selection.train_test_split(), 50, 69, 124, 178, 198
sklearn.multiclass.OneVsOneClassifier, 95
sklearn.neighbors.KNeighborsClassifier, 100, 102
sklearn.neighbors.KNeighborsRegressor, 22
sklearn.pipeline.FeatureUnion, 66
sklearn.pipeline.Pipeline, 66, 125, 148-149, 219
sklearn.preprocessing.Imputer, 60, 66
sklearn.preprocessing.LabelBinarizer, 64, 66
sklearn.preprocessing.LabelEncoder, 62
sklearn.preprocessing.OneHotEncoder, 63
sklearn.preprocessing.PolynomialFeatures, 122-123, 125, 128, 149
sklearn.preprocessing.StandardScaler, 65-66, 96, 114, 128, 146, 148-150, 152, 237, 264
sklearn.svm.LinearSVC, 147-149, 153-154, 156, 165
sklearn.svm.LinearSVR, 155-156
sklearn.svm.SVC, 148, 150, 152-154, 156, 165, 184
sklearn.svm.SVR, 77, 156

sklearn.tree.DecisionTreeClassifier, 173, 179, 186-187, 189, 195
sklearn.tree.DecisionTreeRegressor, 69, 167, 175, 195-196
sklearn.tree.export_graphviz(), 168
StandardScaler, 114, 237, 264
SVM classification classes, 154
TF.Learn, 231
user guide, xvi
score(), 62
search space, 74, 270
second-order partial derivatives (Hessians), 300
self-organizing maps (SOMs), 521-523
semantic hashing, 434
semisupervised learning, 13
sensitivity, 85, 91
sentiment analysis, 379
separable_conv2d(), 376
sequences, 379
sequence_length, 387-388, 409
Shannon's information theory, 172
shortcut connections, 372
show(), 48
show_graph(), 245
shrinkage, 197
shuffle_batch(), 341
shuffle_batch_join(), 341
sigmoid function, 134
sigmoid_cross_entropy_with_logits(), 428
similarity function, 151-152
simulated annealing, 118
simulated environments, 442
(see also OpenAI Gym)
Singular Value Decomposition (SVD), 213
skewed datasets, 84
skip connections, 310, 372
slack variable, 158
smoothing terms, 283, 297, 299, 430
soft margin classification, 146-148
soft placements, 321
soft voting, 184
softmax function, 139, 263, 264
Softmax Regression, 139-142
source ops, 236, 322
spam filters, 3-6, 8
sparse autoencoders, 426-428
sparse matrix, 63
sparse models, 130, 300
sparse_softmax_cross_entropy_with_logits(), 268
sparsity loss, 426
specificity, 91
speech recognition, 6
spurious patterns, 516
stack(), 385
stacked autoencoders, 415-424
TensorFlow implementation, 416
training one-at-a-time, 418-420
tying weights, 417-418
unsupervised pretraining with, 422-424
visualizing the reconstructions, 420-421
stacked denoising autoencoders, 422, 424
stacked denoising encoders, 424
stacked generalization (see stacking)
stacking, 200-202
stale gradients, 348
standard correlation coefficient, 55
standard deviation, 37
standardization, 65
StandardScaler, 66, 237, 264
state-action values, 456
states tensor, 388
state_is_tuple, 398, 401
static unrolling through time, 385-386
static_rnn(), 385-386, 409
stationary point, 503-505
statistical mode, 185
statistical significance, 174
stemming, 103
step functions, 257
step(), 443
Stochastic Gradient Boosting, 199
Stochastic Gradient Descent (SGD), 117-119, 148, 260
training, 136
Stochastic Gradient Descent (SGD) classifier, 82, 129
stochastic neurons, 516
stochastic policy, 440
stratified sampling, 51-53, 83
stride, 357
string kernels, 153
string_input_producer(), 341
strong learners, 182
subderivatives, 164
subgradient vector, 131
subsample, 199, 363

supervised learning, 8-9
Support Vector Machines (SVMs), 94, 145-166
  decision function and predictions, 156-157
  dual problem, 503-505
  kernelized SVM, 161-164
  linear classification, 145-148
  mechanics of, 156-165
  nonlinear classification, 149-154
  online SVMs, 164-165
  Quadratic Programming (QP) problems, 159-160
  SVM regression, 154-165
  the dual problem, 160
  training objective, 157-159
support vectors, 146
svd(), 213
symbolic differentiation, 238, 508-509
synchronous updates, 348

## T

t-Distributed Stochastic Neighbor Embedding (t-SNE), 224
tail heavy, 48
target attributes, 48
target_weights, 409
tasks, 323
Temporal Difference (TD) Learning, 457-458
tensor processing units (TPUs), 315
TensorBoard, 231
TensorFlow, 229-252
  about, xiv
  autodiff, 238-239, 507-513
  Batch Normalization with, 284-286
  construction phase, 234
  control dependencies, 323
  convenience functions, 341
  convolutional layers, 376
  convolutional neural networks and, 360-362
  data parallelism and, 351
  denoising autoencoders, 425-425
  dropout with, 306
  dynamic placer, 318
  execution phase, 234
  feeding data to the training algorithm, 239-241
  Gradient Descent with, 237-239
  graphs, managing, 234
  initial graph creation and session run, 232-234
  installation, 232
  l1 and l2 regularization with, 303
  learning schedules in, 302
  Linear Regression with, 235-236
  max pooling layer in, 364
  max-norm regularization with, 307
  model zoo, 291
  modularity, 246-248
  Momentum optimization in, 295
  name scopes, 245
  neural network policies, 446
  NLP tutorials, 405, 408
  node value lifecycle, 235
  operations (ops), 235
  optimizer, 239
  overview, 229-231
  parallel distributed computing (see parallel distributed computing with TensorFlow)
  Python API
    construction, 265-269
    execution, 269
    using the neural network, 270
  queues (see queues)
  reusing pretrained layers, 287-288
  RNNs in, 384-388
    (see also recurrent neural networks (RNNs))
  saving and restoring models, 241-242
  sharing variables, 248-251
  simple placer, 318
  sklearn.metrics.accuracy_score(), 286
  sparse autoencoders with, 427
  and stacked autoencoders, 416
  TensorBoard, 242-245
  tf.abs(), 303
  tf.add(), 246, 303-304
  tf.add_n(), 247-248, 250-251
  tf.add_to_collection(), 308
  tf.assign(), 237, 288, 307-308, 482
  tf.bfloat16, 350
  tf.bool, 284, 306
  tf.cast(), 268, 391
  tf.clip_by_norm(), 307-308
  tf.clip_by_value(), 286
  tf.concat(), 312, 369, 446, 450
  tf.ConfigProto, 317, 320-321, 345, 487
  tf.constant(), 235-237, 319-320, 323, 325-326
  tf.constant_initializer(), 249-251

tf.container(), 328-330, 351-352, 481
tf.contrib.framework.arg_scope(), 285, 416, 430
tf.contrib.layers.batch_norm(), 284-285
tf.contrib.layers.convolution2d(), 463
tf.contrib.layers.fully_connected(), 267
tf.contrib.layers.l1_regularizer(), 303, 308
tf.contrib.layers.l2_regularizer(), 303, 416-417
tf.contrib.layers.variance_scaling_initializer(), 278-279, 391, 416-417, 430, 446, 450, 463
tf.contrib.learn.DNNClassifier, 264
tf.contrib.learn.infer_real_valued_columns_from_input(), 264
tf.contrib.rnn.BasicLSTMCell, 401, 403
tf.contrib.rnn.BasicRNNCell, 385-387, 390, 392-393, 395, 397-399, 401
tf.contrib.rnn.DropoutWrapper, 399
tf.contrib.rnn.GRUCell, 405
tf.contrib.rnn.LSTMCell, 403
tf.contrib.rnn.MultiRNNCell, 397-399
tf.contrib.rnn.OutputProjectionWrapper, 392-394
tf.contrib.rnn.RNNCell, 398
tf.contrib.rnn.static_rnn(), 385-387, 409-410, 491-492
tf.contrib.slim module, 231, 377
tf.contrib.slim.nets module (nets), 377
tf.control_dependencies(), 323
tf.decode_csv(), 336, 340
tf.device(), 319-321, 326-327, 397-398
tf.exp(), 430-431
tf.FIFOQueue, 330, 332-333, 336, 340
tf.float32, 236, 482
tf.get_collection(), 288-289, 304, 308, 416, 463
tf.get_default_graph(), 234, 242
tf.get_default_session(), 233
tf.get_variable(), 249-251, 288, 303-308
tf.global_variables(), 308
tf.global_variables_initializer(), 233, 237
tf.gradients(), 238
tf.Graph, 232, 234, 242, 335, 343
tf.GraphKeys.REGULARIZATION_LOSSES, 304, 416
tf.GraphKeys.TRAINABLE_VARIABLES, 288-289, 463
tf.group(), 464
tf.int32, 321-332, 337, 387, 390, 406, 466
tf.int64, 265
tf.InteractiveSession, 233
TF.Learn, 264
tf.log(), 427, 430, 446, 450
tf.matmul(), 236-237, 246, 265, 384, 417, 420, 425, 427-428
tf.matrix_inverse(), 236
tf.maximum(), 246, 248-251, 281
tf.multinomial(), 446, 450
tf.name_scope(), 245, 248-249, 265, 267-268, 419-420
tf.nn.conv2d(), 360-361
tf.nn.dynamic_rnn(), 386-387, 390, 392, 395, 397-399, 409-410, 491-492
tf.nn.elu(), 281, 416-417, 430, 446, 450
tf.nn.embedding_lookup(), 406
tf.nn.in_top_k(), 268, 391
tf.nn.max_pool(), 364-365
tf.nn.relu(), 265, 392-393, 395, 463
tf.nn.sigmoid_cross_entropy_with_logits(), 428, 431, 449-450
tf.nn.sparse_softmax_cross_entropy_with_logits(), 267-268, 390
tf.one_hot(), 466
tf.PaddingFIFOQueue, 334
tf.placeholder(), 239-240, 482
tf.placeholder_with_default(), 425
tf.RandomShuffleQueue, 333, 337-338, 340-341
tf.random_normal(), 246, 384, 425, 430
tf.random_uniform(), 237, 241, 406, 482
tf.reduce_mean(), 237, 245, 267-268, 303, 390-391, 414, 416, 418, 420, 425, 427, 466
tf.reduce_sum(), 303, 427-428, 430-431, 465-466
tf.reset_default_graph(), 234
tf.reshape(), 395, 463
tf.RunOptions, 345
tf.Session, 233, 482
tf.shape(), 425, 430
tf.square(), 237, 245, 393, 414, 416, 418, 420, 425, 427, 430-431, 466
tf.stack(), 336, 340, 386
tf.string, 336, 340
tf.summary.FileWriter, 242-243
tf.summary.scalar(), 242

tf.tanh(), 384
tf.TextLineReader, 336, 340
tf.to_float(), 449-450
tf.train.AdamOptimizer, 293, 299, 390, 393, 414, 416-417, 419, 427, 431, 449-450, 466
tf.train.ClusterSpec, 324
tf.train.Coordinator, 338-340
tf.train.exponential_decay(), 302
tf.train.GradientDescentOptimizer, 239, 268, 286, 293, 295
tf.train.MomentumOptimizer, 239, 295-296, 302, 311, 351, 485-486
tf.train.QueueRunner, 338-341
tf.train.replica_device_setter(), 327-328
tf.train.RMSPropOptimizer, 298
tf.train.Saver, 241-242, 268, 377, 399, 450, 466
tf.train.Server, 324
tf.train.start_queue_runners(), 341
tf.transpose(), 236-237, 386, 417
tf.truncated_normal(), 265
tf.unstack(), 385-387, 395, 492
tf.Variable, 232, 482
tf.variable_scope(), 249-251, 288, 307-308, 328, 391, 463
tf.zeros(), 265, 384, 417
truncated backpropagation through time, 400
visualizing graph and training curves, 242-245
TensorFlow Serving, 343
tensorflow.contrib, 267
test set, 29, 49-53, 81
testing and validating, 29-31
text attributes, 62-64
TextLineReader, 336
TF-slim, 231
TF.Learn, 231, 264
thermal equilibrium, 518
thread pools (inter-op/intra-op, in TensorFlow, 322
threshold variable, 248-251
Tikhonov regularization, 127
time series data, 379
toarray(), 63
tolerance hyperparameter, 154
trainable, 288
training data, 4
insufficient quantities, 22
irrelevant features, 25
loading, 335-342
nonrepresentative, 24
overfitting, 26-28
poor quality, 25
underfitting, 28
training instance, 4
training models, 20, 105-143
learning curves in, 123-127
Linear Regression, 105, 106-121
Logistic Regression, 134-142
overview, 105-106
Polynomial Regression, 106, 121-123
training objectives, 157-159
training set, 4, 29, 53, 60, 68-69
cost function of, 135-136
shuffling, 81
transfer learning, 286-293
(see also pretrained layers reuse)
transform(), 61, 66
transformation pipelines, 66-68
transformers, 61
transformers, custom, 64-65
transpose(), 385
true negative rate (TNR), 91
true positive rate (TPR), 85, 91
truncated backpropagation through time, 400
tuples, 332
tying weights, 417


## U


underfitting, 28, 68, 152
univariate regression, 37
unstack(), 385
unsupervised learning, 10-12
anomaly detection, 12
association rule learning, 10, 12
clustering, 10
dimensionality reduction algorithm, 12
visualization algorithms, 11
unsupervised pretraining, 291-292, 422-424
upsampling, 376
utility function, 20


## V


validation set, 30
Value Iteration, 455
value_counts(), 46
vanishing gradients, 276

(see also gradients, vanishing and exploding)
variables, sharing, 248-251
variable_scope(), 249-250
variance
bias/variance tradeoff, 126
variance preservation, 211-212
variance_scaling_initializer(), 278
variational autoencoders, 428-432
VGGNet, 375
visual cortex, 354
visualization, 242-245
visualization algorithms, 11-12
voice recognition, 353
voting classifiers, 181-184

## W

warmup phase, 349
weak learners, 182
weight-tying, 417
weights, 267, 288
freezing, 289
while_loop(), 387
white box models, 170
worker, 324
worker service, 325
worker_device, 327
workspace directory, 40-43

## X

Xavier initialization, 276-279

## Y

YouTube, 253

## Z

zero padding, 356, 361

## About the Author

**Aurélien Géron** is a Machine Learning consultant. A former Googler, he led the YouTube video classification team from 2013 to 2016. He was also a founder and CTO of Wifirst from 2002 to 2012, a leading Wireless ISP in France; and a founder and CTO of Polyconseil in 2001, the firm that now manages the electric car sharing service Autolib'.

Before this he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He published a few technical books (on C++, WiFi, and internet architectures), and was a Computer Science lecturer in a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

## Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn and TensorFlow* is the far eastern fire salamander (*Salamandra infraimmaculata*), an amphibian found in the Middle East. They have black skin featuring large yellow spots on their back and head. These spots are a warning coloration meant to keep predators at bay. Full-grown salamanders can be over a foot in length.

Far eastern fire salamanders live in subtropical shrubland and forests near rivers or other freshwater bodies. They spend most of their life on land, but lay their eggs in the water. They subsist mostly on a diet of insects, worms, and small crustaceans, but occasionally eat other salamanders. Males of the species have been known to live up to 23 years, while females can live up to 21 years.

Although not yet endangered, the far eastern fire salamander population is in decline. Primary threats include damming of rivers (which disrupts the salamander's breeding) and pollution. They are also threatened by the recent introduction of predatory fish, such as the mosquitofish. These fish were intended to control the mosquito population, but they also feed on young salamanders.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.