

# Hands on ML, 2nd edition, learning notes

Sibelius Peng

October 23, 2019

# Contents

<b>I</b>	<b>The Fundamentals of Machine Learning</b>	<b>10</b>
<b>1</b>	<b>The Machine Learning Landscape</b>	<b>11</b>
1.1	What Is ML? . . . . .	11
1.2	Why Use ML? . . . . .	11
1.3	Types . . . . .	11
1.4	Main Challenges . . . . .	12
1.5	Testing and Validating . . . . .	13
<b>2</b>	<b>End-to-End Machine Learning Project</b>	<b>14</b>
<b>3</b>	<b>Classification</b>	<b>16</b>
3.1	MNIST . . . . .	16
3.2	Training a Binary Classifier . . . . .	16
3.3	Performance Measures . . . . .	16
3.4	Multiclass Classification . . . . .	16
3.5	Error Analysis . . . . .	16
3.6	Multilabel Classification . . . . .	16
3.7	Multioutput Classification . . . . .	17

<b>4</b>	<b>Training Models</b>	<b>18</b>
4.1	Linear Regression . . . . .	18
4.2	The Normal Equation . . . . .	19
4.3	Gradient Descent . . . . .	19
4.3.1	Batch Gradient Descent . . . . .	19
4.3.2	Stochastic Gradient Descent . . . . .	20
4.3.3	Mini-batch Gradient Descent . . . . .	20
4.4	Polynomial Regression . . . . .	21
4.5	Learning Curves . . . . .	21
4.6	Regularized Linear Models . . . . .	21
4.6.1	Ridge Regression . . . . .	21
4.6.2	Lasso Regression . . . . .	21
4.6.3	Elastic Net . . . . .	22
4.6.4	Early Stopping . . . . .	22
4.7	Logistic Regression . . . . .	23
4.7.1	Estimating Probabilities . . . . .	23
4.7.2	Training and Cost Function . . . . .	23
4.8	Softmax Regression . . . . .	23
<b>5</b>	<b>Support Vector Machines</b>	<b>25</b>
5.1	Linear SVM classification . . . . .	25
5.1.1	Soft Margin Classification . . . . .	25
5.2	Nonlinear SVM classification . . . . .	25
5.2.1	Polynomial Kernel . . . . .	25

5.2.2	Adding Similarity Features . . . . .	25
5.2.3	Gaussian RBF Kernel . . . . .	26
5.3	SVM Regression . . . . .	26
5.4	Under the Hood . . . . .	26
5.4.1	Decision Function and Predictions . . . . .	26
5.4.2	Quadratic Programming . . . . .	27
5.4.3	The Dual Problem . . . . .	27
5.4.4	Kernelized SVM . . . . .	27
<b>6</b>	<b>Decision Trees</b>	<b>28</b>
6.1	Training and Visualizing a Decision Tree . . . . .	28
6.2	Making Predictions . . . . .	28
6.3	The CART training alg . . . . .	28
6.4	Gini Impurity or Entropy? . . . . .	29
6.5	Regularization Hyperparameters . . . . .	29
6.6	Regression . . . . .	29
6.7	Instability . . . . .	29
<b>7</b>	<b>Ensemble Learning and Random Forests</b>	<b>30</b>
7.1	Voting Classifiers . . . . .	30
7.2	Bagging and Pasting . . . . .	30
7.2.1	Bagging and Pasting in Scikit-Learn . . . . .	30
7.2.2	Out-of-Bag Evaluation . . . . .	30
7.3	Random Patches and Random Subspaces . . . . .	31
7.4	Random Forests . . . . .	31

7.4.1	Extra-Trees . . . . .	31
7.5	Boosting . . . . .	31
7.5.1	AdaBoost . . . . .	31
7.5.2	Gradient Boosting . . . . .	32
7.6	Stacking . . . . .	32
<b>8</b>	<b>Dimension Reduction</b>	<b>33</b>
8.1	The Curse of Dimensionality . . . . .	33
8.2	Main Approaches . . . . .	33
8.2.1	Projection . . . . .	33
8.2.2	Manifold Learning . . . . .	33
8.3	PCA . . . . .	33
8.3.1	Explained Variance Ratio . . . . .	33
8.3.2	Choosing the Right Number of Dimensions . . . . .	34
8.4	Kernel PCA . . . . .	34
8.5	LLE . . . . .	34
8.6	Other Techniques . . . . .	35
<b>9</b>	<b>Unsupervised Learning Techniques</b>	<b>36</b>
9.1	Clustering . . . . .	36
9.1.1	KMeans . . . . .	37
9.1.2	DBSCAN . . . . .	37
9.1.3	Other Clustering Alg . . . . .	37
9.2	Gaussian Mixtures . . . . .	37
9.2.1	Anomaly Detection using Gaussian Mixtures . . . . .	37

9.2.2	Selecting the Number of Clusters . . . . .	38
9.2.3	Bayesian Gaussian Mixture Models . . . . .	38
9.2.4	Other Anomaly Detection and Novelty Detection Algorithms . . . . .	38
<b>II</b>	<b>Neural Networks and Deep Learning</b>	<b>39</b>
<b>10</b>	<b>Introduction to Artificial Neural Networks with Keras</b>	<b>40</b>
10.1	From Biological to Artificial Neurons . . . . .	40
10.1.1	Biological Neurons . . . . .	40
10.1.2	Logical Computations with Neurons . . . . .	40
10.1.3	The Perceptron . . . . .	41
10.1.4	Multi-Layer Perceptron and Backpropagation . . . . .	41
10.1.5	Regression MLPs . . . . .	42
10.1.6	Classification MLPs . . . . .	42
10.2	Implementing MLPs with Keras . . . . .	42
10.2.1	Image Classifier with Sequential API . . . . .	42
10.2.2	Regression with Sequential . . . . .	45
10.2.3	Building Complex Models Using the Functional API . . . . .	45
10.2.4	Building Dynamic Models Using the Subclassing API . . . . .	47
10.2.5	Saving and Restoring a Model . . . . .	47
10.2.6	Using Callbacks . . . . .	48
10.2.7	Visualization Using TensorBoard . . . . .	48
10.3	Fine-Tuning Neural Networks Hyperparameters . . . . .	48
<b>11</b>	<b>Training Deep Neural Networks</b>	<b>50</b>

11.1	Vanishing/Exploding Gradients Problems . . . . .	50
11.1.1	Glorot and He Initialization . . . . .	51
11.1.2	Nonsaturating Activation Functions . . . . .	51
11.1.3	Batch Normalization . . . . .	51
11.1.4	Gradient Clipping . . . . .	51
11.2	Reusing Pretrained Layers . . . . .	52
11.2.1	Transfer Learning With Keras . . . . .	52
11.2.2	Unsupervised Pretraining . . . . .	53
11.2.3	Pretraining on an Auxiliary Task . . . . .	53
11.3	Faster Optimizers . . . . .	53
11.3.1	Momentum Optimization . . . . .	54
11.3.2	Nesterov Accelerated Gradient . . . . .	54
11.3.3	AdaGrad . . . . .	54
11.3.4	RMSProp . . . . .	55
11.3.5	Adam and Nadam Optimization . . . . .	55
11.3.6	Learning Rate Scheduling . . . . .	56
11.4	Avoiding Overfitting Through Regularization . . . . .	57
11.4.1	$\ell_1$ and $\ell_2$ Regularization . . . . .	57
11.4.2	Dropout . . . . .	57
11.4.3	Monte-Carlo (MC) Dropout . . . . .	58
11.4.4	Max-Norm Regularization . . . . .	58
11.5	Summary and Practical Guidelines . . . . .	59

## 12 Custom Models and Training with TensorFlow 60

12.1 A Quick Tour of TensorFlow . . . . .	60
12.2 Using TensorFlow like NumPy . . . . .	60
12.2.1 Tensors and Operations . . . . .	60
12.3 Customizing Models and Training Algorithms . . . . .	60
12.4 TensorFlow Functions and Graphs . . . . .	61
<b>13 Loading and Preprocessing Data with TensorFlow</b>	<b>62</b>
13.1 The Data API . . . . .	62
13.1.1 Shuffling the Data . . . . .	62
13.1.2 Preprocessing . . . . .	63
13.1.3 Putting Everything Together . . . . .	63
13.1.4 Prefetching . . . . .	63
13.1.5 Using the Dataset With tf.keras . . . . .	63
13.2 The TFRecord Format . . . . .	64
13.3 The Features API . . . . .	64
13.4 TF Transform . . . . .	64
13.5 The TensorFlow Datasets (TFDS) Project . . . . .	65
<b>14 Deep Computer Vision Using Convolutional Neural Networks</b>	<b>66</b>
14.1 Convolutional Layers . . . . .	66
<b>15 Processing Sequences Using RNNs and CNNs</b>	<b>67</b>
<b>16 Natural Language Processing with RNNs and Attention</b>	<b>68</b>
<b>Dimension Reduction - Autoencoders</b>	<b>69</b>
What are autoencoders used for? . . . . .	69



Types . . . . .	69
<b>17 Representation Learning and Generative Learning Using Autoencoders and GANs</b>	<b>70</b>
17.1 Autoencoders . . . . .	70
17.1.1 Performing PCA with an Undercomplete Linear Autoencoder . . . . .	70
17.1.2 Implementing a Stacked Autoencoder Using Keras . . . . .	70
17.1.3 Convolutional Autoencoders . . . . .	71
17.1.4 Recurrent Autoencoders . . . . .	71
17.1.5 Denoising Autoencoders . . . . .	72
17.1.6 Sparse Autoencoders . . . . .	72
17.1.7 Variational Autoencoders . . . . .	72
17.2 Generative Adversarial Networks . . . . .	72
17.2.1 The Difficulties of Training GANs . . . . .	72
17.2.2 Progressive Growing of GANs . . . . .	73
17.2.3 StyleGANs . . . . .	73
<b>18 Reinforcement Learning</b>	<b>74</b>
18.1 Learning to Optimize Rewards . . . . .	74
18.2 Introduction to OpenAI Gym . . . . .	74
18.3 Overview of Some Popular RL Algorithms . . . . .	74
<b>19 Training and Deploying TensorFlow Models at Scale</b>	<b>75</b>
19.1 Serving a TensorFlow Model . . . . .	75
19.2 Deploying a Model to a Mobile or Embedded Device . . . . .	75
19.3 Using GPUs to Speed Up Computations . . . . .	75
19.4 Training Models Across Multiple Devices . . . . .	75

<b>Appendices</b>	<b>76</b>
<b>A Machine Learning Project Checklist</b>	<b>77</b>

## Part I

# The Fundamentals of Machine Learning

# 1 | The Machine Learning Landscape

## 1.1 What Is ML?

Machine Learning is the science (and art) of programming computers so they can learn from data.

## 1.2 Why Use ML?

- Problems for which existing solutions require a lot of fine-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better than the traditional approach.
- Complex problems for which using a traditional approach yields no good solution: the best Machine Learning techniques can perhaps find a solution.
- Fluctuating environments: a Machine Learning system can adapt to new data.
- Getting insights about complex problems and large amounts of data.

## 1.3 Types

Supervised, unsupervised, semi-, reinforcement

Batch and Online Learning:

- In batch learning, the system is incapable of learning incrementally: it must be trained using all the available data.
- In online learning, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called mini-batches.

## Instance-Based Versus Model-Based Learning

- This is called instance-based learning: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them).
- Another way to generalize from a set of examples is to build a model of these examples and then use that model to make predictions. This is called model-based learning.

☞ Confusingly, the same word “model” can refer to a type of model (e.g., Linear Regression), to a fully specified model architecture (e.g., Linear Regression with one input and one output), or to the final trained model ready to be used for predictions (e.g., Linear Regression with one input and one output, using  $\theta_0 = 4.85$  and  $\theta_1 = 4.91 \times 10^{-5}$ ). Model selection consists in choosing the type of model and fully specifying its architecture. Training a model means running an algorithm to find the model parameters that will make it best fit the training data (and hopefully make good predictions on new data).

In summary:

- You studied the data.
- You selected a model.
- You trained it on the training data (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you applied the model to make predictions on new cases (this is called inference), hoping that this model will generalize well.

## 1.4 Main Challenges

Insufficient Quantity of Training Data

Nonrepresentative Training Data

Poor-Quality Data

Irrelevant Features

Overfitting/Underfitting the Training Data

## 1.5 Testing and Validating

Hyperparameter Tuning and Model Selection

Data Mismatch

## 2 | End-to-End Machine Learning Project

Here are the main steps you will go through:

1. Look at the big picture.
  - Frame the problem
  - Select a Performance measure
  - Check the assumptions
2. Get the data.
  - create the workspace
  - download the data
  - take a quick look at the data structure
3. Discover and visualize the data to gain insights.
  - visualizing geographical data
  - correlations
  - experimenting with attribute combinations
4. Prepare the data for Machine Learning algorithms.
  - data cleaning
  - handling text and categorical attributes
  - custom transformers
  - feature scaling
  - transformation pipelines
5. Select a model and train it.
  - Training and evaluating on the training set

- better evaluation using cross-validation

## 6. Fine-tune your model.

- grid search

```
1 from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10],
6  'max_features': [2, 3, 4]},
    ]

forest_reg = RandomForestRegressor()

11 grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
    scoring='neg_mean_squared_error',
    return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

- Randomized Search
- Ensemble Methods
- Analyze the Best Models and Their Errors
- Evaluate your system on the test set

## 7. Present your solution.

## 8. Launch, monitor, and maintain your system.



## 3 | Classification

### 3.1 MNIST

### 3.2 Training a Binary Classifier

### 3.3 Performance Measures

Confusion Matrix, Precision and Recall, Precision/Recall Trade-off,

ROC (receiver operating characteristic) curve: true positive rate against the false positive rate

AUC (area under the curve)

### 3.4 Multiclass Classification

Whereas binary classifiers distinguish between two classes, multiclass classifiers (also called multinomial classifiers) can distinguish between more than two classes.

### 3.5 Error Analysis

### 3.6 Multilabel Classification

Until now each instance has always been assigned to just one class. In some cases you may want your classifier to output multiple classes for each instance. Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been

trained to recognize three faces, Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output  $[1, 0, 1]$  (meaning “Alice yes, Bob no, Charlie yes”). Such a classification system that outputs multiple binary tags is called a multilabel classification system.

### 3.7 Multioutput Classification

The last type of classification task we are going to discuss here is called multioutput–multiclass classification (or simply multioutput classification). It is simply a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

## 4 | Training Models

### 4.1 Linear Regression

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

- $\hat{y}$  is the predicted value.
- $n$  is the number of features.
- $x_i$  is the  $i^{th}$  feature value.
- $\theta_j$  is the  $j^{th}$  parameter

$$\hat{y} = h_{\boldsymbol{\theta}}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

- $\boldsymbol{\theta}$  is the model's parameter vector
- $\mathbf{x}$  is the instance's feature vector, with  $\mathbf{x}_0$  always equal to 1
- $h_{\boldsymbol{\theta}}$  is the hypothesis function, using the model parameters  $\boldsymbol{\theta}$ .

MSE cost function for a Linear Regression model

$$\text{MSE}(\mathbf{X}, h_{\boldsymbol{\theta}}) = \frac{1}{m} \sum_{i=1}^m \left( \boldsymbol{\theta}^T \cdot \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

The only difference is that we write  $h_{\boldsymbol{\theta}}$  instead of just  $h$  to make it clear that the model is parametrized by the vector  $\boldsymbol{\theta}$ . To simplify the notations, we will just write  $MSE(\boldsymbol{\theta})$  instead of  $MSE(\boldsymbol{\theta}, h_{\boldsymbol{\theta}})$ .

## 4.2 The Normal Equation

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

🔍 Both the Normal Equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regard to the number of instances in the training set (they are  $O(m)$ ), so they handle large training sets efficiently, provided they can fit in memory.

## 4.3 Gradient Descent

Gradient Descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.

🔍 When using Gradient Descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

### 4.3.1 Batch Gradient Descent

To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter  $\theta_j$ . In other words, you need to calculate how much the cost function will change if you change  $\theta_j$  just a little bit. This is called a partial derivative.

Partial derivatives of the cost function

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m \left( \theta^T \cdot \mathbf{x}^{(i)} - y^{(i)} \right) x_j^{(i)}$$

Gradient vector of the cost function

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$

🔔 Notice that this formula involves calculations over the full training set  $X$ , at each Gradient Descent step! This is why the algorithm is called Batch Gradient Descent: it uses the whole batch of training data at every step (actually, Full Gradient Descent would probably be a better name). As a result it is terribly slow on very large training sets (but we will see much faster Gradient Descent algorithms shortly). However, Gradient Descent scales well with the number of features; training a Linear Regression model when there are hundreds of thousands of features is much faster using Gradient Descent than using the Normal Equation or SVD decomposition.

Gradient Descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

where  $\eta$  is the learning rate

### 4.3.2 Stochastic Gradient Descent

The main problem with Batch Gradient Descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, Stochastic Gradient Descent picks a random instance in the training set at every step and computes the gradients based only on that single instance.

🔔 When using Stochastic Gradient Descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average. A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch). If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

### 4.3.3 Mini-batch Gradient Descent

The last Gradient Descent algorithm we will look at is called Mini-batch Gradient Descent. It is simple to understand once you know Batch and Stochastic Gradient Descent: at each step, instead of computing the gradients based on the full training set (as in Batch GD) or based on just one instance (as in Stochastic GD), Mini-batch GD computes the gradients on small random sets of instances called mini-batches. The main advantage of Mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.

## 4.4 Polynomial Regression

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.

## 4.5 Learning Curves

Root Mean Square Error (RMSE)

## 4.6 Regularized Linear Models

### 4.6.1 Ridge Regression

Ridge Regression (also called Tikhonov regularization) is a regularized version of Linear Regression: a regularization term equal to  $\alpha \sum_{i=1}^n \theta_i^2$  is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularized performance measure to evaluate the model's performance.

Ridge Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Ridge Regression closed-form solution

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

where  $\mathbf{A}$  is the  $n \times n$  identity matrix except with a 0 in the top-left cell, corresponding to the bias term.

### 4.6.2 Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (usually simply called Lasso Regression) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the  $\ell_1$  norm of the weight vector instead of half the square of the  $\ell_2$  norm.

Lasso Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

Lasso Regression subgradient vector

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

### 4.6.3 Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio  $r$ . When  $r = 0$ , Elastic Net is equivalent to Ridge Regression, and when  $r = 1$ , it is equivalent to Lasso Regression.

Elastic Net cost function:

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^n \theta_i^2$$

### 4.6.4 Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called early stopping. Figure 4-20 shows a complex model (in this case, a high-degree Polynomial Regression model) being trained with Batch Gradient Descent. As the epochs go by the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set. After a while though, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch.”

## 4.7 Logistic Regression

### 4.7.1 Estimating Probabilities

Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

Logistic Regression model prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

### 4.7.2 Training and Cost Function

Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

Logistic Regression cost function (log loss)

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

Logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m \left( \sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

## 4.8 Softmax Regression

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in Chapter 3). This is called Softmax Regression, or Multinomial



Logistic Regression.

Softmax score for class  $k$

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}$$

Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

$K$  is number of classes.  $\mathbf{s}(\mathbf{x})$  is a vector containing the scores of each class for the instances  $x$ .  $\sigma(\mathbf{s}(\mathbf{x}))_k$  is the estimated probability that the instance  $\mathbf{x}$  belongs to class  $k$  given the scores of each class for that instance.

Softmax Regression classifier prediction

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k (\theta_k^T \cdot \mathbf{x})$$

The argmax operator returns the value of a variable that maximizes a function. In this equation, it returns the value of  $k$  that maximizes the estimated probability  $\sigma(\mathbf{s}(\mathbf{x}))_k$ .

Minimizing the cost function shown in Equation 4-22, called the cross entropy, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities match the target classes (we will use it again several times in the following chapters).

Equation 4-22. Cross entropy cost function

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

Cross entropy gradient vector for class  $k$

$$\nabla_{\theta_k} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

## 5 | Support Vector Machines

### 5.1 Linear SVM classification

large margin classification

support vectors

#### 5.1.1 Soft Margin Classification

hard margin classification. Two main issues:

1. linearly separable
2. quite sensitive to outliers

limiting margin violations. This is called soft margin classification.

In Scikit-Learn's SVM classes, you can control this balance using the `C` hyperparameter: smaller  $\rightarrow$  wider street.

### 5.2 Nonlinear SVM classification

#### 5.2.1 Polynomial Kernel

#### 5.2.2 Adding Similarity Features

Gaussian Radial Basis Function (RBF)

$$\phi\gamma(\mathbf{x}, \ell) = \exp(-\gamma\|\mathbf{x} - \ell\|^2)$$

It is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark).

### 5.2.3 Gaussian RBF Kernel

## 5.3 SVM Regression

As we mentioned earlier, the SVM algorithm is quite versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. The trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street). The width of the street is controlled by a hyperparameter  $\epsilon$ .

## 5.4 Under the Hood

### 5.4.1 Decision Function and Predictions

Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b < 0 \\ 1 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

If we define  $t^{(i)}$  for negative instances (if  $y^{(i)} = 0$ ) and  $t^{(i)} = 1$  for positive instances (if  $y^{(i)} = 1$ ), then we can express this constraint as  $t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$  for all instances.

Hard margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b}{\text{minimize}} && \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} \\ & \text{subject to} && t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

Soft margin linear SVM classifier objective

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{minimize}} && \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{subject to} && t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \text{ and } \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

### 5.4.2 Quadratic Programming

$$\begin{aligned} & \underset{\mathbf{p}}{\text{Minimize}} && \frac{1}{2} \mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p} \\ & \text{subject to} && \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b} \end{aligned}$$

where

$$\left\{ \begin{array}{l} \mathbf{p} \text{ is an } p\text{-dimensional vector} \\ \mathbf{H} \text{ is an } p \times p \text{ matrix} \\ \mathbf{f} \text{ is an } p\text{-dim vector} \\ \mathbf{A} \text{ } c \times p \text{ matrix} \\ \mathbf{b} \text{ is an } c\text{-dim vector} \end{array} \right.$$

$p$  = number of parameters,  $c$  = number of constraints

### 5.4.3 The Dual Problem

Dual form of the linear SVM:

$$\begin{aligned} & \underset{\alpha}{\text{minimize}} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ & \text{subject to} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \dots, m \end{aligned}$$

### 5.4.4 Kernelized SVM

The function  $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$  is called a second degree polynomial kernel. In ML, a kernel is a function capable of computing the dot product  $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$  based only on the original vectors  $\mathbf{a}$  and  $\mathbf{b}$ , without having to compute (or even to know about) the transformation  $\phi$ .

## 6 | Decision Trees

### 6.1 Training and Visualizing a Decision Tree

### 6.2 Making Predictions

Gini impurity

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

where  $p_{i,k}$  is the ratio of class  $k$  instances among the training instances in the  $i^{th}$  node

### 6.3 The CART training alg

Scikit-Learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees (also called “growing” trees). The idea is really quite simple: the algorithm first splits the training set in two subsets using a single feature  $k$  and a threshold  $t_k$ . How does it choose  $k$  and  $t_k$ ? It searches for the pair  $(k, t_k)$  that produces the purest subsets (weighted by their size).

CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where  $\begin{cases} G_{\text{left/right}} & \text{measures the impurity of the left/right subset,} \\ m_{\text{left/right}} & \text{is the number of instances in the left/right subset.} \end{cases}$

## 6.4 Gini Impurity or Entropy?

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log_2(p_{i,k})$$

## 6.5 Regularization Hyperparameters

## 6.6 Regression

## 6.7 Instability

Hopefully by now you are convinced that Decision Trees have a lot going for them: they are simple to understand and interpret, easy to use, versatile, and powerful. However they do have a few limitations. First, as you may have noticed, Decision Trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to training set rotation. For example, Figure 6-7 shows a simple linearly separable dataset: on the left, a Decision Tree can split it easily, while on the right, after the dataset is rotated by  $45^\circ$ , the decision boundary looks unnecessarily convoluted. Although both Decision Trees fit the training set perfectly, it is very likely that the model on the right will not generalize well. One way to limit this problem is to use PCA (see Chapter 8), which often results in a better orientation of the training data.

## 7 | Ensemble Learning and Random Forests

A group of predictors is called an ensemble; thus, this technique is called Ensemble Learning, and an Ensemble Learning algorithm is called an Ensemble method.

In this chapter we will discuss the most popular Ensemble methods, including bagging, boosting, stacking, and a few others. We will also explore Random Forests.

### 7.1 Voting Classifiers

### 7.2 Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor, but to train them on different random subsets of the training set. When sampling is performed with replacement, this method is called bagging (short for bootstrap aggregating<sup>1</sup>). When sampling is performed without replacement, it is called pasting.

#### 7.2.1 Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression).

#### 7.2.2 Out-of-Bag Evaluation

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable

---

<sup>1</sup>In statistics, resampling with replacement is called bootstrapping.

## 7.3 Random Patches and Random Subspaces

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

## 7.4 Random Forests

As we have discussed, a Random Forest is an ensemble of Decision Trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can instead use the `RandomForestClassifier` class, which is more convenient and optimized for Decision Trees (similarly, there is a `RandomForestRegressor` class for regression tasks).

### 7.4.1 Extra-Trees

When you are growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular Decision Trees do).

A forest of such extremely random trees is simply called an Extremely Randomized Trees ensemble (or Extra-Trees for short).

## 7.5 Boosting

Boosting (originally called hypothesis boosting) refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are AdaBoost (short for Adaptive Boosting) and Gradient Boosting. Let's start with AdaBoost.

### 7.5.1 AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases. This is the technique used by Ada-Boost.



☞ There is one important drawback to this sequential learning technique: it cannot be parallelized (or only partially), since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights  $\alpha_j$ . The predicted class is the one that receives the majority of weighted votes.

AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \quad \sum_{j=1}^N \alpha_j \hat{y}_j^k \quad \text{where } N \text{ is the number of predictors.}$$
$$\hat{y}_j^{(\mathbf{x})} = k$$

Scikit-Learn actually uses a multiclass version of AdaBoost called SAMME (which stands for Stagewise Additive Modeling using a Multiclass Exponential loss function). When there are just two classes, SAMME is equivalent to AdaBoost. Moreover, if the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called SAMME.R (the R stands for “Real”), which relies on class probabilities rather than predictions and generally performs better.

## 7.5.2 Gradient Boosting

Another very popular Boosting algorithm is Gradient Boosting. Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor.

It is worth noting that an optimized implementation of Gradient Boosting is available in the popular python library XGBoost, which stands for Extreme Gradient Boosting.

## 7.6 Stacking

The last Ensemble method we will discuss in this chapter is called stacking (short for stacked generalization). It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?

## 8 | Dimension Reduction

### 8.1 The Curse of Dimensionality

In short, the more dimensions the training set has, the greater the risk of overfitting it.

### 8.2 Main Approaches

#### 8.2.1 Projection

#### 8.2.2 Manifold Learning

Many dimensionality reduction algorithms work by modeling the manifold on which the training instances lie; this is called Manifold Learning. It relies on the manifold assumption, also called the manifold hypothesis, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

### 8.3 PCA

#### 8.3.1 Explained Variance Ratio

Another very useful piece of information is the explained variance ratio of each principal component, available via the `explained_variance_ratio_` variable. It indicates the proportion of the dataset's variance that lies along the axis of each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset:

```
>>> pca.explained_variance_ratio_
```

```
array([0.84248607, 0.14631839])
```

This tells you that 84.2% of the dataset’s variance lies along the first axis, and 14.6% lies along the second axis. This leaves less than 1.2% for the third axis, so it is reasonable to assume that it probably carries little information.

### 8.3.2 Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will generally want to reduce the dimensionality down to 2 or 3.

- Randomized PCA
- Incremental PCA

## 8.4 Kernel PCA

In Chapter 5 we discussed the kernel trick, a mathematical technique that implicitly maps instances into a very high-dimensional space (called the feature space), enabling nonlinear classification and regression with Support Vector Machines. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space. Recall that a linear decision boundary in the high-dimensional feature space corresponds to a complex nonlinear decision boundary in the original space.

It turns out that the same trick can be applied to PCA, making it possible to perform complex nonlinear projections for dimensionality reduction. This is called Kernel (k)PCA. It is often good at preserving clusters of instances after projection, or sometimes even unrolling datasets that lie close to a twisted manifold.

## 8.5 LLE

Locally Linear Embedding (LLE) is another very powerful nonlinear dimensionality reduction (NLDR) technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms. In a nutshell, LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This makes it particularly good at unrolling twisted manifolds, especially when there is not too much noise.

## 8.6 Other Techniques

MDS, Isomap, t-SNE, Linear Discriminant Analysis (LDA)

## 9 | Unsupervised Learning Techniques

Although most of the applications of Machine Learning today are based on supervised learning (and as a result, this is where most of the investments go to), the vast majority of the available data is actually unlabeled: we have the input features  $\mathbf{X}$ , but we do not have the labels  $\mathbf{y}$ . Yann LeCun famously said that “if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake”. In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

### 9.1 Clustering

Wide applications:

- customer segmentation
- data analysis
- dim reduction
- anomaly detection
- semi-supervised
- search engines
- segment an image

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. For example, some algorithms look for instances centered around a particular point, called a centroid. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

### 9.1.1 KMeans

But how exactly does it know which solution is the best? Well of course it uses a performance metric! It is called the model's inertia: this is the mean squared distance between each instance and its closest centroid.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise approach (but also more computationally expensive) is to use the silhouette score, which is the mean silhouette coefficient over all the instances.

**Limitations** Despite its many merits, most notably being fast and scalable, K-Means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid sub-optimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover, K-Means does not behave very well when the clusters have varying sizes, different densities, or non-spherical shapes.

### 9.1.2 DBSCAN

### 9.1.3 Other Clustering Alg

Agglomerative clustering, Birch, Mean-shift, Affinity propagation, Spectral clustering

## 9.2 Gaussian Mixtures

A Gaussian mixture model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown.

### 9.2.1 Anomaly Detection using Gaussian Mixtures

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use.

A closely related task is novelty detection: it differs from anomaly detection in that the algorithm is assumed to be trained on a “clean” dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often precisely used to clean up a dataset.

### 9.2.2 Selecting the Number of Clusters

With K-Means, you could use the inertia or the silhouette score to select the appropriate number of clusters, but with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a theoretical information criterion such as the Bayesian information criterion (BIC) or the Akaike information criterion (AIC)

### 9.2.3 Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters, it is possible to use instead the `BayesianGaussianMixture` class which is capable of giving weights equal (or close) to zero to unnecessary clusters.

Gaussian mixture models work great on clusters with ellipsoidal shapes, but if you try to fit a dataset with different shapes, you may have bad surprises.

### 9.2.4 Other Anomaly Detection and Novelty Detection Algorithms

- Fast-MCD (minimum covariance determinant)
- Isolation forest
- Local outlier factor (LOF)
- One-class SVM: this algorithm is better suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then separating the two classes using a linear SVM classifier within this high-dimensional space (see Chapter 5). Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel, when it is in fact normal. It works great, especially with high-dimensional datasets, but just like all SVMs, it does not scale to large datasets.

## Part II

# Neural Networks and Deep Learning



# 10 | Introduction to Artificial Neural Networks with Keras

This is the key idea that sparked artificial neural networks (ANNs).

## 10.1 From Biological to Artificial Neurons

### 10.1.1 Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in Figure 10-1). It is an unusual-looking cell mostly found in animal cerebral cortexes (大脑皮层) (e.g., your brain), composed of a cell body containing the nucleus and most of the cell's complex components, and many branching extensions called dendrites (树突), plus one very long extension called the axon (轴突). The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called telodendria (轴突的后方分支称为终突), and at the tip of these branches are minuscule structures called synaptic terminals (or simply synapses) (突触终端), which are connected to the dendrites (or directly to the cell body) of other neurons. Biological neurons receive short electrical impulses called signals from other neurons via these synapses. When a neuron receives a sufficient number of signals from other neurons within a few milliseconds, it fires its own signals.

### 10.1.2 Logical Computations with Neurons

$\wedge, \vee, \neg$

### 10.1.3 The Perceptron

The Perceptron (感知器) is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron called a threshold logic unit (TLU), or sometimes a linear threshold unit (LTU): the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ( $z = \sum_{i=1}^n w_i x_i = \mathbf{x}^T \mathbf{w}$ ), then applies a step function to that sum and outputs the result:  $h_w(\mathbf{x}) = \text{step}(z)$  where  $z = \mathbf{x}^T \mathbf{w}$

Thanks to the magic of linear algebra, it is possible to efficiently compute the outputs of a layer of artificial neurons for several instances at once,

Computing the outputs of a fully connected layer

$$h_{\mathbf{W}, \mathbf{b}}(\mathbf{X}) = \phi(\mathbf{XW} + \mathbf{b})$$

- As always,  $\mathbf{X}$  represents the matrix of input features. It has one row per instance, one column per feature.
- The weight matrix  $\mathbf{W}$  contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector  $\mathbf{b}$  contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function  $\phi$  is called the activation function: when the artificial neurons are TLUs, it is a step function (but we will discuss other activation functions shortly).

### 10.1.4 Multi-Layer Perceptron and Backpropagation

For many years researchers struggled to find a way to train MLPs, without success. But in 1986, David Rumelhart, Geoffrey Hinton and Ronald Williams published a groundbreaking paper introducing the backpropagation training algorithm, which is still used today. In short, it is simply Gradient Descent using an efficient technique for computing the gradients automatically: in just two passes through the network (one forward, one backward), the backpropagation algorithm is able to compute the gradient of the network's error with regards to every single model parameter. In other words, it can find out how each connection weight and each bias term should be tweaked in order to reduce the error. Once it has these gradients, it just performs a regular Gradient Descent step, and the whole process is repeated until the network converges to the solution.

This algorithm is so important, it's worth summarizing it again: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes through each layer in reverse

to measure the error contribution from each connection (reverse pass), and finally slightly tweaks the connection weights to reduce the error (Gradient Descent step).

### 10.1.5 Regression MLPs

Regression Task

### 10.1.6 Classification MLPs

MLPs can also be used for classification tasks.

## 10.2 Implementing MLPs with Keras

Keras is a high-level Deep Learning API that allows you to easily build, train, evaluate and execute all sorts of neural networks. Its documentation (or specification) is available at <https://keras.io>. The reference implementation is simply called Keras as well, so to avoid any confusion we will call it keras-team (since it is available at <https://github.com/keras-team/keras>). It was developed by François Chollet as part of a research project<sup>12</sup> and released as an open source project in March 2015. It quickly gained popularity owing to its ease-of-use, flexibility and beautiful design. To perform the heavy computations required by neural networks, keras-team relies on a computation backend. At the present, you can choose from three popular open source deep learning libraries: TensorFlow, Microsoft Cognitive Toolkit (CNTK) or Theano.

### 10.2.1 Image Classifier with Sequential API

```
import tensorflow as tf
from tensorflow import keras

# load
fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()

# split and scale
X_valid, X_train = X_train_full[:5000] / 255., X_train_full[5000:] / 255.
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
X_test = X_test / 255.
```

```

13  ## Here are the corresponding class names:
    class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle_boot"]

18  model = keras.models.Sequential()
    model.add(keras.layers.Flatten(input_shape=[28, 28]))
    model.add(keras.layers.Dense(300, activation="relu"))
    model.add(keras.layers.Dense(100, activation="relu"))
    model.add(keras.layers.Dense(10, activation="softmax"))

23
    keras.backend.clear_session()
    np.random.seed(42)
    tf.random.set_seed(42)

28  # or

    model = keras.models.Sequential([
        keras.layers.Flatten(input_shape=[28, 28]),
        keras.layers.Dense(300, activation="relu"),
        keras.layers.Dense(100, activation="relu"),
        keras.layers.Dense(10, activation="softmax")
    ])

38  model.compile(loss="sparse_categorical_crossentropy",
                optimizer="sgd",
                metrics=["accuracy"])

    history = model.fit(X_train, y_train, epochs=30,
43                        validation_data=(X_valid, y_valid))

    # plot the learning curves
    import pandas as pd

48
    pd.DataFrame(history.history).plot(figsize=(8, 5))
    plt.grid(True)

```

```

plt.gca().set_ylim(0, 1)
save_fig("keras_learning_curves_plot")
53 plt.show()

model.evaluate(X_test, y_test) # 10000/10000 [=====] - 0s
    31us/sample - loss: 0.3343 - accuracy: 0.8857

58

# probability
X_new = X_test[:3]
y_proba = model.predict(X_new)
63 y_proba.round(2)

"""
array([ [0.   , 0.   , 0.   , 0.   , 0.   , 0.02, 0.   , 0.02, 0.   , 0.96],
        [0.   , 0.   , 0.98, 0.   , 0.02, 0.   , 0.   , 0.   , 0.   , 0.   ],
68        [0.   , 1.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   , 0.   ]],
      dtype=float32)
"""

y_pred = model.predict_classes(X_new)
73 # predict class: array([9, 2, 1])

np.array(class_names)[y_pred]
# array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')

78

# plot the first three and their predictions
plt.figure(figsize=(7.2, 2.4))
83 for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)

```

```
88 plt.subplots_adjust(wspace=0.2, hspace=0.5)
save_fig('fashion_mnist_images_plot', tight_layout=False)
plt.show()
```

### 10.2.2 Regression with Sequential

Building, training, evaluating and using a regression MLP using the Sequential API to make predictions is quite similar to what we did for classification. The main differences are the fact that the output layer has a single neuron (since we only want to predict a single value) and uses no activation function, and the loss function is the mean squared error. Since the dataset is quite noisy, we just use a single hidden layer with fewer neurons than before, to avoid overfitting.

As you can see, the Sequential API is quite easy to use. However, although sequential models are extremely common, it is sometimes useful to build neural networks with more complex topologies, or with multiple inputs or outputs. For this purpose, Keras offers the Functional API.

### 10.2.3 Building Complex Models Using the Functional API

One example of a non-sequential neural network is a Wide & Deep neural network. It connects all or part of the inputs directly to the output layer. This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). In contrast, a regular MLP forces all the data to flow through the full stack of layers, thus simple patterns in the data may end up being distorted by this sequence of transformations.

Let's build such a neural network to tackle the California housing problem:

First, we need to create an Input object. This is needed because we may have multiple inputs, as we will see later.

```
input = keras.layers.Input(shape=X_train.shape[1:])
```

Next, we create a Dense layer with 30 neurons and using the ReLU activation function. As soon as it is created, notice that we call it like a function, passing it the input. This is why this is called the Functional API. Note that we are just telling Keras how it should connect the layers together, no actual data is being processed yet.

```
hidden1 = keras.layers.Dense(30, activation="relu")(input)
```

We then create a second hidden layer, and again we use it as a function. Note however that we pass it the output of the first hidden layer.

```
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
```

Next, we create a `Concatenate()` layer, and once again we immediately use it like a function, to concatenate the input and the output of the second hidden layer (you may prefer the `keras.layers.concatenate()` function, which creates a `Concatenate` layer and immediately calls it with the given inputs).

```
concat = keras.layers.concatenate([input, hidden2])
```

Then we create the output layer, with a single neuron and no activation function, and we call it like a function, passing it the result of the concatenation.

```
output = keras.layers.Dense(1)(concat)
```

Lastly, we create a Keras Model, specifying which inputs and outputs to use.

```
model = keras.models.Model(inputs=[input], outputs=[output])
```

Once you have built the Keras model, everything is exactly like earlier, so no need to repeat it here: you must compile the model, train it, evaluate it and use it to make predictions.

But what if you want to send a subset of the features through the wide path, and a different subset (possibly overlapping) through the deep path? In this case, one solution is to use multiple inputs. For example, suppose we want to send 5 features through the deep path (features 0 to 4), and 6 features through the wide path (features 2 to 7):

```
input_A = keras.layers.Input(shape=[5])
input_B = keras.layers.Input(shape=[6])
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
4 hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_A, input_B], outputs=[output])
```

The code is self-explanatory. Note that we specified `inputs=[input_A, input_B]` when creating the model. Now we can compile the model as usual, but when we call the `fit()` method, instead of passing a single input matrix `X_train`, we must pass a pair of matrices (`X_train_A`, `X_train_B`): one per input. The same is true for `X_valid`, and also for `X_test` and `X_new` when you call `evaluate()` or `predict()`:

```
model.compile(loss="mse", optimizer="sgd")
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
5 X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]
history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
```

```
validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

There are also many use cases in which you may want to have multiple outputs. Adding extra outputs is quite easy: just connect them to the appropriate layers and add them to your model's list of outputs.

As you can see, you can build any sort of architecture you want quite easily with the Functional API. Let's look at one last way you can build Keras models.

### 10.2.4 Building Dynamic Models Using the Subclassing API

Simply subclass the `Model` class, create the layers you need in the constructor, and use them to perform the computations you want in the `call()` method. For example, creating an instance of the following `WideAndDeepModel` class gives us an equivalent model to the one we just built with the Functional API. You can then compile it, evaluate it and use it to make predictions, exactly like we just did.

However, this extra flexibility comes at a cost: your model's architecture is hidden within the `call()` method, so Keras cannot easily inspect it, it cannot save or clone it, and when you call the `summary()` method, you only get a list of layers, without any information on how they are connected to each other. Moreover, Keras cannot check types and shapes ahead of time, and it is easier to make mistakes. So unless you really need that extra flexibility, you should probably stick to the Sequential API or the Functional API.

### 10.2.5 Saving and Restoring a Model

```
model.save("my_keras_model.h5")
model = keras.models.load_model("my_keras_model.h5")
```

🔗 This will work when using the Sequential API or the Functional API, but unfortunately not when using Model subclassing. However, you can use `save_weights()` and `load_weights()` to at least save and restore the model parameters (but you will need to save and restore everything else yourself).

But what if training lasts several hours? This is quite common, especially when training on large datasets. In this case, you should not only save your model at the end of training, but also save checkpoints at regular intervals during training. But how can you tell the `fit()` method to save checkpoints? The answer is: using callbacks.



### 10.2.6 Using Callbacks

For example, the `ModelCheckpoint` callback saves checkpoints of your model at regular intervals during training, by default at the end of each epoch:

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")
history = model.fit(X_train, y_train, epochs=10, callbacks=[checkpoint_cb])
```

### 10.2.7 Visualization Using TensorBoard

TensorBoard is a great interactive visualization tool that you can use to view the learning curves during training, compare learning curves between multiple runs, visualize the computation graph, analyze training statistics, view images generated by your model, visualize complex multidimensional data projected down to 3D and automatically clustered for you, and more! This tool is installed automatically when you install TensorFlow, so you already have it!

Let's summarize what you learned so far in this chapter: we saw where neural nets came from, what an MLP is and how you can use it for classification and regression, how to build MLPs using `tf.keras`'s Sequential API, or more complex architectures using the Functional API or `Model Subclassing`, you learned how to save and restore a model, use callbacks for checkpointing, early stopping, and more, and finally how to use TensorBoard for visualization. You can already go ahead and use neural networks to tackle many problems! However, you may wonder how to choose the number of hidden layers, the number of neurons in the network, and all the other hyperparameters. Let's look at this now.

## 10.3 Fine-Tuning Neural Networks Hyperparameters

Note that `RandomizedSearchCV` uses K-fold cross-validation, so it does not use `X_valid` and `y_valid`. These are just used for early stopping.

Fortunately, there are many techniques to explore a search space much more efficiently than randomly. Their core idea is simple: when a region of the space turns out to be good, it should be explored more. This takes care of the "zooming" process for you and leads to much better solutions in much less time. Here are a few Python libraries you can use to optimize hyperparameters:

- Hyperopt
- Hyperas, kopt or Talos
- Scikit-Optimize (skopt)

- Spearmint
- Sklearn-Deap
- And many more!

# 11 | Training Deep Neural Networks

What if you need to tackle a very complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper DNN, perhaps with 10 layers or much more, each containing hundreds of neurons, connected by hundreds of thousands of connections. This would not be a walk in the park:

- First, you would be faced with the tricky vanishing gradients problem (or the related exploding gradients problem) that affects deep neural networks and makes lower layers very hard to train.
- Second, you might not have enough training data for such a large network, or it might be too costly to label.
- Third, training may be extremely slow.
- Fourth, a model with millions of parameters would severely risk overfitting the training set, especially if there are not enough training instances, or they are too noisy.

## 11.1 Vanishing/Exploding Gradients Problems

Recall backpropagation:

Backpropagation is a technique used to train artificial neural networks. It first computes the gradients of the cost function with regard to every model parameter (all the weights and biases), then it performs a Gradient Descent step using these gradients. This backpropagation step is typically performed thousands or millions of times, using many training batches, until the model parameters converge to values that (hopefully) minimize the cost function. To compute the gradients, backpropagation uses reverse-mode autodiff (although it wasn't called that when backpropagation was invented, and it has been reinvented several times). Reverse-mode autodiff performs a forward pass through a computation graph, computing every node's value for the current training batch, and then it performs a reverse pass, computing all the gradients at once (see Appendix D for more details). So what's the difference? Well, backpropagation refers to the whole process of training an artificial neural network using multiple backpropagation steps, each of which computes gradients and uses them to perform a Gradient Descent step. In contrast,

reverse-mode autodiff is just a technique to compute gradients efficiently, and it happens to be used by backpropagation.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution. This is called the **vanishing gradients** problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger, so many layers get insanely large weight updates and the algorithm diverges. This is the **exploding gradients** problem, which is mostly encountered in recurrent neural networks. More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

### 11.1.1 Glorot and He Initialization

fan\_avg

### 11.1.2 Nonsaturating Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the vanishing/ exploding gradients problems were in part due to a poor choice of activation function.

### 11.1.3 Batch Normalization

In a 2015 paper, Sergey Ioffe and Christian Szegedy proposed a technique called Batch Normalization (BN) to address the vanishing/exploding gradients problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer, simply zero-centering and normalizing each input, then scaling and shifting the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, this operation lets the model learn the optimal scale and mean of each of the layer's inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (e.g., using a StandardScaler): the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

### 11.1.4 Gradient Clipping

Another popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold. This is called Gradient Clipping. This technique is most

often used in recurrent neural networks, as Batch Normalization is tricky to use in RNNs. For other types of networks, BN is usually sufficient.

In Keras, implementing Gradient Clipping is just a matter of setting the `clipvalue` or `clipnorm` argument when creating an optimizer. For example:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

This will clip every component of the gradient vector to a value between  $-1.0$  and  $1.0$ .

In practice however, this approach works well. If you want to ensure that Gradient Clipping does not change the direction of the gradient vector, you should clip by norm by setting `clipnorm` instead of `clipvalue`. This will clip the whole gradient if its  $\ell_2$  norm is greater than the threshold you picked.

If you observe that the gradients explode during training (you can track the size of the gradients using TensorBoard), you may want to try both clipping by value and clipping by norm, with different threshold, and see which option performs best on the validation set.

## 11.2 Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch: instead, you should always try to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (we will discuss how to find them in Chapter 14), then just reuse the lower layers of this network: this is called *transfer learning*. It will not only speed up training considerably, but will also require much less training data.

The output layer of the original model should usually be replaced since it is most likely not useful at all for the new task, and it may not even have the right number of outputs for the new task.


Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.

### 11.2.1 Transfer Learning With Keras

Freeze:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False
```

```
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"])
```

 You must always compile your model after you freeze or unfreeze layers.

Well it turns out that transfer learning does not work very well with small dense networks: it works best with deep convolutional neural networks, so we will revisit transfer learning in Chapter 14, using the same techniques.

### 11.2.2 Unsupervised Pretraining

If you can gather plenty of unlabeled training data, you can try to train the layers one by one, starting with the lowest layer and then going up, using an unsupervised feature detector algorithm such as Restricted Boltzmann Machines (RBMs) or autoencoders. Each layer is trained on the output of the previously trained layers (all layers except the one being trained are frozen). Once all layers have been trained this way, you can add the output layer for your task, and fine-tune the final network using supervised learning (i.e., with the labeled training examples). At this point, you can unfreeze all the pretrained layers, or just some of the upper ones.

This is a rather long and tedious process, but it often works well; in fact, it is this technique that Geoffrey Hinton and his team used in 2006 and which led to the revival of neural networks and the success of Deep Learning.

### 11.2.3 Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

Self-supervised learning is when you automatically generate the labels from the data itself, then you train a model on the resulting "labeled" dataset using supervised learning techniques. Since this approach requires no human labeling whatsoever, it is best classified as a form of unsupervised learning.

## 11.3 Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using Batch Normalization, and reusing parts of a pretrained network (possibly built on an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer

than the regular Gradient Descent optimizer. In this section we will present the most popular ones: Momentum optimization, Nesterov Accelerated Gradient, AdaGrad, RMSProp, and finally Adam and Nadam optimization.

### 11.3.1 Momentum Optimization

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the very simple idea behind Momentum optimization, proposed by Boris Polyak in 1964. In contrast, regular Gradient Descent will simply take small regular steps down the slope, so it will take much more time to reach the bottom.

1.  $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta + \mathbf{m}$

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it is good to have a bit of friction in the system: it gets rid of these oscillations and thus speeds up convergence.

Implementing Momentum optimization in Keras is a no-brainer: just use the SGD optimizer and set its momentum hyperparameter, then lie back and profit!

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9)
```

### 11.3.2 Nesterov Accelerated Gradient

The idea of Nesterov Momentum optimization, or Nesterov Accelerated Gradient (NAG), is to measure the gradient of the cost function not at the local position but slightly ahead in the direction of the momentum.

1.  $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta + \beta \mathbf{m})$
2.  $\theta \leftarrow \theta + \mathbf{m}$

### 11.3.3 AdaGrad

The AdaGrad algorithm achieves this by scaling down the gradient vector along the steepest dimensions:

1.  $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$

$$2. \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \epsilon}$$

Recall that  $\otimes, \oslash$  represent element-wise.

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an adaptive learning rate. It helps point the resulting updates more directly toward the global optimum.

AdaGrad often performs well for simple quadratic problems, but unfortunately it often stops too early when training neural networks. The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though Keras has an Adagrad optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as Linear Regression, though). However, understanding Adagrad is helpful to grasp the other adaptive learning rate optimizers.

### 11.3.4 RMSProp

Although AdaGrad slows down a bit too fast and ends up never converging to the global optimum, the RMSProp algorithm fixes this by accumulating only the gradients from the most recent iterations (as opposed to all the gradients since the beginning of training).

As you might expect, Keras has an RMSProp optimizer:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

### 11.3.5 Adam and Nadam Optimization

Adam, which stands for adaptive moment estimation, combines the ideas of Momentum optimization and RMSProp: just like Momentum optimization it keeps track of an exponentially decaying average of past gradients, and just like RMSProp it keeps track of an exponentially decaying average of past squared gradients.

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Since Adam is an adaptive learning rate algorithm (like AdaGrad and RMSProp), it requires less tuning of the learning rate hyperparameter  $\eta$ . You can often use the default value  $\eta = 0.001$ , making Adam even easier to use than Gradient Descent.

Adaptive optimization methods (including RMSProp, Adam and Nadam optimization) are often



great, converging fast to a good solution. However, a 2017 paper<sup>1</sup> by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using plain Nesterov Accelerated Gradient instead: your dataset may just be allergic to adaptive gradients. Also check out the latest research, it is moving fast (e.g., AdaBound).

All the optimization techniques discussed so far only rely on the first-order partial derivatives (Jacobians). The optimization literature contains amazing algorithms based on the second-order partial derivatives (the Hessians, which are the partial derivatives of the Jacobians). Unfortunately, these algorithms are very hard to apply to deep neural networks because there are  $n^2$  Hessians per output (where  $n$  is the number of parameters), as opposed to just  $n$  Jacobians per output. Since DNNs typically have tens of thousands of parameters, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the Hessians is just too slow.

### Sparse Models

However, in some cases these techniques may remain insufficient. One last option is to apply Dual Averaging, often called Follow The Regularized Leader (FTRL), a technique proposed by Yurii Nesterov. When used with  $\ell_1$  regularization, this technique often leads to very sparse models. Keras implements a variant of FTRL called FTRLProximal<sup>21</sup> in the FTRL optimizer.

#### 11.3.6 Learning Rate Scheduling

As we discussed in Chapter 10, one approach is to start with a large learning rate, and divide it by 3 until the training algorithm stops diverging. You will not be too far from the optimal learning rate, which will learn quickly and converge to good solution. However, you can do better than a constant learning rate: if you start with a high learning rate and then reduce it once it stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate. There are many different strategies to reduce the learning rate during training. These strategies are called learning schedules (we briefly introduced this concept in Chapter 4), the most common of which are:

- Power scheduling
- exponential scheduling
- piecewise constant scheduling
- performance scheduling

To sum up, exponential decay or performance scheduling can considerably speed up convergence, so give them a try!

---

<sup>1</sup>“The Marginal Value of Adaptive Gradient Methods in Machine Learning,” A. C. Wilson et al. (2017).

## 11.4 Avoiding Overfitting Through Regularization

### 11.4.1 $\ell_1$ and $\ell_2$ Regularization

Here is how to apply  $\ell$  regularization a Keras layer's connection weights, using a regularization factor of 0.01:

```
layer = keras.layers.Dense(100, activation="elu",
                             kernel_initializer="he_normal",
                             kernel_regularizer=keras.regularizers.l2(0.01))
```

Since you will typically want to apply the same regularizer to all layers in your network, as well as the same activation function and the same initialization strategy in all hidden layers, you may find yourself repeating the same arguments over and over. This makes it ugly and error-prone. To avoid this, you can try refactoring your code to use loops. Another option is to use Python's `functools.partial()` function: it lets you create a thin wrapper for any callable, with some default argument values. For example:

```
from functools import partial

RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
5                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
10    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
11])
```

### 11.4.2 Dropout


Dropout is one of the most popular regularization techniques for deep neural networks.

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability  $p$  of being temporarily “dropped out,” meaning it will be entirely ignored during this training step, but it may be active during the next step. The hyperparameter  $p$  is called the

dropout rate, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

To implement dropout using Keras, you can use the `keras.layers.Dropout` layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all, it just passes the inputs to the next layer. For example, the following code applies dropout regularization before every Dense layer, using a dropout rate of 0.2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```

 Since dropout is only active during training, the training loss is penalized compared to the validation loss, so comparing the two can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So make sure to evaluate the training loss without dropout (e.g., after training). Alternatively, you can call the `fit()` method inside a `keras.backend.learning_phase_scope(1)` block: this will force dropout to be active during both training and validation

### 11.4.3 Monte-Carlo (MC) Dropout

In short, MC Dropout is a fantastic technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

### 11.4.4 Max-Norm Regularization

Another regularization technique that is quite popular for neural networks is called max-norm regularization: for each neuron, it constrains the weights  $\mathbf{w}$  of the incoming connections such that  $\|\mathbf{w}\|_2 \leq r$ , where  $r$  is the max-norm hyperparameter and  $\|\cdot\|_2$  is the  $\ell_2$  norm.

To implement max-norm regularization in Keras, just set every hidden layer's `kernel_constraint` argument to a `max_norm()` constraint, with the appropriate max value, for example:

```
keras.layers.Dense(100, activation="elu", kernel_initializer="he_normal",  
                    kernel_constraint=keras.constraints.max_norm(1.))
```

## 11.5 Summary and Practical Guidelines

In this chapter, we have covered a wide range of techniques and you may be wondering which ones you should use. The configuration in Table 11.1 will work fine in most cases, without requiring much hyperparameter tuning.

Hyperparameter	Default value
Kernel initializer:	LeCun initialization
Activation function:	SELU
Normalization:	None (self-normalization)
Regularization:	Early stopping
Optimizer:	Nadam
Learning rate schedule:	Performance scheduling

Table 11.1: Default DNN configuration

Don't forget to standardize the input features! Of course, you should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

The default configuration in Table 11.1 may need to be tweaked: self-normalize or not, sparse or not, low-latency, risk-sensitive.

However, there may come a time when you need to have even more control, for example to write a custom loss function or to tweak the training algorithm. For such cases, you will need to use TensorFlow's lower-level API, as we will see in the next chapter.

# 12 | Custom Models and Training with TensorFlow

## 12.1 A Quick Tour of TensorFlow

As you know, TensorFlow is a powerful library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning (but you could use it for anything else that requires heavy computations).

## 12.2 Using TensorFlow like NumPy

### 12.2.1 Tensors and Operations

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]]) # matrix
>>> tf.constant(42) # scalar
3 >>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape # t.dtype

# indexing
>>> t[:, 1:]
8 <tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
    array([[2., 3.],
           [5., 6.]], dtype=float32)>
```

## 12.3 Customizing Models and Training Algorithms

Omitted

## 12.4 TensorFlow Functions and Graphs

☞ If you call a TF Function many times with different numerical Python values, then many graphs will be generated, slowing down your program and using up a lot of RAM. Python values should be reserved for arguments that will have few unique values, such as hyperparameters like the number of neurons per layer. This allows TensorFlow to better optimize each variant of your model.

It's time to sum up! In this chapter we started with a brief overview of TensorFlow, then we looked at TensorFlow's low-level API, including tensors, operations, variables and special data structures. We then used these tools to customize almost every component in `tf.keras`. Finally, we looked at how TF Functions can boost performance, how graphs are generated using `autograph` and `tracing`, and what rules to follow when you write TF Functions.

# 13 | Loading and Preprocessing Data with TensorFlow


In this chapter, we will cover the Data API, the TFRecord format and the Features API in detail. We will also take a quick look at a few related projects from TensorFlow's ecosystem.

## 13.1 The Data API

```
>>> X = tf.range(10) # any data tensor
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
<TensorSliceDataset shapes: (), types: tf.int32>
```

```
1 >>> dataset = dataset.repeat(3).batch(7)
```

```
>>> dataset = dataset.map(lambda x: x * 2) # Items: [0,2,4,6,8,10,12]
```

 The dataset methods do not modify datasets, they create new ones, so make sure to keep a reference to these new datasets (e.g., `dataset = ...`), or else nothing will happen.

### 13.1.1 Shuffling the Data

As you know, Gradient Descent works best when the instances in the training set are independent and identically distributed.

It will create a new dataset that will start by filling up a buffer with the first items of the source dataset, then whenever it is asked for an item, it will pull one out randomly from the buffer, and replace it with a fresh one from the source dataset, until it has iterated entirely through the source dataset. At this point it continues to pull out

items randomly from the buffer until it is empty. You must specify the buffer size, and it is important to make it large enough or else shuffling will not be very efficient.

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) # 0 to 9, three times
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
```

### 13.1.2 Preprocessing

```
X_mean, X_std = [...] # mean and scale of each feature in the training set
n_inputs = 8
3 def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
8     return (x - X_mean) / X_std, y
```

### 13.1.3 Putting Everything Together

```
def csv_reader_dataset(filepaths, repeat=None, n_readers=5,
2     n_read_threads=None, shuffle_buffer_size=10000,
    n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths).repeat(repeat)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
7     cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.shuffle(shuffle_buffer_size)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.batch(batch_size)
    return dataset.prefetch(1)
```

### 13.1.4 Prefetching

By calling `prefetch(1)` at the end, we are creating a dataset that will do its best to always be one batch ahead.

### 13.1.5 Using the Dataset With `tf.keras`

```
train_set = csv_reader_dataset(train_filepaths, repeat=None)
```



```

valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)

5 model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, steps_per_epoch=len(X_train) // batch_size, epochs=10,
          validation_data=valid_set,
          validation_steps=len(X_valid) // batch_size)

```

Congratulations, you now know how to build powerful input pipelines using the Data API! However, so far we have used CSV files, which are common, simple and convenient, but they are not really efficient, and they do not support large or complex data structures very well, such as images or audio. So let's use TFRecords instead.

## 13.2 The TFRecord Format

If you are happy with CSV files (or whatever other format you are using), you do not **have** to use TFRecords. As the saying goes, if it ain't broke, don't fix it! TFRecords are useful when the bottleneck during training is loading and parsing the data.

## 13.3 The Features API

Preprocessing your data can be performed in many ways: it can be done ahead of time when preparing your data files, using any tool you like. Or you can preprocess your data on the fly when loading it with the Data API (e.g., using the dataset's `map()` method, as we saw earlier). Or you can include a preprocessing layer directly in your model. Whichever solution you prefer, the **Features API** can help you: it is a set of functions available in the `tf.feature_column` package, which let you define how each feature (or group of features) in your data should be preprocessed (therefore you can think of this API as the analog of Scikit-Learn's `ColumnTransformer` class). We will start by looking at the different types of columns available, and then we will look at how to use them.

## 13.4 TF Transform

But what if you could define your preprocessing operations just once? This is what TF Transform was designed for. It is part of TensorFlow Extended (TFX), an end-to-end platform for productionizing TensorFlow models. First, to use a TFX component, such as TF Transform, you must install it, it does not come bundled with TensorFlow. You define your preprocessing function just once (in Python), by using TF Transform functions for scaling, bucketizing,

crossing features, and more. You can also use any TensorFlow operation you need. Here is what this preprocessing function might look like if we just had two features:

```
import tensorflow_transform as tft
def preprocess(inputs): # inputs is a batch of input features
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
5    standardized_age = tft.scale_to_z_score(median_age - tft.mean(median_age))
    ocean_proximity_id = tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
10    }
```

With the Data API, TFRecords, the Features API and TF Transform, you can build highly scalable input pipelines for training, and also benefit from fast and portable data preprocessing in production.

But what if you just wanted to use a standard dataset? Well in that case, things are much simpler: just use TFDS!

## 13.5 The TensorFlow Datasets (TFDS) Project

The TensorFlow Datasets project makes it trivial to download common datasets, from small ones like MNIST or Fashion MNIST, to huge datasets like ImageNet (you will need quite a bit of disk space!). The list includes image datasets, text datasets (including translation datasets), audio and video datasets, and more.

This was quite a technical chapter, and you may feel that it is a bit far from the abstract beauty of neural networks, but the fact is deep learning often involves large amounts of data, and knowing how to load, parse and preprocess it efficiently is a crucial skill to have. In the next chapter, we will look at Convolutional Neural Networks, which are among the most successful neural net architectures for image processing, and many other applications.

# 14 | Deep Computer Vision Using Convolutional Neural Networks

## 14.1 Convolutional Layers

The most important building block of a CNN is the convolutional layer: neurons in the first convolutional layer are not connected to every single pixel in the input image (like they were in the layers discussed in previous chapters), but only to pixels in their receptive fields. In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer. This architecture allows the network to concentrate on small low-level features in the first hidden layer, then assemble them into larger higher-level features in the next hidden layer, and so on. This hierarchical structure is common in real-world images, which is one of the reasons why CNNs work so well for image recognition.

As you can see, the field of Deep Computer Vision is vast and moving fast, with all sorts of architectures popping out every year, all based on Convolutional Neural Networks. The progress made in just a few years has been astounding, and researchers are now focusing on harder and harder problems, such as adversarial learning (which attempts to make the network more resistant to images designed to fool it), explainability (understanding why the network makes a specific classification), realistic image generation (which we will come back to in ???), single-shot learning (a system that can recognize an object after it has seen it just once), and much more. Some even explore completely novel architectures, such as Geoffrey Hinton's capsule networks<sup>32</sup> (I presented them in a couple videos, with the corresponding code in a notebook). Now on to the next chapter, where we will look at how to process sequential data such as time series using Recurrent Neural Networks and Convolutional Neural Networks.

# 15 | Processing Sequences Using RNNs and CNNs

The batter hits the ball. The outfielder immediately starts running, anticipating the ball's trajectory. He tracks it, adapts his movements, and finally catches it (under a thunder of applause). Predicting the future is something you do all the time, whether you are finishing a friend's sentence or anticipating the smell of coffee at breakfast. In this chapter we will discuss recurrent neural networks (RNNs), a class of nets that can predict the future (well, up to a point, of course). They can analyze time series data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on sequences of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have considered so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing applications such as automatic translation or speech-to-text.

In this chapter we will first look at the fundamental concepts underlying RNNs and how to train them using backpropagation through time, then we will use them to forecast a time series. After that we'll explore the two main difficulties that RNNs face:

- Unstable gradients (discussed in Chapter 11), which can be alleviated using various techniques, including recurrent dropout and recurrent layer normalization
- A (very) limited short-term memory, which can be extended using LSTM and GRU cells

In Chapter 16, we will continue to explore RNNs, and we will see how they can tackle various NLP tasks.

# 16 | Natural Language Processing with RNNs and Attention

A common approach for natural language tasks is to use recurrent neural networks.

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

# Dimension Reduction - Autoencoders

Autoencoders (AE) are neural networks that aims to copy their inputs to their outputs. They work by compressing the input into a latent-space representation, and then reconstructing the output from this representation. This kind of network is composed of two parts :

1. Encoder: This is the part of the network that compresses the input into a latent-space representation. It can be represented by an encoding function  $h=f(x)$ .
2. Decoder: This part aims to reconstruct the input from the latent space representation. It can be represented by a decoding function  $r=g(h)$ .

## What are autoencoders used for?

Today data denoising and dimensionality reduction for data visualization are considered as two main interesting practical applications of autoencoders. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

Autoencoders are learned automatically from data examples. It means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input and that it does not require any new engineering, only the appropriate training data.

## Types

Vanilla autoencoder

Multilayer autoencoder

Convolutional autoencoder

Regularized autoencoder

# 17 | Representation Learning and Generative Learning Using Autoencoders and GANs

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

## 17.1 Autoencoders

### 17.1.1 Performing PCA with an Undercomplete Linear Autoencoder

```
from tensorflow import keras

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

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

### 17.1.2 Implementing a Stacked Autoencoder Using Keras

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
```

```

3     keras.layers.Dense(100, activation="selu"),
        keras.layers.Dense(30, activation="selu"),
    ])

stacked_decoder = keras.models.Sequential([
8     keras.layers.Dense(100, activation="selu", input_shape=[30]),
        keras.layers.Dense(28 * 28, activation="sigmoid"),
        keras.layers.Reshape([28, 28])
    ])

13 stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                   optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                        validation_data=[X_valid, X_valid])

```

### 17.1.3 Convolutional Autoencoders

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

### 17.1.4 Recurrent Autoencoders

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



### 17.1.5 Denoising Autoencoders

### 17.1.6 Sparse Autoencoders

### 17.1.7 Variational Autoencoders

## 17.2 Generative Adversarial Networks

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

```
codings_size = 30

3 generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
8 ])
discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
13    keras.layers.Dense(1, activation="sigmoid")
])

gan = keras.models.Sequential([generator, discriminator])

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

### 17.2.1 The Difficulties of Training GANs

Unfortunately, it's not that simple: nothing guarantees that the equilibrium will ever be reached.

The biggest difficulty is called mode collapse: this is when the generator's outputs gradually become less diverse.

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable.

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

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

## **17.2.2 Progressive Growing of GANs**

### **17.2.3 StyleGANs**

# 18 | Reinforcement Learning

## 18.1 Learning to Optimize Rewards

## 18.2 Introduction to OpenAI Gym

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

## 18.3 Overview of Some Popular RL Algorithms

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

- Actor-Critic algorithms
- Asynchronous Advantage Actor-Critic (A3C)
- Advantage Actor-Critic (A2C)
- Soft Actor-Critic (SAC)
- Proximal Policy Optimization (PPO)
- Curiosity-based exploration

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

# 19 | Training and Deploying TensorFlow Models at Scale

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

## 19.1 Serving a TensorFlow Model

## 19.2 Deploying a Model to a Mobile or Embedded Device

## 19.3 Using GPUs to Speed Up Computations

## 19.4 Training Models Across Multiple Devices

Now you have all the tools and knowledge you need to create state-of-the-art neural net architectures and train them at scale using various distribution strategies, on your own infrastructure or on the cloud—and you can even perform powerful Bayesian optimization to fine-tune the hyperparameters!

## Part III

# Appendices

# Appendices

# A | Machine Learning Project Checklist

This checklist can guide you through your Machine Learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.



## Appendix B. Machine Learning Project Checklist

This checklist can guide you through your Machine Learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and shortlist the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

### Frame the Problem and Look at the Big Picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions/workarounds (if any)?
4. How should you frame this problem (supervised/unsupervised, online/offline, etc.)?
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?
7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

### Get the Data

Note: automate as much as possible so you can easily get fresh data.



1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

### Explore the Data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
  - Name
  - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)
  - % of missing values
  - Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
  - Usefulness for the task
  - Type of distribution (Gaussian, uniform, logarithmic, etc.)
4. For supervised learning tasks, identify the target attribute(s).
5. Visualize the data.
6. Study the correlations between attributes.
7. Study how you would solve the problem manually.
8. Identify the promising transformations you may want to apply.
9. Identify extra data that would be useful (go back to "Get the Data").
10. Document what you have learned.

### Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
- Write functions for all data transformations you apply, for five reasons:
  - So you can easily prepare the data the next time you get a fresh dataset
  - So you can apply these transformations in future projects
  - To clean and prepare the test set
  - To clean and prepare new data instances once your solution is live
  - To make it easy to treat your preparation choices as hyperparameters

1. Data cleaning:

- Fix or remove outliers (optional).
- Fill in missing values (e.g., with zero, mean, median...) or drop their rows (or columns).

2. Feature selection (optional):

- Drop the attributes that provide no useful information for the task.

3. Feature engineering, where appropriate:

- Discretize continuous features.
- Decompose features (e.g., categorical, date/time, etc.).
- Add promising transformations of features (e.g.,  $\log(x)$ ,  $\sqrt{x}$ ,  $x^2$ , etc.).
- Aggregate features into promising new features.

4. Feature scaling:

- Standardize or normalize features.

### Shortlist Promising Models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
  - Once again, try to automate these steps as much as possible.
1. Train many quick-and-dirty models from different categories (e.g., linear, naive Bayes, SVM, Random Forest, neural net, etc.) using standard parameters.
  2. Measure and compare their performance.
    - For each model, use  $N$ -fold cross-validation and compute the mean and standard deviation of the performance measure on the  $N$  folds.
  3. Analyze the most significant variables for each algorithm.
  4. Analyze the types of errors the models make.
    - What data would a human have used to avoid these errors?
  5. Perform a quick round of feature selection and engineering.
  6. Perform one or two more quick iterations of the five previous steps.
  7. Shortlist the top three to five most promising models, preferring models that make different types of errors.

### Fine-Tune the System

Notes:

- You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.
  - As always, automate what you can.
1. Fine-tune the hyperparameters using cross-validation:
    - Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g., if you're not sure whether to replace missing values with zeros or with the median value, or to just drop the rows).
    - Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, as described by Jasper Snoek et al.).<sup>1</sup>
  2. Try Ensemble methods. Combining your best models will often produce better performance than running them individually.
  3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.

## WARNING

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").

### Launch!

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
  - Beware of slow degradation: models tend to "rot" as data evolves.
  - Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
  - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

1 Jasper Snoek et al., "Practical Bayesian Optimization of Machine Learning Algorithms," *Proceedings of the 25th International Conference on Neural Information Processing Systems 2* (2012): 2951 – 2959.

---

Settings / Support / Sign Out



PREV

A. Exercise Solutions

NEXT

C. SVM Dual Problem

