We only looked at a very simple bagging example in this section. In practice, more complex classification tasks and a dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to its strengths. Finally, we must note that the bagging algorithm can be an effective approach to reducing the variance of a model. However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trends in the data well. This is why we want to perform bagging on an ensemble of classifiers with low bias, for example, unpruned decision trees.

Leveraging weak learners via adaptive boosting

In this last section about ensemble methods, we will discuss **boosting**, with a special focus on its most common implementation: **Adaptive Boosting** (**AdaBoost**).

AdaBoost recognition

The original idea behind AdaBoost was formulated by Robert E. Schapire in 1990 in *The Strength of Weak Learnability*, *Machine Learning*, 5(2): 197-227, 1990, URL: `http://rob.schapire.net/papers/strengthofweak.pdf`. After Robert Schapire and Yoav Freund presented the AdaBoost algorithm in the *Proceedings of the Thirteenth International Conference* (ICML 1996), AdaBoost became one of the most widely used ensemble methods in the years that followed (*Experiments with a New Boosting Algorithm* by *Y. Freund*, *R. E. Schapire*, and others, *ICML*, volume 96, 148-156, 1996). In 2003, Freund and Schapire received the Gödel Prize for their groundbreaking work, which is a prestigious prize for the most outstanding publications in the field of computer science.

In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, which often only have a slight performance advantage over random guessing—a typical example of a weak learner is a decision tree stump. The key concept behind boosting is to focus on training examples that are hard to classify, that is, to let the weak learners subsequently learn from misclassified training examples to improve the performance of the ensemble.

The following subsections will introduce the algorithmic procedure behind the general concept of boosting and AdaBoost. Lastly, we will use scikit-learn for a practical classification example.

How adaptive boosting works

In contrast to bagging, the initial formulation of the boosting algorithm uses random subsets of training examples drawn from the training dataset without replacement; the original boosting procedure can be summarized in the following four key steps:

1. Draw a random subset (sample) of training examples, d_1, without replacement from the training dataset, D, to train a weak learner, C_1.
2. Draw a second random training subset, d_2, without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner, C_2.

3. Find the training examples, d_3, in the training dataset, D, which C_1 and C_2 disagree upon, to train a third weak learner, C_3.
4. Combine the weak learners C_1, C_2, and C_3 via majority voting.

As discussed by Leo Breiman (*Bias, variance, and arcing classifiers*, 1996), boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, boosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data (*An improvement of AdaBoost to avoid overfitting* by *G. Raetsch*, *T. Onoda*, and *K. R. Mueller*. *Proceedings of the International Conference on Neural Information Processing*, *CiteSeer*, 1998).

In contrast to the original boosting procedure described here, AdaBoost uses the complete training dataset to train the weak learners, where the training examples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.

Before we dive deeper into the specific details of the AdaBoost algorithm, let's take a look at *Figure 7.9* to get a better grasp of the basic concept behind AdaBoost:

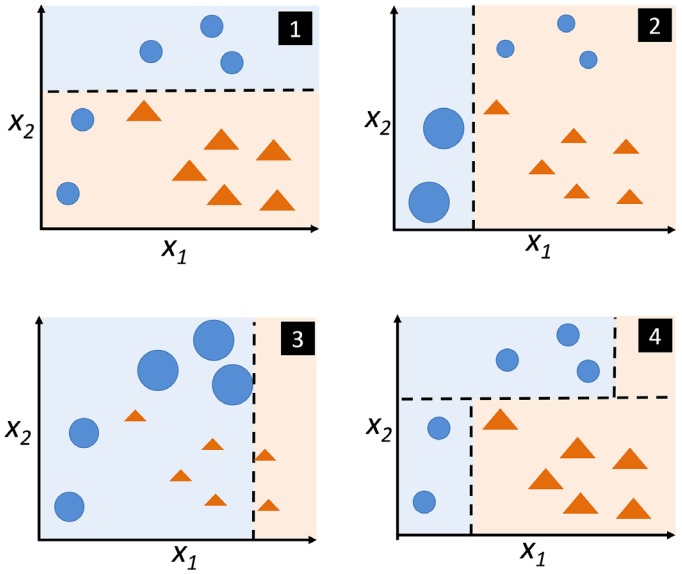


Figure 7.9: The concept of AdaBoost to improve weak learners

To walk through the AdaBoost illustration step by step, we will start with subfigure **1**, which represents a training dataset for binary classification where all training examples are assigned equal weights. Based on this training dataset, we train a decision stump (shown as a dashed line) that tries to classify the examples of the two classes (triangles and circles), as well as possibly minimizing the loss function (or the impurity score in the special case of decision tree ensembles).

For the next round (subfigure **2**), we assign a larger weight to the two previously misclassified examples (circles). Furthermore, we lower the weight of the correctly classified examples. The next decision stump will now be more focused on the training examples that have the largest weights—the training examples that are supposedly hard to classify.

The weak learner shown in subfigure **2** misclassifies three different examples from the circle class, which are then assigned a larger weight, as shown in subfigure **3**.

Assuming that our AdaBoost ensemble only consists of three rounds of boosting, we then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote, as shown in subfigure **4**.

Now that we have a better understanding of the basic concept of AdaBoost, let's take a more detailed look at the algorithm using pseudo code. For clarity, we will denote element-wise multiplication by the cross symbol (×) and the dot-product between two vectors by a dot symbol (·):

1. Set the weight vector, $\boldsymbol{w}$, to uniform weights, where $\sum_i w_i = 1$.
2. For j in m boosting rounds, do the following:
    a. Train a weighted weak learner: $C_j = \text{train}(\boldsymbol{X}, \boldsymbol{y}, \boldsymbol{w})$.
    b. Predict class labels: $\hat{\boldsymbol{y}} = \text{predict}(C_j, \boldsymbol{X})$.
    c. Compute the weighted error rate: $\varepsilon = \boldsymbol{w} \cdot (\hat{\boldsymbol{y}} \neq \boldsymbol{y})$.
    d. Compute the coefficient: $\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$.
    e. Update the weights: $\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$.
    f. Normalize the weights to sum to 1: $\boldsymbol{w} := \boldsymbol{w} / \sum_i w_i$.
3. Compute the final prediction: $\hat{\boldsymbol{y}} = \left(\sum_{j=1}^{m} (\alpha_j \times \text{predict}(C_j, \boldsymbol{X})) > 0\right)$.

Note that the expression $(\hat{\boldsymbol{y}} \neq \boldsymbol{y})$ in *step 2c* refers to a binary vector consisting of 1s and 0s, where a 1 is assigned if the prediction is incorrect and 0 is assigned otherwise.

Although the AdaBoost algorithm seems to be pretty straightforward, let's walk through a more concrete example using a training dataset consisting of 10 training examples, as illustrated in *Figure 7.10*:

Index	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

Figure 7.10: Running 10 training examples through the AdaBoost algorithm

The first column of the table depicts the indices of training examples 1 to 10. In the second column, you can see the feature values of the individual samples, assuming this is a one-dimensional dataset. The third column shows the true class label, y_i, for each training sample, x_i, where $y_i \in \{1, -1\}$. The initial weights are shown in the fourth column; we initialize the weights uniformly (assigning the same constant value) and normalize them to sum to 1. In the case of the 10-sample training dataset, we therefore assign 0.1 to each weight, w_i, in the weight vector, $\boldsymbol{w}$. The predicted class labels, $\hat{\boldsymbol{y}}$, are shown in the fifth column, assuming that our splitting criterion is $x \leq 3.0$. The last column of the table then shows the updated weights based on the update rules that we defined in the pseudo code.

Since the computation of the weight updates may look a little bit complicated at first, we will now follow the calculation step by step. We will start by computing the weighted error rate, ε (`epsilon`), as described in *step 2c*:

```
>>> y = np.array([1, 1, 1, -1, -1, -1,  1,  1,  1, -1])
>>> yhat = np.array([1, 1, 1, -1, -1, -1, -1, -1, -1, -1])
>>> correct = (y == yhat)
>>> weights = np.full(10, 0.1)
>>> print(weights)
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
>>> epsilon = np.mean(~correct)
>>> print(epsilon)
0.3
```

Note that `correct` is a Boolean array consisting of `True` and `False` values where `True` indicates that a prediction is correct. Via `~correct`, we invert the array such that `np.mean(~correct)` computes the proportion of incorrect predictions (`True` counts as the value 1 and `False` as 0), that is, the classification error.

Next, we will compute the coefficient, α_j—shown in *step 2d*—which will later be used in *step 2e* to update the weights, as well as for the weights in the majority vote prediction (*step 3*):

```
>>> alpha_j = 0.5 * np.log((1-epsilon) / epsilon)
>>> print(alpha_j)
0.42364893019360184
```

After we have computed the coefficient, α_j (`alpha_j`), we can now update the weight vector using the following equation:

$$\boldsymbol{w} := \boldsymbol{w} \times \exp(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y})$$

Here, $\hat{\boldsymbol{y}} \times \boldsymbol{y}$ is an element-wise multiplication between the vectors of the predicted and true class labels, respectively. Thus, if a prediction, $\hat{y}_i$, is correct, $\hat{y}_i \times y_i$ will have a positive sign so that we decrease the ith weight, since α_j is a positive number as well:

```
>>> update_if_correct = 0.1 * np.exp(-alpha_j * 1 * 1)
>>> print(update_if_correct)
0.06546536707079771
```

Similarly, we will increase the ith weight if $\hat{y}_i$ predicted the label incorrectly, like this:

```
>>> update_if_wrong_1 = 0.1 * np.exp(-alpha_j * 1 * -1)
>>> print(update_if_wrong_1)
0.1527525231651947
```

Alternatively, it's like this:

```
>>> update_if_wrong_2 = 0.1 * np.exp(-alpha_j * -1 * 1)
>>> print(update_if_wrong_2)
0.1527525231651947
```

We can use these values to update the weights as follows:

```
>>> weights = np.where(correct == 1,
...                    update_if_correct,
...                    update_if_wrong_1)
>>> print(weights)
array([0.06546537, 0.06546537, 0.06546537, 0.06546537, 0.06546537,
       0.06546537, 0.15275252, 0.15275252, 0.15275252, 0.06546537])
```

The code above assigned the `update_if_correct` value to all correct predictions and the `update_if_wrong_1` value to all wrong predictions. We omitted using `update_if_wrong_2` for simplicity, since it is similar to `update_if_wrong_1` anyway.

After we have updated each weight in the weight vector, we normalize the weights so that they sum up to 1 (*step 2f*):

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

In code, we can accomplish that as follows:

```
>>> normalized_weights = weights / np.sum(weights)
>>> print(normalized_weights)
[0.07142857 0.07142857 0.07142857 0.07142857 0.07142857 0.07142857
 0.16666667 0.16666667 0.16666667 0.07142857]
```

Thus, each weight that corresponds to a correctly classified example will be reduced from the initial value of 0.1 to 0.0714 for the next round of boosting. Similarly, the weights of the incorrectly classified examples will increase from 0.1 to 0.1667.

Applying AdaBoost using scikit-learn

The previous subsection introduced AdaBoost in a nutshell. Skipping to the more practical part, let's now train an AdaBoost ensemble classifier via scikit-learn. We will use the same Wine subset that we used in the previous section to train the bagging meta-classifier.

Via the `base_estimator` attribute, we will train the `AdaBoostClassifier` on 500 decision tree stumps:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=1)
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                          n_estimators=500,
...                          learning_rate=0.1,
...                          random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Decision tree train/test accuracies '
...       f'{tree_train:.3f}/{tree_test:.3f}')
Decision tree train/test accuracies 0.916/0.875
```

As you can see, the decision tree stump seems to underfit the training data in contrast to the unpruned decision tree that we saw in the previous section:

```
>>> ada = ada.fit(X_train, y_train)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print(f'AdaBoost train/test accuracies '
...       f'{ada_train:.3f}/{ada_test:.3f}')
AdaBoost train/test accuracies 1.000/0.917
```

Here, you can see that the AdaBoost model predicts all class labels of the training dataset correctly and also shows a slightly improved test dataset performance compared to the decision tree stump. However, you can also see that we introduced additional variance with our attempt to reduce the model bias—a greater gap between training and test performance.

Although we used another simple example for demonstration purposes, we can see that the performance of the AdaBoost classifier is slightly improved compared to the decision stump and achieved very similar accuracy scores as the bagging classifier that we trained in the previous section. However, we must note that it is considered bad practice to select a model based on the repeated usage of the test dataset. The estimate of the generalization performance may be overoptimistic, which we discussed in more detail in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Lastly, let's check what the decision regions look like:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(1, 2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Decision tree', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue',
...                        marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green',
...                        marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('OD280/OD315 of diluted wines', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='Alcohol',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()
```

By looking at the decision regions, you can see that the decision boundary of the AdaBoost model is substantially more complex than the decision boundary of the decision stump. In addition, note that the AdaBoost model separates the feature space very similarly to the bagging classifier that we trained in the previous section:

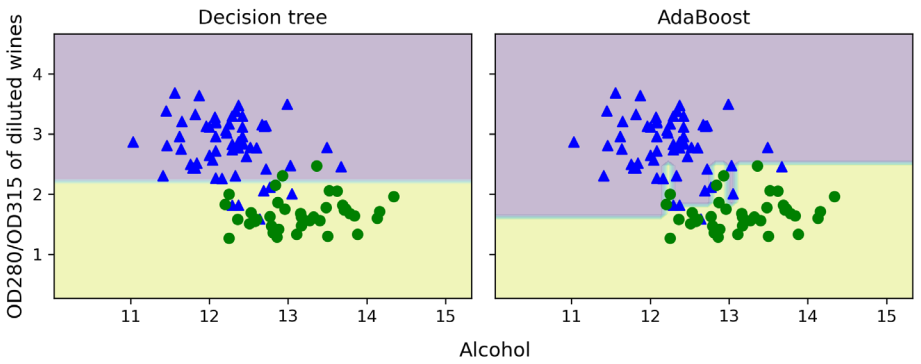


Figure 7.11: The decision boundaries of the decision tree versus AdaBoost

As concluding remarks about ensemble techniques, it is worth noting that ensemble learning increases the computational complexity compared to individual classifiers. In practice, we need to think carefully about whether we want to pay the price of increased computational costs for an often relatively modest improvement in predictive performance.

An often-cited example of this tradeoff is the famous $1 million *Netflix Prize*, which was won using ensemble techniques. The details about the algorithm were published in *The BigChaos Solution to the Netflix Grand Prize* by *A. Toescher*, *M. Jahrer*, and *R. M. Bell*, *Netflix Prize documentation*, 2009, which is available at `http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf`. The winning team received the $1 million grand prize money; however, Netflix never implemented their model due to its complexity, which made it infeasible for a real-world application:

> *"We evaluated some of the new methods offline but the additional accuracy gains that we measured did not seem to justify the engineering effort needed to bring them into a production environment."*
>
> `http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html`

Gradient boosting — training an ensemble based on loss gradients

Gradient boosting is another variant of the boosting concept introduced in the previous section, that is, successively training weak learners to create a strong ensemble. Gradient boosting is an extremely important topic because it forms the basis of popular machine learning algorithms such as XGBoost, which is well-known for winning Kaggle competitions.

The gradient boosting algorithm may appear a bit daunting at first. So, in the following subsections, we will cover it step by step, starting with a general overview. Then, we will see how gradient boosting is used for classification and walk through an example. Finally, after we've introduced the fundamental concepts of gradient boosting, we will take a brief look at popular implementations, such as XGBoost, and we will see how we can use gradient boosting in practice.

Comparing AdaBoost with gradient boosting

Fundamentally, gradient boosting is very similar to AdaBoost, which we discussed previously in this chapter. AdaBoost trains decision tree stumps based on errors of the previous decision tree stump. In particular, the errors are used to compute sample weights in each round as well as for computing a classifier weight for each decision tree stump when combining the individual stumps into an ensemble. We stop training once a maximum number of iterations (decision tree stumps) is reached. Like AdaBoost, gradient boosting fits decision trees in an iterative fashion using prediction errors. However, gradient boosting trees are usually deeper than decision tree stumps and have typically a maximum depth of 3 to 6 (or a maximum number of 8 to 64 leaf nodes). Also, in contrast to AdaBoost, gradient boosting does not use the prediction errors for assigning sample weights; they are used directly to form the target variable for fitting the next tree. Moreover, instead of having an individual weighting term for each tree, like in AdaBoost, gradient boosting uses a global learning rate that is the same for each tree.

As you can see, AdaBoost and gradient boosting share several similarities but differ in certain key aspects. In the following subsection, we will sketch the general outline of the gradient boosting algorithm.

Outlining the general gradient boosting algorithm

In this section, we will look at gradient boosting for classification. For simplicity, we will look at a binary classification example. Interested readers can find the generalization to the multi-class setting with logistic loss in *Section 4.6. Multiclass logistic regression and classification* of the original gradient boosting paper written by Friedman in 2001, *Greedy function approximation: A gradient boosting machine*, `https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedy-function-approximation-A-gradient-boostingmachine/10.1214/aos/1013203451.full`.

Gradient boosting for regression

Note that the procedure behind gradient boosting is a bit more complicated than AdaBoost. We omit a simpler regression example, which was given in Friedman's paper, for brevity, but interested readers are encouraged to also consider my complementary video tutorial on gradient boosting for regression, which is available at: `https://www.youtube.com/watch?v=zblsrxc7XpM`.

In essence, gradient boosting builds a series of trees, where each tree is fit on the error—the difference between the label and the predicted value—of the previous tree. In each round, the tree ensemble improves as we are nudging each tree more in the right direction via small updates. These updates are based on a loss gradient, which is how gradient boosting got its name.

The following steps will introduce the general algorithm behind gradient boosting. After illustrating the main steps, we will dive into some of its parts in more detail and walk through a hands-on example in the next subsections.

1. Initialize a model to return a constant prediction value. For this, we use a decision tree root node; that is, a decision tree with a single leaf node. We denote the value returned by the tree as $\hat{y}$, and we find this value by minimizing a differentiable loss function L that we will define later:

$$F_0(x) = \arg\min_{\hat{y}} \sum_{i=1}^{n} L(y_i, \hat{y})$$

   Here, n refers to the n training examples in our dataset.

2. For each tree $m = 1, \ldots, M$, where M is a user-specified total number of trees, we carry out the following computations outlined in *steps 2a* to *2d* below:

   a. Compute the difference between a predicted value $F(x_i) = \hat{y}_i$ and the class label y_i. This value is sometimes called the *pseudo-response* or *pseudo-residual*. More formally, we can write this pseudo-residual as the negative gradient of the loss function with respect to the predicted values:

$$r_{im} = -\left[\frac{\partial\, L(y_i, F(x_i))}{\partial\, F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n$$

   Note that in the notation above $F(x)$ is the prediction of the previous tree, $F_{m-1}(x)$. So, in the first round, this refers to the constant value from the tree (single leaf node) from step 1.

   b. Fit a tree to the pseudo-residuals r_{im}. We use the notation R_{jm} to denote the $j = 1 \ldots J_m$ leaf nodes of the resulting tree in iteration m.

c. For each leaf node R_{jm}, we compute the following output value:

$$\gamma_{jm} = \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

In the next subsection, we will dive deeper into how this γ_{jm} is computed by minimizing the loss function. At this point, we can already note that leaf nodes R_{jm} may contain more than one training example, hence the summation.

d. Update the model by adding the output values γ_m to the previous tree:

$$F_m(x) = F_{m-1}(x) + \eta\gamma_m$$

However, instead of adding the full predicted values of the current tree γ_m to the previous tree F_{m-1}, we scale γ_m by a learning rate η, which is typically a small value between 0.01 and 1. In other words, we update the model incrementally by taking small steps, which helps avoid overfitting.

Now, after looking at the general structure of gradient boosting, we will adopt these mechanics to look at gradient boosting for classification.

Explaining the gradient boosting algorithm for classification

In this subsection, we will go over the details for implementing the gradient boosting algorithm for binary classification. In this context, we will be using the logistic loss function that we introduced for logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*. For a single training example, we can specify the logistic loss as follows:

$$L_i = -y_i \log p_i + (1 - y_i) \log(1 - p_i)$$

In *Chapter 3*, we also introduced the log(odds):

$$\hat{y} = \log(\text{odds}) = \log\left(\frac{p}{1-p}\right)$$

For reasons that will make sense later, we will use these log(odds) to rewrite the logistic function as follows (omitting intermediate steps here):

$$L_i = \log\left(1 + e^{\hat{y}_i}\right) - y_i\hat{y}_i$$

Now, we can define the partial derivative of the loss function with respect to these log(odds), $\hat{y}$. The derivative of this loss function with respect to the log(odds) is:

$$\frac{\partial L_i}{\partial \hat{y}_i} = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} - y_i = p_i - y_i$$

After specifying these mathematical definitions, let us now revisit the general gradient boosting *steps 1* to *2d* from the previous section and reformulate them for this binary classification scenario.

1. Create a root node that minimizes the logistic loss. It turns out that the loss is minimized if the root node returns the log(odds), $\hat{y}$.
2. For each tree $m = 1, ..., M$, where M is a user-specified number of total trees, we carry out the following computations outlined in *steps 2a* to *2d*:

    a. We convert the log(odds) into a probability using the familiar logistic function that we used in logistic regression (in *Chapter 3*):

    $$p = \frac{1}{1 + e^{-\hat{y}}}$$

    Then, we compute the pseudo-residual, which is the negative partial derivative of the loss with respect to the log(odds), which turns out to be the difference between the class label and the predicted probability:

    $$-\frac{\partial L_i}{\partial \hat{y}_i} = y_i - p_i$$

    b. Fit a new tree to the pseudo-residuals.

    c. For each leaf node R_{jm}, compute a value γ_{jm} that minimizes the logistic loss function. This includes a summarization step for dealing with leaf nodes that contain multiple training examples:

    $$\begin{aligned}\gamma_{jm} &= \arg\min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) \\ &= \log\left(1 + e^{\hat{y}_i + \gamma}\right) - y_i(\hat{y}_i + \gamma)\end{aligned}$$

    Skipping over intermediate mathematical details, this results in the following:

    $$\gamma_{jm} = \frac{\sum_i y_i - p_i}{\sum_i p_i(i - p_i)}$$

    Note that the summation here is only over the examples at the node corresponding to the leaf node R_{jm} and not the complete training set.

    d. Update the model by adding the gamma value from *step 2c* with learning rate η:

    $$F_m(x) = F_{m-1}(x) + \eta\gamma_m$$

Outputting log(odds) vs probabilities

Why do the trees return log(odds) values and not probabilities? This is because we cannot just add up probability values and arrive at a meaningful result. (So, technically speaking, gradient boosting for classification uses regression trees.)

In this section, we adopted the general gradient boosting algorithm and specified it for binary classification, for instance, by replacing the generic loss function with the logistic loss and the predicted values with the log(odds). However, many of the individual steps may still seem very abstract, and in the next section, we will apply these steps to a concrete example.

Illustrating gradient boosting for classification

The previous two subsections went over the condensed mathematical details of the gradient boosting algorithm for binary classification. To make these concepts clearer, let's apply it to a small toy example, that is, a training dataset of the following three examples shown in *Figure 7.12*:

	Feature x_1	Feature x_2	Class label y
1	1.12	1.4	1
2	2.45	2.1	0
3	3.54	1.2	1

Figure 7.12: Toy dataset for explaining gradient boosting

Let's start with *step 1*, constructing the root node and computing the log(odds), and *step 2a*, converting the log(odds) into class-membership probabilities and computing the pseudo-residuals. Note that based on what we have learned in *Chapter 3*, the odds can be computed as the number of successes divided by the number of failures. Here, we regard label 1 as success and label 0 as failure, so the odds are computed as: odds = 2/1. Carrying out steps *1* and *2a*, we get the following results shown in *Figure 7.13*:

	Feature x_1	Feature x_2	Class label y	Step 1: $\hat{y} = \log(\text{odds})$	Step 2a: $p = \frac{1}{1+e^{-\hat{y}}}$	Step 2a: $r = y - p$
1	1.12	1.4	1	0.69	0.67	0.33
2	2.45	2.1	0	0.69	0.67	-0.67
3	3.54	1.2	1	0.69	0.67	0.33

Figure 7.13: Results from the first round of applying step 1 and step 2a

Next, in *step 2b*, we fit a new tree on the pseudo-residuals r. Then, in *step 2c*, we compute the output values, γ, for this tree as shown in *Figure 7.14*:

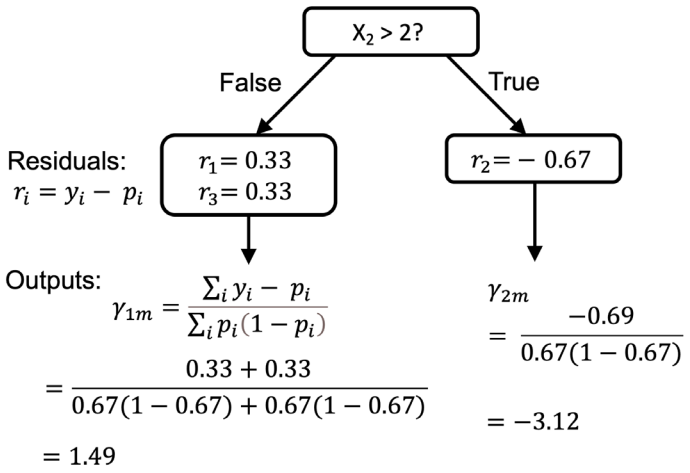
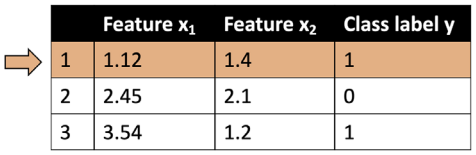


Figure 7.14: An illustration of steps 2b and 2c, which fits a tree to the residuals and computes the output values for each leaf node

(Note that we artificially limit the tree to have only two leaf nodes, which helps illustrate what happens if a leaf node contains more than one example.)

Then, in the final *step 2d*, we update the previous model and the current model. Assuming a learning rate of $\eta = 0.1$, the resulting prediction for the first training example is shown in *Figure 7.15*:



	Feature x_1	Feature x_2	Class label y
1	1.12	1.4	1
2	2.45	2.1	0
3	3.54	1.2	1

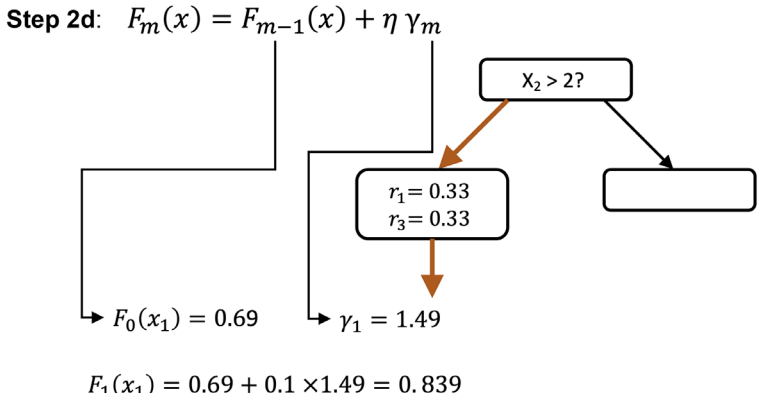


Figure 7.15: The update of the previous model shown in the context of the first training example

Now that we have completed *steps 2a* to *2d* of the first round, $m = 1$, we can proceed to execute *steps 2a* to *2d* for the second round, $m = 2$. In the second round, we use the log(odds) returned by the updated model, for example, $F_1(x_1) = 0.839$, as input to *step 2A*. The new values we obtain in the second round are shown in *Figure 7.16*:

	x_1	x_2	y	Step 1: $F_0(x) = \hat{y}$ = log(odds)	Step 2a: $p = \frac{1}{1+e^{-\hat{y}}}$	Step 2a: $r = y - p$	New log(odds) $\hat{y} = F_1(x)$	Step 2a: p	Step 2a: r
1	1.12	1.4	1	0.69	0.67	0.33	0.839	0.698	0.302
2	2.45	2.1	0	0.69	0.67	-0.67	0.378	0.593	-0.593
3	3.54	1.2	1	0.69	0.67	0.33	0.839	0.698	0.302

Round $m = 1$ Round $m = 2$

Figure 7.16: Values from the second round next to the values from the first round

We can already see that the predicted probabilities are higher for the positive class and lower for the negative class. Consequently, the residuals are getting smaller, too. Note that the process of *steps 2a* to *2d* is repeated until we have fit M trees or the residuals are smaller than a user-specified threshold value. Then, once the gradient boosting algorithm has completed, we can use it to predict the class labels by thresholding the probability values of the final model, $F_M(x)$ at 0.5, like logistic regression in *Chapter 3*. However, in contrast to logistic regression, gradient boosting consists of multiple trees and produces nonlinear decision boundaries. In the next section, we will look at how gradient boosting looks in action.

Using XGBoost

After covering the nitty-gritty details behind gradient boosting, let's finally look at how we can use gradient boosting code implementations.

In scikit-learn, gradient boosting is implemented as `sklearn.ensemble.GradientBoostingClassifier` (see `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html` for more details). It is important to note that gradient boosting is a sequential process that can be slow to train. However, in recent years a more popular implementation of gradient boosting has emerged, namely, XGBoost.

XGBoost proposed several tricks and approximations that speed up the training process substantially. Hence, the name XGBoost, which stands for extreme gradient boosting. Moreover, these approximations and tricks result in very good predictive performances. In fact, XGBoost gained popularity as it has been the winning solution for many Kaggle competitions.

Next to XGBoost, there are also other popular implementations of gradient boosting, for example, LightGBM and CatBoost. Inspired by LightGBM, scikit-learn now also implements a `HistGradientBoostingClassifier`, which is more performant than the original gradient boosting classifier (`GradientBoostingClassifier`).

You can find more details about these methods via the resources below:

- **XGBoost**: `https://xgboost.readthedocs.io/en/stable/`
- **LightGBM**: `https://lightgbm.readthedocs.io/en/latest/`
- **CatBoost**: `https://catboost.ai`
- **HistGradientBoostingClassifier**: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html`

However, since XGBoost is still among the most popular gradient boosting implementations, we will see how we can use it in practice. First, we need to install it, for example via `pip`:

```
pip install xgboost
```

Installing XGBoost

For this chapter, we used XGBoost version 1.5.0, which can be installed via:

```
pip install XGBoost==1.5.0
```

You can find more information about the installation details at `https://xgboost.readthedocs.io/en/stable/install.html`

Fortunately, XGBoost's `XGBClassifier` follows the scikit-learn API. So, using it is relatively straightforward:

```
>>> import xgboost as xgb
>>> model = xgb.XGBClassifier(n_estimators=1000, learning_rate=0.01,
...                           max_depth=4, random_state=1,
...                           use_label_encoder=False)

>>> gbm = model.fit(X_train, y_train)
>>> y_train_pred = gbm.predict(X_train)
>>> y_test_pred = gbm.predict(X_test)

>>> gbm_train = accuracy_score(y_train, y_train_pred)
>>> gbm_test = accuracy_score(y_test, y_test_pred)
>>> print(f'XGboost train/test accuracies '
...       f'{gbm_train:.3f}/{gbm_test:.3f}')
XGboost train/test accuracies 0.968/0.917
```

Here, we fit the gradient boosting classifier with 1,000 trees (rounds) and a learning rate of 0.01. Typically, a learning rate between 0.01 and 0.1 is recommended. However, remember that the learning rate is used for scaling the predictions from the individual rounds. So, intuitively, the lower the learning rate, the more estimators are required to achieve accurate predictions.

Next, we have the `max_depth` for the individual decision trees, which we set to 4. Since we are still boosting weak learners, a value between 2 and 6 is reasonable, but larger values may also work well depending on the dataset.

Finally, `use_label_encoder=False` disables a warning message which informs users that XGBoost is not converting labels by default anymore, and it expects users to provide labels in an integer format starting with label 0. (There is nothing to worry about here, since we have been following this format throughout this book.)

There are many more settings available, and a detailed discussion is out of the scope of this book. However, interested readers can find more details in the original documentation at `https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier`.

Summary

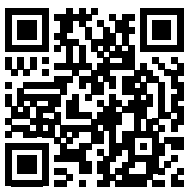
In this chapter, we looked at some of the most popular and widely used techniques for ensemble learning. Ensemble methods combine different classification models to cancel out their individual weaknesses, which often results in stable and well-performing models that are very attractive for industrial applications as well as machine learning competitions.

At the beginning of this chapter, we implemented `MajorityVoteClassifier` in Python, which allows us to combine different algorithms for classification. We then looked at bagging, a useful technique for reducing the variance of a model by drawing random bootstrap samples from the training dataset and combining the individually trained classifiers via majority vote. Lastly, we learned about boosting in the form of AdaBoost and gradient boosting, which are algorithms based on training weak learners that subsequently learn from mistakes.

Throughout the previous chapters, we learned a lot about different learning algorithms, tuning, and evaluation techniques. In the next chapter, we will look at a particular application of machine learning, sentiment analysis, which has become an interesting topic in the internet and social media era.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`



8

Applying Machine Learning to Sentiment Analysis

In the modern internet and social media age, people's opinions, reviews, and recommendations have become a valuable resource for political science and businesses. Thanks to modern technologies, we are now able to collect and analyze such data most efficiently. In this chapter, we will delve into a subfield of **natural language processing** (**NLP**) called **sentiment analysis** and learn how to use machine learning algorithms to classify documents based on their sentiment: the attitude of the writer. In particular, we are going to work with a dataset of 50,000 movie reviews from the **Internet Movie Database** (**IMDb**) and build a predictor that can distinguish between positive and negative reviews.

The topics that we will cover in this chapter include the following:

- Cleaning and preparing text data
- Building feature vectors from text documents
- Training a machine learning model to classify positive and negative movie reviews
- Working with large text datasets using out-of-core learning
- Inferring topics from document collections for categorization

Preparing the IMDb movie review data for text processing

As mentioned, sentiment analysis, sometimes also called **opinion mining**, is a popular subdiscipline of the broader field of NLP; it is concerned with analyzing the sentiment of documents. A popular task in sentiment analysis is the classification of documents based on the expressed opinions or emotions of the authors with regard to a particular topic.

In this chapter, we will be working with a large dataset of movie reviews from IMDb that has been collected by Andrew Maas and others (*Learning Word Vectors for Sentiment Analysis* by *A. L. Maas*, *R. E. Daly*, *P. T. Pham*, *D. Huang*, *A. Y. Ng*, and *C. Potts*, *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA, Association for Computational Linguistics, June 2011). The movie review dataset consists of 50,000 polar movie reviews that are labeled as either positive or negative; here, positive means that a movie was rated with more than six stars on IMDb, and negative means that a movie was rated with fewer than five stars on IMDb. In the following sections, we will download the dataset, preprocess it into a useable format for machine learning tools, and extract meaningful information from a subset of these movie reviews to build a machine learning model that can predict whether a certain reviewer liked or disliked a movie.

Obtaining the movie review dataset

A compressed archive of the movie review dataset (84.1 MB) can be downloaded from `http://ai.stanford.edu/~amaas/data/sentiment/` as a gzip-compressed tarball archive:

- If you are working with Linux or macOS, you can open a new terminal window, `cd` into the download directory, and execute `tar -zxf aclImdb_v1.tar.gz` to decompress the dataset.
- If you are working with Windows, you can download a free archiver, such as 7-Zip (`http://www.7-zip.org`), to extract the files from the download archive.
- Alternatively, you can unpack the gzip-compressed tarball archive directly in Python as follows:

```
>>> import tarfile
>>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:
...     tar.extractall()
```

Preprocessing the movie dataset into a more convenient format

Having successfully extracted the dataset, we will now assemble the individual text documents from the decompressed download archive into a single CSV file. In the following code section, we will be reading the movie reviews into a pandas `DataFrame` object, which can take up to 10 minutes on a standard desktop computer.

To visualize the progress and estimated time until completion, we will use the **Python Progress Indicator** (**PyPrind**, `https://pypi.python.org/pypi/PyPrind/`) package, which was developed several years ago for such purposes. PyPrind can be installed by executing the `pip install pyprind` command:

```
>>> import pyprind
>>> import pandas as pd
>>> import os
>>> import sys
>>> # change the 'basepath' to the directory of the
>>> # unzipped movie dataset
>>> basepath = 'aclImdb'
```

```
>>>
>>> labels = {'pos': 1, 'neg': 0}
>>> pbar = pyprind.ProgBar(50000, stream=sys.stdout)
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path = os.path.join(basepath, s, l)
...         for file in sorted(os.listdir(path)):
...             with open(os.path.join(path, file),
...                       'r', encoding='utf-8') as infile:
...                 txt = infile.read()
...             df = df.append([[txt, labels[l]]],
...                            ignore_index=True)
...             pbar.update()
>>> df.columns = ['review', 'sentiment']
0%                          100%
[##############################] | ETA: 00:00:00
Total time elapsed: 00:00:25
```

In the preceding code, we first initialized a new progress bar object, `pbar`, with 50,000 iterations, which was the number of documents we were going to read in. Using the nested `for` loops, we iterated over the `train` and `test` subdirectories in the main `aclImdb` directory and read the individual text files from the `pos` and `neg` subdirectories that we eventually appended to the `df` pandas `DataFrame`, together with an integer class label (1 = positive and 0 = negative).

Since the class labels in the assembled dataset are sorted, we will now shuffle the `DataFrame` using the `permutation` function from the `np.random` submodule—this will be useful for splitting the dataset into training and test datasets in later sections, when we will stream the data from our local drive directly.

For our own convenience, we will also store the assembled and shuffled movie review dataset as a CSV file:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

Since we are going to use this dataset later in this chapter, let's quickly confirm that we have successfully saved the data in the right format by reading in the CSV and printing an excerpt of the first three examples:

```
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>>> # the following column renaming is necessary on some computers:
>>> df = df.rename(columns={"0": "review", "1": "sentiment"})
>>> df.head(3)
```

If you are running the code examples in a Jupyter notebook, you should now see the first three examples of the dataset, as shown in *Figure 8.1*:

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Figure 8.1: The first three rows of the movie review dataset

As a sanity check, before we proceed to the next section, let's make sure that the `DataFrame` contains all 50,000 rows:

```
>>> df.shape
(50000, 2)
```

Introducing the bag-of-words model

You may remember from *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, that we have to convert categorical data, such as text or words, into a numerical form before we can pass it on to a machine learning algorithm. In this section, we will introduce the **bag-of-words** model, which allows us to represent text as numerical feature vectors. The idea behind bag-of-words is quite simple and can be summarized as follows:

1. We create a vocabulary of unique tokens—for example, words—from the entire set of documents.
2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

Since the unique words in each document represent only a small subset of all the words in the bag-of-words vocabulary, the feature vectors will mostly consist of zeros, which is why we call them **sparse**. Do not worry if this sounds too abstract; in the following subsections, we will walk through the process of creating a simple bag-of-words model step by step.

Transforming words into feature vectors

To construct a bag-of-words model based on the word counts in the respective documents, we can use the `CountVectorizer` class implemented in scikit-learn. As you will see in the following code section, `CountVectorizer` takes an array of text data, which can be documents or sentences, and constructs the bag-of-words model for us:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array(['The sun is shining',
```

```
...                 'The weather is sweet',
...                 'The sun is shining, the weather is sweet,'
...                 'and one and one is two'])
>>> bag = count.fit_transform(docs)
```

By calling the `fit_transform` method on `CountVectorizer`, we constructed the vocabulary of the bag-of-words model and transformed the following three sentences into sparse feature vectors:

- `'The sun is shining'`
- `'The weather is sweet'`
- `'The sun is shining, the weather is sweet, and one and one is two'`

Now, let's print the contents of the vocabulary to get a better understanding of the underlying concepts:

```
>>> print(count.vocabulary_)
{'and': 0,
'two': 7,
'shining': 3,
'one': 2,
'sun': 4,
'weather': 8,
'the': 6,
'sweet': 5,
'is': 1}
```

As you can see from executing the preceding command, the vocabulary is stored in a Python dictionary that maps the unique words to integer indices. Next, let's print the feature vectors that we just created:

```
>>> print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Each index position in the feature vectors shown here corresponds to the integer values that are stored as dictionary items in the `CountVectorizer` vocabulary. For example, the first feature at index position `0` resembles the count of the word `'and'`, which only occurs in the last document, and the word `'is'`, at index position `1` (the second feature in the document vectors), occurs in all three sentences. These values in the feature vectors are also called the **raw term frequencies**: $tf(t, d)$—the number of times a term, t, occurs in a document, d. It should be noted that, in the bag-of-words model, the word or term order in a sentence or document does not matter. The order in which the term frequencies appear in the feature vector is derived from the vocabulary indices, which are usually assigned alphabetically.

N-gram models

The sequence of items in the bag-of-words model that we just created is also called the 1-gram or unigram model—each item or token in the vocabulary represents a single word. More generally, the contiguous sequences of items in NLP—words, letters, or symbols—are also called *n-grams*. The choice of the number, *n*, in the n-gram model depends on the particular application; for example, a study by Ioannis Kanaris and others revealed that n-grams of size 3 and 4 yield good performances in the anti-spam filtering of email messages (*Words versus character n-grams for anti-spam filtering* by *Ioannis Kanaris*, *Konstantinos Kanaris*, *Ioannis Houvardas*, and *Efstathios Stamatatos*, *International Journal on Artificial Intelligence Tools*, *World Scientific Publishing Company*, 16(06): 1047-1067, 2007).

To summarize the concept of the n-gram representation, the 1-gram and 2-gram representations of our first document, "the sun is shining", would be constructed as follows:

- 1-gram: "the", "sun", "is", "shining"
- 2-gram: "the sun", "sun is", "is shining"

The `CountVectorizer` class in scikit-learn allows us to use different n-gram models via its `ngram_range` parameter. While a 1-gram representation is used by default, we could switch to a 2-gram representation by initializing a new `CountVectorizer` instance with `ngram_range=(2,2)`.

Assessing word relevancy via term frequency-inverse document frequency

When we are analyzing text data, we often encounter words that occur across multiple documents from both classes. These frequently occurring words typically don't contain useful or discriminatory information. In this subsection, you will learn about a useful technique called the **term frequency-inverse document frequency** (**tf-idf**), which can be used to downweight these frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the term frequency and the inverse document frequency:

$$tf\text{-}idf(t, d) = tf(t, d) \times idf(t, d)$$

Here, $tf(t, d)$ is the term frequency that we introduced in the previous section, and $idf(t, d)$ is the inverse document frequency, which can be calculated as follows:

$$idf(t, d) = \log \frac{n_d}{1 + df(d, t)}$$

Here, n_d is the total number of documents, and $df(d, t)$ is the number of documents, d, that contain the term t. Note that adding the constant 1 to the denominator is optional and serves the purpose of assigning a non-zero value to terms that occur in none of the training examples; the *log* is used to ensure that low document frequencies are not given too much weight.

The scikit-learn library implements yet another transformer, the `TfidfTransformer` class, which takes the raw term frequencies from the `CountVectorizer` class as input and transforms them into tf-idfs:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer(use_idf=True,
...                          norm='l2',
...                          smooth_idf=True)
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs))
...       .toarray())
[[ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.  ]
 [ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
 [ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

As you saw in the previous subsection, the word `'is'` had the largest term frequency in the third document, being the most frequently occurring word. However, after transforming the same feature vector into tf-idfs, the word `'is'` is now associated with a relatively small tf-idf (0.45) in the third document, since it is also present in the first and second document and thus is unlikely to contain any useful discriminatory information.

However, if we'd manually calculated the tf-idfs of the individual terms in our feature vectors, we would have noticed that `TfidfTransformer` calculates the tf-idfs slightly differently compared to the standard textbook equations that we defined previously. The equation for the inverse document frequency implemented in scikit-learn is computed as follows:

$$idf(t,d) = \log\frac{1+n_d}{1+df(d,t)}$$

Similarly, the tf-idf computed in scikit-learn deviates slightly from the default equation we defined earlier:

$$tf\text{-}idf(t,\, d) = tf(t,\, d) \times (idf(t,\, d) + 1)$$

Note that the "+1" in the previous idf equation is due to setting `smooth_idf=True` in the previous code example, which is helpful for assigning zero weight (that is, $idf(t,\, d) = \log(1) = 0$) to terms that occur in all documents.

While it is also more typical to normalize the raw term frequencies before calculating the tf-idfs, the `TfidfTransformer` class normalizes the tf-idfs directly. By default (`norm='l2'`), scikit-learn's `TfidfTransformer` applies the L2-normalization, which returns a vector of length 1 by dividing an unnormalized feature vector, v, by its L2-norm:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \cdots + v_n^2}} = \frac{v}{(\sum_{i=1}^{n} v_i^2)^{1/2}}$$

To make sure that we understand how `TfidfTransformer` works, let's walk through an example and calculate the tf-idf of the word `'is'` in the third document. The word `'is'` has a term frequency of 3 (*tf* = 3) in the third document, and the document frequency of this term is 3 since the term `'is'` occurs in all three documents (*df* = 3). Thus, we can calculate the inverse document frequency as follows:

$$idf(\text{"is"}, d_3) = \log\frac{1+3}{1+3} = 0$$

Now, in order to calculate the tf-idf, we simply need to add 1 to the inverse document frequency and multiply it by the term frequency:

$$tf\text{-}idf(\text{"is"}, d_3) = 3 \times (0 + 1) = 3$$

If we repeated this calculation for all terms in the third document, we'd obtain the following tf-idf vectors: `[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]`. However, notice that the values in this feature vector are different from the values that we obtained from `TfidfTransformer` that we used previously. The final step that we are missing in this tf-idf calculation is the L2-normalization, which can be applied as follows:

$$\begin{aligned} tf\text{-}idf(d_3)_{norm} &= \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}} \\ &= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19] \\ \\ tf\text{-}idf(\text{"is"}, d_3) &= 0.45 \end{aligned}$$

As you can see, the results now match the results returned by scikit-learn's `TfidfTransformer`, and since you now understand how tf-idfs are calculated, let's proceed to the next section and apply those concepts to the movie review dataset.

Cleaning text data

In the previous subsections, we learned about the bag-of-words model, term frequencies, and tf-idfs. However, the first important step—before we build our bag-of-words model—is to clean the text data by stripping it of all unwanted characters.

To illustrate why this is important, let's display the last 50 characters from the first document in the reshuffled movie review dataset:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

As you can see here, the text contains HTML markup as well as punctuation and other non-letter characters. While HTML markup does not contain many useful semantics, punctuation marks can represent useful, additional information in certain NLP contexts. However, for simplicity, we will now remove all punctuation marks except for emoticon characters, such as :), since those are certainly useful for sentiment analysis.

To accomplish this task, we will use Python's **regular expression** (**regex**) library, `re`, as shown here:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text)
...     text = (re.sub('[\W]+', ' ', text.lower()) +
...             ' '.join(emoticons).replace('-', ''))
...     return text
```

Via the first regex, `<[^>]*>`, in the preceding code section, we tried to remove all of the HTML markup from the movie reviews. Although many programmers generally advise against the use of regex to parse HTML, this regex should be sufficient to *clean* this particular dataset. Since we are only interested in removing HTML markup and do not plan to use the HTML markup further, using regex to do the job should be acceptable. However, if you prefer to use sophisticated tools for removing HTML markup from text, you can take a look at Python's HTML parser module, which is described at `https://docs.python.org/3/library/html.parser.html`. After we removed the HTML markup, we used a slightly more complex regex to find emoticons, which we temporarily stored as emoticons. Next, we removed all non-word characters from the text via the regex `[\W]+` and converted the text into lowercase characters.

Dealing with word capitalization

In the context of this analysis, we assume that the capitalization of a word—for example, whether it appears at the beginning of a sentence—does not contain semantically relevant information. However, note that there are exceptions; for instance, we remove the notation of proper names. But again, in the context of this analysis, it is a simplifying assumption that the letter case does not contain information that is relevant for sentiment analysis.

Eventually, we added the temporarily stored emoticons to the end of the processed document string. Additionally, we removed the *nose* character (- in :-)) from the emoticons for consistency.

Regular expressions

Although regular expressions offer an efficient and convenient approach to searching for characters in a string, they also come with a steep learning curve. Unfortunately, an in-depth discussion of regular expressions is beyond the scope of this book. However, you can find a great tutorial on the Google Developers portal at `https://developers.google.com/edu/python/regular-expressions` or you can check out the official documentation of Python's `re` module at `https://docs.python.org/3.9/library/re.html`.

Although the addition of the emoticon characters to the end of the cleaned document strings may not look like the most elegant approach, we must note that the order of the words doesn't matter in our bag-of-words model if our vocabulary consists of only one-word tokens. But before we talk more about the splitting of documents into individual terms, words, or tokens, let's confirm that our `preprocessor` function works correctly:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Lastly, since we will make use of the *cleaned* text data over and over again during the next sections, let's now apply our `preprocessor` function to all the movie reviews in our `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

Processing documents into tokens

After successfully preparing the movie review dataset, we now need to think about how to split the text corpora into individual elements. One way to *tokenize* documents is to split them into individual words by splitting the cleaned documents at their whitespace characters:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

In the context of tokenization, another useful technique is **word stemming**, which is the process of transforming a word into its root form. It allows us to map related words to the same stem. The original stemming algorithm was developed by Martin F. Porter in 1979 and is hence known as the **Porter stemmer** algorithm (*An algorithm for suffix stripping* by *Martin F. Porter*, *Program: Electronic Library and Information Systems*, 14(3): 130–137, 1980). The **Natural Language Toolkit** (**NLTK**, `http://www.nltk.org`) for Python implements the Porter stemming algorithm, which we will use in the following code section. To install the NLTK, you can simply execute `conda install nltk` or `pip install nltk`.

NLTK online book

Although the NLTK is not the focus of this chapter, I highly recommend that you visit the NLTK website as well as read the official NLTK book, which is freely available at `http://www.nltk.org/book/`, if you are interested in more advanced applications in NLP.

The following code shows how to use the Porter stemming algorithm:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
```

```
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Using the `PorterStemmer` from the `nltk` package, we modified our `tokenizer` function to reduce words to their root form, which was illustrated by the simple preceding example where the word `'running'` was *stemmed* to its root form `'run'`.

Stemming algorithms

The Porter stemming algorithm is probably the oldest and simplest stemming algorithm. Other popular stemming algorithms include the newer Snowball stemmer (Porter2 or English stemmer) and the Lancaster stemmer (Paice/Husk stemmer). While both the Snowball and Lancaster stemmers are faster than the original Porter stemmer, the Lancaster stemmer is also notorious for being more aggressive than the Porter stemmer, which means that it will produce shorter and more obscure words. These alternative stemming algorithms are also available through the NLTK package (`http://www.nltk.org/api/nltk.stem.html`).

While stemming can create non-real words, such as `'thu'` (from `'thus'`), as shown in the previous example, a technique called *lemmatization* aims to obtain the canonical (grammatically correct) forms of individual words—the so-called *lemmas*. However, lemmatization is computationally more difficult and expensive compared to stemming and, in practice, it has been observed that stemming and lemmatization have little impact on the performance of text classification (*Influence of Word Normalization on Text Classification*, by *Michal Toman*, *Roman Tesar*, and *Karel Jezek*, *Proceedings of InSciT*, pages 354–358, 2006).

Before we jump into the next section, where we will train a machine learning model using the bag-of-words model, let's briefly talk about another useful topic called **stop word removal**. Stop words are simply those words that are extremely common in all sorts of texts and probably bear no (or only a little) useful information that can be used to distinguish between different classes of documents. Examples of stop words are *is*, *and*, *has*, and *like*. Removing stop words can be useful if we are working with raw or normalized term frequencies rather than tf-idfs, which already downweight the frequently occurring words.

To remove stop words from the movie reviews, we will use the set of 127 English stop words that is available from the NLTK library, which can be obtained by calling the `nltk.download` function:

```
>>> import nltk
>>> nltk.download('stopwords')
```

After we download the stop words set, we can load and apply the English stop word set as follows:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> [w for w in tokenizer_porter('a runner likes'
...  ' running and runs a lot')
...  if w not in stop]
['runner', 'like', 'run', 'run', 'lot']
```

Training a logistic regression model for document classification

In this section, we will train a logistic regression model to classify the movie reviews into *positive* and *negative* reviews based on the bag-of-words model. First, we will divide the `DataFrame` of cleaned text documents into 25,000 documents for training and 25,000 documents for testing:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Next, we will use a `GridSearchCV` object to find the optimal set of parameters for our logistic regression model using 5-fold stratified cross-validation:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> small_param_grid = [
...     {
...         'vect__ngram_range': [(1, 1)],
...         'vect__stop_words': [None],
...         'vect__tokenizer': [tokenizer, tokenizer_porter],
...         'clf__penalty': ['l2'],
...         'clf__C': [1.0, 10.0]
...     },
...     {
...         'vect__ngram_range': [(1, 1)],
...         'vect__stop_words': [stop, None],
...         'vect__tokenizer': [tokenizer],
```

```
...         'vect__use_idf':[False],
...         'vect__norm':[None],
...         'clf__penalty': ['l2'],
...         'clf__C': [1.0, 10.0]
...     },
... ]
>>> lr_tfidf = Pipeline([
...     ('vect', tfidf),
...     ('clf', LogisticRegression(solver='liblinear'))
... ])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, small_param_grid,
...                            scoring='accuracy', cv=5,
...                            verbose=2, n_jobs=1)
>>> gs_lr_tfidf.fit(X_train, y_train)
```

Note that for the logistic regression classifier, we are using the LIBLINEAR solver as it can perform better than the default choice (`'lbfgs'`) for relatively large datasets.

Multiprocessing via the n_jobs parameter

Please note that we highly recommend setting `n_jobs=-1` (instead of `n_jobs=1`, as in the previous code example) to utilize all available cores on your machine and speed up the grid search. However, some Windows users reported issues when running the previous code with the `n_jobs=-1` setting related to pickling the `tokenizer` and `tokenizer_porter` functions for multiprocessing on Windows. Another workaround would be to replace those two functions, `[tokenizer, tokenizer_porter]`, with `[str.split]`. However, note that replacement by the simple `str.split` would not support stemming.

When we initialized the `GridSearchCV` object and its parameter grid using the preceding code, we restricted ourselves to a limited number of parameter combinations, since the number of feature vectors, as well as the large vocabulary, can make the grid search computationally quite expensive. Using a standard desktop computer, our grid search may take 5-10 minutes to complete.

In the previous code example, we replaced `CountVectorizer` and `TfidfTransformer` from the previous subsection with `TfidfVectorizer`, which combines `CountVectorizer` with the `TfidfTransformer`. Our `param_grid` consisted of two parameter dictionaries. In the first dictionary, we used `TfidfVectorizer` with its default settings (`use_idf=True`, `smooth_idf=True`, and `norm='l2'`) to calculate the tf-idfs; in the second dictionary, we set those parameters to `use_idf=False`, `smooth_idf=False`, and `norm=None` in order to train a model based on raw term frequencies. Furthermore, for the logistic regression classifier itself, we trained models using L2 regularization via the penalty parameter and compared different regularization strengths by defining a range of values for the inverse-regularization parameter `C`. As an optional exercise, you are also encouraged to add L1 regularization to the parameter grid by changing `'clf__penalty': ['l2']` to `'clf__penalty': ['l2', 'l1']`.

After the grid search has finished, we can print the best parameter set:

```
>>> print(f'Best parameter set: {gs_lr_tfidf.best_params_}')
Best parameter set: {'clf__C': 10.0, 'clf__penalty': 'l2', 'vect__ngram_range':
(1, 1), 'vect__stop_words': None, 'vect__tokenizer': <function tokenizer at
0x169932dc0>}
```

As you can see in the preceding output, we obtained the best grid search results using the regular `tokenizer` without Porter stemming, no stop word library, and tf-idfs in combination with a logistic regression classifier that uses L2-regularization with the regularization strength `C` of `10.0`.

Using the best model from this grid search, let's print the average 5-fold cross-validation accuracy scores on the training dataset and the classification accuracy on the test dataset:

```
>>> print(f'CV Accuracy: {gs_lr_tfidf.best_score_:.3f}')
CV Accuracy: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print(f'Test Accuracy: {clf.score(X_test, y_test):.3f}')
Test Accuracy: 0.899
```

The results reveal that our machine learning model can predict whether a movie review is positive or negative with 90 percent accuracy.

The naïve Bayes classifier

A still very popular classifier for text classification is the naïve Bayes classifier, which gained popularity in applications of email spam filtering. Naïve Bayes classifiers are easy to implement, computationally efficient, and tend to perform particularly well on relatively small datasets compared to other algorithms. Although we don't discuss naïve Bayes classifiers in this book, the interested reader can find an article about naïve Bayes text classification that is freely available on arXiv (*Naive Bayes and Text Classification I – Introduction and Theory* by *S. Raschka*, *Computing Research Repository* (*CoRR*), abs/1410.5329, 2014, `http://arxiv.org/pdf/1410.5329v3.pdf`). Different versions of naïve Bayes classifiers referenced in this article are implemented in scikit-learn. You can find an overview page with links to the respective code classes here: `https://scikit-learn.org/stable/modules/naive_bayes.html`.

Working with bigger data – online algorithms and out-of-core learning

If you executed the code examples in the previous section, you may have noticed that it could be computationally quite expensive to construct the feature vectors for the 50,000-movie review dataset during a grid search. In many real-world applications, it is not uncommon to work with even larger datasets that can exceed our computer's memory.

Since not everyone has access to supercomputer facilities, we will now apply a technique called **out-of-core learning**, which allows us to work with such large datasets by fitting the classifier incrementally on smaller batches of a dataset.

Text classification with recurrent neural networks

In *Chapter 15, Modeling Sequential Data Using Recurrent Neural Networks*, we will revisit this dataset and train a deep learning-based classifier (a recurrent neural network) to classify the reviews in the IMDb movie review dataset. This neural network-based classifier follows the same out-of-core principle using the stochastic gradient descent optimization algorithm, but does not require the construction of a bag-of-words model.

Back in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*, the concept of **stochastic gradient descent** was introduced; it is an optimization algorithm that updates the model's weights using one example at a time. In this section, we will make use of the `partial_fit` function of `SGDClassifier` in scikit-learn to stream the documents directly from our local drive and train a logistic regression model using small mini-batches of documents.

First, we will define a `tokenizer` function that cleans the unprocessed text data from the `movie_data.csv` file that we constructed at the beginning of this chapter and separates it into word tokens while removing stop words:

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(?::|;|=)(?:-)?(?:\)|\(|D|P)',
...                            text)
...     text = re.sub('[\W]+', ' ', text.lower()) \
...                   + ' '.join(emoticons).replace('-', '')
...     tokenized = [w for w in text.split() if w not in stop]
...     return tokenized
```

Next, we will define a generator function, `stream_docs`, that reads in and returns one document at a time:

```
>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # skip header
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label
```

To verify that our `stream_docs` function works correctly, let's read in the first document from the `movie_data.csv` file, which should return a tuple consisting of the review text as well as the corresponding class label:

```
>>> next(stream_docs(path='movie_data.csv'))
('"In 1974, the teenager Martha Moxley ... ',1)
```

We will now define a function, `get_minibatch`, that will take a document stream from the `stream_docs` function and return a particular number of documents specified by the `size` parameter:

```
>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     try:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(label)
...     except StopIteration:
...         return None, None
...     return docs, y
```

Unfortunately, we can't use `CountVectorizer` for out-of-core learning since it requires holding the complete vocabulary in memory. Also, `TfidfVectorizer` needs to keep all the feature vectors of the training dataset in memory to calculate the inverse document frequencies. However, another useful vectorizer for text processing implemented in scikit-learn is `HashingVectorizer`. `HashingVectorizer` is data-independent and makes use of the hashing trick via the 32-bit `MurmurHash3` function by Austin Appleby (you can find more information about MurmurHash at `https://en.wikipedia.org/wiki/MurmurHash`):

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier
>>> vect = HashingVectorizer(decode_error='ignore',
...                          n_features=2**21,
...                          preprocessor=None,
...                          tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1)
>>> doc_stream = stream_docs(path='movie_data.csv')
```

Using the preceding code, we initialized `HashingVectorizer` with our `tokenizer` function and set the number of features to `2**21`. Furthermore, we reinitialized a logistic regression classifier by setting the `loss` parameter of `SGDClassifier` to `'log'`. Note that by choosing a large number of features in `HashingVectorizer`, we reduce the chance of causing hash collisions, but we also increase the number of coefficients in our logistic regression model.

Now comes the really interesting part—having set up all the complementary functions, we can start the out-of-core learning using the following code:

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                          100%
[##############################] | ETA: 00:00:00
Total time elapsed: 00:00:21
```

Again, we made use of the PyPrind package to estimate the progress of our learning algorithm. We initialized the progress bar object with 45 iterations and, in the following `for` loop, we iterated over 45 mini-batches of documents where each mini-batch consists of 1,000 documents. Having completed the incremental learning process, we will use the last 5,000 documents to evaluate the performance of our model:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print(f'Accuracy: {clf.score(X_test, y_test):.3f}')
Accuracy: 0.868
```

NoneType error

Please note that if you encounter a `NoneType` error, you may have executed the `X_test, y_test = get_minibatch(...)` code twice. Via the previous loop, we have 45 iterations where we fetch 1,000 documents each. Hence, there are exactly 5,000 documents left for testing, which we assign via:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
```

If we execute this code twice, then there are not enough documents left in the generator, and `X_test` returns `None`. Hence, if you encounter the `NoneType` error, you have to start at the previous `stream_docs(...)` code again.

As you can see, the accuracy of the model is approximately 87 percent, slightly below the accuracy that we achieved in the previous section using the grid search for hyperparameter tuning. However, out-of-core learning is very memory efficient, and it took less than a minute to complete.

Finally, we can use the last 5,000 documents to update our model:

```
>>> clf = clf.partial_fit(X_test, y_test)
```

The word2vec model

A more modern alternative to the bag-of-words model is word2vec, an algorithm that Google released in 2013 (*Efficient Estimation of Word Representations in Vector Space* by *T. Mikolov*, *K. Chen*, *G. Corrado*, and *J. Dean*, `https://arxiv.org/abs/1301.3781`).

The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters, and via clever vector spacing, the model can reproduce certain words using simple vector math, for example, *king – man + woman = queen*.

The original C-implementation with useful links to the relevant papers and alternative implementations can be found at `https://code.google.com/p/word2vec/`.

Topic modeling with latent Dirichlet allocation

Topic modeling describes the broad task of assigning topics to unlabeled text documents. For example, a typical application is the categorization of documents in a large text corpus of newspaper articles. In applications of topic modeling, we then aim to assign category labels to those articles, for example, sports, finance, world news, politics, and local news. Thus, in the context of the broad categories of machine learning that we discussed in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, we can consider topic modeling as a clustering task, a subcategory of unsupervised learning.

In this section, we will discuss a popular technique for topic modeling called **latent Dirichlet allocation** (**LDA**). However, note that while latent Dirichlet allocation is often abbreviated as LDA, it is not to be confused with *linear discriminant analysis*, a supervised dimensionality reduction technique that was introduced in *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

Decomposing text documents with LDA

Since the mathematics behind LDA is quite involved and requires knowledge of Bayesian inference, we will approach this topic from a practitioner's perspective and interpret LDA using layman's terms. However, the interested reader can read more about LDA in the following research paper: *Latent Dirichlet Allocation*, by *David M. Blei*, *Andrew Y. Ng*, and *Michael I. Jordan*, *Journal of Machine Learning Research 3*, pages: 993-1022, Jan 2003, `https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf`.

LDA is a generative probabilistic model that tries to find groups of words that appear frequently together across different documents. These frequently appearing words represent our topics, assuming that each document is a mixture of different words. The input to an LDA is the bag-of-words model that we discussed earlier in this chapter.

Given a bag-of-words matrix as input, LDA decomposes it into two new matrices:

- A document-to-topic matrix
- A word-to-topic matrix

LDA decomposes the bag-of-words matrix in such a way that if we multiply those two matrices together, we will be able to reproduce the input, the bag-of-words matrix, with the lowest possible error. In practice, we are interested in those topics that LDA found in the bag-of-words matrix. The only downside may be that we must define the number of topics beforehand—the number of topics is a hyperparameter of LDA that has to be specified manually.

LDA with scikit-learn

In this subsection, we will use the `LatentDirichletAllocation` class implemented in scikit-learn to decompose the movie review dataset and categorize it into different topics. In the following example, we will restrict the analysis to 10 different topics, but readers are encouraged to experiment with the hyperparameters of the algorithm to further explore the topics that can be found in this dataset.

First, we are going to load the dataset into a pandas `DataFrame` using the local `movie_data.csv` file of the movie reviews that we created at the beginning of this chapter:

```
>>> import pandas as pd
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>>> # the following is necessary on some computers:
>>> df = df.rename(columns={"0": "review", "1": "sentiment"})
```

Next, we are going to use the already familiar `CountVectorizer` to create the bag-of-words matrix as input to the LDA.

For convenience, we will use scikit-learn's built-in English stop word library via `stop_words='english'`:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer(stop_words='english',
...                         max_df=.1,
...                         max_features=5000)
>>> X = count.fit_transform(df['review'].values)
```

Notice that we set the maximum document frequency of words to be considered to 10 percent (`max_df=.1`) to exclude words that occur too frequently across documents. The rationale behind the removal of frequently occurring words is that these might be common words appearing across all documents that are, therefore, less likely to be associated with a specific topic category of a given document. Also, we limited the number of words to be considered to the most frequently occurring 5,000 words (`max_features=5000`), to limit the dimensionality of this dataset to improve the inference performed by LDA. However, both `max_df=.1` and `max_features=5000` are hyperparameter values chosen arbitrarily, and readers are encouraged to tune them while comparing the results.

The following code example demonstrates how to fit a `LatentDirichletAllocation` estimator to the bag-of-words matrix and infer the 10 different topics from the documents (note that the model fitting can take up to 5 minutes or more on a laptop or standard desktop computer):

```
>>> from sklearn.decomposition import LatentDirichletAllocation
>>> lda = LatentDirichletAllocation(n_components=10,
...                                 random_state=123,
...                                 learning_method='batch')
>>> X_topics = lda.fit_transform(X)
```

By setting `learning_method='batch'`, we let the `lda` estimator do its estimation based on all available training data (the bag-of-words matrix) in one iteration, which is slower than the alternative `'online'` learning method, but can lead to more accurate results (setting `learning_method='online'` is analogous to online or mini-batch learning, which we discussed in *Chapter 2, Training Simple Machine Learning Algorithms for Classification*, and previously in this chapter).

Expectation-maximization

The scikit-learn library's implementation of LDA uses the **expectation-maximization** (**EM**) algorithm to update its parameter estimates iteratively. We haven't discussed the EM algorithm in this chapter, but if you are curious to learn more, please see the excellent overview on Wikipedia (`https://en.wikipedia.org/wiki/Expectation-maximization_algorithm`) and the detailed tutorial on how it is used in LDA in Colorado Reed's tutorial, *Latent Dirichlet Allocation: Towards a Deeper Understanding*, which is freely available at `http://obphio.us/pdfs/lda_tutorial.pdf`.

After fitting the LDA, we now have access to the `components_` attribute of the `lda` instance, which stores a matrix containing the word importance (here, `5000`) for each of the 10 topics in increasing order:

```
>>> lda.components_.shape
(10, 5000)
```

To analyze the results, let's print the five most important words for each of the 10 topics. Note that the word importance values are ranked in increasing order. Thus, to print the top five words, we need to sort the topic array in reverse order:

```
>>> n_top_words = 5
>>> feature_names = count.get_feature_names_out()
>>> for topic_idx, topic in enumerate(lda.components_):
...     print(f'Topic {(topic_idx + 1)}:')
...     print(' '.join([feature_names[i]
...                     for i in topic.argsort()\
...                     [:-n_top_words - 1:-1]]))
Topic 1:
```

```
worst minutes awful script stupid
Topic 2:
family mother father children girl
Topic 3:
american war dvd music tv
Topic 4:
human audience cinema art sense
Topic 5:
police guy car dead murder
Topic 6:
horror house sex girl woman
Topic 7:
role performance comedy actor performances
Topic 8:
series episode war episodes tv
Topic 9:
book version original read novel
Topic 10:
action fight guy guys cool
```

Based on reading the five most important words for each topic, you may guess that the LDA identified the following topics:

1. Generally bad movies (not really a topic category)
2. Movies about families
3. War movies
4. Art movies
5. Crime movies
6. Horror movies
7. Comedy movie reviews
8. Movies somehow related to TV shows
9. Movies based on books
10. Action movies

To confirm that the categories make sense based on the reviews, let's plot three movies from the horror movie category (horror movies belong to category 6 at index position 5):

```
>>> horror = X_topics[:, 5].argsort()[::-1]
>>> for iter_idx, movie_idx in enumerate(horror[:3]):
...     print(f'\nHorror movie #{(iter_idx + 1)}:')
...     print(df['review'][movie_idx][:300], '...')
Horror movie #1:
House of Dracula works from the same basic premise as House of Frankenstein
from the year before; namely that Universal's three most famous monsters;
Dracula, Frankenstein's Monster and The Wolf Man are appearing in the movie
together. Naturally, the film is rather messy therefore, but the fact that ...
```

```
Horror movie #2:
Okay, what the hell kind of TRASH have I been watching now? "The Witches'
Mountain" has got to be one of the most incoherent and insane Spanish
exploitation flicks ever and yet, at the same time, it's also strangely
compelling. There's absolutely nothing that makes sense here and I even doubt
there ...
Horror movie #3:
<br /><br />Horror movie time, Japanese style. Uzumaki/Spiral was a total
freakfest from start to finish. A fun freakfest at that, but at times it was
a tad too reliant on kitsch rather than the horror. The story is difficult to
summarize succinctly: a carefree, normal teenage girl starts coming fac ...
```

Using the preceding code example, we printed the first 300 characters from the top three horror movies. The reviews—even though we don't know which exact movie they belong to—sound like reviews of horror movies (however, one might argue that `Horror movie #2` could also be a good fit for topic category 1: *Generally bad movies*).

Summary

In this chapter, you learned how to use machine learning algorithms to classify text documents based on their polarity, which is a basic task in sentiment analysis in the field of NLP. Not only did you learn how to encode a document as a feature vector using the bag-of-words model, but you also learned how to weight the term frequency by relevance using tf-idf.

Working with text data can be computationally quite expensive due to the large feature vectors that are created during this process; in the last section, we covered how to utilize out-of-core or incremental learning to train a machine learning algorithm without loading the whole dataset into a computer's memory.

Lastly, you were introduced to the concept of topic modeling using LDA to categorize the movie reviews into different categories in an unsupervised fashion.

So far, in this book, we have covered many machine learning concepts, best practices, and supervised models for classification. In the next chapter, we will look at another subcategory of supervised learning, *regression analysis*, which lets us predict outcome variables on a continuous scale, in contrast to the categorical class labels of the classification models that we have been working with so far.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`

9

Predicting Continuous Target Variables with Regression Analysis

Throughout the previous chapters, you learned a lot about the main concepts behind **supervised learning** and trained many different models for classification tasks to predict group memberships or categorical variables. In this chapter, we will dive into another subcategory of supervised learning: **regression analysis**.

Regression models are used to predict target variables on a continuous scale, which makes them attractive for addressing many questions in science. They also have applications in industry, such as understanding relationships between variables, evaluating trends, or making forecasts. One example is predicting the sales of a company in future months.

In this chapter, we will discuss the main concepts of regression models and cover the following topics:

- Exploring and visualizing datasets
- Looking at different approaches to implementing linear regression models
- Training regression models that are robust to outliers
- Evaluating regression models and diagnosing common problems
- Fitting regression models to nonlinear data

Introducing linear regression

The goal of linear regression is to model the relationship between one or multiple features and a continuous target variable. In contrast to classification—a different subcategory of supervised learning—regression analysis aims to predict outputs on a continuous scale rather than categorical class labels.

In the following subsections, you will be introduced to the most basic type of linear regression, **simple linear regression**, and understand how to relate it to the more general, multivariate case (linear regression with multiple features).

Simple linear regression

The goal of simple (**univariate**) linear regression is to model the relationship between a single feature (**explanatory variable**, x) and a continuous-valued **target** (**response variable**, y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_1 x + b$$

Here, the parameter (bias unit), b, represents the y axis intercept and w_1 is the weight coefficient of the explanatory variable. Our goal is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which can then be used to predict the responses of new explanatory variables that were not part of the training dataset.

Based on the linear equation that we defined previously, linear regression can be understood as finding the best-fitting straight line through the training examples, as shown in *Figure 9.1*:

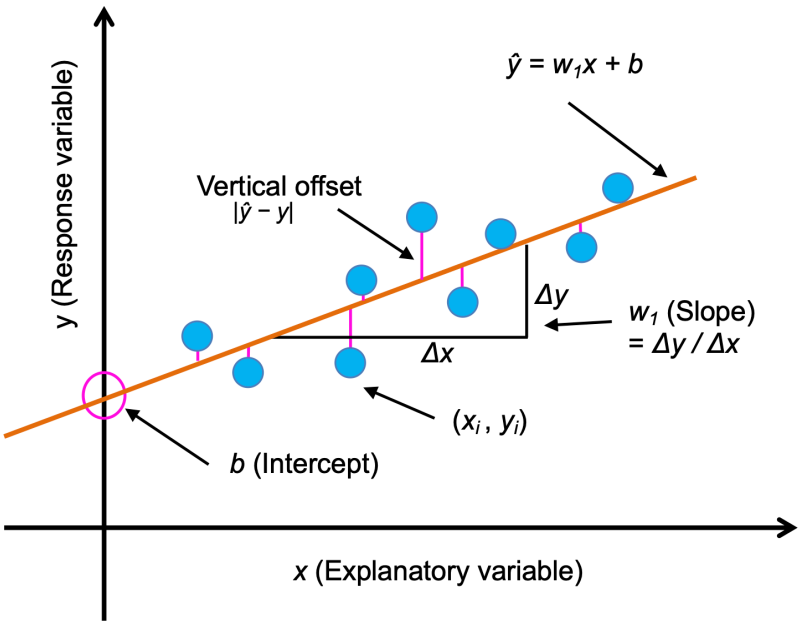


Figure 9.1: A simple one-feature linear regression example

This best-fitting line is also called the **regression line**, and the vertical lines from the regression line to the training examples are the so-called **offsets** or **residuals**—the errors of our prediction.

Multiple linear regression

The previous section introduced simple linear regression, a special case of linear regression with one explanatory variable. Of course, we can also generalize the linear regression model to multiple explanatory variables; this process is called **multiple linear regression**:

$$y = w_1 x_1 + \ldots + w_m x_m + b = \sum_{i=1}^{m} w_i x_i + b = w^T \boldsymbol{x} + b$$

Figure 9.2 shows how the two-dimensional, fitted hyperplane of a multiple linear regression model with two features could look:

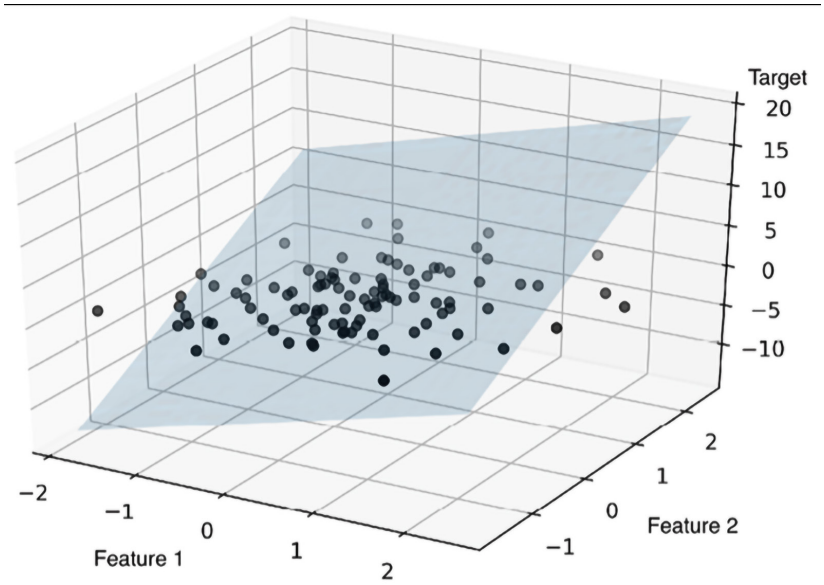


Figure 9.2: A two-feature linear regression model

As you can see, visualizations of multiple linear regression hyperplanes in a three-dimensional scatterplot are already challenging to interpret when looking at static figures. Since we have no good means of visualizing hyperplanes with two dimensions in a scatterplot (multiple linear regression models fit to datasets with three or more features), the examples and visualizations in this chapter will mainly focus on the univariate case, using simple linear regression. However, simple and multiple linear regression are based on the same concepts and the same evaluation techniques; the code implementations that we will discuss in this chapter are also compatible with both types of regression model.

Exploring the Ames Housing dataset

Before we implement the first linear regression model, we will discuss a new dataset, the Ames Housing dataset, which contains information about individual residential property in Ames, Iowa, from 2006 to 2010. The dataset was collected by Dean De Cock in 2011, and additional information is available via the following links:

- A report describing the dataset: `http://jse.amstat.org/v19n3/decock.pdf`
- Detailed documentation regarding the dataset's features: `http://jse.amstat.org/v19n3/decock/DataDocumentation.txt`
- The dataset in a tab-separated format: `http://jse.amstat.org/v19n3/decock/AmesHousing.txt`

As with each new dataset, it is always helpful to explore the data through a simple visualization, to get a better feeling of what we are working with, which is what we will do in the following subsections.

Loading the Ames Housing dataset into a DataFrame

In this section, we will load the Ames Housing dataset using the pandas `read_csv` function, which is fast and versatile and a recommended tool for working with tabular data stored in a plaintext format.

The Ames Housing dataset consists of 2,930 examples and 80 features. For simplicity, we will only work with a subset of the features, shown in the following list. However, if you are curious, follow the link to the full dataset description provided at the beginning of this section, and you are encouraged to explore other variables in this dataset after reading this chapter.

The features we will be working with, including the target variable, are as follows:

- `Overall Qual`: Rating for the overall material and finish of the house on a scale from 1 (very poor) to 10 (excellent)
- `Overall Cond`: Rating for the overall condition of the house on a scale from 1 (very poor) to 10 (excellent)
- `Gr Liv Area`: Above grade (ground) living area in square feet
- `Central Air`: Central air conditioning (N=no, Y=yes)
- `Total Bsmt SF`: Total square feet of the basement area
- `SalePrice`: Sale price in U.S. dollars ($)

For the rest of this chapter, we will regard the sale price (`SalePrice`) as our target variable—the variable that we want to predict using one or more of the five explanatory variables. Before we explore this dataset further, let's load it into a pandas `DataFrame`:

```
import pandas as pd

columns = ['Overall Qual', 'Overall Cond', 'Gr Liv Area',
           'Central Air', 'Total Bsmt SF', 'SalePrice']
```

```
df = pd.read_csv('http://jse.amstat.org/v19n3/decock/AmesHousing.txt',
                 sep='\t',
                 usecols=columns)

df.head()
```

To confirm that the dataset was loaded successfully, we can display the first five lines of the dataset, as shown in *Figure 9.3*:

	Overall Qual	Overall Cond	Total Bsmt SF	Central Air	Gr Liv Area	SalePrice
0	6	5	1080.0	Y	1656	215000
1	5	6	882.0	Y	896	105000
2	6	6	1329.0	Y	1329	172000
3	7	5	2110.0	Y	2110	244000
4	5	5	928.0	Y	1629	189900

Figure 9.3: The first five rows of the housing dataset

After loading the dataset, let's also check the dimensions of the `DataFrame` to make sure that it contains the expected number of rows:

```
>>> df.shape
(2930, 6)
```

As we can see, the `DataFrame` contains 2,930 rows, as expected.

Another aspect we have to take care of is the `'Central Air'` variable, which is encoded as type `string`, as we can see in *Figure 9.3*. As we learned in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, we can use the `.map` method to convert `DataFrame` columns. The following code will convert the string `'Y'` to the integer 1, and the string `'N'` to the integer 0:

```
>>> df['Central Air'] = df['Central Air'].map({'N': 0, 'Y': 1})
```

Lastly, let's check whether any of the data frame columns contain missing values:

```
>>> df.isnull().sum()
Overall Qual     0
Overall Cond     0
Total Bsmt SF    1
Central Air      0
Gr Liv Area      0
SalePrice        0
dtype: int64
```

As we can see, the `Total Bsmt SF` feature variable contains one missing value. Since we have a relatively large dataset, the easiest way to deal with this missing feature value is to remove the corresponding example from the dataset (for alternative methods, please see *Chapter 4*):

```
>>> df = df.dropna(axis=0)
>>> df.isnull().sum()

Overall Qual     0
Overall Cond     0
Total Bsmt SF    0
Central Air      0
Gr Liv Area      0
SalePrice        0
dtype: int64
```

Visualizing the important characteristics of a dataset

Exploratory data analysis (**EDA**) is an important and recommended first step prior to the training of a machine learning model. In the rest of this section, we will use some simple yet useful techniques from the graphical EDA toolbox that may help us to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

First, we will create a **scatterplot matrix** that allows us to visualize the pair-wise correlations between the different features in this dataset in one place. To plot the scatterplot matrix, we will use the `scatterplotmatrix` function from the mlxtend library (`http://rasbt.github.io/mlxtend/`), which is a Python library that contains various convenience functions for machine learning and data science applications in Python.

You can install the `mlxtend` package via `conda install mlxtend` or `pip install mlxtend`. For this chapter, we used mlxtend version 0.19.0.

Once the installation is complete, you can import the package and create the scatterplot matrix as follows:

```
>>> import matplotlib.pyplot as plt
>>> from mlxtend.plotting import scatterplotmatrix
>>> scatterplotmatrix(df.values, figsize=(12, 10),
...                   names=df.columns, alpha=0.5)
>>> plt.tight_layout()
plt.show()
```

As you can see in *Figure 9.4*, the scatterplot matrix provides us with a useful graphical summary of the relationships in a dataset:

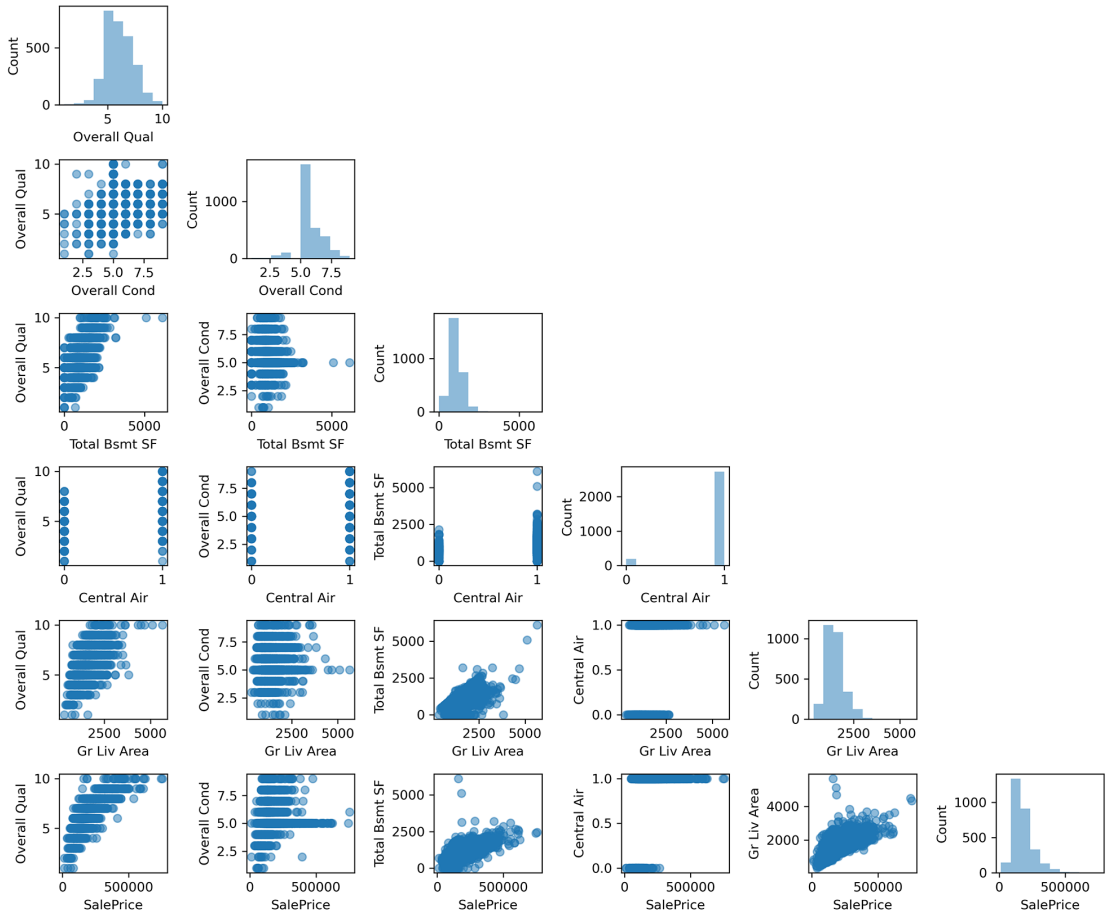


Figure 9.4: A scatterplot matrix of our data

Using this scatterplot matrix, we can now quickly see how the data is distributed and whether it contains outliers. For example, we can see (fifth column from the left of the bottom row) that there is a somewhat linear relationship between the size of the living area above ground (`Gr Liv Area`) and the sale price (`SalePrice`).

Furthermore, we can see in the histogram—the lower-right subplot in the scatterplot matrix—that the `SalePrice` variable seems to be skewed by several outliers.

The normality assumption of linear regression

Note that in contrast to common belief, training a linear regression model does not require that the explanatory or target variables are normally distributed. The normality assumption is only a requirement for certain statistics and hypothesis tests that are beyond the scope of this book (for more information on this topic, please refer to *Introduction to Linear Regression Analysis* by *Douglas C. Montgomery, Elizabeth A. Peck*, and *G. Geoffrey Vining*, *Wiley*, pages: 318-319, 2012).

Looking at relationships using a correlation matrix

In the previous section, we visualized the data distributions of the Ames Housing dataset variables in the form of histograms and scatterplots. Next, we will create a correlation matrix to quantify and summarize linear relationships between variables. A correlation matrix is closely related to the covariance matrix that we covered in the section *Unsupervised dimensionality reduction via principal component analysis* in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. We can interpret the correlation matrix as being a rescaled version of the covariance matrix. In fact, the correlation matrix is identical to a covariance matrix computed from standardized features.

The correlation matrix is a square matrix that contains the **Pearson product-moment correlation coefficient** (often abbreviated as **Pearson's r**), which measures the linear dependence between pairs of features. The correlation coefficients are in the range –1 to 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$. As mentioned previously, Pearson's correlation coefficient can simply be calculated as the covariance between two features, x and y (numerator), divided by the product of their standard deviations (denominator):

$$r = \frac{\sum_{i=1}^{n}\left[\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)\right]}{\sqrt{\sum_{i=1}^{n}\left(x^{(i)} - \mu_x\right)^2}\sqrt{\sum_{i=1}^{n}\left(y^{(i)} - \mu_y\right)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$$

Here, μ denotes the mean of the corresponding feature, σ_{xy} is the covariance between the features x and y, and σ_x and σ_y are the features' standard deviations.

Covariance versus correlation for standardized features

We can show that the covariance between a pair of standardized features is, in fact, equal to their linear correlation coefficient. To show this, let's first standardize the features x and y to obtain their z-scores, which we will denote as x' and y', respectively:

$$x' = \frac{x - \mu_x}{\sigma_x},\ y' = \frac{y - \mu_y}{\sigma_y}$$

Remember that we compute the (population) covariance between two features as follows:

$$\sigma_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Since standardization centers a feature variable at mean zero, we can now calculate the covariance between the scaled features as follows:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(x'^{(i)} - 0\right)\left(y'^{(i)} - 0\right)$$

Through resubstitution, we then get the following result:

$$\sigma'_{xy} = \frac{1}{n}\sum_{i}^{n}\left(\frac{x - \mu_x}{\sigma_x}\right)\left(\frac{y - \mu_y}{\sigma_y}\right)$$

$$\sigma'_{xy} = \frac{1}{n \cdot \sigma_x\sigma_y}\sum_{i}^{n}\left(x^{(i)} - \mu_x\right)\left(y^{(i)} - \mu_y\right)$$

Finally, we can simplify this equation as follows:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x\sigma_y}$$

In the following code example, we will use NumPy's `corrcoef` function on the five feature columns that we previously visualized in the scatterplot matrix, and we will use mlxtend's `heatmap` function to plot the correlation matrix array as a heat map:

```
>>> import numpy as np
>>> from mlxtend.plotting import heatmap


>>> cm = np.corrcoef(df.values.T)
>>> hm = heatmap(cm, row_names=df.columns, column_names=df.columns)
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 9.5*, the correlation matrix provides us with another useful summary graphic that can help us to select features based on their respective linear correlations:

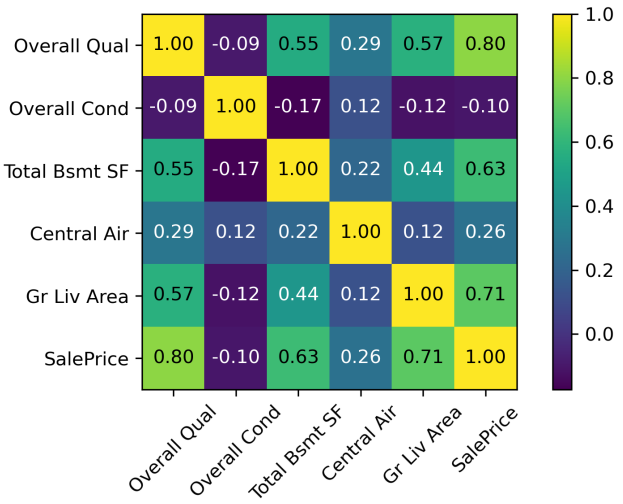


Figure 9.5: A correlation matrix of the selected variables

To fit a linear regression model, we are interested in those features that have a high correlation with our target variable, `SalePrice`. Looking at the previous correlation matrix, we can see that `SalePrice` shows the largest correlation with the `Gr Liv Area` variable (`0.71`), which seems to be a good choice for an exploratory variable to introduce the concepts of a simple linear regression model in the following section.

Implementing an ordinary least squares linear regression model

At the beginning of this chapter, we mentioned that linear regression can be understood as obtaining the best-fitting straight line through the examples of our training data. However, we have neither defined the term *best-fitting* nor have we discussed the different techniques of fitting such a model. In the following subsections, we will fill in the missing pieces of this puzzle using the **ordinary least squares** (**OLS**) method (sometimes also called **linear least squares**) to estimate the parameters of the linear regression line that minimizes the sum of the squared vertical distances (residuals or errors) to the training examples.

Solving regression for regression parameters with gradient descent

Consider our implementation of the **Adaptive Linear Neuron** (**Adaline**) from *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. You will remember that the artificial neuron uses a linear activation function. Also, we defined a loss function, $L(\boldsymbol{w})$, which we minimized to learn the weights via optimization algorithms, such as **gradient descent** (**GD**) and **stochastic gradient descent** (**SGD**).

This loss function in Adaline is the **mean squared error** (**MSE**), which is identical to the loss function that we use for OLS:

$$L(\boldsymbol{w}, b) = \frac{1}{2n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

Here, $\hat{y}$ is the predicted value $\hat{y} = \boldsymbol{w}^T\boldsymbol{x} + b$ (note that the term $\frac{1}{2}$ is just used for convenience to derive the update rule of GD). Essentially, OLS regression can be understood as Adaline without the threshold function so that we obtain continuous target values instead of the class labels `0` and `1`. To demonstrate this, let's take the GD implementation of Adaline from *Chapter 2* and remove the threshold function to implement our first linear regression model:

```
class LinearRegressionGD:
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.array([0.])
        self.losses_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        return self.net_input(X)
```

Weight updates with gradient descent

If you need a refresher about how the weights are updated—taking a step in the opposite direction of the gradient—please revisit the *Adaptive linear neurons and the convergence of learning* section in *Chapter 2*.

To see our `LinearRegressionGD` regressor in action, let's use the `Gr Living Area` (size of the living area above ground in square feet) feature from the Ames Housing dataset as the explanatory variable and train a model that can predict `SalePrice`. Furthermore, we will standardize the variables for better convergence of the GD algorithm. The code is as follows:

```
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD(eta=0.1)
>>> lr.fit(X_std, y_std)
```

Notice the workaround regarding `y_std`, using `np.newaxis` and `flatten`. Most data preprocessing classes in scikit-learn expect data to be stored in two-dimensional arrays. In the previous code example, the use of `np.newaxis` in `y[:, np.newaxis]` added a new dimension to the array. Then, after `StandardScaler` returned the scaled variable, we converted it back to the original one-dimensional array representation using the `flatten()` method for our convenience.

We discussed in *Chapter 2* that it is always a good idea to plot the loss as a function of the number of epochs (complete iterations) over the training dataset when we are using optimization algorithms, such as GD, to check that the algorithm converged to a loss minimum (here, a *global* loss minimum):

```
>>> plt.plot(range(1, lr.n_iter+1), lr.losses_)
>>> plt.ylabel('MSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

As you can see in *Figure 9.6*, the GD algorithm converged approximately after the tenth epoch:

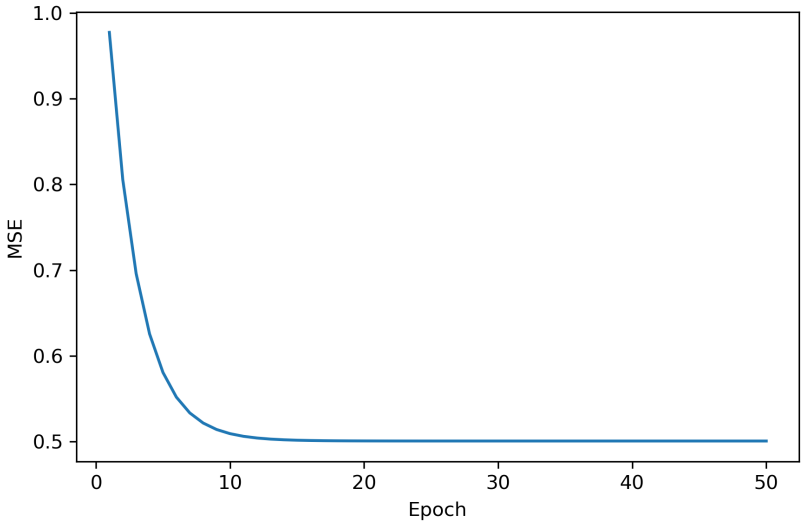


Figure 9.6: The loss function versus the number of epochs

Next, let's visualize how well the linear regression line fits the training data. To do so, we will define a simple helper function that will plot a scatterplot of the training examples and add the regression line:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
```

Now, we will use this `lin_regplot` function to plot the living area against the sale price:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel(' Living area above ground (standardized)')
>>> plt.ylabel('Sale price (standardized)')
>>> plt.show()
```

As you can see in *Figure 9.7*, the linear regression line reflects the general trend that house prices tend to increase with the size of the living area:

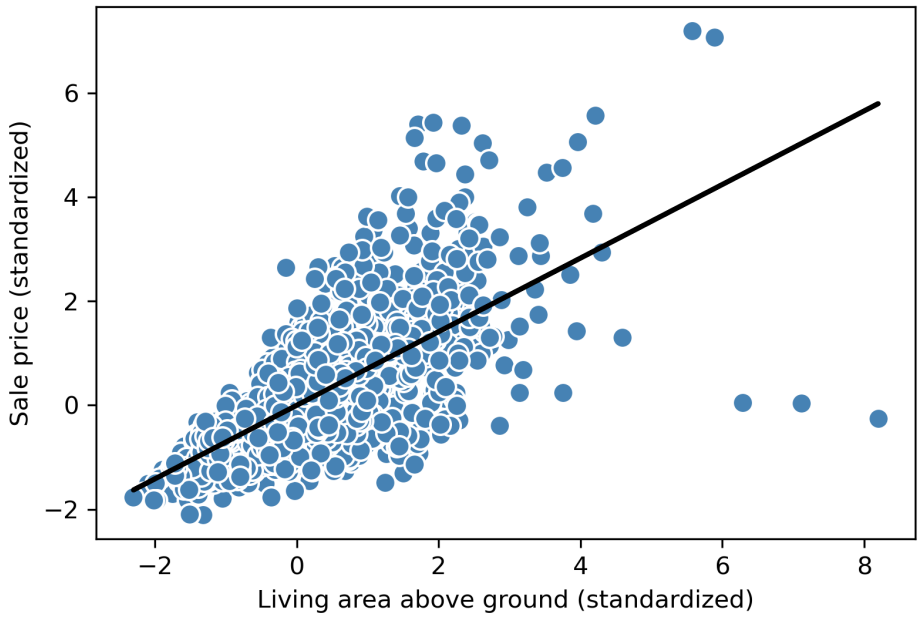


Figure 9.7: A linear regression plot of sale prices versus living area size

Although this observation makes sense, the data also tells us that the living area size does not explain house prices very well in many cases. Later in this chapter, we will discuss how to quantify the performance of a regression model. Interestingly, we can also observe several outliers, for example, the three data points corresponding to a standardized living area greater than 6. We will discuss how we can deal with outliers later in this chapter.

In certain applications, it may also be important to report the predicted outcome variables on their original scale. To scale the predicted price back onto the original *price in U.S. dollars* scale, we can simply apply the `inverse_transform` method of `StandardScaler`:

```
>>> feature_std = sc_x.transform(np.array([[2500]]))
>>> target_std = lr.predict(feature_std)
>>> target_reverted = sc_y.inverse_transform(target_std.reshape(-1, 1))
>>> print(f'Sales price: ${target_reverted.flatten()[0]:.2f}')
Sales price: $292507.07
```

In this code example, we used the previously trained linear regression model to predict the price of a house with an aboveground living area of 2,500 square feet. According to our model, such a house will be worth $292,507.07.

As a side note, it is also worth mentioning that we technically don't have to update the intercept parameter (for instance, the bias unit, *b*) if we are working with standardized variables, since the *y* axis intercept is always 0 in those cases. We can quickly confirm this by printing the model parameters:

```
>>> print(f'Slope: {lr.w_[0]:.3f}')
Slope: 0.707
>>> print(f'Intercept: {lr.b_[0]:.3f}')
Intercept: -0.000
```

Estimating the coefficient of a regression model via scikit-learn

In the previous section, we implemented a working model for regression analysis; however, in a real-world application, we may be interested in more efficient implementations. For example, many of scikit-learn's estimators for regression make use of the least squares implementation in SciPy (`scipy.linalg.lstsq`), which, in turn, uses highly optimized code optimizations based on the **Linear Algebra Package** (**LAPACK**). The linear regression implementation in scikit-learn also works (better) with unstandardized variables, since it does not use (S)GD-based optimization, so we can skip the standardization step:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> y_pred = slr.predict(X)
>>> print(f'Slope: {slr.coef_[0]:.3f}')
Slope: 111.666
>>> print(f'Intercept: {slr.intercept_:.3f}')
Intercept: 13342.979
```

As you can see from executing this code, scikit-learn's `LinearRegression` model, fitted with the unstandardized `Gr Liv Area` and `SalePrice` variables, yielded different model coefficients, since the features have not been standardized. However, when we compare it to our GD implementation by plotting `SalePrice` against `Gr Liv Area`, we can qualitatively see that it fits the data similarly well:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')
>>> plt.tight_layout()
>>> plt.show()
```

For instance, we can see that the overall result looks identical to our GD implementation:

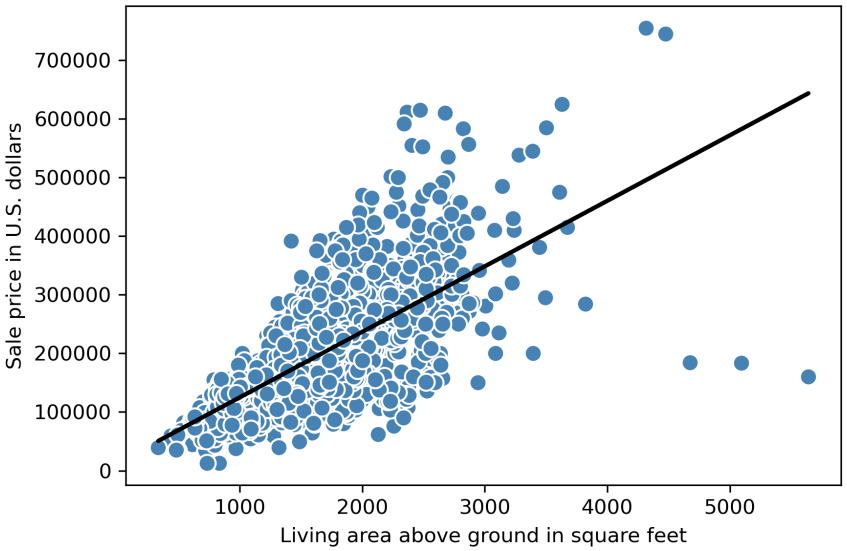


Figure 9.8: A linear regression plot using scikit-learn

Analytical solutions of linear regression

As an alternative to using machine learning libraries, there is also a closed-form solution for solving OLS involving a system of linear equations that can be found in most introductory statistics textbooks:

$$w = (X^T X)^{-1} X^T y$$

We can implement it in Python as follows:

```
# adding a column vector of "ones"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print(f'Slope: {w[1]:.3f}')
Slope: 111.666
>>> print(f'Intercept: {w[0]:.3f}')
Intercept: 13342.979
```

The advantage of this method is that it is guaranteed to find the optimal solution analytically. However, if we are working with very large datasets, it can be computationally too expensive to invert the matrix in this formula (sometimes also called the normal equation), or the matrix containing the training examples may be singular (non-invertible), which is why we may prefer iterative methods in certain cases.

If you are interested in more information on how to obtain normal equations, take a look at Dr. Stephen Pollock's chapter *The Classical Linear Regression Model*, from his lectures at the University of Leicester, which is available for free at `http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf`.

Also, if you want to compare linear regression solutions obtained via GD, SGD, the closed-form solution, QR factorization, and singular vector decomposition, you can use the `LinearRegression` class implemented in mlxtend (`http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/`), which lets users toggle between these options. Another great library to recommend for regression modeling in Python is statsmodels, which implements more advanced linear regression models, as illustrated at `https://www.statsmodels.org/stable/examples/index.html#regression`.

Fitting a robust regression model using RANSAC

Linear regression models can be heavily impacted by the presence of outliers. In certain situations, a very small subset of our data can have a big effect on the estimated model coefficients. Many statistical tests can be used to detect outliers, but these are beyond the scope of the book. However, removing outliers always requires our own judgment as data scientists as well as our domain knowledge.

As an alternative to throwing out outliers, we will look at a robust method of regression using the **RANdom SAmple Consensus** (**RANSAC**) algorithm, which fits a regression model to a subset of the data, the so-called **inliers**.

We can summarize the iterative RANSAC algorithm as follows:

1. Select a random number of examples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user-given tolerance to the inliers.
3. Refit the model using all inliers.
4. Estimate the error of the fitted model versus the inliers.
5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of iterations was reached; go back to *step 1* otherwise.

Let's now use a linear model in combination with the RANSAC algorithm as implemented in scikit-learn's `RANSACRegressor` class:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(
...     LinearRegression(),
...     max_trials=100, # default value
...     min_samples=0.95,
...     residual_threshold=None, # default value
...     random_state=123)
>>> ransac.fit(X, y)
```

We set the maximum number of iterations of the `RANSACRegressor` to 100, and using `min_samples=0.95`, we set the minimum number of the randomly chosen training examples to be at least 95 percent of the dataset.

By default (via `residual_threshold=None`), scikit-learn uses the **MAD** estimate to select the inlier threshold, where MAD stands for the **median absolute deviation** of the target values, `y`. However, the choice of an appropriate value for the inlier threshold is problem-specific, which is one disadvantage of RANSAC.

Many different approaches have been developed in recent years to select a good inlier threshold automatically. You can find a detailed discussion in *Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting* by *R. Toldo* and *A. Fusiello*, *Springer*, 2009 (in *Image Analysis and Processing–ICIAP 2009*, pages: 123-131).

Once we have fitted the RANSAC model, let's obtain the inliers and outliers from the fitted RANSAC linear regression model and plot them together with the linear fit:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...             c='steelblue', edgecolor='white',
...             marker='o', label='Inliers')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...             c='limegreen', edgecolor='white',
...             marker='s', label='Outliers')
>>> plt.plot(line_X, line_y_ransac, color='black', lw=2)
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 9.9*, the linear regression model was fitted on the detected set of inliers, which are shown as circles:

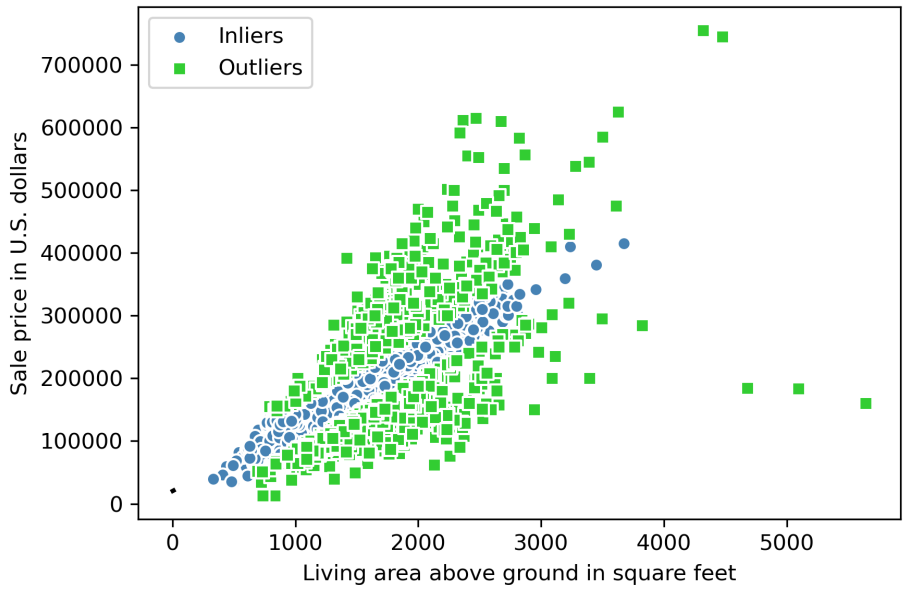


Figure 9.9: Inliers and outliers identified via a RANSAC linear regression model

When we print the slope and intercept of the model by executing the following code, the linear regression line will be slightly different from the fit that we obtained in the previous section without using RANSAC:

```
>>> print(f'Slope: {ransac.estimator_.coef_[0]:.3f}')
Slope: 106.348
>>> print(f'Intercept: {ransac.estimator_.intercept_:.3f}')
Intercept: 20190.093
```

Remember that we set the `residual_threshold` parameter to `None`, so RANSAC was using the MAD to compute the threshold for flagging inliers and outliers. The MAD, for this dataset, can be computed as follows:

```
>>> def median_absolute_deviation(data):
...     return np.median(np.abs(data - np.median(data)))
>>> median_absolute_deviation(y)
37000.00
```

So, if we want to identify fewer data points as outliers, we can choose a `residual_threshold` value greater than the preceding MAD. For example, *Figure 9.10* shows the inliers and outliers of a RANSAC linear regression model with a residual threshold of 65,000:

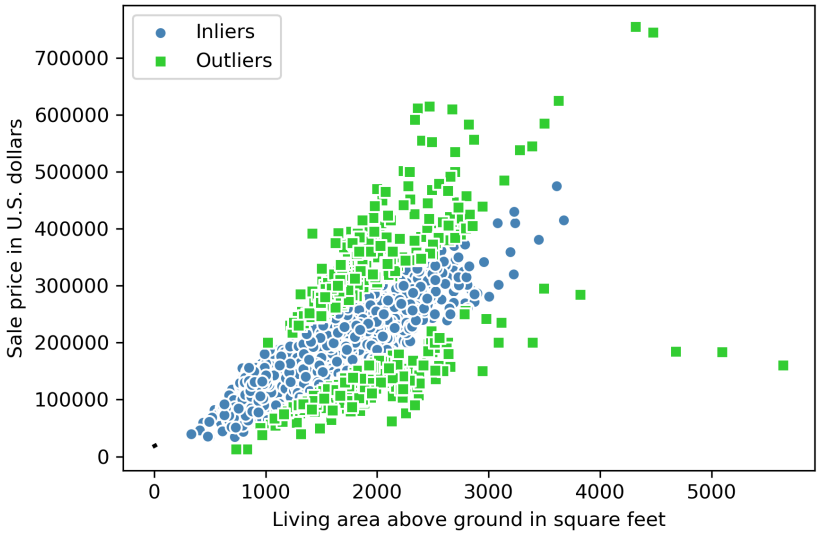


Figure 9.10: Inliers and outliers determined by a RANSAC linear regression model with a larger residual threshold

Using RANSAC, we reduced the potential effect of the outliers in this dataset, but we don't know whether this approach will have a positive effect on the predictive performance for unseen data or not. Thus, in the next section, we will look at different approaches for evaluating a regression model, which is a crucial part of building systems for predictive modeling.

Evaluating the performance of linear regression models

In the previous section, you learned how to fit a regression model on training data. However, you discovered in previous chapters that it is crucial to test the model on data that it hasn't seen during training to obtain a more unbiased estimate of its generalization performance.

As you may remember from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we want to split our dataset into separate training and test datasets, where we will use the former to fit the model and the latter to evaluate its performance on unseen data to estimate the generalization performance. Instead of proceeding with the simple regression model, we will now use all five features in the dataset and train a multiple regression model:

```
>>> from sklearn.model_selection import train_test_split
>>> target = 'SalePrice'
>>> features = df.columns[df.columns != target]
>>> X = df[features].values
>>> y = df[target].values
```

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=123)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Since our model uses multiple explanatory variables, we can't visualize the linear regression line (or hyperplane, to be precise) in a two-dimensional plot, but we can plot the residuals (the differences or vertical distances between the actual and predicted values) versus the predicted values to diagnose our regression model. **Residual plots** are a commonly used graphical tool for diagnosing regression models. They can help to detect nonlinearity and outliers and check whether the errors are randomly distributed.

Using the following code, we will now plot a residual plot where we simply subtract the true target variables from our predicted responses:

```
>>> x_max = np.max(
...     [np.max(y_train_pred), np.max(y_test_pred)])
>>> x_min = np.min(
...     [np.min(y_train_pred), np.min(y_test_pred)])

>>> fig, (ax1, ax2) = plt.subplots(
...     1, 2, figsize=(7, 3), sharey=True)

>>> ax1.scatter(
...     y_test_pred, y_test_pred - y_test,
...     c='limegreen', marker='s',
...     edgecolor='white',
...     label='Test data')
>>> ax2.scatter(
...     y_train_pred, y_train_pred - y_train,
...     c='steelblue', marker='o', edgecolor='white',
...     label='Training data')
>>> ax1.set_ylabel('Residuals')

>>> for ax in (ax1, ax2):
...     ax.set_xlabel('Predicted values')
...     ax.legend(loc='upper left')
...     ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,\
...         color='black', lw=2)
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we should see residual plots for the test and training datasets with a line passing through the *x* axis origin, as shown in *Figure 9.11*:

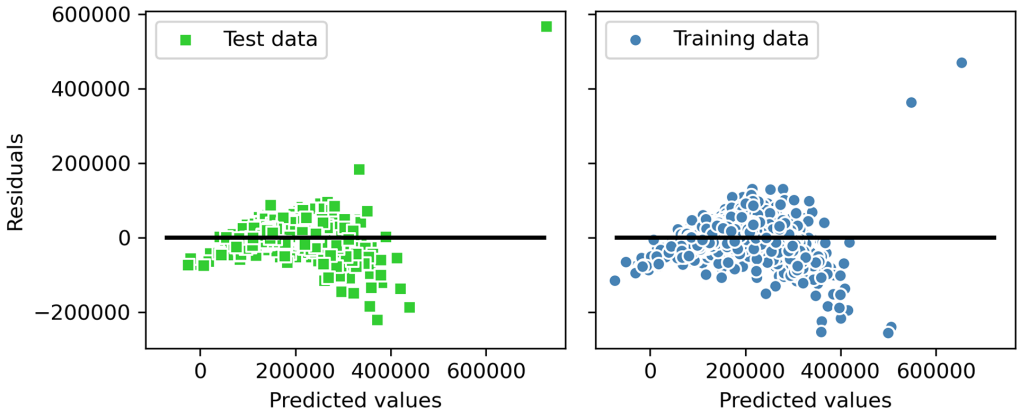


Figure 9.11: Residual plots of our data

In the case of a perfect prediction, the residuals would be exactly zero, which we will probably never encounter in realistic and practical applications. However, for a good regression model, we would expect the errors to be randomly distributed and the residuals to be randomly scattered around the centerline. If we see patterns in a residual plot, it means that our model is unable to capture some explanatory information, which has leaked into the residuals, as you can see to a degree in our previous residual plot. Furthermore, we can also use residual plots to detect outliers, which are represented by the points with a large deviation from the centerline.

Another useful quantitative measure of a model's performance is the **mean squared error** (**MSE**) that we discussed earlier as our loss function that we minimized to fit the linear regression model. The following is a version of the MSE without the $\frac{1}{2}$ scaling factor that is often used to simplify the loss derivative in gradient descent:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2$$

Similar to prediction accuracy in classification contexts, we can use the MSE for cross-validation and model selection as discussed in *Chapter 6*.

Like classification accuracy, MSE also normalizes according to the sample size, n. This makes it possible to compare across different sample sizes (for example, in the context of learning curves) as well.

Let's now compute the MSE of our training and test predictions:

```
>>> from sklearn.metrics import mean_squared_error
>>> mse_train = mean_squared_error(y_train, y_train_pred)
>>> mse_test = mean_squared_error(y_test, y_test_pred)
```

```
>>> print(f'MSE train: {mse_train:.2f}')
MSE train: 1497216245.85
>>> print(f'MSE test: {mse_test:.2f}')
MSE test: 1516565821.00
```

We can see that the MSE on the training dataset is less than on the test set, which is an indicator that our model is slightly overfitting the training data in this case. Note that it can be more intuitive to show the error on the original unit scale (here, dollar instead of dollar-squared), which is why we may choose to compute the square root of the MSE, called *root mean squared error*, or the **mean absolute error** (**MAE**), which emphasizes incorrect prediction slightly less:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|$$

We can compute the MAE similar to the MSE:

```
>>> from sklearn.metrics import mean_absolute_error
>>> mae_train = mean_absolute_error(y_train, y_train_pred)
>>> mae_test = mean_absolute_error(y_test, y_test_pred)
>>> print(f'MAE train: {mae_train:.2f}')
MAE train: 25983.03
>>> print(f'MAE test: {mae_test:.2f}')
MAE test: 24921.29
```

Based on the test set MAE, we can say that the model makes an error of approximately $25,000 on average.

When we use the MAE or MSE for comparing models, we need to be aware that these are unbounded in contrast to the classification accuracy, for example. In other words, the interpretations of the MAE and MSE depend on the dataset and feature scaling. For example, if the sale prices were presented as multiples of 1,000 (with the K suffix), the same model would yield a lower MAE compared to a model that worked with unscaled features. To further illustrate this point,

$$|\$500\text{K} - 550\text{K}| < |\$500{,}000 - 550{,}000|$$

Thus, it may sometimes be more useful to report the **coefficient of determination** (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model's performance. Or, in other words, R^2 is the fraction of response variance that is captured by the model. The R^2 value is defined as:

$$R^2 = 1 - \frac{SSE}{SST}$$

Here, SSE is the sum of squared errors, which is similar to the MSE but does not include the normalization by sample size n:

$$SSE = \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

And SST is the total sum of squares:

$$SST = \sum_{i=1}^{n} (y^{(i)} - \mu_y)^2$$

In other words, SST is simply the variance of the response.

Now, let's briefly show that R^2 is indeed just a rescaled version of the MSE:

$$\begin{aligned} R^2 &= 1 - \frac{\frac{1}{n}SSE}{\frac{1}{n}SST} \\ &= \frac{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \mu_y)^2} \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

For the training dataset, R^2 is bounded between 0 and 1, but it can become negative for the test dataset. A negative R^2 means that the regression model fits the data worse than a horizontal line representing the sample mean. (In practice, this often happens in the case of extreme overfitting, or if we forget to scale the test set in the same manner we scaled the training set.) If $R^2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

Evaluated on the training data, the R^2 of our model is 0.77, which isn't great but also not too bad given that we only work with a small set of features. However, the R^2 on the test dataset is only slightly smaller, at 0.75, which indicates that the model is only overfitting slightly:

```
>>> from sklearn.metrics import r2_score
>>> train_r2 = r2_score(y_train, y_train_pred)
>>> test_r2 = r2_score(y_test, y_test_pred)
>>> print(f'R^2 train: {train_r2:.3f}, {test_r2:.3f}')
R^2 train: 0.77, test: 0.75
```

Using regularized methods for regression

As we discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-Learn*, regularization is one approach to tackling the problem of overfitting by adding additional information and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called **ridge regression**, **least absolute shrinkage and selection operator** (**LASSO**), and **elastic net**.

Ridge regression is an L2 penalized model where we simply add the squared sum of the weights to the MSE loss function:

$$L(\boldsymbol{w})_{Ridge} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda ||\boldsymbol{w}||_2^2$$

Here, the L2 term is defined as follows:

$$\lambda ||\boldsymbol{w}||_2^2 = \lambda \sum_{j=1}^{m} w_j^2$$

By increasing the value of hyperparameter λ, we increase the regularization strength and thereby shrink the weights of our model. Please note that, as mentioned in *Chapter 3*, the bias unit b is not regularized.

An alternative approach that can lead to sparse models is LASSO. Depending on the regularization strength, certain weights can become zero, which also makes LASSO useful as a supervised feature selection technique:

$$L(\boldsymbol{w})_{Lasso} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda ||\boldsymbol{w}||_1$$

Here, the L1 penalty for LASSO is defined as the sum of the absolute magnitudes of the model weights, as follows:

$$\lambda ||\boldsymbol{w}||_1 = \lambda \sum_{j=1}^{m} \left|w_j\right|$$

However, a limitation of LASSO is that it selects at most n features if $m > n$, where n is the number of training examples. This may be undesirable in certain applications of feature selection. In practice, however, this property of LASSO is often an advantage because it avoids saturated models. The saturation of a model occurs if the number of training examples is equal to the number of features, which is a form of overparameterization. As a consequence, a saturated model can always fit the training data perfectly but is merely a form of interpolation and thus is not expected to generalize well.

A compromise between ridge regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than n features if $m > n$:

$$L(\boldsymbol{w})_{Elastic\ Net} = \sum_{i=1}^{n} \left(y^{(i)} - \hat{y}^{(i)}\right)^2 + \lambda_2 ||\boldsymbol{w}||_2^2 + \lambda_1 ||\boldsymbol{w}||_1$$

Those regularized regression models are all available via scikit-learn, and their usage is similar to the regular regression model except that we have to specify the regularization strength via the parameter λ, for example, optimized via k-fold cross-validation.

A ridge regression model can be initialized via:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Note that the regularization strength is regulated by the parameter `alpha`, which is similar to the parameter λ. Likewise, we can initialize a LASSO regressor from the `linear_model` submodule:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Lastly, the `ElasticNet` implementation allows us to vary the L1 to L2 ratio:

```
>>> from sklearn.linear_model import ElasticNet
>>> elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

For example, if we set `l1_ratio` to 1.0, the `ElasticNet` regressor would be equal to LASSO regression. For more detailed information about the different implementations of linear regression, please refer to the documentation at `http://scikit-learn.org/stable/modules/linear_model.html`.

Turning a linear regression model into a curve – polynomial regression

In the previous sections, we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_1 x + w_2 x^2 + \ldots + w_d x^d + b$$

Here, d denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients, $\boldsymbol{w}$. In the following subsections, we will see how we can add such polynomial terms to an existing dataset conveniently and fit a polynomial regression model.

Adding polynomial terms using scikit-learn

We will now learn how to use the `PolynomialFeatures` transformer class from scikit-learn to add a quadratic term ($d = 2$) to a simple regression problem with one explanatory variable. Then, we will compare the polynomial to the linear fit by following these steps:

1. Add a second-degree polynomial term:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([ 258.0, 270.0, 294.0, 320.0, 342.0,
...                368.0, 396.0, 446.0, 480.0, 586.0])\
...              [:, np.newaxis]
>>> y = np.array([ 236.4, 234.4, 252.8, 298.6, 314.2,
...                342.2, 360.8, 368.0, 391.2, 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

2. Fit a simple linear regression model for comparison:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

3. Fit a multiple regression model on the transformed features for polynomial regression:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

4. Plot the results:

```
>>> plt.scatter(X, y, label='Training points')
>>> plt.plot(X_fit, y_lin_fit,
...          label='Linear fit', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...          label='Quadratic fit')
>>> plt.xlabel('Explanatory variable')
>>> plt.ylabel('Predicted or known target values')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In the resulting plot, you can see that the polynomial fit captures the relationship between the response and explanatory variables much better than the linear fit:

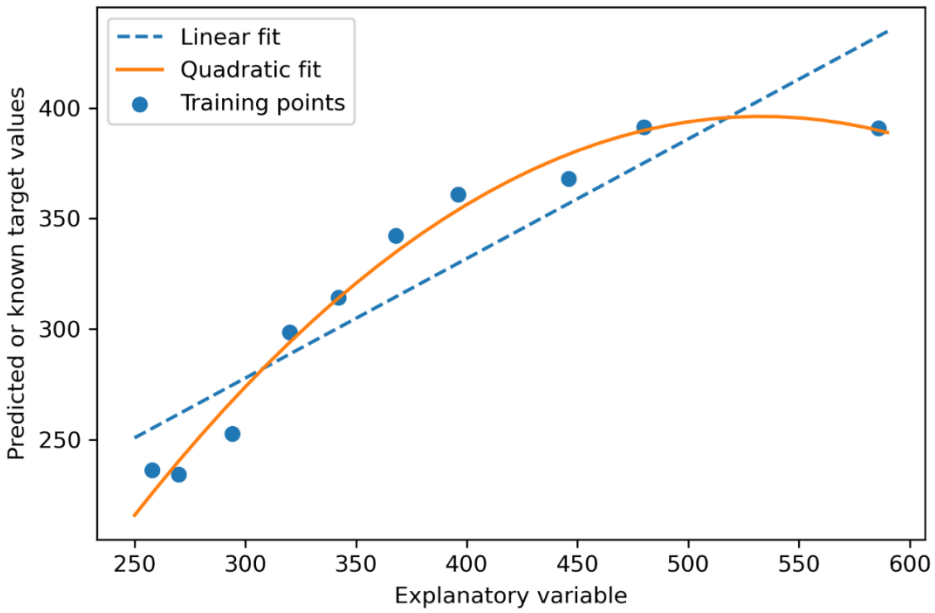


Figure 9.12: A comparison of a linear and quadratic model

Next, we will compute the MSE and R^2 evaluation metrics:

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> mse_lin = mean_squared_error(y, y_lin_pred)
>>> mse_quad = mean_squared_error(y, y_quad_pred)
>>> print(f'Training MSE linear: {mse_lin:.3f}'
          f', quadratic: {mse_quad:.3f}')
Training MSE linear: 569.780, quadratic: 61.330
>>> r2_lin = r2_score(y, y_lin_pred)
>>> r2_quad = r2_score(y, y_quad_pred)
>>> print(f'Training R^2 linear: {r2_lin:.3f}'
          f', quadratic: {r2_quad:.3f}')
Training R^2 linear: 0.832, quadratic: 0.982
```

As you can see after executing the code, the MSE decreased from 570 (linear fit) to 61 (quadratic fit); also, the coefficient of determination reflects a closer fit of the quadratic model ($R^2 = 0.982$) as opposed to the linear fit ($R^2 = 0.832$) in this particular toy problem.

Modeling nonlinear relationships in the Ames Housing dataset

In the preceding subsection, you learned how to construct polynomial features to fit nonlinear relationships in a toy problem; let's now take a look at a more concrete example and apply those concepts to the data in the Ames Housing dataset. By executing the following code, we will model the relationship between sale prices and the living area above ground using second-degree (quadratic) and third-degree (cubic) polynomials and compare that to a linear fit.

We start by removing the three outliers with a living area greater than 4,000 square feet, which we can see in previous figures, such as in *Figure 9.8*, so that these outliers don't skew our regression fits:

```
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> X = X[(df['Gr Liv Area'] < 4000)]
>>> y = y[(df['Gr Liv Area'] < 4000)]
```

Next, we fit the regression models:

```
>>> regr = LinearRegression()

>>> # create quadratic and cubic features
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

>>> # fit to features
>>> X_fit = np.arange(X.min()-1, X.max()+2, 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

>>> # plot results
>>> plt.scatter(X, y, label='Training points', color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...          label=f'Linear (d=1), $R^2$={linear_r2:.2f}',
```

```
...           color='blue',
...           lw=2,
...           linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...           label=f'Quadratic (d=2), $R^2$={quadratic_r2:.2f}',
...           color='red',
...           lw=2,
...           linestyle='-')
>>> plt.plot(X_fit, y_cubic_fit,
...           label=f'Cubic (d=3), $R^2$={cubic_r2:.2f}',
...           color='green',
...           lw=2,
...           linestyle='--')
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

The resulting plot is shown in *Figure 9.13*:

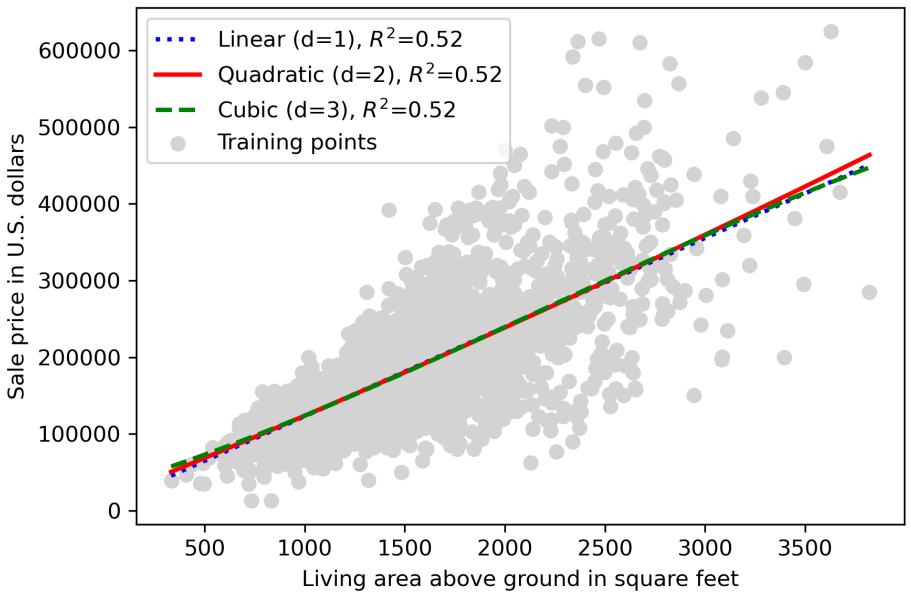


Figure 9.13: A comparison of different curves fitted to the sale price and living area data

As we can see, using quadratic or cubic features does not really have an effect. That's because the relationship between the two variables appears to be linear. So, let's take a look at another feature, namely, `Overall Qual`. The `Overall Qual` variable rates the overall quality of the material and finish of the houses and is given on a scale from 1 to 10, where 10 is best:

```
>>> X = df[['Overall Qual']].values
>>> y = df['SalePrice'].values
```

After specifying the `X` and `y` variables, we can reuse the previous code and obtain the plot in *Figure 9.14*:

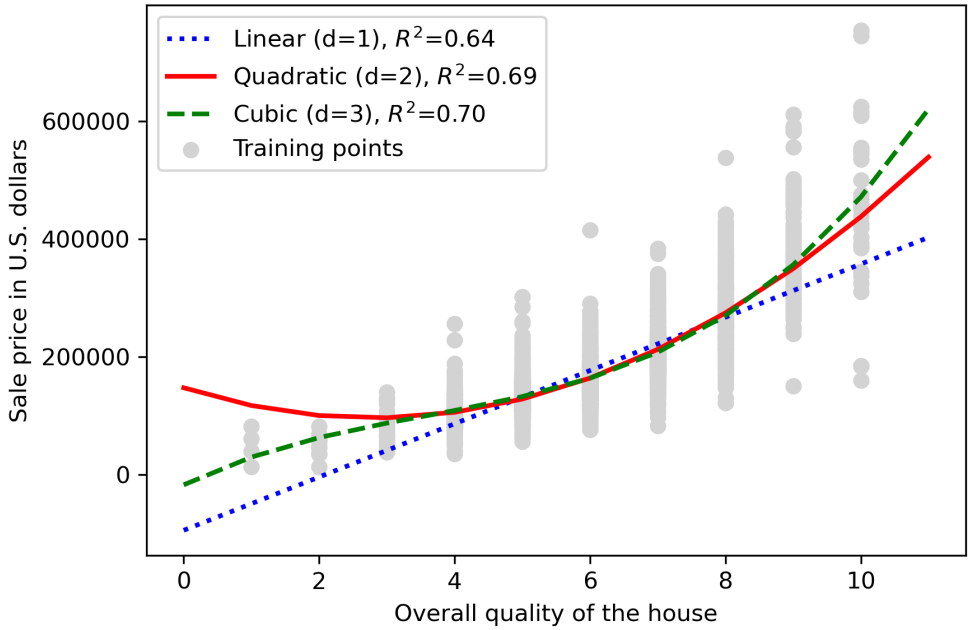


Figure 9.14: A linear, quadratic, and cubic fit on the sale price and house quality data

As you can see, the quadratic and cubic fits capture the relationship between sale prices and the overall quality of the house better than the linear fit. However, you should be aware that adding more and more polynomial features increases the complexity of a model and therefore increases the chance of overfitting. Thus, in practice, it is always recommended to evaluate the performance of the model on a separate test dataset to estimate the generalization performance.

Dealing with nonlinear relationships using random forests

In this section, we are going to look at **random forest** regression, which is conceptually different from the previous regression models in this chapter. A random forest, which is an ensemble of multiple **decision trees**, can be understood as the sum of piecewise linear functions, in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we subdivide the input space into smaller regions that become more manageable.

Decision tree regression

An advantage of the decision tree algorithm is that it works with arbitrary features and does not require any transformation of the features if we are dealing with nonlinear data because decision trees analyze one feature at a time, rather than taking weighted combinations into account. (Likewise, normalizing or standardizing features is not required for decision trees.) As mentioned in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, we grow a decision tree by iteratively splitting its nodes until the leaves are pure or a stopping criterion is satisfied. When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **information gain** (**IG**), which can be defined as follows for a binary split:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

Here, x_i is the feature to perform the split, N_p is the number of training examples in the parent node, I is the impurity function, D_p is the subset of training examples at the parent node, and D_{left} and D_{right} are the subsets of training examples at the left and right child nodes after the split. Remember that our goal is to find the feature split that maximizes the information gain; in other words, we want to find the feature split that reduces the impurities in the child nodes most. In *Chapter 3*, we discussed Gini impurity and entropy as measures of impurity, which are both useful criteria for classification. To use a decision tree for regression, however, we need an impurity metric that is suitable for continuous variables, so we define the impurity measure of a node, t, as the MSE instead:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2$$

Here, N_t is the number of training examples at node t, D_t is the training subset at node t, $y^{(i)}$ is the true target value, and $\hat{y}_t$ is the predicted target value (sample mean):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}$$

In the context of decision tree regression, the MSE is often referred to as **within-node variance**, which is why the splitting criterion is also better known as **variance reduction**.

To see what the line fit of a decision tree looks like, let's use the `DecisionTreeRegressor` implemented in scikit-learn to model the relationship between the `SalePrice` and `Gr Living Area` variables. Note that `SalePrice` and `Gr Living Area` do not necessarily represent a nonlinear relationship, but this feature combination still demonstrates the general aspects of a regression tree quite nicely:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
```

```
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('Living area above ground in square feet')
>>> plt.ylabel('Sale price in U.S. dollars')>>> plt.show()
```

As you can see in the resulting plot, the decision tree captures the general trend in the data. And we can imagine that a regression tree could also capture trends in nonlinear data relatively well. However, a limitation of this model is that it does not capture the continuity and differentiability of the desired prediction. In addition, we need to be careful about choosing an appropriate value for the depth of the tree so as to not overfit or underfit the data; here, a depth of three seemed to be a good choice.

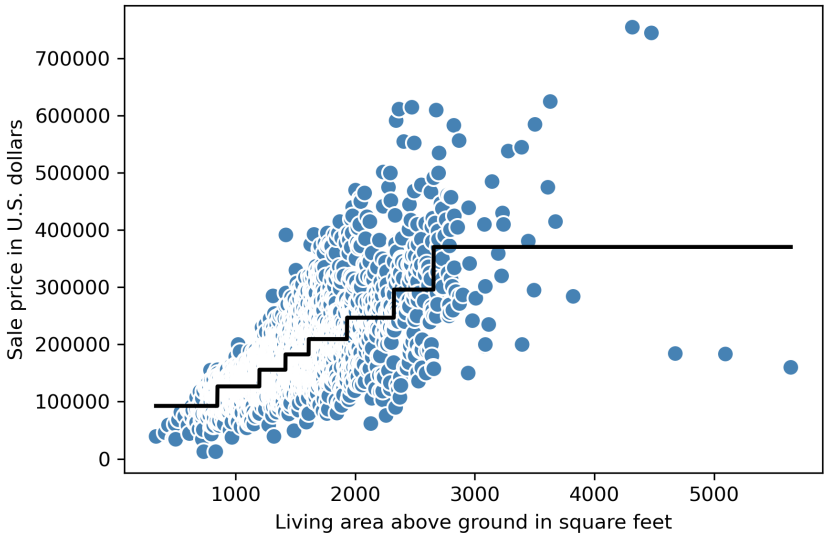


Figure 9.15: A decision tree regression plot

You are encouraged to experiment with deeper decision trees. Note that the relationship between `Gr Living Area` and `SalePrice` is rather linear, so you are also encouraged to apply the decision tree to the `Overall Qual` variable instead.

In the next section, we will look at a more robust way of fitting regression trees: random forests.

Random forest regression

As you learned in *Chapter 3*, the random forest algorithm is an ensemble technique that combines multiple decision trees. A random forest usually has a better generalization performance than an individual decision tree due to randomness, which helps to decrease the model's variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble. The basic random forest algorithm for regression is almost identical to the random forest algorithm for classification that we discussed in *Chapter 3*. The only difference is that we use the MSE criterion to grow the individual decision trees, and the predicted target variable is calculated as the average prediction across all decision trees.

Now, let's use all the features in the Ames Housing dataset to fit a random forest regression model on 70 percent of the examples and evaluate its performance on the remaining 30 percent, as we have done previously in the *Evaluating the performance of linear regression models* section. The code is as follows:

```
>>> target = 'SalePrice'
>>> features = df.columns[df.columns != target]
>>> X = df[features].values
>>> y = df[target].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=123)

>>> from sklearn.ensemble import RandomForestRegressor
>>> forest = RandomForestRegressor(
...     n_estimators=1000,
...     criterion='squared_error',
...     random_state=1,
...     n_jobs=-1)
>>> forest.fit(X_train, y_train)
>>> y_train_pred = forest.predict(X_train)
>>> y_test_pred = forest.predict(X_test)
>>> mae_train = mean_absolute_error(y_train, y_train_pred)
>>> mae_test = mean_absolute_error(y_test, y_test_pred)
>>> print(f'MAE train: {mae_train:.2f}')
MAE train: 8305.18
>>> print(f'MAE test: {mae_test:.2f}')
MAE test: 20821.77
>>> r2_train = r2_score(y_train, y_train_pred)
>>> r2_test =r2_score(y_test, y_test_pred)
>>> print(f'R^2 train: {r2_train:.2f}')
R^2 train: 0.98
>>> print(f'R^2 test: {r2_test:.2f}')
R^2 test: 0.85
```

Unfortunately, you can see that the random forest tends to overfit the training data. However, it's still able to explain the relationship between the target and explanatory variables relatively well ($R^2 = 0.85$ on the test dataset). For comparison, the linear model from the previous section, *Evaluating the performance of linear regression models*, which was fit to the same dataset, was overfitting less but performed worse on the test set ($R^2 = 0.75$).

Lastly, let's also take a look at the residuals of the prediction:

```
>>> x_max = np.max([np.max(y_train_pred), np.max(y_test_pred)])
>>> x_min = np.min([np.min(y_train_pred), np.min(y_test_pred)])

>>> fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(7, 3), sharey=True)
```

```
>>> ax1.scatter(y_test_pred, y_test_pred - y_test,
...             c='limegreen', marker='s', edgecolor='white',
...             label='Test data')
>>> ax2.scatter(y_train_pred, y_train_pred - y_train,
...             c='steelblue', marker='o', edgecolor='white',
...             label='Training data')
>>> ax1.set_ylabel('Residuals')

>>> for ax in (ax1, ax2):
...     ax.set_xlabel('Predicted values')
...     ax.legend(loc='upper left')
...     ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,
...               color='black', lw=2)

>>> plt.tight_layout()
>>> plt.show()
```

As it was already summarized by the R^2 coefficient, you can see that the model fits the training data better than the test data, as indicated by the outliers in the y axis direction. Also, the distribution of the residuals does not seem to be completely random around the zero center point, indicating that the model is not able to capture all the exploratory information. However, the residual plot indicates a large improvement over the residual plot of the linear model that we plotted earlier in this chapter.

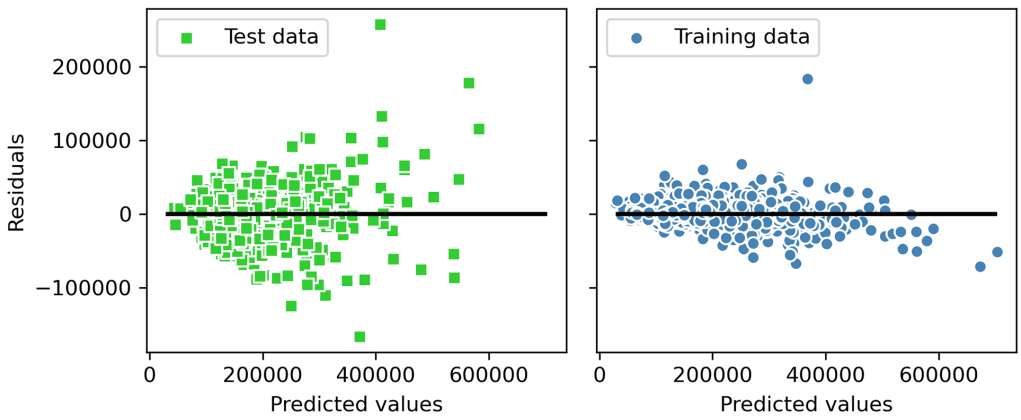


Figure 9.16: The residuals of the random forest regression

Ideally, our model error should be random or unpredictable. In other words, the error of the predictions should not be related to any of the information contained in the explanatory variables; rather, it should reflect the randomness of the real-world distributions or patterns. If we find patterns in the prediction errors, for example, by inspecting the residual plot, it means that the residual plots contain predictive information. A common reason for this could be that explanatory information is leaking into those residuals.

Unfortunately, there is not a universal approach for dealing with non-randomness in residual plots, and it requires experimentation. Depending on the data that is available to us, we may be able to improve the model by transforming variables, tuning the hyperparameters of the learning algorithm, choosing simpler or more complex models, removing outliers, or including additional variables.

Summary

At the beginning of this chapter, you learned about simple linear regression analysis to model the relationship between a single explanatory variable and a continuous response variable. We then discussed a useful explanatory data analysis technique to look at patterns and anomalies in data, which is an important first step in predictive modeling tasks.

We built our first model by implementing linear regression using a gradient-based optimization approach. You then saw how to utilize scikit-learn's linear models for regression and also implement a robust regression technique (RANSAC) as an approach for dealing with outliers. To assess the predictive performance of regression models, we computed the mean sum of squared errors and the related R^2 metric. Furthermore, we also discussed a useful graphical approach for diagnosing the problems of regression models: the residual plot.

After we explored how regularization can be applied to regression models to reduce the model complexity and avoid overfitting, we also covered several approaches for modeling nonlinear relationships, including polynomial feature transformation and random forest regressors.

We discussed supervised learning, classification, and regression analysis in detail in the previous chapters. In the next chapter, we are going to learn about another interesting subfield of machine learning, unsupervised learning, and also how to use cluster analysis to find hidden structures in data in the absence of target variables.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`

10

Working with Unlabeled Data – Clustering Analysis

In the previous chapters, we used supervised learning techniques to build machine learning models, using data where the answer was already known—the class labels were already available in our training data. In this chapter, we will switch gears and explore cluster analysis, a category of **unsupervised learning** techniques that allows us to discover hidden structures in data where we do not know the right answer upfront. The goal of **clustering** is to find a natural grouping in data so that items in the same cluster are more similar to each other than to those from different clusters.

Given its exploratory nature, clustering is an exciting topic, and in this chapter, you will learn about the following concepts, which can help us to organize data into meaningful structures:

- Finding centers of similarity using the popular **k-means** algorithm
- Taking a bottom-up approach to building hierarchical clustering trees
- Identifying arbitrary shapes of objects using a density-based clustering approach

Grouping objects by similarity using k-means

In this section, we will learn about one of the most popular clustering algorithms, k-means, which is widely used in academia as well as in industry. Clustering (or cluster analysis) is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines.

k-means clustering using scikit-learn

As you will see in a moment, the k-means algorithm is extremely easy to implement, but it is also computationally very efficient compared to other clustering algorithms, which might explain its popularity. The k-means algorithm belongs to the category of **prototype-based clustering**.

We will discuss two other categories of clustering, **hierarchical** and **density-based clustering**, later in this chapter.

Prototype-based clustering means that each cluster is represented by a prototype, which is usually either the **centroid** (*average*) of similar points with continuous features, or the **medoid** (the most *representative* or the point that minimizes the distance to all other points that belong to a particular cluster) in the case of categorical features. While k-means is very good at identifying clusters with a spherical shape, one of the drawbacks of this clustering algorithm is that we have to specify the number of clusters, *k*, *a priori*. An inappropriate choice for *k* can result in poor clustering performance. Later in this chapter, we will discuss the **elbow** method and **silhouette plots**, which are useful techniques to evaluate the quality of a clustering to help us determine the optimal number of clusters, *k*.

Although k-means clustering can be applied to data in higher dimensions, we will walk through the following examples using a simple two-dimensional dataset for the purpose of visualization:

```
>>> from sklearn.datasets import make_blobs
>>> X, y = make_blobs(n_samples=150,
...                   n_features=2,
...                   centers=3,
...                   cluster_std=0.5,
...                   shuffle=True,
...                   random_state=0)
>>> import matplotlib.pyplot as plt
>>> plt.scatter(X[:, 0],
...             X[:, 1],
...             c='white',
...             marker='o',
...             edgecolor='black',
...             s=50)
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

The dataset that we just created consists of 150 randomly generated points that are roughly grouped into three regions with higher density, which is visualized via a two-dimensional scatterplot:

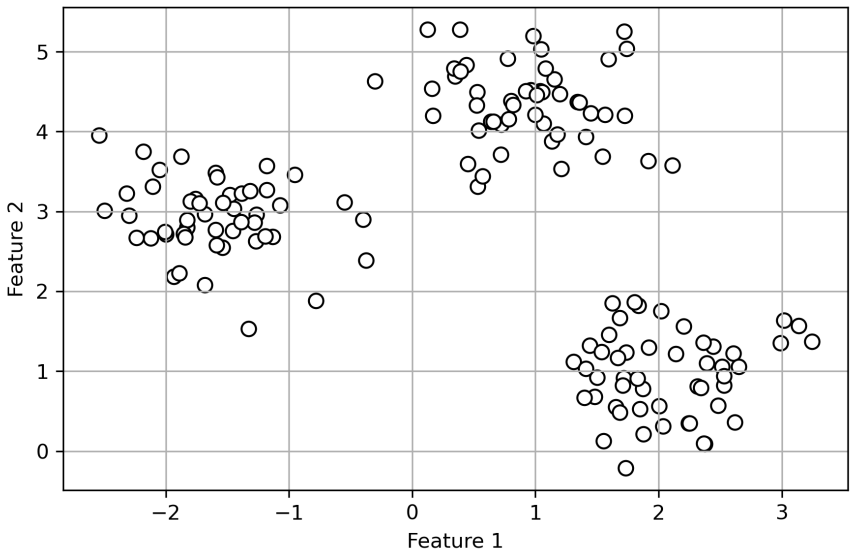


Figure 10.1: A scatterplot of our unlabeled dataset

In real-world applications of clustering, we do not have any ground-truth category information (information provided as empirical evidence as opposed to inference) about those examples; if we were given class labels, this task would fall into the category of supervised learning. Thus, our goal is to group the examples based on their feature similarities, which can be achieved using the k-means algorithm, as summarized by the following four steps:

1. Randomly pick k centroids from the examples as initial cluster centers
2. Assign each example to the nearest centroid, $\mu^{(j)}, j \in \{1, \ldots, k\}$
3. Move the centroids to the center of the examples that were assigned to it
4. Repeat *steps 2* and *3* until the cluster assignments do not change or a user-defined tolerance or maximum number of iterations is reached

Now, the next question is, *how do we measure similarity between objects*? We can define similarity as the opposite of distance, and a commonly used distance for clustering examples with continuous features is the **squared Euclidean distance** between two points, $\boldsymbol{x}$ and $\boldsymbol{y}$, in m-dimensional space:

$$d(\boldsymbol{x}, \boldsymbol{y})^2 = \sum_{j=1}^{m} \left(x_j - y_j\right)^2 = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$$

Note that, in the preceding equation, the index j refers to the jth dimension (feature column) of the example inputs, $\boldsymbol{x}$ and $\boldsymbol{y}$. In the rest of this section, we will use the superscripts i and j to refer to the index of the example (data record) and cluster index, respectively.

Based on this Euclidean distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing the within-cluster **sum of squared errors** (**SSE**), which is sometimes also called **cluster inertia**:

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

Here, $\boldsymbol{\mu}^{(j)}$ is the representative point (centroid) for cluster j. $w^{(i,j)} = 1$ if the example, $\boldsymbol{x}^{(i)}$, is in cluster j, or 0 otherwise.

$$w^{(i,j)} = \begin{cases} 1, & \text{if } \boldsymbol{x}^{(i)} \in j \\ 0, & \text{otherwise} \end{cases}$$

Now that you have learned how the simple k-means algorithm works, let's apply it to our example dataset using the `KMeans` class from scikit-learn's `cluster` module:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...             init='random',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
```

Using the preceding code, we set the number of desired clusters to `3`; having to specify the number of clusters *a priori* is one of the limitations of k-means. We set `n_init=10` to run the k-means clustering algorithms 10 times independently, with different random centroids to choose the final model as the one with the lowest SSE. Via the `max_iter` parameter, we specify the maximum number of iterations for each single run (here, `300`). Note that the k-means implementation in scikit-learn stops early if it converges before the maximum number of iterations is reached. However, it is possible that k-means does not reach convergence for a particular run, which can be problematic (computationally expensive) if we choose relatively large values for `max_iter`. One way to deal with convergence problems is to choose larger values for `tol`, which is a parameter that controls the tolerance with regard to the changes in the within-cluster SSE to declare convergence. In the preceding code, we chose a tolerance of `1e-04` (=0.0001).

A problem with k-means is that one or more clusters can be empty. Note that this problem does not exist for k-medoids or fuzzy C-means, an algorithm that we will discuss later in this section. However, this problem is accounted for in the current k-means implementation in scikit-learn. If a cluster is empty, the algorithm will search for the example that is farthest away from the centroid of the empty cluster. Then, it will reassign the centroid to be this farthest point.

Feature scaling

When we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the features are measured on the same scale and apply z-score standardization or min-max scaling if necessary.

Having predicted the cluster labels, `y_km`, and discussed some of the challenges of the k-means algorithm, let's now visualize the clusters that k-means identified in the dataset together with the cluster centroids. These are stored under the `cluster_centers_` attribute of the fitted `KMeans` object:

```
>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             marker='s', edgecolor='black',
...             label='Cluster 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50, c='orange',
...             marker='o', edgecolor='black',
...             label='Cluster 2')
>>> plt.scatter(X[y_km == 2, 0],
...             X[y_km == 2, 1],
...             s=50, c='lightblue',
...             marker='v', edgecolor='black',
...             label='Cluster 3')
>>> plt.scatter(km.cluster_centers_[:, 0],
...             km.cluster_centers_[:, 1],
...             s=250, marker='*',
...             c='red', edgecolor='black',
...             label='Centroids')
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.legend(scatterpoints=1)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

In *Figure 10.2*, you can see that k-means placed the three centroids at the center of each sphere, which looks like a reasonable grouping given this dataset:

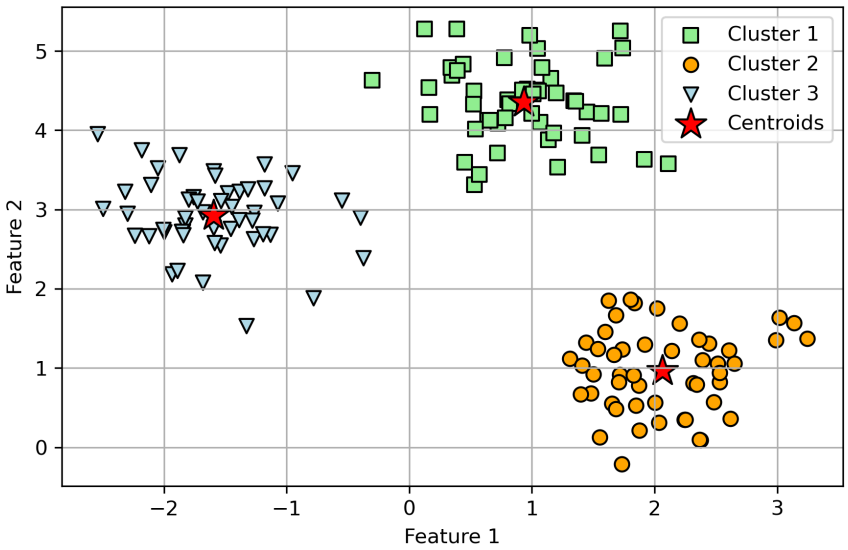


Figure 10.2: The k-means clusters and their centroids

Although k-means worked well on this toy dataset, we still have the drawback of having to specify the number of clusters, *k*, *a priori*. The number of clusters to choose may not always be so obvious in real-world applications, especially if we are working with a higher-dimensional dataset that cannot be visualized. The other properties of k-means are that clusters do not overlap and are not hierarchical, and we also assume that there is at least one item in each cluster. Later in this chapter, we will encounter different types of clustering algorithms, hierarchical and density-based clustering. Neither type of algorithm requires us to specify the number of clusters upfront or assume spherical structures in our dataset.

In the next subsection, we will cover a popular variant of the classic k-means algorithm called **k-means++**. While it doesn't address those assumptions and drawbacks of k-means that were discussed in the previous paragraph, it can greatly improve the clustering results through more clever seeding of the initial cluster centers.

A smarter way of placing the initial cluster centroids using k-means++

So far, we have discussed the classic k-means algorithm, which uses a random seed to place the initial centroids, which can sometimes result in bad clusterings or slow convergence if the initial centroids are chosen poorly. One way to address this issue is to run the k-means algorithm multiple times on a dataset and choose the best-performing model in terms of the SSE.

Another strategy is to place the initial centroids far away from each other via the k-means++ algorithm, which leads to better and more consistent results than the classic k-means (*k-means++: The Advantages of Careful Seeding* by *D. Arthur* and *S. Vassilvitskii* in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027-1035. *Society for Industrial and Applied Mathematics*, 2007).

The initialization in k-means++ can be summarized as follows:

1. Initialize an empty set, $\mathbf{M}$, to store the k centroids being selected.
2. Randomly choose the first centroid, $\boldsymbol{\mu}^{(j)}$, from the input examples and assign it to $\mathbf{M}$.
3. For each example, $\boldsymbol{x}^{(i)}$, that is not in $\mathbf{M}$, find the minimum squared distance, $d(\boldsymbol{x}^{(i)}, \mathbf{M})^2$, to any of the centroids in $\mathbf{M}$.
4. To randomly select the next centroid, $\boldsymbol{\mu}^{(p)}$, use a weighted probability distribution equal to $\frac{d(\boldsymbol{\mu}^{(p)}, \mathbf{M})^2}{\sum_i d(\boldsymbol{x}^{(i)}, \mathbf{M})^2}$. For instance, we collect all points in an array and choose a weighted random sampling, such that the larger the squared distance, the more likely a point gets chosen as the centroid.
5. Repeat *steps 3* and *4* until k centroids are chosen.
6. Proceed with the classic k-means algorithm.

To use k-means++ with scikit-learn's `KMeans` object, we just need to set the `init` parameter to `'k-means++'`. In fact, `'k-means++'` is the default argument to the `init` parameter, which is strongly recommended in practice. The only reason we didn't use it in the previous example was to not introduce too many concepts all at once. The rest of this section on k-means will use k-means++, but you are encouraged to experiment more with the two different approaches (classic k-means via `init='random'` versus k-means++ via `init='k-means++'`) for placing the initial cluster centroids.

Hard versus soft clustering

Hard clustering describes a family of algorithms where each example in a dataset is assigned to exactly one cluster, as in the k-means and k-means++ algorithms that we discussed earlier in this chapter. In contrast, algorithms for **soft clustering** (sometimes also called **fuzzy clustering**) assign an example to one or more clusters. A popular example of soft clustering is the **fuzzy C-means** (**FCM**) algorithm (also called **soft k-means** or **fuzzy k-means**). The original idea goes back to the 1970s, when Joseph C. Dunn first proposed an early version of fuzzy clustering to improve k-means (*A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters*, 1973). Almost a decade later, James C. Bedzek published his work on the improvement of the fuzzy clustering algorithm, which is now known as the FCM algorithm (*Pattern Recognition with Fuzzy Objective Function Algorithms*, *Springer Science+Business Media*, 2013).

The FCM procedure is very similar to k-means. However, we replace the hard cluster assignment with probabilities for each point belonging to each cluster. In k-means, we could express the cluster membership of an example, x, with a sparse vector of binary values:

$$\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 1 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0 \end{bmatrix}$$

Here, the index position with value 1 indicates the cluster centroid, $\boldsymbol{\mu}^{(j)}$, that the example is assigned to (assuming $k = 3$, $j \in \{1, 2, 3\}$). In contrast, a membership vector in FCM could be represented as follows:

$$\begin{bmatrix} x \in \mu^{(1)} & \rightarrow & w^{(i,j)} = 0.1 \\ x \in \mu^{(2)} & \rightarrow & w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} & \rightarrow & w^{(i,j)} = 0.05 \end{bmatrix}$$

Here, each value falls in the range [0, 1] and represents a probability of membership of the respective cluster centroid. The sum of the memberships for a given example is equal to 1. As with the k-means algorithm, we can summarize the FCM algorithm in four key steps:

1. Specify the number of k centroids and randomly assign the cluster memberships for each point
2. Compute the cluster centroids, $\boldsymbol{\mu}^{(j)}, j \in \{1, \ldots, k\}$
3. Update the cluster memberships for each point
4. Repeat *steps 2* and *3* until the membership coefficients do not change or a user-defined tolerance or maximum number of iterations is reached

The objective function of FCM—we abbreviate it as J_m—looks very similar to the within-cluster SSE that we minimize in k-means:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)^m} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

However, note that the membership indicator, $w^{(i,j)}$, is not a binary value as in k-means ($w^{(i,j)} \in \{0, 1\}$), but a real value that denotes the cluster membership probability ($w^{(i,j)} \in [0, 1]$). You also may have noticed that we added an additional exponent to $w^{(i,j)}$; the exponent m, any number greater than or equal to one (typically $m = 2$), is the so-called **fuzziness coefficient** (or simply **fuzzifier**), which controls the degree of *fuzziness*.

The larger the value of m, the smaller the cluster membership, $w^{(i,j)}$, becomes, which leads to fuzzier clusters. The cluster membership probability itself is calculated as follows:

$$w^{(i,j)} = \left[\sum_{c=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(c)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

For example, if we chose three cluster centers, as in the previous k-means example, we could calculate the membership of $\boldsymbol{x}^{(i)}$ belonging to the $\boldsymbol{\mu}^{(j)}$ cluster as follows:

$$w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

The center, $\boldsymbol{\mu}^{(j)}$, of a cluster itself is calculated as the mean of all examples weighted by the degree to which each example belongs to that cluster ($w^{(i,j)^m}$):

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{(i,j)^m} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{(i,j)^m}}$$

Just by looking at the equation to calculate the cluster memberships, we can say that each iteration in FCM is more expensive than an iteration in k-means. On the other hand, FCM typically requires fewer iterations overall to reach convergence. However, it has been found, in practice, that both k-means and FCM produce very similar clustering outputs, as described in a study (*Comparative Analysis of k-means and Fuzzy C-Means Algorithms* by *S. Ghosh* and *S. K. Dubey*, *IJACSA*, 4: 35–38, 2013). Unfortunately, the FCM algorithm is not implemented in scikit-learn currently, but interested readers can try out the FCM implementation from the scikit-fuzzy package, which is available at `https://github.com/scikit-fuzzy/scikit-fuzzy`.

Using the elbow method to find the optimal number of clusters

One of the main challenges in unsupervised learning is that we do not know the definitive answer. We don't have the ground-truth class labels in our dataset that allow us to apply the techniques that we used in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, to evaluate the performance of a supervised model. Thus, to quantify the quality of clustering, we need to use intrinsic metrics—such as the within-cluster SSE (distortion)—to compare the performance of different k-means clustering models.

Conveniently, we don't need to compute the within-cluster SSE explicitly when we are using scikit-learn, as it is already accessible via the `inertia_` attribute after fitting a `KMeans` model:

```
>>> print(f'Distortion: {km.inertia_:.2f}')
Distortion: 72.48
```

Based on the within-cluster SSE, we can use a graphical tool, the so-called **elbow method**, to estimate the optimal number of clusters, k, for a given task. We can say that if k increases, the distortion will decrease. This is because the examples will be closer to the centroids they are assigned to. The idea behind the elbow method is to identify the value of k where the distortion begins to increase most rapidly, which will become clearer if we plot the distortion for different values of k:

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 random_state=0)
...     km.fit(X)
...     distortions.append(km.inertia_)
```

```
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Number of clusters')
>>> plt.ylabel('Distortion')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 10.3*, the *elbow* is located at $k = 3$, so this is supporting evidence that $k = 3$ is indeed a good choice for this dataset:

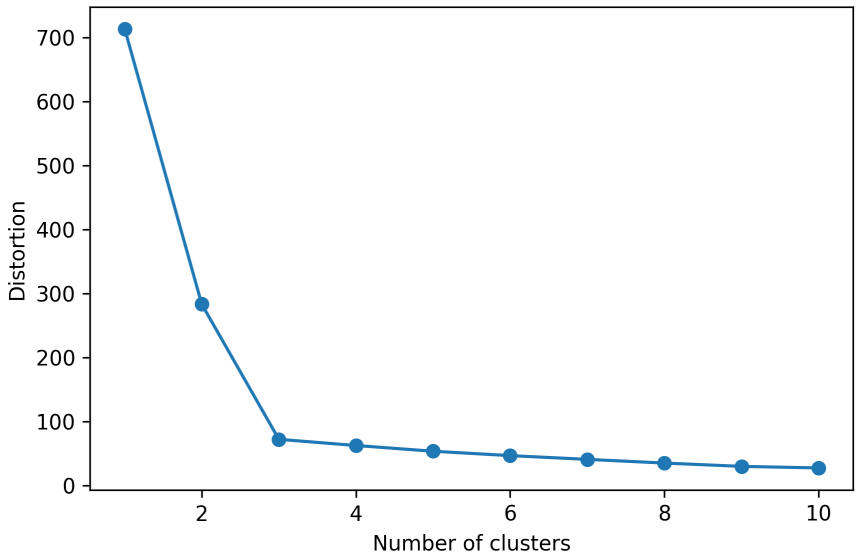


Figure 10.3: Finding the optimal number of clusters using the elbow method

Quantifying the quality of clustering via silhouette plots

Another intrinsic metric to evaluate the quality of a clustering is **silhouette analysis**, which can also be applied to clustering algorithms other than k-means that we will discuss later in this chapter. Silhouette analysis can be used as a graphical tool to plot a measure of how tightly grouped the examples in the clusters are. To calculate the **silhouette coefficient** of a single example in our dataset, we can apply the following three steps:

1. Calculate the **cluster cohesion**, $a^{(i)}$, as the average distance between an example, $\boldsymbol{x}^{(i)}$, and all other points in the same cluster.
2. Calculate the **cluster separation**, $b^{(i)}$, from the next closest cluster as the average distance between the example, $\boldsymbol{x}^{(i)}$, and all examples in the nearest cluster.
3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesion and separation divided by the greater of the two, as shown here:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{max\{b^{(i)}, a^{(i)}\}}$$

The silhouette coefficient is bounded in the range –1 to 1. Based on the preceding equation, we can see that the silhouette coefficient is 0 if the cluster separation and cohesion are equal ($b^{(i)} = a^{(i)}$). Furthermore, we get close to an ideal silhouette coefficient of 1 if $b^{(i)} >> a^{(i)}$, since $b^{(i)}$ quantifies how dissimilar an example is from other clusters, and $a^{(i)}$ tells us how similar it is to the other examples in its own cluster.

The silhouette coefficient is available as `silhouette_samples` from scikit-learn's `metric` module, and optionally, the `silhouette_scores` function can be imported for convenience. The `silhouette_scores` function calculates the average silhouette coefficient across all examples, which is equivalent to `numpy.mean(silhouette_samples(...))`. By executing the following code, we will now create a plot of the silhouette coefficients for a k-means clustering with $k = 3$:

```
>>> km = KMeans(n_clusters=3,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(
...     X, y_km, metric='euclidean'
... )
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
```

```
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...             color="red",
...             linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

Through a visual inspection of the silhouette plot, we can quickly scrutinize the sizes of the different clusters and identify clusters that contain *outliers*:

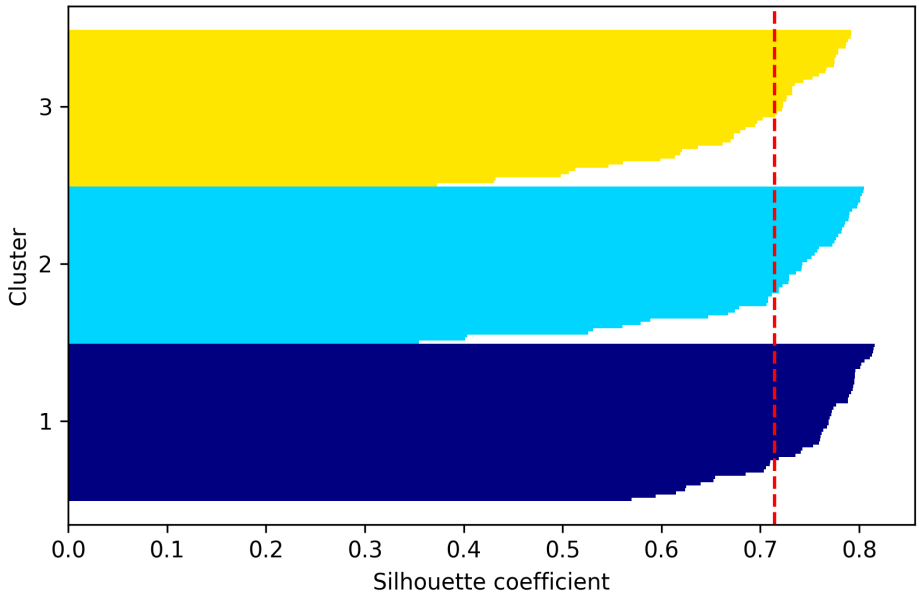


Figure 10.4: A silhouette plot for a good example of clustering

However, as you can see in the preceding silhouette plot, the silhouette coefficients are not close to 0 and are approximately equally far away from the average silhouette score, which is, in this case, an indicator of *good* clustering. Furthermore, to summarize the goodness of our clustering, we added the average silhouette coefficient to the plot (dotted line).

To see what a silhouette plot looks like for a relatively *bad* clustering, let's seed the k-means algorithm with only two centroids:

```
>>> km = KMeans(n_clusters=2,
...             init='k-means++',
...             n_init=10,
...             max_iter=300,
...             tol=1e-04,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> plt.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             s=50, c='lightgreen',
...             edgecolor='black',
...             marker='s',
...             label='Cluster 1')
>>> plt.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             s=50,
...             c='orange',
...             edgecolor='black',
...             marker='o',
...             label='Cluster 2')
>>> plt.scatter(km.cluster_centers_[:, 0],
...             km.cluster_centers_[:, 1],
...             s=250,
...             marker='*',
...             c='red',
...             label='Centroids')
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.legend()
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 10.5*, one of the centroids falls between two of the three spherical groupings of the input data.

Although the clustering does not look completely terrible, it is suboptimal:

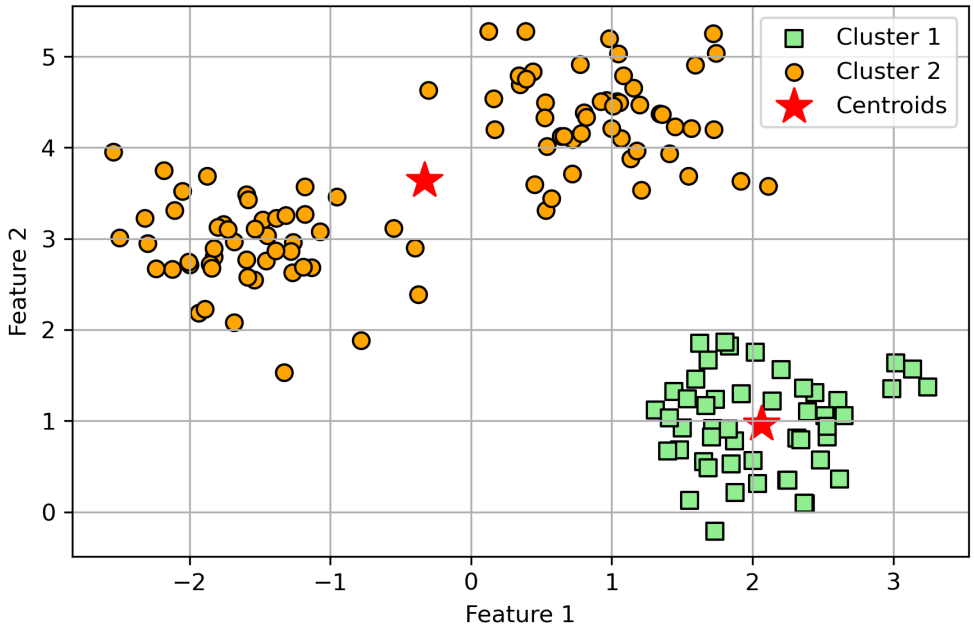


Figure 10.5: A suboptimal example of clustering

Please keep in mind that we typically do not have the luxury of visualizing datasets in two-dimensional scatterplots in real-world problems, since we typically work with data in higher dimensions. So, next, we will create the silhouette plot to evaluate the results:

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(
...     X, y_km, metric='euclidean'
... )
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...              c_silhouette_vals,
...              height=1.0,
...              edgecolor='none',
...              color=color)
```

```
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Cluster')
>>> plt.xlabel('Silhouette coefficient')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in *Figure 10.6*, the silhouettes now have visibly different lengths and widths, which is evidence of a relatively *bad* or at least *suboptimal* clustering:

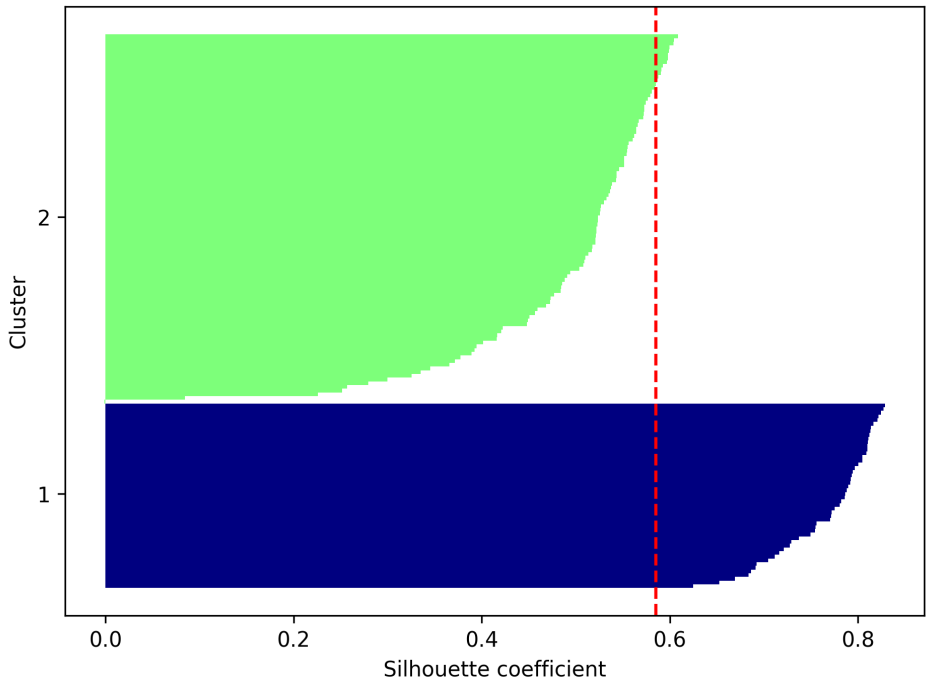


Figure 10.6: A silhouette plot for a suboptimal example of clustering

Now, after we have gained a good understanding of how clustering works, the next section will introduce hierarchical clustering as an alternative approach to k-means.

Organizing clusters as a hierarchical tree

In this section, we will look at an alternative approach to prototype-based clustering: **hierarchical clustering**. One advantage of the hierarchical clustering algorithm is that it allows us to plot **dendrograms** (visualizations of a binary hierarchical clustering), which can help with the interpretation of the results by creating meaningful taxonomies. Another advantage of this hierarchical approach is that we do not need to specify the number of clusters upfront.

The two main approaches to hierarchical clustering are **agglomerative** and **divisive** hierarchical clustering. In divisive hierarchical clustering, we start with one cluster that encompasses the complete dataset, and we iteratively split the cluster into smaller clusters until each cluster only contains one example. In this section, we will focus on agglomerative clustering, which takes the opposite approach. We start with each example as an individual cluster and merge the closest pairs of clusters until only one cluster remains.

Grouping clusters in a bottom-up fashion

The two standard algorithms for agglomerative hierarchical clustering are **single linkage** and **complete linkage**. Using single linkage, we compute the distances between the most similar members for each pair of clusters and merge the two clusters for which the distance between the most similar members is the smallest. The complete linkage approach is similar to single linkage but, instead of comparing the most similar members in each pair of clusters, we compare the most dissimilar members to perform the merge. This is shown in *Figure 10.7*:

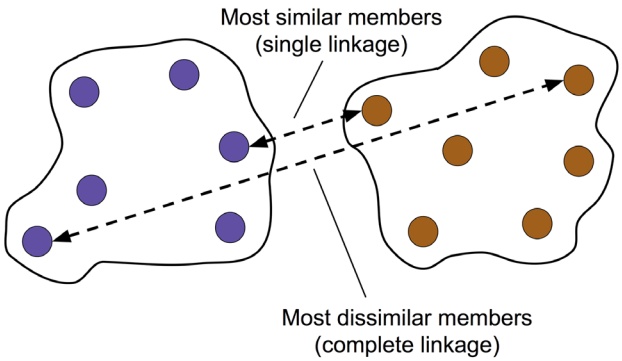


Figure 10.7: The complete linkage approach

Alternative types of linkages

Other commonly used algorithms for agglomerative hierarchical clustering include average linkage and Ward's linkage. In average linkage, we merge the cluster pairs based on the minimum average distances between all group members in the two clusters. In Ward's linkage, the two clusters that lead to the minimum increase of the total within-cluster SSE are merged.

In this section, we will focus on agglomerative clustering using the complete linkage approach. Hierarchical complete linkage clustering is an iterative procedure that can be summarized by the following steps:

1. Compute a pair-wise distance matrix of all examples.
2. Represent each data point as a singleton cluster.

3. Merge the two closest clusters based on the distance between the most dissimilar (distant) members.
4. Update the cluster linkage matrix.
5. Repeat *steps 2-4* until one single cluster remains.

Next, we will discuss how to compute the distance matrix (*step 1*). But first, let's generate a random data sample to work with. The rows represent different observations (IDs 0-4), and the columns are the different features (`X`, `Y`, `Z`) of those examples:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random_sample([5, 3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

After executing the preceding code, we should now see the following `DataFrame` containing the randomly generated examples:

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807642	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Figure 10.8: A randomly generated data sample

Performing hierarchical clustering on a distance matrix

To calculate the distance matrix as input for the hierarchical clustering algorithm, we will use the `pdist` function from SciPy's `spatial.distance` submodule:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...                         pdist(df, metric='euclidean')),
...                         columns=labels, index=labels)
>>> row_dist
```

Using the preceding code, we calculated the Euclidean distance between each pair of input examples in our dataset based on the features `X`, `Y`, and `Z`.

We provided the condensed distance matrix—returned by `pdist`—as input to the `squareform` function to create a symmetrical matrix of the pair-wise distances, as shown here:

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Figure 10.9: The calculated pair-wise distances of our data

Next, we will apply the complete linkage agglomeration to our clusters using the `linkage` function from SciPy's `cluster.hierarchy` submodule, which returns a so-called **linkage matrix**.

However, before we call the `linkage` function, let's take a careful look at the function documentation:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
[...]
Parameters:
  y : ndarray
    A condensed or redundant distance matrix. A condensed
    distance matrix is a flat array containing the upper
    triangular of the distance matrix. This is the form
    that pdist returns. Alternatively, a collection of m
    observation vectors in n dimensions may be passed as
    an m by n array.

  method : str, optional
    The linkage algorithm to use. See the Linkage Methods
    section below for full descriptions.

  metric : str, optional
    The distance metric to use. See the distance.pdist
    function for a list of valid distance metrics.

  Returns:
  Z : ndarray
```

```
    The hierarchical clustering encoded as a linkage matrix.
[...]
```

Based on the function description, we understand that we can use a condensed distance matrix (upper triangular) from the `pdist` function as an input attribute. Alternatively, we could also provide the initial data array and use the `'euclidean'` metric as a function argument in `linkage`. However, we should not use the `squareform` distance matrix that we defined earlier, since it would yield different distance values than expected. To sum it up, the three possible scenarios are listed here:

- **Incorrect approach**: Using the `squareform` distance matrix as shown in the following code snippet leads to incorrect results:

```
>>> row_clusters = linkage(row_dist,
...                        method='complete',
...                        metric='euclidean')
```

- **Correct approach**: Using the condensed distance matrix as shown in the following code example yields the correct linkage matrix:

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                        method='complete')
```

- **Correct approach**: Using the complete input example matrix (the so-called design matrix) as shown in the following code snippet also leads to a correct linkage matrix similar to the preceding approach:

```
>>> row_clusters = linkage(df.values,
...                        method='complete',
...                        metric='euclidean')
```

To take a closer look at the clustering results, we can turn those results into a pandas `DataFrame` (best viewed in a Jupyter notebook) as follows:

```
>>> pd.DataFrame(row_clusters,
...              columns=['row label 1',
...                       'row label 2',
...                       'distance',
...                       'no. of items in clust.'],
...              index=[f'cluster {(i + 1)}' for i in
...                     range(row_clusters.shape[0])])
```

As shown in *Figure 10.10*, the linkage matrix consists of several rows where each row represents one merge. The first and second columns denote the most dissimilar members in each cluster, and the third column reports the distance between those members.

The last column returns the count of the members in each cluster:

	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0.0	4.0	3.835396	2.0
cluster 2	1.0	2.0	4.347073	2.0
cluster 3	3.0	5.0	5.899885	3.0
cluster 4	6.0	7.0	8.316594	5.0

Figure 10.10: The linkage matrix

Now that we have computed the linkage matrix, we can visualize the results in the form of a dendrogram:

```
>>> from scipy.cluster.hierarchy import dendrogram
>>> # make dendrogram black (part 1/2)
>>> # from scipy.cluster.hierarchy import set_link_color_palette
>>> # set_link_color_palette(['black'])
>>> row_dendr = dendrogram(
...     row_clusters,
...     labels=labels,
...     # make dendrogram black (part 2/2)
...     # color_threshold=np.inf
... )
>>> plt.tight_layout()
>>> plt.ylabel('Euclidean distance')
>>> plt.show()
```

If you are executing the preceding code or reading an e-book version of this book, you will notice that the branches in the resulting dendrogram are shown in different colors. The color scheme is derived from a list of Matplotlib colors that are cycled for the distance thresholds in the dendrogram. For example, to display the dendrograms in black, you can uncomment the respective sections that were inserted in the preceding code:

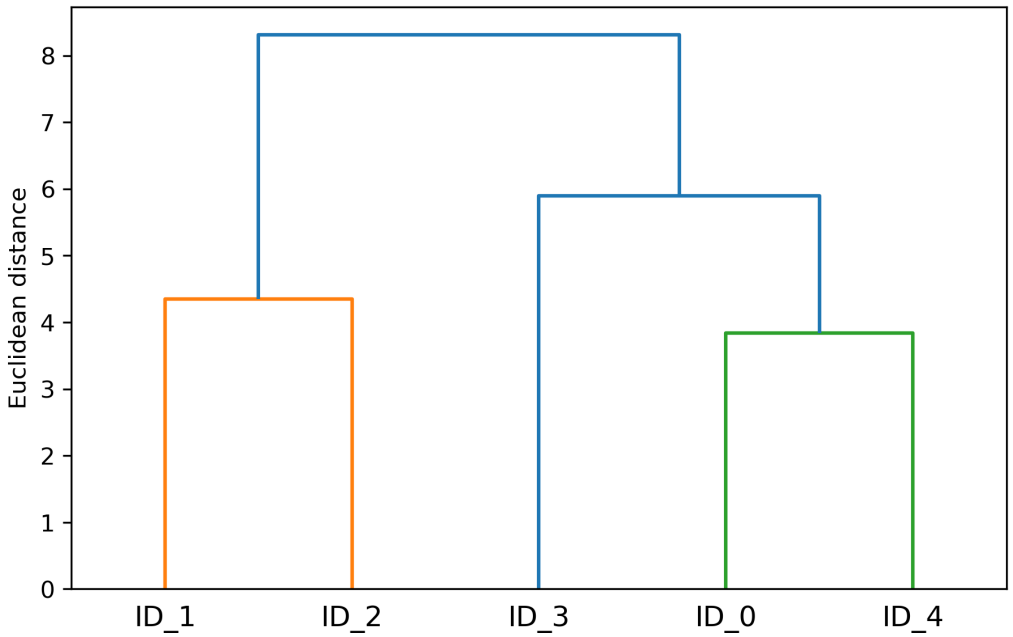


Figure 10.11: A dendrogram of our data

Such a dendrogram summarizes the different clusters that were formed during the agglomerative hierarchical clustering; for example, you can see that the examples `ID_0` and `ID_4`, followed by `ID_1` and `ID_2`, are the most similar ones based on the Euclidean distance metric.

Attaching dendrograms to a heat map

In practical applications, hierarchical clustering dendrograms are often used in combination with a **heat map**, which allows us to represent the individual values in the data array or matrix containing our training examples with a color code. In this section, we will discuss how to attach a dendrogram to a heat map plot and order the rows in the heat map correspondingly.

However, attaching a dendrogram to a heat map can be a little bit tricky, so let's go through this procedure step by step:

1. We create a new `figure` object and define the *x* axis position, *y* axis position, width, and height of the dendrogram via the `add_axes` attribute. Furthermore, we rotate the dendrogram 90 degrees counterclockwise. The code is as follows:

```
>>> fig = plt.figure(figsize=(8, 8), facecolor='white')
>>> axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
>>> row_dendr = dendrogram(row_clusters,
...                        orientation='left')
>>> # note: for matplotlib < v1.5.1, please use
>>> # orientation='right'
```

2. Next, we reorder the data in our initial `DataFrame` according to the clustering labels that can be accessed from the `dendrogram` object, which is essentially a Python dictionary, via the `leaves` key. The code is as follows:

```
>>> df_rowclust = df.iloc[row_dendr['leaves'][::-1]]
```

3. Now, we construct the heat map from the reordered `DataFrame` and position it next to the dendrogram:

```
>>> axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])
>>> cax = axm.matshow(df_rowclust,
...                   interpolation='nearest',
...                   cmap='hot_r')
```

4. Finally, we modify the aesthetics of the dendrogram by removing the axis ticks and hiding the axis spines. Also, we add a color bar and assign the feature and data record names to the *x* and *y* axis tick labels, respectively:

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels(['']  + list(df_rowclust.columns))
>>> axm.set_yticklabels(['']  + list(df_rowclust.index))
>>> plt.show()
```

After following the previous steps, the heat map should be displayed with the dendrogram attached:

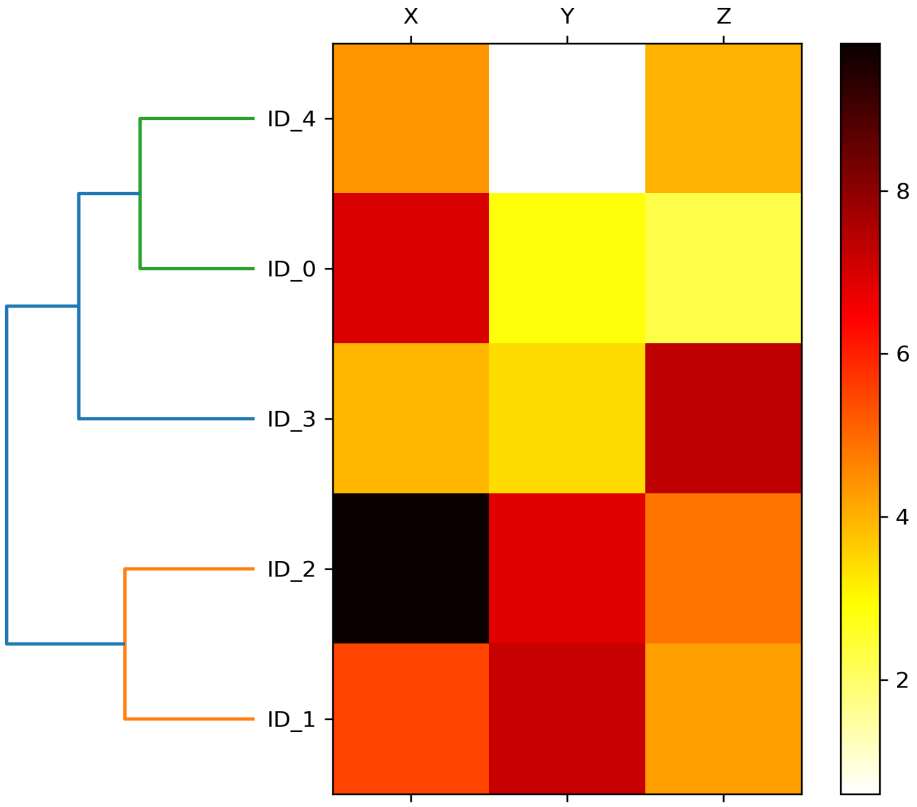


Figure 10.12: A heat map and dendrogram of our data

As you can see, the order of rows in the heat map reflects the clustering of the examples in the dendrogram. In addition to a simple dendrogram, the color-coded values of each example and feature in the heat map provide us with a nice summary of the dataset.

Applying agglomerative clustering via scikit-learn

In the previous subsection, you saw how to perform agglomerative hierarchical clustering using SciPy. However, there is also an `AgglomerativeClustering` implementation in scikit-learn, which allows us to choose the number of clusters that we want to return. This is useful if we want to prune the hierarchical cluster tree.

By setting the `n_cluster` parameter to `3`, we will now cluster the input examples into three groups using the same complete linkage approach based on the Euclidean distance metric as before:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=3,
...                              affinity='euclidean',
...                              linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print(f'Cluster labels: {labels}')
Cluster labels: [1 0 0 2 1]
```

Looking at the predicted cluster labels, we can see that the first and the fifth examples (`ID_0` and `ID_4`) were assigned to one cluster (label `1`), and the examples `ID_1` and `ID_2` were assigned to a second cluster (label `0`). The example `ID_3` was put into its own cluster (label `2`). Overall, the results are consistent with the results that we observed in the dendrogram. We should note, though, that `ID_3` is more similar to `ID_4` and `ID_0` than to `ID_1` and `ID_2`, as shown in the preceding dendrogram figure; this is not clear from scikit-learn's clustering results. Let's now rerun the `AgglomerativeClustering` using `n_cluster=2` in the following code snippet:

```
>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print(f'Cluster labels: {labels}')
Cluster labels: [0 1 1 0 0]
```

As you can see, in this *pruned* clustering hierarchy, label `ID_3` was assigned to the same cluster as `ID_0` and `ID_4`, as expected.

Locating regions of high density via DBSCAN

Although we can't cover the vast number of different clustering algorithms in this chapter, let's at least include one more approach to clustering: **density-based spatial clustering of applications with noise** (**DBSCAN**), which does not make assumptions about spherical clusters like k-means, nor does it partition the dataset into hierarchies that require a manual cut-off point. As its name implies, density-based clustering assigns cluster labels based on dense regions of points. In DBSCAN, the notion of density is defined as the number of points within a specified radius, ε.

According to the DBSCAN algorithm, a special label is assigned to each example (data point) using the following criteria:

- A point is considered a **core point** if at least a specified number (MinPts) of neighboring points fall within the specified radius, ε
- A **border point** is a point that has fewer neighbors than MinPts within ε, but lies within the ε radius of a core point
- All other points that are neither core nor border points are considered **noise points**

After labeling the points as core, border, or noise, the DBSCAN algorithm can be summarized in two simple steps:

1. Form a separate cluster for each core point or connected group of core points. (Core points are connected if they are no farther away than ε.)
2. Assign each border point to the cluster of its corresponding core point.

To get a better understanding of what the result of DBSCAN can look like, before jumping to the implementation, let's summarize what we have just learned about core points, border points, and noise points in *Figure 10.13*:

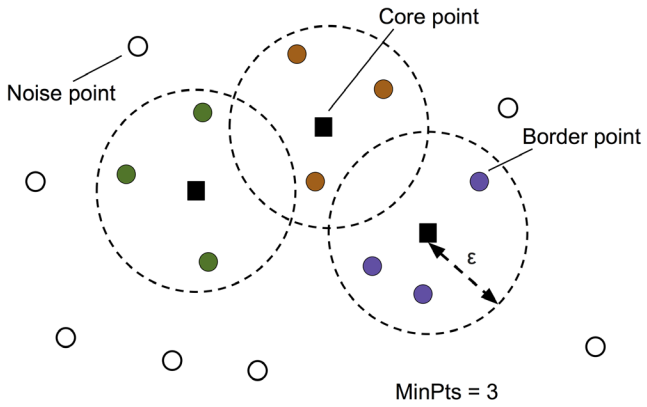


Figure 10.13: Core, noise, and border points for DBSCAN

One of the main advantages of using DBSCAN is that it does not assume that the clusters have a spherical shape as in k-means. Furthermore, DBSCAN is different from k-means and hierarchical clustering in that it doesn't necessarily assign each point to a cluster but is capable of removing noise points.

For a more illustrative example, let's create a new dataset of half-moon-shaped structures to compare k-means clustering, hierarchical clustering, and DBSCAN:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                   noise=0.05,
...                   random_state=0)
>>> plt.scatter(X[:, 0], X[:, 1])
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see in the resulting plot, there are two visible, half-moon-shaped groups consisting of 100 examples (data points) each:

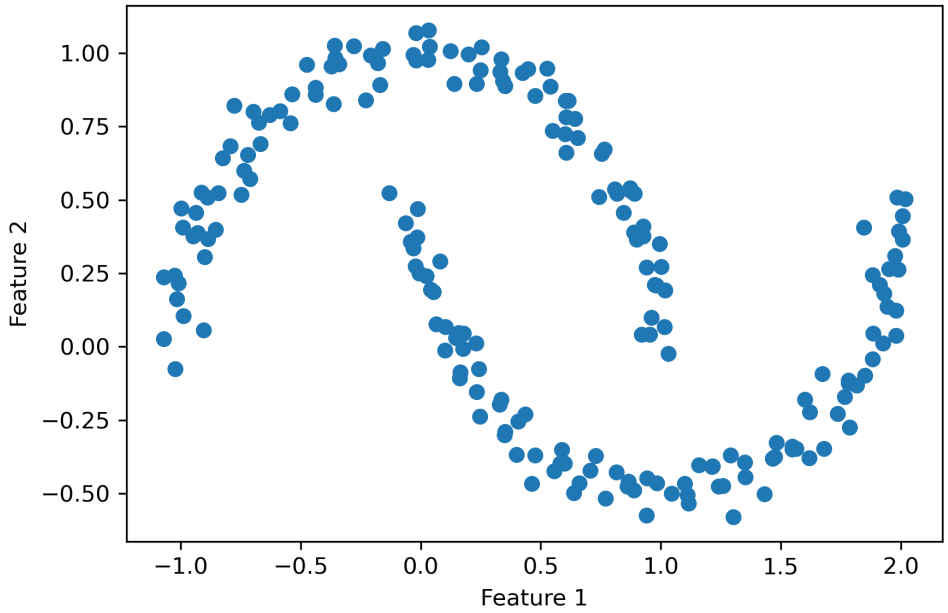


Figure 10.14: A two-feature half-moon-shaped dataset

We will start by using the k-means algorithm and complete linkage clustering to see if one of those previously discussed clustering algorithms can successfully identify the half-moon shapes as separate clusters. The code is as follows:

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
>>> km = KMeans(n_clusters=2,
...             random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km == 0, 0],
...             X[y_km == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='cluster 1')
```

```
>>> ax1.scatter(X[y_km == 1, 0],
...             X[y_km == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='cluster 2')
>>> ax1.set_title('K-means clustering')
>>> ax1.set_xlabel('Feature 1')
>>> ax1.set_ylabel('Feature 2')

>>> ac = AgglomerativeClustering(n_clusters=2,
...                              affinity='euclidean',
...                              linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac == 0, 0],
...             X[y_ac == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='Cluster 1')
>>> ax2.scatter(X[y_ac == 1, 0],
...             X[y_ac == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='Cluster 2')
>>> ax2.set_title('Agglomerative clustering')
>>> ax2.set_xlabel('Feature 1')
>>> ax2.set_ylabel('Feature 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

Based on the visualized clustering results, we can see that the k-means algorithm was unable to separate the two clusters, and also, the hierarchical clustering algorithm was challenged by those complex shapes:

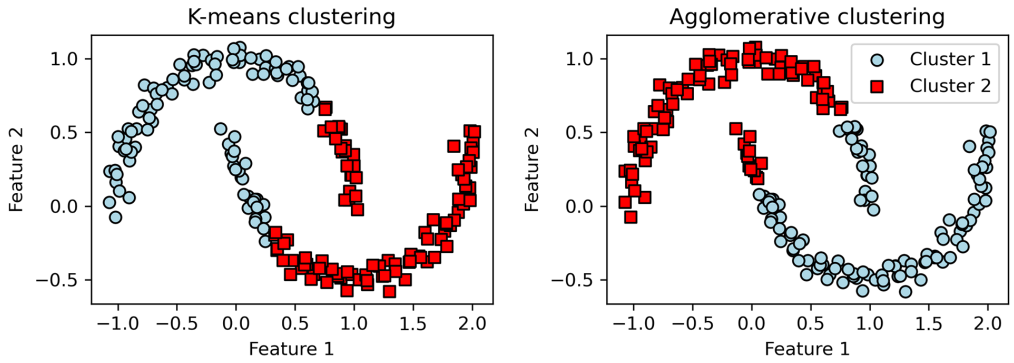


Figure 10.15: k-means and agglomerative clustering on the half-moon-shaped dataset

Finally, let's try the DBSCAN algorithm on this dataset to see if it can find the two half-moon-shaped clusters using a density-based approach:

```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...             min_samples=5,
...             metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db == 0, 0],
...             X[y_db == 0, 1],
...             c='lightblue',
...             edgecolor='black',
...             marker='o',
...             s=40,
...             label='Cluster 1')
>>> plt.scatter(X[y_db == 1, 0],
...             X[y_db == 1, 1],
...             c='red',
...             edgecolor='black',
...             marker='s',
...             s=40,
...             label='Cluster 2')
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

The DBSCAN algorithm can successfully detect the half-moon shapes, which highlights one of the strengths of DBSCAN—clustering data of arbitrary shapes:

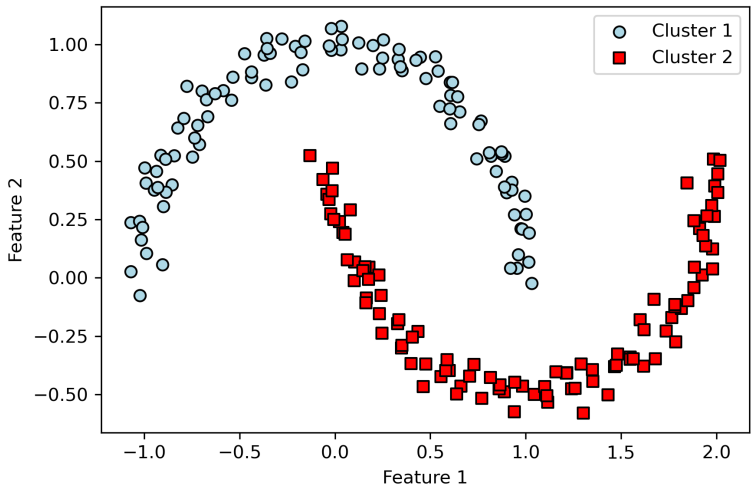


Figure 10.16: DBSCAN clustering on the half-moon-shaped dataset

However, we should also note some of the disadvantages of DBSCAN. With an increasing number of features in our dataset—assuming a fixed number of training examples—the negative effect of the **curse of dimensionality** increases. This is especially a problem if we are using the Euclidean distance metric. However, the problem of the curse of dimensionality is not unique to DBSCAN: it also affects other clustering algorithms that use the Euclidean distance metric, for example, k-means and hierarchical clustering algorithms. In addition, we have two hyperparameters in DBSCAN (MinPts and ε) that need to be optimized to yield good clustering results. Finding a good combination of MinPts and ε can be problematic if the density differences in the dataset are relatively large.

Graph-based clustering

So far, we have seen three of the most fundamental categories of clustering algorithms: prototype-based clustering with k-means, agglomerative hierarchical clustering, and density-based clustering via DBSCAN. However, there is also a fourth class of more advanced clustering algorithms that we have not covered in this chapter: graph-based clustering. Probably the most prominent members of the graph-based clustering family are the spectral clustering algorithms.

Although there are many different implementations of spectral clustering, what they all have in common is that they use the eigenvectors of a similarity or distance matrix to derive the cluster relationships. Since spectral clustering is beyond the scope of this book, you can read the excellent tutorial by Ulrike von Luxburg to learn more about this topic (*A tutorial on spectral clustering*, *Statistics and Computing*, 17(4): 395-416, 2007). It is freely available from arXiv at `http://arxiv.org/pdf/0711.0189v1.pdf`.

Note that, in practice, it is not always obvious which clustering algorithm will perform best on a given dataset, especially if the data comes in multiple dimensions that make it hard or impossible to visualize. Furthermore, it is important to emphasize that a successful clustering does not only depend on the algorithm and its hyperparameters; rather, the choice of an appropriate distance metric and the use of domain knowledge that can help to guide the experimental setup can be even more important.

In the context of the curse of dimensionality, it is thus common practice to apply dimensionality reduction techniques prior to performing clustering. Such dimensionality reduction techniques for unsupervised datasets include principal component analysis and t-SNE, which we covered in *Chapter 5*, *Compressing Data via Dimensionality Reduction*. Also, it is particularly common to compress datasets down to two-dimensional subspaces, which allows us to visualize the clusters and assigned labels using two-dimensional scatterplots, which are particularly helpful for evaluating the results.

Summary

In this chapter, you learned about three different clustering algorithms that can help us with the discovery of hidden structures or information in data. We started with a prototype-based approach, k-means, which clusters examples into spherical shapes based on a specified number of cluster centroids. Since clustering is an unsupervised method, we do not enjoy the luxury of ground-truth labels to evaluate the performance of a model. Thus, we used intrinsic performance metrics, such as the elbow method or silhouette analysis, as an attempt to quantify the quality of clustering.

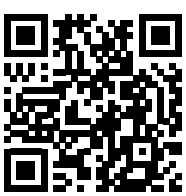
We then looked at a different approach to clustering: agglomerative hierarchical clustering. Hierarchical clustering does not require specifying the number of clusters upfront, and the result can be visualized in a dendrogram representation, which can help with the interpretation of the results. The last clustering algorithm that we covered in this chapter was DBSCAN, an algorithm that groups points based on local densities and is capable of handling outliers and identifying non-globular shapes.

After this excursion into the field of unsupervised learning, it is now time to introduce some of the most exciting machine learning algorithms for supervised learning: multilayer artificial neural networks. After their recent resurgence, neural networks are once again the hottest topic in machine learning research. Thanks to recently developed deep learning algorithms, neural networks are considered state of the art for many complex tasks such as image classification, natural language processing, and speech recognition. In *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*, we will construct our own multilayer neural network. In *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, we will work with the PyTorch library, which specializes in training neural network models with multiple layers very efficiently by utilizing graphics processing units.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:

`https://packt.link/MLwPyTorch`



11

Implementing a Multilayer Artificial Neural Network from Scratch

As you may know, deep learning is getting a lot of attention from the press and is, without doubt, the hottest topic in the machine learning field. Deep learning can be understood as a subfield of machine learning that is concerned with training artificial **neural networks** (**NNs**) with many layers efficiently. In this chapter, you will learn the basic concepts of artificial NNs so that you are well equipped for the following chapters, which will introduce advanced Python-based deep learning libraries and **deep neural network** (**DNN**) architectures that are particularly well suited for image and text analyses.

The topics that we will cover in this chapter are as follows:

- Gaining a conceptual understanding of multilayer NNs
- Implementing the fundamental backpropagation algorithm for NN training from scratch
- Training a basic multilayer NN for image classification

Modeling complex functions with artificial neural networks

At the beginning of this book, we started our journey through machine learning algorithms with artificial neurons in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*. Artificial neurons represent the building blocks of the multilayer artificial NNs that we will discuss in this chapter.

The basic concept behind artificial NNs was built upon hypotheses and models of how the human brain works to solve complex problem tasks. Although artificial NNs have gained a lot of popularity in recent years, early studies of NNs go back to the 1940s, when Warren McCulloch and Walter Pitts first described how neurons could work. (*A logical calculus of the ideas immanent in nervous activity*, by *W. S. McCulloch* and *W. Pitts*, *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.)

However, in the decades that followed the first implementation of the **McCulloch-Pitts neuron** model—Rosenblatt's perceptron in the 1950s—many researchers and machine learning practitioners slowly began to lose interest in NNs since no one had a good solution for training an NN with multiple layers. Eventually, interest in NNs was rekindled in 1986 when D.E. Rumelhart, G.E. Hinton, and R.J. Williams were involved in the (re)discovery and popularization of the backpropagation algorithm to train NNs more efficiently, which we will discuss in more detail later in this chapter (*Learning representations by backpropagating errors*, by *D.E. Rumelhart*, *G.E. Hinton*, and *R.J. Williams*, *Nature*, 323 (6088): 533–536, 1986). Readers who are interested in the history of **artificial intelligence** (**AI**), machine learning, and NNs are also encouraged to read the Wikipedia article on the so-called *AI winters*, which are the periods of time where a large portion of the research community lost interest in the study of NNs (`https://en.wikipedia.org/wiki/AI_winter`).

However, NNs are more popular today than ever thanks to the many breakthroughs that have been made in the previous decade, which resulted in what we now call deep learning algorithms and architectures—NNs that are composed of many layers. NNs are a hot topic not only in academic research but also in big technology companies, such as Facebook, Microsoft, Amazon, Uber, Google, and many more that invest heavily in artificial NNs and deep learning research.

As of today, complex NNs powered by deep learning algorithms are considered state-of-the-art solutions for complex problem solving such as image and voice recognition. Some of the recent applications include:

- Predicting COVID-19 resource needs from a series of X-rays (`https://arxiv.org/abs/2101.04909`)
- Modeling virus mutations (`https://science.sciencemag.org/content/371/6526/284`)
- Leveraging data from social media platforms to manage extreme weather events (`https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-5973.12311`)
- Improving photo descriptions for people who are blind or visually impaired (`https://tech.fb.com/how-facebook-is-using-ai-to-improve-photo-descriptions-for-people-who-are-blind-or-visually-impaired/`)

Single-layer neural network recap

This chapter is all about multilayer NNs, how they work, and how to train them to solve complex problems. However, before we dig deeper into a particular multilayer NN architecture, let's briefly reiterate some of the concepts of single-layer NNs that we introduced in *Chapter 2*, namely, the **ADAptive LInear NEuron** (**Adaline**) algorithm, which is shown in *Figure 11.1*:

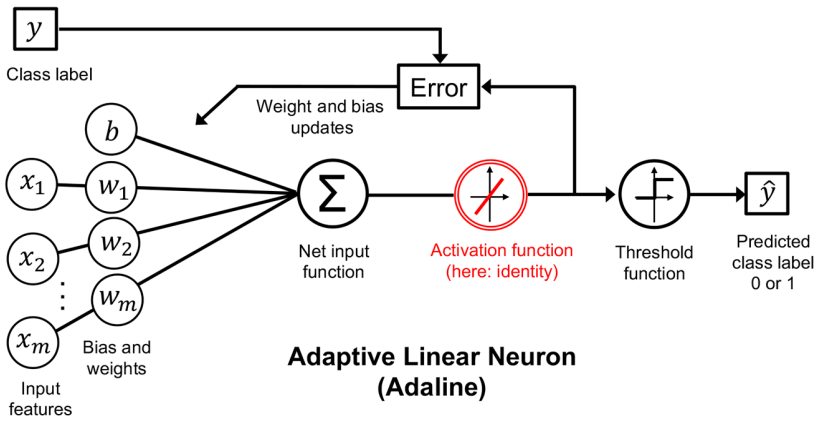


Figure 11.1: The Adaline algorithm

In *Chapter 2*, we implemented the Adaline algorithm to perform binary classification, and we used the gradient descent optimization algorithm to learn the weight coefficients of the model. In every epoch (pass over the training dataset), we updated the weight vector $\boldsymbol{w}$ and bias unit b using the following update rule:

$$\boldsymbol{w} \coloneqq \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b \coloneqq b + \Delta b$$

where $\Delta w_j = -\eta \frac{\partial L}{\partial w_j}$ and $\Delta b = -\eta \frac{\partial L}{\partial b}$ for the bias unit and each weight w_j in the weight vector $\boldsymbol{w}$.

In other words, we computed the gradient based on the whole training dataset and updated the weights of the model by taking a step in the opposite direction of the loss gradient $\nabla L(\boldsymbol{w})$. (For simplicity, we will focus on the weights and omit the bias unit in the following paragraphs; however, as you remember from *Chapter 2*, the same concepts apply.) In order to find the optimal weights of the model, we optimized an objective function that we defined as the **mean of squared errors** (**MSE**) loss function $L(\boldsymbol{w})$. Furthermore, we multiplied the gradient by a factor, the **learning rate** η, which we had to choose carefully to balance the speed of learning against the risk of overshooting the global minimum of the loss function.

In gradient descent optimization, we updated all weights simultaneously after each epoch, and we defined the partial derivative for each weight w_j in the weight vector, $\boldsymbol{w}$, as follows:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{n}\sum_i \left(y^{(i)} - a^{(i)}\right)^2 = -\frac{2}{n}\sum_i \left(y^{(i)} - a^{(i)}\right)x_j^{(i)}$$

Here, $y^{(i)}$ is the target class label of a particular sample $x^{(i)}$, and $a^{(i)}$ is the activation of the neuron, which is a linear function in the special case of Adaline.

Furthermore, we defined the activation function $\sigma(\cdot)$ as follows:

$$\sigma(\cdot) = z = a$$

Here, the net input, z, is a linear combination of the weights that are connecting the input layer to the output layer:

$$z = \sum_j w_j x_j + b = \boldsymbol{w}^T\boldsymbol{x} + b$$

While we used the activation $\sigma(\cdot)$ to compute the gradient update, we implemented a threshold function to squash the continuous-valued output into binary class labels for prediction:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0; \\ 0 & \text{otherwise} \end{cases}$$

Single-layer naming convention

Note that although Adaline consists of two layers, one input layer and one output layer, it is called a single-layer network because of its single link between the input and output layers.

Also, we learned about a certain *trick* to accelerate the model learning, the so-called **stochastic gradient descent** (**SGD**) optimization. SGD approximates the loss from a single training sample (online learning) or a small subset of training examples (mini-batch learning). We will make use of this concept later in this chapter when we implement and train a **multilayer perceptron** (**MLP**). Apart from faster learning—due to the more frequent weight updates compared to gradient descent—its noisy nature is also regarded as beneficial when training multilayer NNs with nonlinear activation functions, which do not have a convex loss function. Here, the added noise can help to escape local loss minima, but we will discuss this topic in more detail later in this chapter.

Introducing the multilayer neural network architecture

In this section, you will learn how to connect multiple single neurons to a multilayer feedforward NN; this special type of *fully connected* network is also called **MLP**.

Figure 11.2 illustrates the concept of an MLP consisting of two layers:

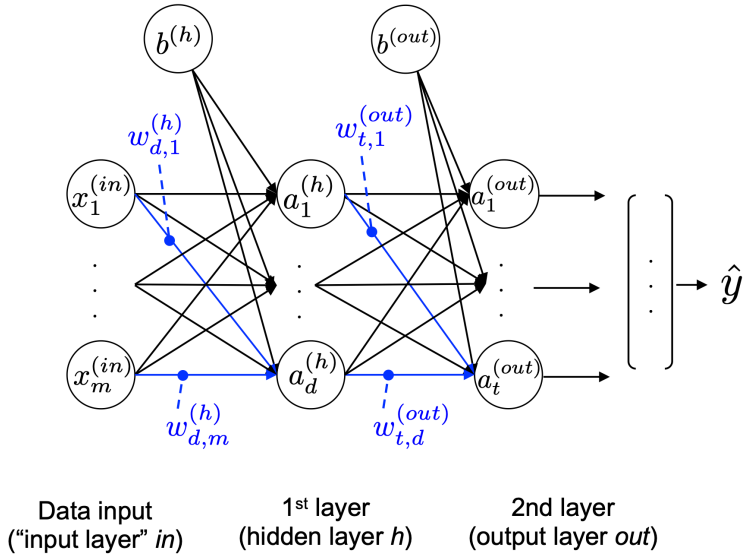


Figure 11.2: A two-layer MLP

Next to the data input, the MLP depicted in *Figure 11.2* has one hidden layer and one output layer. The units in the hidden layer are fully connected to the input features, and the output layer is fully connected to the hidden layer. If such a network has more than one hidden layer, we also call it a **deep NN**. (Note that in some contexts, the inputs are also regarded as a layer. However, in this case, it would make the Adaline model, which is a single-layer neural network, a two-layer neural network, which may be counterintuitive.)

Adding additional hidden layers

We can add any number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in an NN as additional hyperparameters that we want to optimize for a given problem task using the cross-validation technique, which we discussed in *Chapter 6, Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

However, the loss gradients for updating the network's parameters, which we will calculate later via backpropagation, will become increasingly small as more layers are added to a network. This vanishing gradient problem makes model learning more challenging. Therefore, special algorithms have been developed to help train such DNN structures; this is known as **deep learning**, which we will discuss in more detail in the following chapters.

As shown in *Figure 11.2*, we denote the *i*th activation unit in the *l*th layer as $a_i^{(l)}$. To make the math and code implementations a bit more intuitive, we will not use numerical indices to refer to layers, but we will use the *in* superscript for the input features, the *h* superscript for the hidden layer, and the *out* superscript for the output layer. For instance, $x_i^{(in)}$ refers to the *i*th input feature value, $a_i^{(h)}$ refers to the *i*th unit in the hidden layer, and $a_i^{(out)}$ refers to the *i*th unit in the output layer. Note that the $\boldsymbol{b}$'s in *Figure 11.2* denote the bias units. In fact, $\boldsymbol{b}^{(h)}$ and $\boldsymbol{b}^{(out)}$ are vectors with the number of elements being equal to the number of nodes in the layer they correspond to. For example, $\boldsymbol{b}^{(h)}$ stores d bias units, where d is the number of nodes in the hidden layer. If this sounds confusing, don't worry. Looking at the code implementation later, where we initialize weight matrices and bias unit vectors, will help clarify these concepts.

Each node in layer l is connected to all nodes in layer $l + 1$ via a weight coefficient. For example, the connection between the *k*th unit in layer l to the *j*th unit in layer $l + 1$ will be written as $w_{j,k}^{(l+1)}$. Referring back to *Figure 11.2*, we denote the weight matrix that connects the input to the hidden layer as $\boldsymbol{W}^{(h)}$, and we write the matrix that connects the hidden layer to the output layer as $\boldsymbol{W}^{(out)}$.

While one unit in the output layer would suffice for a binary classification task, we saw a more general form of an NN in the preceding figure, which allows us to perform multiclass classification via a generalization of the **one-versus-all** (**OvA**) technique. To better understand how this works, remember the **one-hot** representation of categorical variables that we introduced in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*.

For example, we can encode the three class labels in the familiar Iris dataset (0=*Setosa*, 1=*Versicolor*, 2=*Virginica*) as follows:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix}, 1 = \begin{bmatrix}0\\1\\0\end{bmatrix}, 2 = \begin{bmatrix}0\\0\\1\end{bmatrix}$$

This one-hot vector representation allows us to tackle classification tasks with an arbitrary number of unique class labels present in the training dataset.

If you are new to NN representations, the indexing notation (subscripts and superscripts) may look a little bit confusing at first. What may seem overly complicated at first will make much more sense in later sections when we vectorize the NN representation. As introduced earlier, we summarize the weights that connect the input and hidden layers by a $d \times m$ dimensional matrix $\boldsymbol{W}^{(h)}$, where d is the number of hidden units and m is the number of input units.

Activating a neural network via forward propagation

In this section, we will describe the process of **forward propagation** to calculate the output of an MLP model. To understand how it fits into the context of learning an MLP model, let's summarize the MLP learning procedure in three simple steps:

1. Starting at the input layer, we forward propagate the patterns of the training data through the network to generate an output.
2. Based on the network's output, we calculate the loss that we want to minimize using a loss function that we will describe later.

3. We backpropagate the loss, find its derivative with respect to each weight and bias unit in the network, and update the model.

Finally, after we repeat these three steps for multiple epochs and learn the weight and bias parameters of the MLP, we use forward propagation to calculate the network output and apply a threshold function to obtain the predicted class labels in the one-hot representation, which we described in the previous section.

Now, let's walk through the individual steps of forward propagation to generate an output from the patterns in the training data. Since each unit in the hidden layer is connected to all units in the input layers, we first calculate the activation unit of the hidden layer $a_1^{(h)}$ as follows:

$$z_1^{(h)} = x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \cdots + x_m^{(in)} w_{1,m}^{(h)}$$

$$a_1^{(h)} = \sigma\left(z_1^{(h)}\right)$$

Here, $z_1^{(h)}$ is the net input and $\sigma(\cdot)$ is the activation function, which has to be differentiable to learn the weights that connect the neurons using a gradient-based approach. To be able to solve complex problems such as image classification, we need nonlinear activation functions in our MLP model, for example, the sigmoid (logistic) activation function that we remember from the section about logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

As you may recall, the sigmoid function is an *S*-shaped curve that maps the net input z onto a logistic distribution in the range 0 to 1, which cuts the y axis at $z = 0$, as shown in *Figure 11.3*:

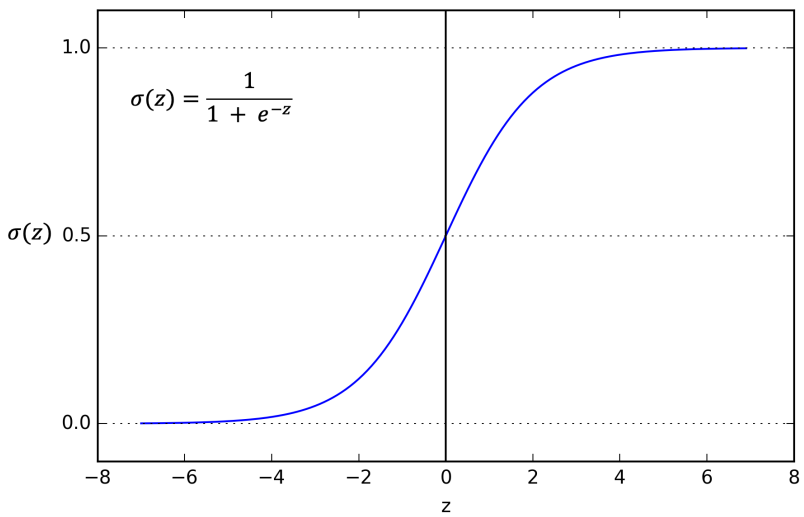


Figure 11.3: The sigmoid activation function

MLP is a typical example of a feedforward artificial NN. The term **feedforward** refers to the fact that each layer serves as the input to the next layer without loops, in contrast to recurrent NNs—an architecture that we will discuss later in this chapter and discuss in more detail in *Chapter 15*, *Modeling Sequential Data Using Recurrent Neural Networks*. The term *multilayer perceptron* may sound a little bit confusing since the artificial neurons in this network architecture are typically sigmoid units, not perceptrons. We can think of the neurons in the MLP as logistic regression units that return values in the continuous range between 0 and 1.

For purposes of code efficiency and readability, we will now write the activation in a more compact form using the concepts of basic linear algebra, which will allow us to vectorize our code implementation via NumPy rather than writing multiple nested and computationally expensive Python `for` loops:

$$z^{(h)} = \boldsymbol{x}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)}$$

$$a^{(h)} = \sigma\left(\boldsymbol{z}^{(h)}\right)$$

Here, $\boldsymbol{x}^{(in)}$ is our $1\times m$ dimensional feature vector. $\boldsymbol{W}^{(h)}$ is a $d\times m$ dimensional weight matrix where d is the number of units in the hidden layer; consequently, the transposed matrix $\boldsymbol{W}^{(h)T}$ is $m\times d$ dimensional. The bias vector $\boldsymbol{b}^{(h)}$ consists of d bias units (one bias unit per hidden node).

After matrix-vector multiplication, we obtain the $1\times d$ dimensional net input vector $\boldsymbol{z}^{(h)}$ to calculate the activation $\boldsymbol{a}^{(h)}$ (where $\boldsymbol{a}^{(h)} \in \mathbb{R}^{1\times d}$).

Furthermore, we can generalize this computation to all n examples in the training dataset:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{X}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)}$$

Here, $\boldsymbol{X}^{(in)}$ is now an $n\times m$ matrix, and the matrix multiplication will result in an $n\times d$ dimensional net input matrix, $\boldsymbol{Z}^{(h)}$. Finally, we apply the activation function $\sigma(\cdot)$ to each value in the net input matrix to get the $n\times d$ activation matrix in the next layer (here, the output layer):

$$\boldsymbol{A}^{(h)} = \sigma\left(\boldsymbol{Z}^{(h)}\right)$$

Similarly, we can write the activation of the output layer in vectorized form for multiple examples:

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)T} + \boldsymbol{b}^{(out)}$$

Here, we multiply the transpose of the $t\times d$ matrix $\boldsymbol{W}^{(out)}$ (t is the number of output units) by the $n\times d$ dimensional matrix, $\boldsymbol{A}^{(h)}$, and add the t dimensional bias vector $\boldsymbol{b}^{(out)}$ to obtain the $n\times t$ dimensional matrix, $\boldsymbol{Z}^{(out)}$. (The rows in this matrix represent the outputs for each example.)

Lastly, we apply the sigmoid activation function to obtain the continuous-valued output of our network:

$$\boldsymbol{A}^{(out)} = \sigma\left(\boldsymbol{Z}^{(out)}\right)$$

Similar to $\boldsymbol{Z}^{(out)}$, $\boldsymbol{A}^{(out)}$ is an $n\times t$ dimensional matrix.

Classifying handwritten digits

In the previous section, we covered a lot of the theory around NNs, which can be a little bit overwhelming if you are new to this topic. Before we continue with the discussion of the algorithm for learning the weights of the MLP model, backpropagation, let's take a short break from the theory and see an NN in action.

Additional resources on backpropagation

The NN theory can be quite complex; thus, we want to provide readers with additional resources that cover some of the topics we discuss in this chapter in more detail or from a different perspective:

- *Chapter 6*, *Deep Feedforward Networks*, *Deep Learning*, by *I. Goodfellow*, *Y. Bengio*, and *A. Courville*, MIT Press, 2016 (manuscripts freely accessible at `http://www.deeplearningbook.org`).
- *Pattern Recognition and Machine Learning*, by *C. M. Bishop*, Springer New York, 2006.
- Lecture video slides from Sebastian Raschka's deep learning course:
  `https://sebastianraschka.com/blog/2021/dl-course.html#l08-multinomial-logistic-regression--softmax-regression`
  `https://sebastianraschka.com/blog/2021/dl-course.html#l09-multilayer-perceptrons-and-backpropration`

In this section, we will implement and train our first multilayer NN to classify handwritten digits from the popular **Mixed National Institute of Standards and Technology** (**MNIST**) dataset that has been constructed by Yann LeCun and others and serves as a popular benchmark dataset for machine learning algorithms (*Gradient-Based Learning Applied to Document Recognition* by *Y. LeCun*, *L. Bottou*, *Y. Bengio*, and *P. Haffner*, *Proceedings of the IEEE*, 86(11): 2278-2324, 1998).

Obtaining and preparing the MNIST dataset

The MNIST dataset is publicly available at `http://yann.lecun.com/exdb/mnist/` and consists of the following four parts:

1. **Training dataset images**: `train-images-idx3-ubyte.gz` (9.9 MB, 47 MB unzipped, and 60,000 examples)
2. **Training dataset labels**: `train-labels-idx1-ubyte.gz` (29 KB, 60 KB unzipped, and 60,000 labels)
3. **Test dataset images**: `t10k-images-idx3-ubyte.gz` (1.6 MB, 7.8 MB unzipped, and 10,000 examples)
4. **Test dataset labels**: `t10k-labels-idx1-ubyte.gz` (5 KB, 10 KB unzipped, and 10,000 labels)

The MNIST dataset was constructed from two datasets of the US **National Institute of Standards and Technology** (**NIST**). The training dataset consists of handwritten digits from 250 different people, 50 percent high school students, and 50 percent employees from the Census Bureau. Note that the test dataset contains handwritten digits from different people following the same split.

Instead of downloading the abovementioned dataset files and preprocessing them into NumPy arrays ourselves, we will use scikit-learn's new `fetch_openml` function, which allows us to load the MNIST dataset more conveniently:

```
>>> from sklearn.datasets import fetch_openml
>>> X, y = fetch_openml('mnist_784', version=1,
...                     return_X_y=True)
>>> X = X.values
>>> y = y.astype(int).values
```

In scikit-learn, the `fetch_openml` function downloads the MNIST dataset from OpenML (`https://www.openml.org/d/554`) as pandas `DataFrame` and Series objects, which is why we use the `.values` attribute to obtain the underlying NumPy arrays. (If you are using a scikit-learn version older than 1.0, `fetch_openml` downloads NumPy arrays directly so you can omit using the `.values` attribute.) The $n \times m$ dimensional `X` array consists of 70,000 images with 784 pixels each, and the `y` array stores the corresponding 70,000 class labels, which we can confirm by checking the dimensions of the arrays as follows:

```
>>> print(X.shape)
(70000, 784)
>>> print(y.shape)
(70000,)
```

The images in the MNIST dataset consist of 28×28 pixels, and each pixel is represented by a grayscale intensity value. Here, `fetch_openml` already unrolled the 28×28 pixels into one-dimensional row vectors, which represent the rows in our `X` array (784 per row or image) above. The second array (`y`) returned by the `fetch_openml` function contains the corresponding target variable, the class labels (integers 0-9) of the handwritten digits.

Next, let's normalize the pixels values in MNIST to the range –1 to 1 (originally 0 to 255) via the following code line:

```
>>> X = ((X / 255.) - .5) * 2
```

The reason behind this is that gradient-based optimization is much more stable under these conditions, as discussed in *Chapter 2*. Note that we scaled the images on a pixel-by-pixel basis, which is different from the feature-scaling approach that we took in previous chapters.

Previously, we derived scaling parameters from the training dataset and used these to scale each column in the training dataset and test dataset. However, when working with image pixels, centering them at zero and rescaling them to a [–1, 1] range is also common and usually works well in practice.

To get an idea of how those images in MNIST look, let's visualize examples of the digits 0-9 after reshaping the 784-pixel vectors from our feature matrix into the original 28×28 image that we can plot via Matplotlib's `imshow` function:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                        sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X[y == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a plot of the 2×5 subfigures showing a representative image of each unique digit:

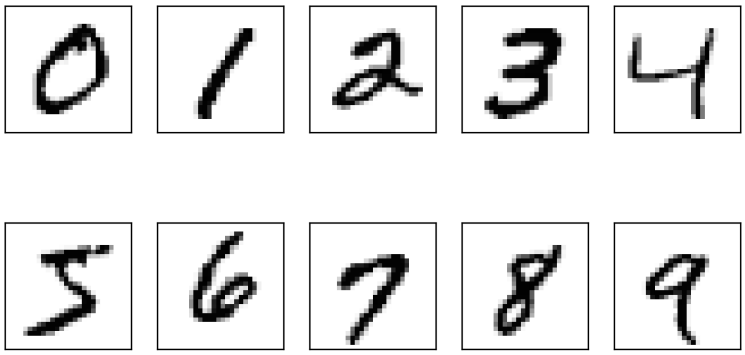


Figure 11.4: A plot showing one randomly chosen handwritten digit from each class

In addition, let's also plot multiple examples of the same digit to see how different the handwriting for each really is:

```
>>> fig, ax = plt.subplots(nrows=5,
...                        ncols=5,
...                        sharex=True,
...                        sharey=True)
>>> ax = ax.flatten()
```

```
>>> for i in range(25):
...     img = X[y == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we should now see the first 25 variants of the digit 7:

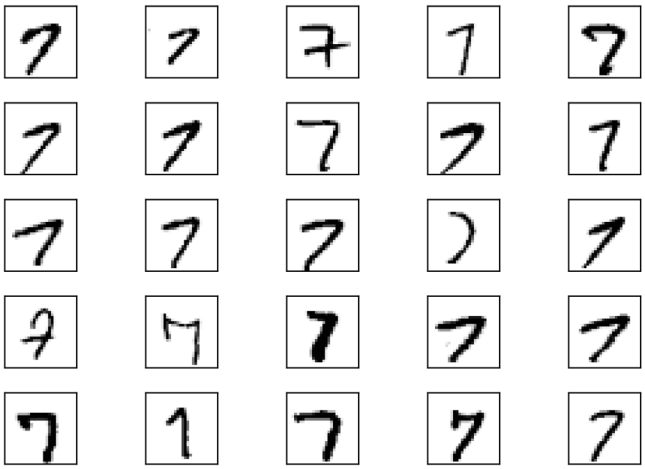


Figure 11.5: Different variants of the handwritten digit 7

Finally, let's divide the dataset into training, validation, and test subsets. The following code will split the dataset such that 55,000 images are used for training, 5,000 images for validation, and 10,000 images for testing:

```
>>> from sklearn.model_selection import train_test_split
>>> X_temp, X_test, y_temp, y_test = train_test_split(
...     X, y, test_size=10000, random_state=123, stratify=y
... )
>>> X_train, X_valid, y_train, y_valid = train_test_split(
...     X_temp, y_temp, test_size=5000,
...     random_state=123, stratify=y_temp
... )
```

Implementing a multilayer perceptron

In this subsection, we will now implement an MLP from scratch to classify the images in the MNIST dataset. To keep things simple, we will implement an MLP with only one hidden layer. Since the approach may seem a little bit complicated at first, you are encouraged to download the sample code for this chapter from the Packt Publishing website or from GitHub (`https://github.com/rasbt/machine-learning-book`) so that you can view this MLP implementation annotated with comments and syntax highlighting for better readability.

If you are not running the code from the accompanying Jupyter Notebook file or don't have access to the internet, copy the `NeuralNetMLP` code from this chapter into a Python script file in your current working directory (for example, `neuralnet.py`), which you can then import into your current Python session via the following command:

```
from neuralnet import NeuralNetMLP
```

The code will contain parts that we have not talked about yet, such as the backpropagation algorithm. Do not worry if not all the code makes immediate sense to you; we will follow up on certain parts later in this chapter. However, going over the code at this stage can make it easier to follow the theory later.

So, let's look at the following implementation of an MLP, starting with the two helper functions to compute the logistic sigmoid activation and to convert integer class label arrays to one-hot encoded labels:

```
import numpy as np

def sigmoid(z):
    return 1. / (1. + np.exp(-z))


def int_to_onehot(y, num_labels):

    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1

    return ary
```

Below, we implement the main class for our MLP, which we call `NeuralNetMLP`. There are three class methods, `.__init__()`, `.forward()`, and `.backward()`, that we will discuss one by one, starting with the `__init__` constructor:

```
class NeuralNetMLP:

    def __init__(self, num_features, num_hidden,
                 num_classes, random_seed=123):
        super().__init__()

        self.num_classes = num_classes

        # hidden
        rng = np.random.RandomState(random_seed)

        self.weight_h = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # output
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)
```

The `__init__` constructor instantiates the weight matrices and bias vectors for the hidden and the output layer. Next, let's see how these are used in the `forward` method to make predictions:

```
    def forward(self, x):
        # Hidden layer

        # input dim: [n_examples, n_features]
        #        dot [n_hidden, n_features].T
        # output dim: [n_examples, n_hidden]
        z_h = np.dot(x, self.weight_h.T) + self.bias_h
        a_h = sigmoid(z_h)

        # Output layer
        # input dim: [n_examples, n_hidden]
        #        dot [n_classes, n_hidden].T
        # output dim: [n_examples, n_classes]
        z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
        a_out = sigmoid(z_out)
        return a_h, a_out
```

The `forward` method takes in one or more training examples and returns the predictions. In fact, it returns both the activation values from the hidden layer and the output layer, `a_h` and `a_out`. While `a_out` represents the class-membership probabilities that we can convert to class labels, which we care about, we also need the activation values from the hidden layer, `a_h`, to optimize the model parameters; that is, the weight and bias units of the hidden and output layers.

Finally, let's talk about the `backward` method, which updates the weight and bias parameters of the neural network:

```
    def backward(self, x, a_h, a_out, y):

        ##########################
        ### Output layer weights
        ##########################

        # one-hot encoding
        y_onehot = int_to_onehot(y, self.num_classes)

        # Part 1: dLoss/dOutWeights
        ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
        ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet
        ## for convenient re-use

        # input/output dim: [n_examples, n_classes]
        d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

        # input/output dim: [n_examples, n_classes]
        d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative

        # output dim: [n_examples, n_classes]
        delta_out = d_loss__d_a_out * d_a_out__d_z_out

        # gradient for output weights

        # [n_examples, n_hidden]
        d_z_out__dw_out = a_h

        # input dim: [n_classes, n_examples]
        #           dot [n_examples, n_hidden]
        # output dim: [n_classes, n_hidden]
        d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
        d_loss__db_out = np.sum(delta_out, axis=0)
```

```
        ##################################
        # Part 2: dLoss/dHiddenWeights
        ## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet
        #    * dHiddenNet/dWeight

        # [n_classes, n_hidden]
        d_z_out__a_h = self.weight_out

        # output dim: [n_examples, n_hidden]
        d_loss__a_h = np.dot(delta_out, d_z_out__a_h)

        # [n_examples, n_hidden]
        d_a_h__d_z_h = a_h * (1. - a_h) # sigmoid derivative

        # [n_examples, n_features]
        d_z_h__d_w_h = x

        # output dim: [n_hidden, n_features]
        d_loss__d_w_h = np.dot((d_loss__a_h * d_a_h__d_z_h).T,
                               d_z_h__d_w_h)
        d_loss__d_b_h = np.sum((d_loss__a_h * d_a_h__d_z_h), axis=0)

        return (d_loss__dw_out, d_loss__db_out,
                d_loss__d_w_h, d_loss__d_b_h)
```

The `backward` method implements the so-called *backpropagation* algorithm, which calculates the gradients of the loss with respect to the weight and bias parameters. Similar to Adaline, these gradients are then used to update these parameters via gradient descent. Note that multilayer NNs are more complex than their single-layer siblings, and we will go over the mathematical concepts of how to compute the gradients in a later section after discussing the code. For now, just consider the `backward` method as a way for computing gradients that are used for the gradient descent updates. For simplicity, the loss function this derivation is based on is the same MSE loss that we used in Adaline. In later chapters, we will look at alternative loss functions, such as multi-category cross-entropy loss, which is a generalization of the binary logistic regression loss to multiple classes.

Looking at this code implementation of the `NeuralNetMLP` class, you may have noticed that this object-oriented implementation differs from the familiar scikit-learn API that is centered around the `.fit()` and `.predict()` methods. Instead, the main methods of the `NeuralNetMLP` class are the `.forward()` and `.backward()` methods. One of the reasons behind this is that it makes a complex neural network a bit easier to understand in terms of how the information flows through the networks.

Another reason is that this implementation is relatively similar to how more advanced deep learning libraries such as PyTorch operate, which we will introduce and use in the upcoming chapters to implement more complex neural networks.

After we have implemented the `NeuralNetMLP` class, we use the following code to instantiate a new `NeuralNetMLP` object:

```
>>> model = NeuralNetMLP(num_features=28*28,
...                      num_hidden=50,
...                      num_classes=10)
```

The `model` accepts MNIST images reshaped into 784-dimensional vectors (in the format of `X_train`, `X_valid`, or `X_test`, which we defined previously) for the 10 integer classes (digits 0-9). The hidden layer consists of 50 nodes. Also, as you may be able to tell from looking at the previously defined `.forward()` method, we use a sigmoid activation function after the first hidden layer and output layer to keep things simple. In later chapters, we will learn about alternative activation functions for both the hidden and output layers.

Figure 11.6 summarizes the neural network architecture that we instantiated above:

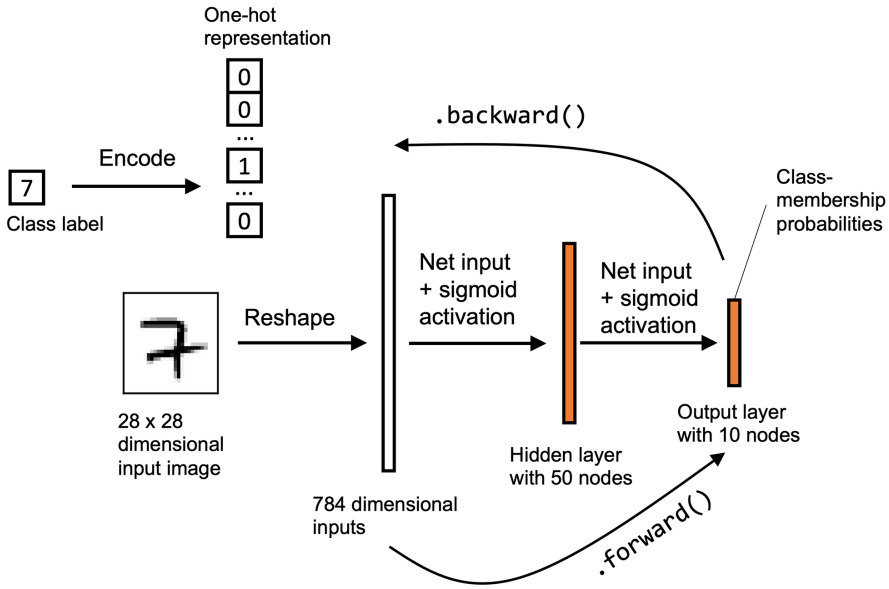


Figure 11.6: The NN architecture for labeling handwritten digits

In the next subsection, we are going to implement the training function that we can use to train the network on mini-batches of the data via backpropagation.

Coding the neural network training loop

Now that we have implemented the `NeuralNetMLP` class in the previous subsection and initiated a model, the next step is to train the model. We will tackle this in multiple steps. First, we will define some helper functions for data loading. Second, we will embed these functions into the training loop that iterates over the dataset in multiple epochs.

The first function we are going to define is a mini-batch generator, which takes in our dataset and divides it into mini-batches of a desired size for stochastic gradient descent training. The code is as follows:

```
>>> import numpy as np
>>> num_epochs = 50
>>> minibatch_size = 100

>>> def minibatch_generator(X, y, minibatch_size):
...     indices = np.arange(X.shape[0])
...     np.random.shuffle(indices)
...     for start_idx in range(0, indices.shape[0] - minibatch_size
...                            + 1, minibatch_size):
...         batch_idx = indices[start_idx:start_idx + minibatch_size]
...         yield X[batch_idx], y[batch_idx]
```

Before we move on to the next functions, let's confirm that the mini-batch generator works as intended and produces mini-batches of the desired size. The following code will attempt to iterate through the dataset, and then we will print the dimension of the mini-batches. Note that in the following code examples, we will remove the `break` statements. The code is as follows:

```
>>> # iterate over training epochs
>>> for i in range(num_epochs):
...     # iterate over minibatches
...     minibatch_gen = minibatch_generator(
...         X_train, y_train, minibatch_size)
...     for X_train_mini, y_train_mini in minibatch_gen:
...         break
...     break
>>> print(X_train_mini.shape)
(100, 784)
>>> print(y_train_mini.shape)
(100,)
```

As we can see, the network returns mini-batches of size 100 as intended.

Next, we have to define our loss function and performance metric that we can use to monitor the training process and evaluate the model. The MSE loss and accuracy function can be implemented as follows:

```
>>> def mse_loss(targets, probas, num_labels=10):
...     onehot_targets = int_to_onehot(
...         targets, num_labels=num_labels
...     )
...     return np.mean((onehot_targets - probas)**2)

>>> def accuracy(targets, predicted_labels):
...     return np.mean(predicted_labels == targets)
```

Let's test the preceding function and compute the initial validation set MSE and accuracy of the model we instantiated in the previous section:

```
>>> _, probas = model.forward(X_valid)
>>> mse = mse_loss(y_valid, probas)
>>> print(f'Initial validation MSE: {mse:.1f}')
Initial validation MSE: 0.3

>>> predicted_labels = np.argmax(probas, axis=1)
>>> acc = accuracy(y_valid, predicted_labels)
>>> print(f'Initial validation accuracy: {acc*100:.1f}%')
Initial validation accuracy: 9.4%
```

In this code example, note that `model.forward()` returns the hidden and output layer activations. Remember that we have 10 output nodes (one corresponding to each unique class label). Hence, when computing the MSE, we first converted the class labels into one-hot encoded class labels in the `mse_loss()` function. In practice, it does not make a difference whether we average over the row or the columns of the squared-difference matrix first, so we simply call `np.mean()` without any axis specification so that it returns a scalar.

The output layer activations, since we used the logistic sigmoid function, are values in the range [0, 1]. For each input, the output layer produces 10 values in the range [0, 1], so we used the `np.argmax()` function to select the index position of the largest value, which yields the predicted class label. We then compared the true labels with the predicted class labels to compute the accuracy via the `accuracy()` function we defined. As we can see from the preceding output, the accuracy is not very high. However, given that we have a balanced dataset with 10 classes, a prediction accuracy of approximately 10 percent is what we would expect for an untrained model producing random predictions.

Using the previous code, we can compute the performance on, for example, the whole training set if we provide `y_train` as input to targets and the predicted labels from feeding the model with `X_train`. However, in practice, our computer memory is usually a limiting factor for how much data the model can ingest in one forward pass (due to the large matrix multiplications). Hence, we are defining our MSE and accuracy computation based on our previous mini-batch generator. The following function will compute the MSE and accuracy incrementally by iterating over the dataset one mini-batch at a time to be more memory-efficient:

```
>>> def compute_mse_and_acc(nnet, X, y, num_labels=10,
...                         minibatch_size=100):
...     mse, correct_pred, num_examples = 0., 0, 0
...     minibatch_gen = minibatch_generator(X, y, minibatch_size)
...     for i, (features, targets) in enumerate(minibatch_gen):
...         _, probas = nnet.forward(features)
...         predicted_labels = np.argmax(probas, axis=1)
...         onehot_targets = int_to_onehot(
...             targets, num_labels=num_labels
...         )
...         loss = np.mean((onehot_targets - probas)**2)
...         correct_pred += (predicted_labels == targets).sum()
...         num_examples += targets.shape[0]
...         mse += loss
...     mse = mse/i
...     acc = correct_pred/num_examples
...     return mse, acc
```

Before we implement the training loop, let's test the function and compute the initial training set MSE and accuracy of the model we instantiated in the previous section and make sure it works as intended:

```
>>> mse, acc = compute_mse_and_acc(model, X_valid, y_valid)
>>> print(f'Initial valid MSE: {mse:.1f}')
Initial valid MSE: 0.3
>>> print(f'Initial valid accuracy: {acc*100:.1f}%')
Initial valid accuracy: 9.4%
```

As we can see from the results, our generator approach produces the same results as the previously defined MSE and accuracy functions, except for a small rounding error in the MSE (0.27 versus 0.28), which is negligible for our purposes.

Let's now get to the main part and implement the code to train our model:

```
>>> def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
...           learning_rate=0.1):
...     epoch_loss = []
```

```
...     epoch_train_acc = []
...     epoch_valid_acc = []
...
...     for e in range(num_epochs):
...         # iterate over minibatches
...         minibatch_gen = minibatch_generator(
...             X_train, y_train, minibatch_size)
...         for X_train_mini, y_train_mini in minibatch_gen:
...             #### Compute outputs ####
...             a_h, a_out = model.forward(X_train_mini)

...             #### Compute gradients ####
...             d_loss__d_w_out, d_loss__d_b_out, \
...             d_loss__d_w_h, d_loss__d_b_h = \
...                 model.backward(X_train_mini, a_h, a_out,
...                                y_train_mini)
...
...             #### Update weights ####
...             model.weight_h -= learning_rate * d_loss__d_w_h
...             model.bias_h -= learning_rate * d_loss__d_b_h
...             model.weight_out -= learning_rate * d_loss__d_w_out
...             model.bias_out -= learning_rate * d_loss__d_b_out
...
...         #### Epoch Logging ####
...         train_mse, train_acc = compute_mse_and_acc(
...             model, X_train, y_train
...         )
...         valid_mse, valid_acc = compute_mse_and_acc(
...             model, X_valid, y_valid
...         )
...         train_acc, valid_acc = train_acc*100, valid_acc*100
...         epoch_train_acc.append(train_acc)
...         epoch_valid_acc.append(valid_acc)
...         epoch_loss.append(train_mse)
...         print(f'Epoch: {e+1:03d}/{num_epochs:03d} '
...               f'| Train MSE: {train_mse:.2f} '
...               f'| Train Acc: {train_acc:.2f}% '
...               f'| Valid Acc: {valid_acc:.2f}%')
...
...     return epoch_loss, epoch_train_acc, epoch_valid_acc
```

On a high level, the `train()` function iterates over multiple epochs, and in each epoch, it used the previously defined `minibatch_generator()` function to iterate over the whole training set in mini-batches for stochastic gradient descent training. Inside the mini-batch generator `for` loop, we obtain the outputs from the model, `a_h` and `a_out`, via its `.forward()` method. Then, we compute the loss gradients via the model's `.backward()` method—the theory will be explained in a later section. Using the loss gradients, we update the weights by adding the negative gradient multiplied by the learning rate. This is the same concept that we discussed earlier for Adaline. For example, to update the model weights of the hidden layer, we defined the following line:

```
model.weight_h -= learning_rate * d_loss__d_w_h
```

For a single weight, w_j, this corresponds to the following partial derivative-based update:

$$w_j := w_j - \eta \frac{\partial L}{\partial w_j}$$

Finally, the last portion of the previous code computes the losses and prediction accuracies on the training and test sets to track the training progress.

Let's now execute this function to train our model for 50 epochs, which may take a few minutes to finish:

```
>>> np.random.seed(123) # for the training set shuffling
>>> epoch_loss, epoch_train_acc, epoch_valid_acc = train(
...     model, X_train, y_train, X_valid, y_valid,
...     num_epochs=50, learning_rate=0.1)
```

During training, we should see the following output:

```
Epoch: 001/050 | Train MSE: 0.05 | Train Acc: 76.17% | Valid Acc: 76.02%
Epoch: 002/050 | Train MSE: 0.03 | Train Acc: 85.46% | Valid Acc: 84.94%
Epoch: 003/050 | Train MSE: 0.02 | Train Acc: 87.89% | Valid Acc: 87.64%
Epoch: 004/050 | Train MSE: 0.02 | Train Acc: 89.36% | Valid Acc: 89.38%
Epoch: 005/050 | Train MSE: 0.02 | Train Acc: 90.21% | Valid Acc: 90.16%
...
Epoch: 048/050 | Train MSE: 0.01 | Train Acc: 95.57% | Valid Acc: 94.58%
Epoch: 049/050 | Train MSE: 0.01 | Train Acc: 95.55% | Valid Acc: 94.54%
Epoch: 050/050 | Train MSE: 0.01 | Train Acc: 95.59% | Valid Acc: 94.74%
```

The reason why we print all this output is that, in NN training, it is really useful to compare training and validation accuracy. This helps us judge whether the network model performs well, given the architecture and hyperparameters. For example, if we observe a low training and validation accuracy, there is likely an issue with the training dataset, or the hyperparameters' settings are not ideal.

In general, training (deep) NNs is relatively expensive compared with the other models we've discussed so far. Thus, we want to stop it early in certain circumstances and start over with different hyperparameter settings. On the other hand, if we find that it increasingly tends to overfit the training data (noticeable by an increasing gap between training and validation dataset performance), we may want to stop the training early, as well.

In the next subsection, we will discuss the performance of our NN model in more detail.

Evaluating the neural network performance

Before we discuss backpropagation, the training procedure of NNs, in more detail in the next section, let's look at the performance of the model that we trained in the previous subsection.

In `train()`, we collected the training loss and the training and validation accuracy for each epoch so that we can visualize the results using Matplotlib. Let's look at the training MSE loss first:

```
>>> plt.plot(range(len(epoch_loss)), epoch_loss)
>>> plt.ylabel('Mean squared error')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

The preceding code plots the loss over the 50 epochs, as shown in *Figure 11.7*:

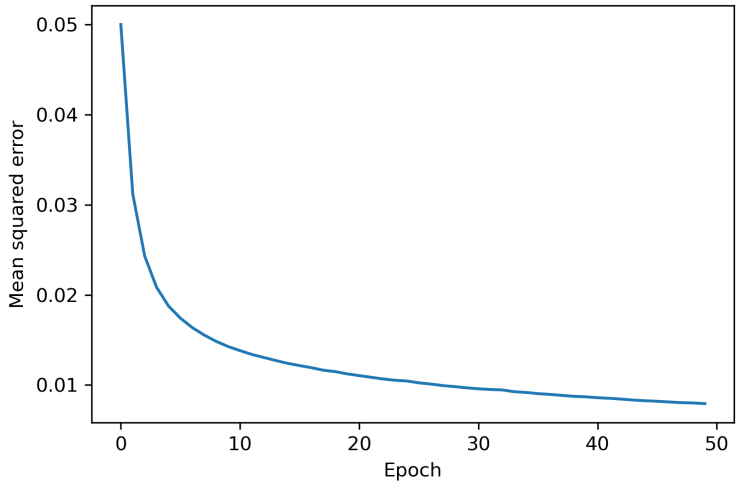


Figure 11.7: A plot of the MSE by the number of training epochs

As we can see, the loss decreased substantially during the first 10 epochs and seems to slowly converge in the last 10 epochs. However, the small slope between epoch 40 and epoch 50 indicates that the loss would further decrease with training over additional epochs.

Next, let's take a look at the training and validation accuracy:

```
>>> plt.plot(range(len(epoch_train_acc)), epoch_train_acc,
...          label='Training')
>>> plt.plot(range(len(epoch_valid_acc)), epoch_valid_acc,
...          label='Validation')
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Epochs')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

The preceding code examples plot those accuracy values over the 50 training epochs, as shown in *Figure 11.8*:

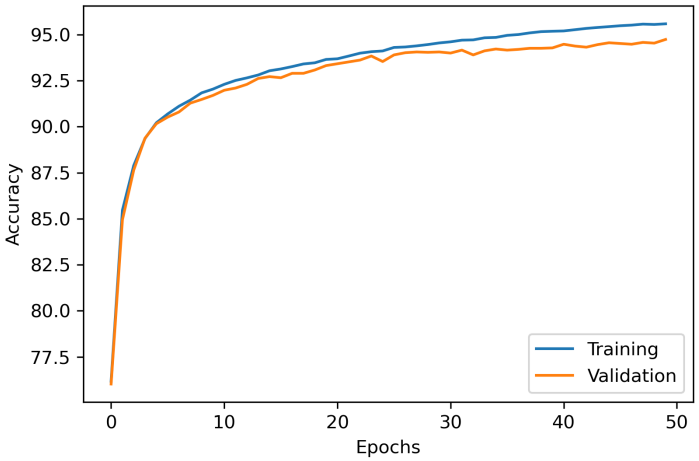


Figure 11.8: Classification accuracy by the number of training epochs

The plot reveals that the gap between training and validation accuracy increases as we train for more epochs. At approximately the 25th epoch, the training and validation accuracy values are almost equal, and then, the network starts to slightly overfit the training data.

Reducing overfitting

One way to decrease the effect of overfitting is to increase the regularization strength via L2 regularization, which we introduced in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-Learn*. Another useful technique for tackling overfitting in NNs is dropout, which will be covered in *Chapter 14, Classifying Images with Deep Convolutional Neural Networks*.

Finally, let's evaluate the generalization performance of the model by calculating the prediction accuracy on the test dataset:

```
>>> test_mse, test_acc = compute_mse_and_acc(model, X_test, y_test)
>>> print(f'Test accuracy: {test_acc*100:.2f}%')
Test accuracy: 94.51%
```

We can see that the test accuracy is very close to the validation set accuracy corresponding to the last epoch (94.74%), which we reported during the training in the last subsection. Moreover, the respective training accuracy is only minimally higher at 95.59%, reaffirming that our model only slightly overfits the training data.

To further fine-tune the model, we could change the number of hidden units, the learning rate, or use various other tricks that have been developed over the years but are beyond the scope of this book. In *Chapter 14, Classifying Images with Deep Convolutional Neural Networks*, you will learn about a different NN architecture that is known for its good performance on image datasets.

Also, the chapter will introduce additional performance-enhancing tricks such as adaptive learning rates, more sophisticated SGD-based optimization algorithms, batch normalization, and dropout.

Other common tricks that are beyond the scope of the following chapters include:

- Adding skip-connections, which are the main contribution of residual NNs (*Deep residual learning for image recognition* by *K. He*, *X. Zhang*, *S. Ren*, and *J. Sun*, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770-778, 2016)
- Using learning rate schedulers that change the learning rate during training (*Cyclical learning rates for training neural networks* by *L.N. Smith*, *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 464-472, 2017)
- Attaching loss functions to earlier layers in the networks as it's being done in the popular Inception v3 architecture (*Rethinking the Inception architecture for computer vision by C. Szegedy*, *V. Vanhoucke*, *S. Ioffe*, *J. Shlens*, and *Z. Wojna*, *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2818-2826, 2016)

Lastly, let's take a look at some of the images that our MLP struggles with by extracting and plotting the first 25 misclassified samples from the test set:

```
>>> X_test_subset = X_test[:1000, :]
>>> y_test_subset = y_test[:1000]
>>> _, probas = model.forward(X_test_subset)
>>> test_pred = np.argmax(probas, axis=1)
>>> misclassified_images = \
...      X_test_subset[y_test_subset != test_pred][:25]
>>> misclassified_labels = test_pred[y_test_subset != test_pred][:25]
>>> correct_labels = y_test_subset[y_test_subset != test_pred][:25]

>>> fig, ax = plt.subplots(nrows=5, ncols=5,
...                        sharex=True, sharey=True,
...                        figsize=(8, 8))
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = misclassified_images[i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys', interpolation='nearest')
...     ax[i].set_title(f'{i+1}) '
...                     f'True: {correct_labels[i]}\n'
...                     f' Predicted: {misclassified_labels[i]}')

>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

We should now see a 5×5 subplot matrix where the first number in the subtitles indicates the plot index, the second number represents the true class label (`True`), and the third number stands for the predicted class label (`Predicted`):

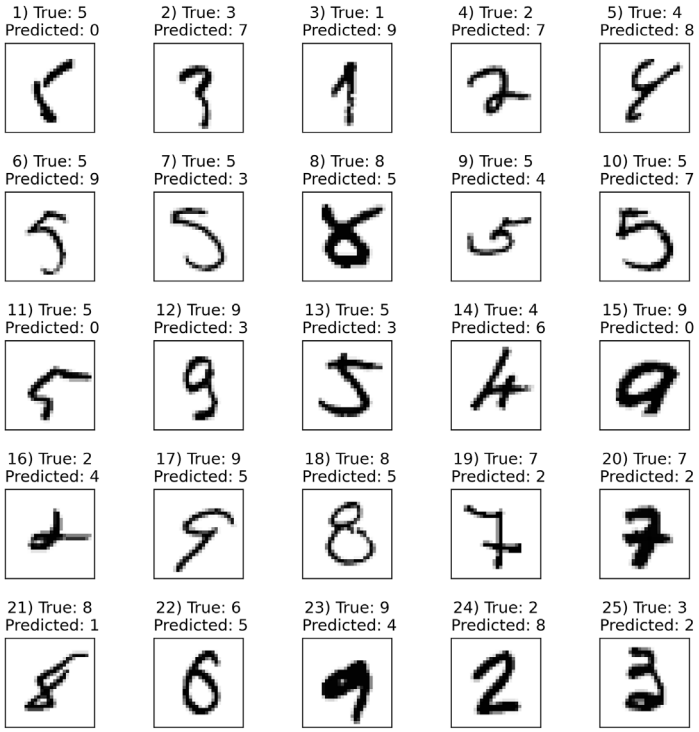


Figure 11.9: Handwritten digits that the model fails to classify correctly

As we can see in *Figure 11.9*, among others, the network finds 7s challenging when they include a horizontal line as in examples 19 and 20. Looking back at an earlier figure in this chapter where we plotted different training examples of the number 7, we can hypothesize that the handwritten digit 7 with a horizontal line is underrepresented in our dataset and is often misclassified.

Training an artificial neural network

Now that we have seen an NN in action and have gained a basic understanding of how it works by looking over the code, let's dig a little bit deeper into some of the concepts, such as the loss computation and the backpropagation algorithm that we implemented to learn the model parameters.

Computing the loss function

As mentioned previously, we used an MSE loss (as in Adaline) to train the multilayer NN as it makes the derivation of the gradients a bit easier to follow. In later chapters, we will discuss other loss functions, such as the multi-category cross-entropy loss (a generalization of the binary logistic regression loss), which is a more common choice for training NN classifiers.

In the previous section, we implemented an MLP for multiclass classification that returns an output vector of t elements that we need to compare to the $t \times 1$ dimensional target vector in the one-hot encoding representation. If we predict the class label of an input image with class label 2, using this MLP, the activation of the third layer and the target may look like this:

$$a^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

Thus, our MSE loss either has to sum or average over the t activation units in our network in addition to averaging over the n examples in the dataset or mini-batch:

$$L(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{t} \sum_{j=1}^{t} \left(y_j^{[i]} - a_j^{(out)[i]} \right)^2$$

Here, again, the superscript $[i]$ is the index of a particular example in our training dataset.

Remember that our goal is to minimize the loss function $L(\boldsymbol{W})$; thus, we need to calculate the partial derivative of the parameters $\boldsymbol{W}$ with respect to each weight for every layer in the network:

$$\frac{\partial}{\partial w_{j,l}^{(l)}} = L(\boldsymbol{W}, \boldsymbol{b})$$

In the next section, we will talk about the backpropagation algorithm, which allows us to calculate those partial derivatives to minimize the loss function.

Note that $\boldsymbol{W}$ consists of multiple matrices. In an MLP with one hidden layer, we have the weight matrix, $\boldsymbol{W}^{(h)}$, which connects the input to the hidden layer, and $\boldsymbol{W}^{(out)}$, which connects the hidden layer to the output layer. A visualization of the three-dimensional tensor $\boldsymbol{W}$ is provided in *Figure 11.10*:

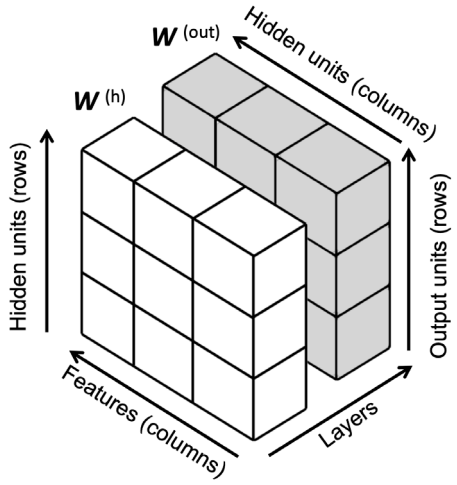


Figure 11.10: A visualization of a three-dimensional tensor

In this simplified figure, it may seem that both $\boldsymbol{W}^{(h)}$ and $\boldsymbol{W}^{(out)}$ have the same number of rows and columns, which is typically not the case unless we initialize an MLP with the same number of hidden units, output units, and input features.

If this sounds confusing, stay tuned for the next section, where we will discuss the dimensionality of $\boldsymbol{W}^{(h)}$ and $\boldsymbol{W}^{(out)}$ in more detail in the context of the backpropagation algorithm. Also, you are encouraged to read through the code of `NeuralNetMLP` again, which is annotated with helpful comments about the dimensionality of the different matrices and vector transformations.

Developing your understanding of backpropagation

Although backpropagation was introduced to the neural network community more than 30 years ago (*Learning representations by backpropagating errors*, by *D.E. Rumelhart*, *G.E. Hinton*, and *R.J. Williams*, *Nature*, 323: 6088, pages 533–536, 1986), it remains one of the most widely used algorithms for training artificial NNs very efficiently. If you are interested in additional references regarding the history of backpropagation, Juergen Schmidhuber wrote a nice survey article, *Who Invented Backpropagation?*, which you can find online at `http://people.idsia.ch/~juergen/who-invented-backpropagation.html`.

This section will provide both a short, clear summary and the bigger picture of how this fascinating algorithm works before we dive into more mathematical details. In essence, we can think of backpropagation as a very computationally efficient approach to compute the partial derivatives of a complex, non-convex loss function in multilayer NNs. Here, our goal is to use those derivatives to learn the weight coefficients for parameterizing such a multilayer artificial NN. The challenge in the parameterization of NNs is that we are typically dealing with a very large number of model parameters in a high-dimensional feature space. In contrast to loss functions of single-layer NNs such as Adaline or logistic regression, which we have seen in previous chapters, the error surface of an NN loss function is not convex or smooth with respect to the parameters. There are many bumps in this high-dimensional loss surface (local minima) that we have to overcome in order to find the global minimum of the loss function.

You may recall the concept of the chain rule from your introductory calculus classes. The chain rule is an approach to compute the derivative of a complex, nested function, such as $f(g(x))$, as follows:

$$\frac{d}{dx}[f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}$$

Similarly, we can use the chain rule for an arbitrarily long function composition. For example, let's assume that we have five different functions, $f(x)$, $g(x)$, $h(x)$, $u(x)$, and $v(x)$, and let F be the function composition: $F(x) = f(g(h(u(v(x)))))$. Applying the chain rule, we can compute the derivative of this function as follows:

$$\frac{dF}{dx} = \frac{d}{dx}F(x) = \frac{d}{dx}f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}$$

In the context of computer algebra, a set of techniques, known as **automatic differentiation**, has been developed to solve such problems very efficiently. If you are interested in learning more about automatic differentiation in machine learning applications, read A.G. Baydin and B.A. Pearlmutter's article, *Automatic Differentiation of Algorithms for Machine Learning*, arXiv preprint arXiv:1404.7456, 2014, which is freely available on arXiv at `http://arxiv.org/pdf/1404.7456.pdf`.

Automatic differentiation comes with two modes, the forward and reverse modes; backpropagation is simply a special case of reverse-mode automatic differentiation. The key point is that applying the chain rule in forward mode could be quite expensive since we would have to multiply large matrices for each layer (Jacobians) that we would eventually multiply by a vector to obtain the output.

The trick of reverse mode is that we traverse the chain rule from right to left. We multiply a matrix by a vector, which yields another vector that is multiplied by the next matrix, and so on. Matrix-vector multiplication is computationally much cheaper than matrix-matrix multiplication, which is why backpropagation is one of the most popular algorithms used in NN training.

A basic calculus refresher

To fully understand backpropagation, we need to borrow certain concepts from differential calculus, which is outside the scope of this book. However, you can refer to a review chapter of the most fundamental concepts, which you might find useful in this context. It discusses function derivatives, partial derivatives, gradients, and the Jacobian. This text is freely accessible at `https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus.pdf`. If you are unfamiliar with calculus or need a brief refresher, consider reading this text as an additional supporting resource before reading the next section.

Training neural networks via backpropagation

In this section, we will go through the math of backpropagation to understand how you can learn the weights in an NN very efficiently. Depending on how comfortable you are with mathematical representations, the following equations may seem relatively complicated at first.

In a previous section, we saw how to calculate the loss as the difference between the activation of the last layer and the target class label. Now, we will see how the backpropagation algorithm works to update the weights in our MLP model from a mathematical perspective, which we implemented in the `.backward()` method of the `NeuralNetMLP()` class. As we recall from the beginning of this chapter, we first need to apply forward propagation to obtain the activation of the output layer, which we formulated as follows:

$$\boldsymbol{Z}^{(h)} = \boldsymbol{X}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)} \quad \text{(net input of the hidden layer)}$$

$$\boldsymbol{A}^{(h)} = \sigma\left(\boldsymbol{Z}^{(h)}\right) \quad \text{(activation of the hidden layer)}$$

$$\boldsymbol{Z}^{(out)} = \boldsymbol{A}^{(h)}\boldsymbol{W}^{(out)T} + \boldsymbol{b}^{(out)} \quad \text{(net input of the output layer)}$$

$$\boldsymbol{A}^{(out)} = \sigma\left(\boldsymbol{Z}^{(out)}\right) \quad \text{(activation of the output layer)}$$

Concisely, we just forward-propagate the input features through the connections in the network, as shown by the arrows in *Figure 11.11* for a network with two input features, three hidden nodes, and two output nodes:

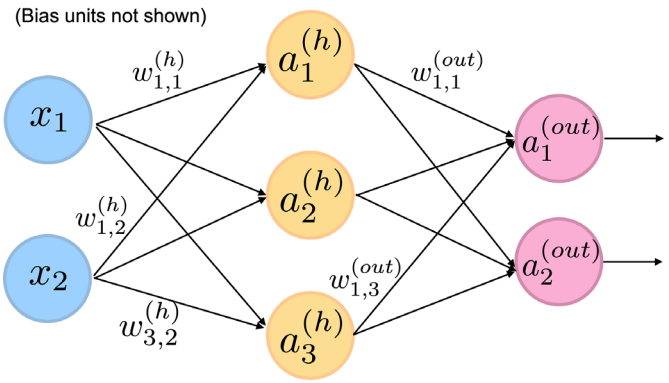
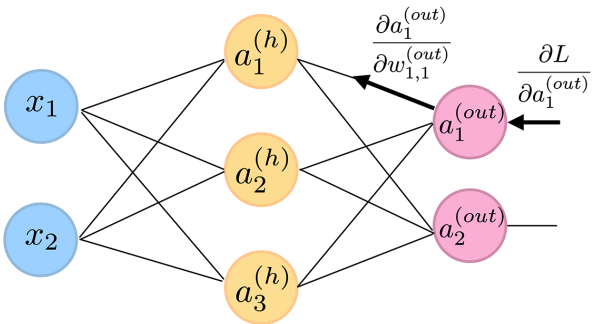


Figure 11.11: Forward-propagating the input features of an NN

In backpropagation, we propagate the error from right to left. We can think of this as an application of the chain rule to the computation of the forward pass to compute the gradient of the loss with respect to the model weights (and bias units). For simplicity, we will illustrate this process for the partial derivative used to update the first weight in the weight matrix of the output layer. The paths of the computation we backpropagate are highlighted via the bold arrows below:



Gradient for output layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

Figure 11.12: Backpropagating the error of an NN

If we include the net inputs z explicitly, the partial derivative computation shown in the previous figure expands as follows:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

To compute this partial derivative, which is used to update $w_{1,1}^{(out)}$, we can compute the three individual partial derivative terms and multiply the results. For simplicity, we will omit averaging over the individual examples in the mini-batch, so we drop the $\frac{1}{n}\sum_{i=1}^{n}$ averaging term from the following equations.

Let's start with $\frac{\partial L}{\partial a_1^{(out)}}$, which is the partial derivative of the MSE loss (which simplifies to the squared error if we omit the mini-batch dimension) with respect to the predicted output score of the first output node:

$$\frac{\partial L}{\partial a_1^{(out)}} = \frac{\partial}{\partial a_1^{(out)}} \left(y_1 - a_1^{(out)}\right)^2 = 2\left(a_1^{(out)} - y\right)$$

The next term is the derivative of the logistic sigmoid activation function that we used in the output layer:

$$\frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} = \frac{\partial}{\partial z_1^{(out)}} \frac{1}{1 + e^{z_1^{(out)}}} = \quad \ldots \quad = \left(\frac{1}{1 + e^{z_1^{(out)}}}\right)\left(1 - \frac{1}{1 + e^{z_1^{(out)}}}\right)$$

$$= a_1^{(out)}\left(1 - a_1^{(out)}\right)$$

Lastly, we compute the derivative of the net input with respect to the weight:

$$\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = \frac{\partial}{\partial w_{1,1}^{(out)}} a_1^{(h)} w_{1,1}^{(out)} + b_1^{(out)} = a_1^{(h)}$$

Putting all of it together, we get the following:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = 2\left(a_1^{(out)} - y\right) \cdot a_1^{(out)}\left(1 - a_1^{(out)}\right) \cdot a_1^{(h)}$$

We then use this value to update the weight via the familiar stochastic gradient descent update with a learning rate of η:

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}$$

In our code implementation of `NeuralNetMLP()`, we implemented the computation $\frac{\partial L}{\partial w_{1,1}^{(out)}}$ in vectorized form in the `.backward()` method as follows:

```
        # Part 1: dLoss/dOutWeights
        ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
        ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet for convenient re-use

        # input/output dim: [n_examples, n_classes]
        d_loss__d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

        # input/output dim: [n_examples, n_classes]
        d_a_out__d_z_out = a_out * (1. - a_out) # sigmoid derivative

        # output dim: [n_examples, n_classes]
        delta_out = d_loss__d_a_out * d_a_out__d_z_out # "delta (rule)
                                                       # placeholder"

        # gradient for output weights

        # [n_examples, n_hidden]
```

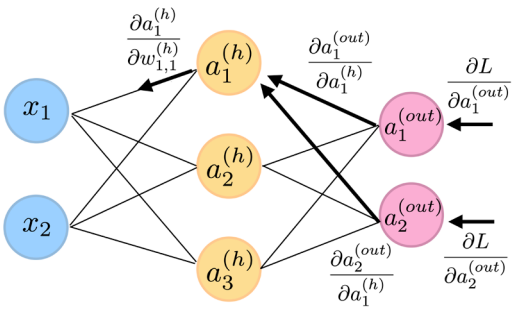
```
        d_z_out__dw_out = a_h

        # input dim: [n_classes, n_examples] dot [n_examples, n_hidden]
        # output dim: [n_classes, n_hidden]
        d_loss__dw_out = np.dot(delta_out.T, d_z_out__dw_out)
        d_loss__db_out = np.sum(delta_out, axis=0)
```

As annotated in the code snippet above, we created the following "delta" placeholder variable:

$$\delta_1^{(out)} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}$$

This is because $\delta^{(out)}$ terms are involved in computing the partial derivatives (or gradients) of the hidden layer weights as well; hence, we can reuse $\delta^{(out)}$.

Speaking of hidden layer weights, *Figure 11.13* illustrates how to compute the partial derivative of the loss with respect to the first weight of the hidden layer:



Gradient for hidden layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}} + \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

Figure 11.13: Computing the partial derivatives of the loss with respect to the first hidden layer weight

It is important to highlight that since the weight $w_{1,1}^{(h)}$ is connected to both output nodes, we have to use the *multi-variable* chain rule to sum the two paths highlighted with bold arrows. As before, we can expand it to include the net inputs z and then solve the individual terms:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

Notice that if we reuse $\delta^{(out)}$ computed previously, this equation can be simplified as follows:

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \delta_1^{(out)} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \delta_2^{(out)} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

The preceding terms can be individually solved relatively easily, as we have done previously, because there are no new derivatives involved. For example, $\frac{\partial a_1^{(h)}}{\partial z_1^{(h)}}$ is the derivative of the sigmoid activation, that is, $a_1^{(h)}\left(1 - a_1^{(h)}\right)$, and so forth. We'll leave solving the individual parts as an optional exercise for you.

About convergence in neural networks

You might be wondering why we did not use regular gradient descent but instead used mini-batch learning to train our NN for the handwritten digit classification earlier. You may recall our discussion on SGD that we used to implement online learning. In online learning, we compute the gradient based on a single training example ($k = 1$) at a time to perform the weight update. Although this is a stochastic approach, it often leads to very accurate solutions with a much faster convergence than regular gradient descent. Mini-batch learning is a special form of SGD where we compute the gradient based on a subset k of the n training examples with $1 < k < n$. Mini-batch learning has an advantage over online learning in that we can make use of our vectorized implementations to improve computational efficiency. However, we can update the weights much faster than in regular gradient descent. Intuitively, you can think of mini-batch learning as predicting the voter turnout of a presidential election from a poll by asking only a representative subset of the population rather than asking the entire population (which would be equal to running the actual election).

Multilayer NNs are much harder to train than simpler algorithms such as Adaline, logistic regression, or support vector machines. In multilayer NNs, we typically have hundreds, thousands, or even billions of weights that we need to optimize. Unfortunately, the output function has a rough surface, and the optimization algorithm can easily become trapped in local minima, as shown in *Figure 11.14*:

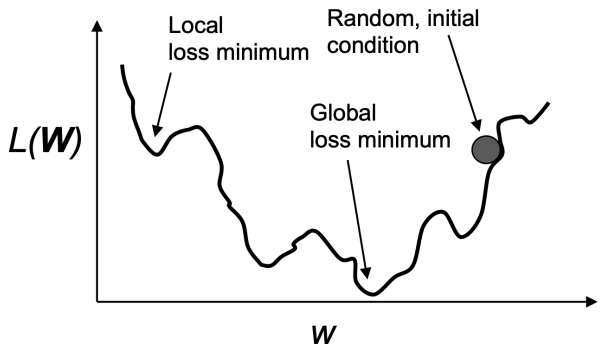


Figure 11.14: Optimization algorithms can become trapped in local minima

Note that this representation is extremely simplified since our NN has many dimensions; it makes it impossible to visualize the actual loss surface for the human eye. Here, we only show the loss surface for a single weight on the x axis. However, the main message is that we do not want our algorithm to get trapped in local minima. By increasing the learning rate, we can more readily escape such local minima. On the other hand, we also increase the chance of overshooting the global optimum if the learning rate is too large. Since we initialize the weights randomly, we start with a solution to the optimization problem that is typically hopelessly wrong.

A few last words about the neural network implementation

You may be wondering why we went through all of this theory just to implement a simple multilayer artificial network that can classify handwritten digits instead of using an open source Python machine learning library. In fact, we will introduce more complex NN models in the next chapters, which we will train using the open source PyTorch library (`https://pytorch.org`).

Although the from-scratch implementation in this chapter seems a bit tedious at first, it was a good exercise for understanding the basics behind backpropagation and NN training. A basic understanding of algorithms is crucial for applying machine learning techniques appropriately and successfully.

Now that you have learned how feedforward NNs work, we are ready to explore more sophisticated DNNs using PyTorch, which allows us to construct NNs more efficiently, as we will see in *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*.

PyTorch, which was originally released in September 2016, has gained a lot of popularity among machine learning researchers, who use it to construct DNNs because of its ability to optimize mathematical expressions for computations on multidimensional arrays utilizing **graphics processing units** (**GPUs**).

Lastly, we should note that scikit-learn also includes a basic MLP implementation, `MLPClassifier`, which you can find at `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html`. While this implementation is great and very convenient for training basic MLPs, we strongly recommend specialized deep learning libraries, such as PyTorch, for implementing and training multilayer NNs.

Summary

In this chapter, you have learned the basic concepts behind multilayer artificial NNs, which are currently the hottest topic in machine learning research. In *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, we started our journey with simple single-layer NN structures and now we have connected multiple neurons to a powerful NN architecture to solve complex problems such as handwritten digit recognition. We demystified the popular backpropagation algorithm, which is one of the building blocks of many NN models that are used in deep learning. After learning about the backpropagation algorithm in this chapter, we are well equipped for exploring more complex DNN architectures. In the remaining chapters, we will cover more advanced deep learning concepts and PyTorch, an open source library that allows us to implement and train multilayer NNs more efficiently.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`

12

Parallelizing Neural Network Training with PyTorch

In this chapter, we will move on from the mathematical foundations of machine learning and deep learning to focus on PyTorch. PyTorch is one of the most popular deep learning libraries currently available, and it lets us implement **neural networks** (**NNs**) much more efficiently than any of our previous NumPy implementations. In this chapter, we will start using PyTorch and see how it brings significant benefits to training performance.

This chapter will begin the next stage of our journey into machine learning and deep learning, and we will explore the following topics:

- How PyTorch improves training performance
- Working with PyTorch's `Dataset` and `DataLoader` to build input pipelines and enable efficient model training
- Working with PyTorch to write optimized machine learning code
- Using the `torch.nn` module to implement common deep learning architectures conveniently
- Choosing activation functions for artificial NNs

PyTorch and training performance

PyTorch can speed up our machine learning tasks significantly. To understand how it can do this, let's begin by discussing some of the performance challenges we typically run into when we execute expensive calculations on our hardware. Then, we will take a high-level look at what PyTorch is and what our learning approach will be in this chapter.

Performance challenges

The performance of computer processors has, of course, been continuously improving in recent years. That allows us to train more powerful and complex learning systems, which means that we can improve the predictive performance of our machine learning models. Even the cheapest desktop computer hardware that's available right now comes with processing units that have multiple cores.

In the previous chapters, we saw that many functions in scikit-learn allow us to spread those computations over multiple processing units. However, by default, Python is limited to execution on one core due to the **global interpreter lock** (**GIL**). So, although we indeed take advantage of Python's multiprocessing library to distribute our computations over multiple cores, we still have to consider that the most advanced desktop hardware rarely comes with more than 8 or 16 such cores.

You will recall from *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*, that we implemented a very simple **multilayer perceptron** (**MLP**) with only one hidden layer consisting of 100 units. We had to optimize approximately 80,000 weight parameters ([784*100 + 100] + [100 * 10] + 10 = 79,510) for a very simple image classification task. The images in MNIST are rather small (28×28), and we can only imagine the explosion in the number of parameters if we wanted to add additional hidden layers or work with images that have higher pixel densities. Such a task would quickly become unfeasible for a single processing unit. The question then becomes, how can we tackle such problems more effectively?

The obvious solution to this problem is to use **graphics processing units** (**GPUs**), which are real workhorses. You can think of a graphics card as a small computer cluster inside your machine. Another advantage is that modern GPUs are great value compared to the state-of-the-art **central processing units** (**CPUs**), as you can see in the following overview:

Specifications	Intel® Core™ i9-11900KB Processor	NVIDIA GeForce® RTX™ 3080 Ti
Base Clock Frequency	3.3 GHz	1.37 GHz
Cores	16 (32 threads)	10240
Memory Bandwidth	45.8 GB/s	912.1 GB/s
Floating-Point Calculations	742 GFLOPS	34.10 TFLOPS
Cost	~ $540.00	~ $1200.00

Figure 12.1: Comparison of a state-of-the-art CPU and GPU

The sources for the information in *Figure 12.1* are the following websites (date accessed: July 2021):

- `https://ark.intel.com/content/www/us/en/ark/products/215570/intel-core-i9-11900kb-processor-24m-cache-up-to-4-90-ghz.html`
- `https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/`

At 2.2 times the price of a modern CPU, we can get a GPU that has 640 times more cores and is capable of around 46 times more floating-point calculations per second. So, what is holding us back from utilizing GPUs for our machine learning tasks? The challenge is that writing code to target GPUs is not as simple as executing Python code in our interpreter. There are special packages, such as CUDA and OpenCL, that allow us to target the GPU. However, writing code in CUDA or OpenCL is probably not the most convenient way to implement and run machine learning algorithms. The good news is that this is what PyTorch was developed for!

What is PyTorch?

PyTorch is a scalable and multiplatform programming interface for implementing and running machine learning algorithms, including convenience wrappers for deep learning. PyTorch was primarily developed by the researchers and engineers from the **Facebook AI Research** (**FAIR**) lab. Its development also involves many contributions from the community. PyTorch was initially released in September 2016 and is free and open source under the modified BSD license. Many machine learning researchers and practitioners from academia and industry have adapted PyTorch to develop deep learning solutions, such as Tesla Autopilot, Uber's Pyro, and Hugging Face's Transformers (`https://pytorch.org/ecosystem/`).

To improve the performance of training machine learning models, PyTorch allows execution on CPUs, GPUs, and XLA devices such as TPUs. However, its greatest performance capabilities can be discovered when using GPUs and XLA devices. PyTorch supports CUDA-enabled and ROCm GPUs officially. PyTorch's development is based on the Torch library (`www.torch.ch`). As its name implies, the Python interface is the primary development focus of PyTorch.

PyTorch is built around a computation graph composed of a set of nodes. Each node represents an operation that may have zero or more inputs or outputs. PyTorch provides an imperative programming environment that evaluates operations, executes computation, and returns concrete values immediately. Hence, the computation graph in PyTorch is defined implicitly, rather than constructed in advance and executed after.

Mathematically, tensors can be understood as a generalization of scalars, vectors, matrices, and so on. More concretely, a scalar can be defined as a rank-0 tensor, a vector can be defined as a rank-1 tensor, a matrix can be defined as a rank-2 tensor, and matrices stacked in a third dimension can be defined as rank-3 tensors. Tensors in PyTorch are similar to NumPy's arrays, except that tensors are optimized for automatic differentiation and can run on GPUs.

To make the concept of a tensor clearer, consider *Figure 12.2*, which represents tensors of ranks 0 and 1 in the first row, and tensors of ranks 2 and 3 in the second row:

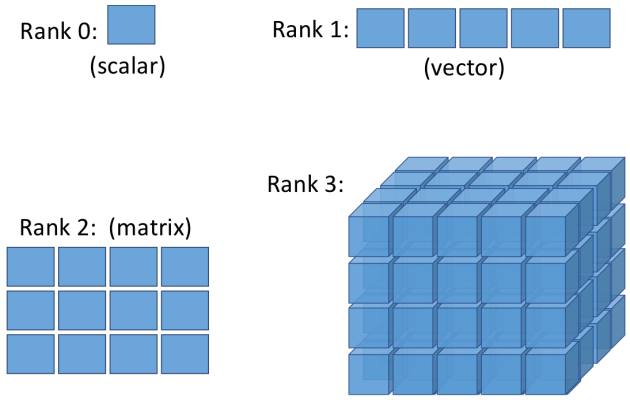


Figure 12.2: Different types of tensor in PyTorch

Now that we know what PyTorch is, let's see how to use it.

How we will learn PyTorch

First, we are going to cover PyTorch's programming model, in particular, creating and manipulating tensors. Then, we will see how to load data and utilize the `torch.utils.data` module, which will allow us to iterate through a dataset efficiently. In addition, we will discuss the existing, ready-to-use datasets in the `torch.utils.data.Dataset` submodule and learn how to use them.

After learning about these basics, the PyTorch neural network `torch.nn` module will be introduced. Then, we will move forward to building machine learning models, learn how to compose and train the models, and learn how to save the trained models on disk for future evaluation.

First steps with PyTorch

In this section, we will take our first steps in using the low-level PyTorch API. After installing PyTorch, we will cover how to create tensors in PyTorch and different ways of manipulating them, such as changing their shape, data type, and so on.

Installing PyTorch

To install PyTorch, we recommend consulting the latest instructions on the official `https://pytorch.org` website. Below, we will outline the basic steps that will work on most systems.

Depending on how your system is set up, you can typically just use Python's `pip` installer and install PyTorch from PyPI by executing the following from your terminal:

```
pip install torch torchvision
```

This will install the latest *stable* version, which is 1.9.0 at the time of writing. To install the 1.9.0 version, which is guaranteed to be compatible with the following code examples, you can modify the preceding command as follows:

```
pip install torch==1.9.0 torchvision==0.10.0
```

If you want to use GPUs (recommended), you need a compatible NVIDIA graphics card that supports CUDA and cuDNN. If your machine satisfies these requirements, you can install PyTorch with GPU support, as follows:

```
pip install torch==1.9.0+cu111 torchvision==0.10.0+cu111 -f https://download.
pytorch.org/whl/torch_stable.html
```

for CUDA 11.1 or:

```
pip install torch==1.9.0 torchvision==0.10.0\  -f https://download.pytorch.org/
whl/torch_stable.html
```

for CUDA 10.2 as of the time of writing.

As macOS binaries don't support CUDA, you can install from source: `https://pytorch.org/get-started/locally/#mac-from-source`.

For more information about the installation and setup process, please see the official recommendations at `https://pytorch.org/get-started/locally/`.

Note that PyTorch is under active development; therefore, every couple of months, new versions are released with significant changes. You can verify your PyTorch version from your terminal, as follows:

```
python -c 'import torch; print(torch.__version__)'
```

Troubleshooting your installation of PyTorch

If you experience problems with the installation procedure, read more about system- and platform-specific recommendations that are provided at `https://pytorch.org/get-started/locally/`. Note that all the code in this chapter can be run on your CPU; using a GPU is entirely optional but recommended if you want to fully enjoy the benefits of PyTorch. For example, while training some NN models on a CPU could take a week, the same models could be trained in just a few hours on a modern GPU. If you have a graphics card, refer to the installation page to set it up appropriately. In addition, you may find this setup guide helpful, which explains how to install the NVIDIA graphics card drivers, CUDA, and cuDNN on Ubuntu (not required but recommended requirements for running PyTorch on a GPU): `https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf`. Furthermore, as you will see in *Chapter 17*, *Generative Adversarial Networks for Synthesizing New Data*, you can also train your models using a GPU for free via Google Colab.

Creating tensors in PyTorch

Now, let's consider a few different ways of creating tensors, and then see some of their properties and how to manipulate them. Firstly, we can simply create a tensor from a list or a NumPy array using the `torch.tensor` or the `torch.from_numpy` function as follows:

```
>>> import torch
>>> import numpy as np
>>> np.set_printoptions(precision=3)
>>> a = [1, 2, 3]
>>> b = np.array([4, 5, 6], dtype=np.int32)
>>> t_a = torch.tensor(a)
>>> t_b = torch.from_numpy(b)
>>> print(t_a)
>>> print(t_b)
tensor([1, 2, 3])
tensor([4, 5, 6], dtype=torch.int32)
```

This resulted in tensors `t_a` and `t_b`, with their properties, `shape=(3,)` and `dtype=int32`, adopted from their source. Similar to NumPy arrays, we can also see these properties:

```
>>> t_ones = torch.ones(2, 3)
>>> t_ones.shape
torch.Size([2, 3])
>>> print(t_ones)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

Finally, creating a tensor of random values can be done as follows:

```
>>> rand_tensor = torch.rand(2,3)
>>> print(rand_tensor)
tensor([[0.1409, 0.2848, 0.8914],
        [0.9223, 0.2924, 0.7889]])
```

Manipulating the data type and shape of a tensor

Learning ways to manipulate tensors is necessary to make them compatible for input to a model or an operation. In this section, you will learn how to manipulate tensor data types and shapes via several PyTorch functions that cast, reshape, transpose, and squeeze (remove dimensions).

The `torch.to()` function can be used to change the data type of a tensor to a desired type:

```
>>> t_a_new = t_a.to(torch.int64)
>>> print(t_a_new.dtype)
torch.int64
```

See `https://pytorch.org/docs/stable/tensor_attributes.html` for all other data types.

As you will see in upcoming chapters, certain operations require that the input tensors have a certain number of dimensions (that is, rank) associated with a certain number of elements (shape). Thus, we might need to change the shape of a tensor, add a new dimension, or squeeze an unnecessary dimension. PyTorch provides useful functions (or operations) to achieve this, such as `torch.transpose()`, `torch.reshape()`, and `torch.squeeze()`. Let's take a look at some examples:

- Transposing a tensor:

```
>>> t = torch.rand(3, 5)
>>> t_tr = torch.transpose(t, 0, 1)
>>> print(t.shape, ' --> ', t_tr.shape)
torch.Size([3, 5])  -->  torch.Size([5, 3])
```

- Reshaping a tensor (for example, from a 1D vector to a 2D array):

```
>>> t = torch.zeros(30)
>>> t_reshape = t.reshape(5, 6)
>>> print(t_reshape.shape)
torch.Size([5, 6])
```

- Removing the unnecessary dimensions (dimensions that have size 1, which are not needed):

```
>>> t = torch.zeros(1, 2, 1, 4, 1)
>>> t_sqz = torch.squeeze(t, 2)
>>> print(t.shape, ' --> ', t_sqz.shape)
torch.Size([1, 2, 1, 4, 1])  -->  torch.Size([1, 2, 4, 1])
```

Applying mathematical operations to tensors

Applying mathematical operations, in particular linear algebra operations, is necessary for building most machine learning models. In this subsection, we will cover some widely used linear algebra operations, such as element-wise product, matrix multiplication, and computing the norm of a tensor.

First, let's instantiate two random tensors, one with uniform distribution in the range [–1, 1) and the other with a standard normal distribution:

```
>>> torch.manual_seed(1)
>>> t1 = 2 * torch.rand(5, 2) - 1
>>> t2 = torch.normal(mean=0, std=1, size=(5, 2))
```

Note that `torch.rand` returns a tensor filled with random numbers from a uniform distribution in the range of [0, 1).

Notice that `t1` and `t2` have the same shape. Now, to compute the element-wise product of `t1` and `t2`, we can use the following:

```
>>> t3 = torch.multiply(t1, t2)
>>> print(t3)
tensor([[ 0.4426, -0.3114],
        [ 0.0660, -0.5970],
        [ 1.1249,  0.0150],
        [ 0.1569,  0.7107],
        [-0.0451, -0.0352]])
```

To compute the mean, sum, and standard deviation along a certain axis (or axes), we can use `torch.mean()`, `torch.sum()`, and `torch.std()`. For example, the mean of each column in `t1` can be computed as follows:

```
>>> t4 = torch.mean(t1, axis=0)
>>> print(t4)
tensor([-0.1373,  0.2028])
```

The matrix-matrix product between `t1` and `t2` (that is, $t_1 \times t_2^T$, where the superscript T is for transpose) can be computed by using the `torch.matmul()` function as follows:

```
>>> t5 = torch.matmul(t1, torch.transpose(t2, 0, 1))
>>> print(t5)
tensor([[ 0.1312,  0.3860, -0.6267, -1.0096, -0.2943],
        [ 0.1647, -0.5310,  0.2434,  0.8035,  0.1980],
        [-0.3855, -0.4422,  1.1399,  1.5558,  0.4781],
        [ 0.1822, -0.5771,  0.2585,  0.8676,  0.2132],
        [ 0.0330,  0.1084, -0.1692, -0.2771, -0.0804]])
```

On the other hand, computing $t_1^T \times t_2$ is performed by transposing `t1`, resulting in an array of size 2×2:

```
>>> t6 = torch.matmul(torch.transpose(t1, 0, 1), t2)
>>> print(t6)
tensor([[ 1.7453,  0.3392],
        [-1.6038, -0.2180]])
```

Finally, the `torch.linalg.norm()` function is useful for computing the L^p norm of a tensor. For example, we can calculate the L^2 norm of `t1` as follows:

```
>>> norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
>>> print(norm_t1)
tensor([0.6785, 0.5078, 1.1162, 0.5488, 0.1853])
```

To verify that this code snippet computes the L^2 norm of `t1` correctly, you can compare the results with the following NumPy function: `np.sqrt(np.sum(np.square(t1.numpy()), axis=1))`.

Split, stack, and concatenate tensors

In this subsection, we will cover PyTorch operations for splitting a tensor into multiple tensors, or the reverse: stacking and concatenating multiple tensors into a single one.

Assume that we have a single tensor, and we want to split it into two or more tensors. For this, PyTorch provides a convenient `torch.chunk()` function, which divides an input tensor into a list of equally sized tensors. We can determine the desired number of splits as an integer using the `chunks` argument to split a tensor along the desired dimension specified by the `dim` argument. In this case, the total size of the input tensor along the specified dimension must be divisible by the desired number of splits. Alternatively, we can provide the desired sizes in a list using the `torch.split()` function. Let's have a look at an example of both these options:

- Providing the number of splits:

  ```
  >>> torch.manual_seed(1)
  >>> t = torch.rand(6)
  >>> print(t)
  tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293, 0.7999])
  >>> t_splits = torch.chunk(t, 3)
  >>> [item.numpy() for item in t_splits]
  [array([0.758, 0.279], dtype=float32),
   array([0.403, 0.735], dtype=float32),
   array([0.029, 0.8  ], dtype=float32)]
  ```

  In this example, a tensor of size 6 was divided into a list of three tensors each with size 2. If the tensor size is not divisible by the `chunks` value, the last chunk will be smaller.

- Providing the sizes of different splits:

  Alternatively, instead of defining the number of splits, we can also specify the sizes of the output tensors directly. Here, we are splitting a tensor of size `5` into tensors of sizes `3` and `2`:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(5)
>>> print(t)
tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293])
>>> t_splits = torch.split(t, split_size_or_sections=[3, 2])
>>> [item.numpy() for item in t_splits]
[array([0.758, 0.279, 0.403], dtype=float32),
 array([0.735, 0.029], dtype=float32)]
```

Sometimes, we are working with multiple tensors and need to concatenate or stack them to create a single tensor. In this case, PyTorch functions such as `torch.stack()` and `torch.cat()` come in handy. For example, let's create a 1D tensor, `A`, containing 1s with size `3`, and a 1D tensor, `B`, containing 0s with size `2`, and concatenate them into a 1D tensor, `C`, of size `5`:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(2)
>>> C = torch.cat([A, B], axis=0)
>>> print(C)
tensor([1., 1., 1., 0., 0.])
```

If we create 1D tensors `A` and `B`, both with size `3`, then we can stack them together to form a 2D tensor, `S`:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(3)
>>> S = torch.stack([A, B], axis=1)
>>> print(S)
tensor([[1., 0.],
        [1., 0.],
        [1., 0.]])
```

The PyTorch API has many operations that you can use for building a model, processing your data, and more. However, covering every function is outside the scope of this book, where we will focus on the most essential ones. For the full list of operations and functions, you can refer to the documentation page of PyTorch at `https://pytorch.org/docs/stable/index.html`.

Building input pipelines in PyTorch

When we are training a deep NN model, we usually train the model incrementally using an iterative optimization algorithm such as stochastic gradient descent, as we have seen in previous chapters.

As mentioned at the beginning of this chapter, `torch.nn` is a module for building NN models. In cases where the training dataset is rather small and can be loaded as a tensor into the memory, we can directly use this tensor for training. In typical use cases, however, when the dataset is too large to fit into the computer memory, we will need to load the data from the main storage device (for example, the hard drive or solid-state drive) in chunks, that is, batch by batch. (Note the use of the term "batch" instead of "mini-batch" in this chapter to stay close to the PyTorch terminology.) In addition, we may need to construct a data-processing pipeline to apply certain transformations and preprocessing steps to our data, such as mean centering, scaling, or adding noise to augment the training procedure and to prevent overfitting.

Applying preprocessing functions manually every time can be quite cumbersome. Luckily, PyTorch provides a special class for constructing efficient and convenient preprocessing pipelines. In this section, we will see an overview of different methods for constructing a PyTorch `Dataset` and `DataLoader`, and implementing data loading, shuffling, and batching.

Creating a PyTorch DataLoader from existing tensors

If the data already exists in the form of a tensor object, a Python list, or a NumPy array, we can easily create a dataset loader using the `torch.utils.data.DataLoader()` class. It returns an object of the `DataLoader` class, which we can use to iterate through the individual elements in the input dataset. As a simple example, consider the following code, which creates a dataset from a list of values from 0 to 5:

```
>>> from torch.utils.data import DataLoader
>>> t = torch.arange(6, dtype=torch.float32)
>>> data_loader = DataLoader(t)
```

We can easily iterate through a dataset entry by entry as follows:

```
>>> for item in data_loader:
...     print(item)
tensor([0.])
tensor([1.])
tensor([2.])
tensor([3.])
tensor([4.])
tensor([5.])
```

If we want to create batches from this dataset, with a desired batch size of `3`, we can do this with the `batch_size` argument as follows:

```
>>> data_loader = DataLoader(t, batch_size=3, drop_last=False)
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', batch)
batch 1: tensor([0., 1., 2.])
batch 2: tensor([3., 4., 5.])
```

This will create two batches from this dataset, where the first three elements go into batch #1, and the remaining elements go into batch #2. The optional `drop_last` argument is useful for cases when the number of elements in the tensor is not divisible by the desired batch size. We can drop the last non-full batch by setting `drop_last` to `True`. The default value for `drop_last` is `False`.

We can always iterate through a dataset directly, but as you just saw, `DataLoader` provides an automatic and customizable batching to a dataset.

Combining two tensors into a joint dataset

Often, we may have the data in two (or possibly more) tensors. For example, we could have a tensor for features and a tensor for labels. In such cases, we need to build a dataset that combines these tensors, which will allow us to retrieve the elements of these tensors in tuples.

Assume that we have two tensors, `t_x` and `t_y`. Tensor `t_x` holds our feature values, each of size `3`, and `t_y` stores the class labels. For this example, we first create these two tensors as follows:

```
>>> torch.manual_seed(1)
>>> t_x = torch.rand([4, 3], dtype=torch.float32)
>>> t_y = torch.arange(4)
```

Now, we want to create a joint dataset from these two tensors. We first need to create a `Dataset` class as follows:

```
>>> from torch.utils.data import Dataset
>>> class JointDataset(Dataset):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
...     def __len__(self):
...         return len(self.x)
...
```

```
...     def __getitem__(self, idx):
...         return self.x[idx], self.y[idx]
```

A custom `Dataset` class must contain the following methods to be used by the data loader later on:

- `__init__()`: This is where the initial logic happens, such as reading existing arrays, loading a file, filtering data, and so forth.
- `__getitem__()`: This returns the corresponding sample to the given index.

Then we create a joint dataset of `t_x` and `t_y` with the custom `Dataset` class as follows:

```
>>> from torch.utils.data import TensorDataset
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Finally, we can print each example of the joint dataset as follows:

```
>>> for example in joint_dataset:
...     print('  x: ', example[0], '  y: ', example[1])
  x:  tensor([0.7576, 0.2793, 0.4031])   y:  tensor(0)
  x:  tensor([0.7347, 0.0293, 0.7999])   y:  tensor(1)
  x:  tensor([0.3971, 0.7544, 0.5695])   y:  tensor(2)
  x:  tensor([0.4388, 0.6387, 0.5247])   y:  tensor(3)
```

We can also simply utilize the `torch.utils.data.TensorDataset` class, if the second dataset is a labeled dataset in the form of tensors. So, instead of using our self-defined `Dataset` class, `JointDataset`, we can create a joint dataset as follows:

```
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Note that a common source of error could be that the element-wise correspondence between the original features (x) and labels (y) might be lost (for example, if the two datasets are shuffled separately). However, once they are merged into one dataset, it is safe to apply these operations.

If we have a dataset created from the list of image filenames on disk, we can define a function to load the images from these filenames. You will see an example of applying multiple transformations to a dataset later in this chapter.

Shuffle, batch, and repeat

As was mentioned in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, when training an NN model using stochastic gradient descent optimization, it is important to feed training data as randomly shuffled batches. You have already seen how to specify the batch size using the `batch_size` argument of a data loader object. Now, in addition to creating batches, you will see how to shuffle and reiterate over the datasets. We will continue working with the previous joint dataset.

First, let's create a shuffled version data loader from the `joint_dataset` dataset:

```
>>> torch.manual_seed(1)
>>> data_loader = DataLoader(dataset=joint_dataset, batch_size=2, shuffle=True)
```

Here, each batch contains two data records (x) and the corresponding labels (y). Now we iterate through the data loader entry by entry as follows:

```
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', 'x:', batch[0],
              '\n         y:', batch[1])
batch 1: x: tensor([[0.4388, 0.6387, 0.5247],
        [0.3971, 0.7544, 0.5695]])
         y: tensor([3, 2])
batch 2: x: tensor([[0.7576, 0.2793, 0.4031],
        [0.7347, 0.0293, 0.7999]])
         y: tensor([0, 1])
```

The rows are shuffled without losing the one-to-one correspondence between the entries in `x` and `y`.

In addition, when training a model for multiple epochs, we need to shuffle and iterate over the dataset by the desired number of epochs. So, let's iterate over the batched dataset twice:

```
>>> for epoch in range(2):
>>>     print(f'epoch {epoch+1}')
>>>     for i, batch in enumerate(data_loader, 1):
...         print(f'batch {i}:', 'x:', batch[0],
                  '\n         y:', batch[1])
epoch 1
batch 1: x: tensor([[0.7347, 0.0293, 0.7999],
        [0.3971, 0.7544, 0.5695]])
         y: tensor([1, 2])
batch 2: x: tensor([[0.4388, 0.6387, 0.5247],
        [0.7576, 0.2793, 0.4031]])
         y: tensor([3, 0])
epoch 2
batch 1: x: tensor([[0.3971, 0.7544, 0.5695],
        [0.7576, 0.2793, 0.4031]])
         y: tensor([2, 0])
batch 2: x: tensor([[0.7347, 0.0293, 0.7999],
        [0.4388, 0.6387, 0.5247]])
         y: tensor([1, 3])
```

This results in two different sets of batches. In the first epoch, the first batch contains a pair of values `[y=1, y=2]`, and the second batch contains a pair of values `[y=3, y=0]`. In the second epoch, two batches contain a pair of values, `[y=2, y=0]` and `[y=1, y=3]` respectively. For each iteration, the elements within a batch are also shuffled.

Creating a dataset from files on your local storage disk

In this section, we will build a dataset from image files stored on disk. There is an image folder associated with the online content of this chapter. After downloading the folder, you should be able to see six images of cats and dogs in JPEG format.

This small dataset will show how building a dataset from stored files generally works. To accomplish this, we are going to use two additional modules: `Image` in `PIL` to read the image file contents and `transforms` in `torchvision` to decode the raw contents and resize the images.

The `PIL.Image` and `torchvision.transforms` modules provide a lot of additional and useful functions, which are beyond the scope of the book. You are encouraged to browse through the official documentation to learn more about these functions:

`https://pillow.readthedocs.io/en/stable/reference/Image.html` for `PIL.Image`

`https://pytorch.org/vision/stable/transforms.html` for `torchvision.transforms`

Before we start, let's take a look at the content of these files. We will use the `pathlib` library to generate a list of image files:

```
>>> import pathlib
>>> imgdir_path = pathlib.Path('cat_dog_images')
>>> file_list = sorted([str(path) for path in
... imgdir_path.glob('*.jpg')])
>>> print(file_list)
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg', 'cat_dog_images/cat-
02.jpg', 'cat_dog_images/cat-03.jpg', 'cat_dog_images/dog-01.jpg', 'cat_dog_
images/dog-02.jpg']
```

Next, we will visualize these image examples using Matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> import os
>>> from PIL import Image
>>> fig = plt.figure(figsize=(10, 5))
>>> for i, file in enumerate(file_list):
...     img = Image.open(file)
...     print('Image shape:', np.array(img).shape)
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
```

```
...     ax.imshow(img)
...     ax.set_title(os.path.basename(file), size=15)
>>> plt.tight_layout()
>>> plt.show()
Image shape: (900, 1200, 3)
Image shape: (900, 1200, 3)
Image shape: (900, 1200, 3)
Image shape: (900, 742, 3)
Image shape: (800, 1200, 3)
Image shape: (800, 1200, 3)
```

Figure 12.3 shows the example images:

Figure 12.3: Images of cats and dogs

Just from this visualization and the printed image shapes, we can already see that the images have different aspect ratios. If you print the aspect ratios (or data array shapes) of these images, you will see that some images are 900 pixels high and 1200 pixels wide (900×1200), some are 800×1200, and one is 900×742. Later, we will preprocess these images to a consistent size. Another point to consider is that the labels for these images are provided within their filenames. So, we extract these labels from the list of filenames, assigning label `1` to dogs and label `0` to cats:

```
>>> labels = [1 if 'dog' in
...               os.path.basename(file) else 0
...                       for file in file_list]
>>> print(labels)
[0, 0, 0, 1, 1, 1]
```

Now, we have two lists: a list of filenames (or paths of each image) and a list of their labels. In the previous section, you learned how to create a joint dataset from two arrays. Here, we will do the following:

```
>>> class ImageDataset(Dataset):
...     def __init__(self, file_list, labels):
...         self.file_list = file_list
```

```
...         self.labels = labels
...
...     def __getitem__(self, index):
...         file = self.file_list[index]
...         label = self.labels[index]
...         return file, label
...
...     def __len__(self):
...         return len(self.labels)

>>> image_dataset = ImageDataset(file_list, labels)
>>> for file, label in image_dataset:
...     print(file, label)


cat_dog_images/cat-01.jpg 0
cat_dog_images/cat-02.jpg 0
cat_dog_images/cat-03.jpg 0
cat_dog_images/dog-01.jpg 1
cat_dog_images/dog-02.jpg 1
cat_dog_images/dog-03.jpg 1
```

The joint dataset has filenames and labels.

Next, we need to apply transformations to this dataset: load the image content from its file path, decode the raw content, and resize it to a desired size, for example, 80×120. As mentioned before, we use the `torchvision.transforms` module to resize the images and convert the loaded pixels into tensors as follows:

```
>>> import torchvision.transforms as transforms
>>> img_height, img_width = 80, 120
>>> transform = transforms.Compose([
...     transforms.ToTensor(),
...     transforms.Resize((img_height, img_width)),
... ])
```

Now we update the `ImageDataset` class with the `transform` we just defined:

```
>>> class ImageDataset(Dataset):
...     def __init__(self, file_list, labels, transform=None):
...         self.file_list = file_list
...         self.labels = labels
...         self.transform = transform
...
```

```
...     def __getitem__(self, index):
...         img = Image.open(self.file_list[index])
...         if self.transform is not None:
...             img = self.transform(img)
...         label = self.labels[index]
...         return img, label
...
...     def __len__(self):
...         return len(self.labels)
>>>
>>> image_dataset = ImageDataset(file_list, labels, transform)
```

Finally, we visualize these transformed image examples using Matplotlib:

```
>>> fig = plt.figure(figsize=(10, 6))
>>> for i, example in enumerate(image_dataset):
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(example[0].numpy().transpose((1, 2, 0)))
...     ax.set_title(f'{example[1]}', size=15)
...
>>> plt.tight_layout()
>>> plt.show()
```

This results in the following visualization of the retrieved example images, along with their labels:

Figure 12.4: Images are labeled

The `__getitem__` method in the `ImageDataset` class wraps all four steps into a single function, including the loading of the raw content (images and labels), decoding the images into tensors, and resizing the images. The function then returns a dataset that we can iterate over and apply other operations that we learned about in the previous sections via a data loader, such as shuffling and batching.

Fetching available datasets from the torchvision.datasets library

The `torchvision.datasets` library provides a nice collection of freely available image datasets for training or evaluating deep learning models. Similarly, the `torchtext.datasets` library provides datasets for natural language. Here, we use `torchvision.datasets` as an example.

The `torchvision` datasets (`https://pytorch.org/vision/stable/datasets.html`) are nicely formatted and come with informative descriptions, including the format of features and labels and their type and dimensionality, as well as the link to the original source of the dataset. Another advantage is that these datasets are all subclasses of `torch.utils.data.Dataset`, so all the functions we covered in the previous sections can be used directly. So, let's see how to use these datasets in action.

First, if you haven't already installed `torchvision` together with PyTorch earlier, you need to install the `torchvision` library via `pip` from the command line:

```
pip install torchvision
```

You can take a look at the list of available datasets at `https://pytorch.org/vision/stable/datasets.html`.

In the following paragraphs, we will cover fetching two different datasets: CelebA (`celeb_a`) and the MNIST digit dataset.

Let's first work with the CelebA dataset (`http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`) with `torchvision.datasets.CelebA` (`https://pytorch.org/vision/stable/datasets.html#celeba`). The description of `torchvision.datasets.CelebA` provides some useful information to help us understand the structure of this dataset:

- The database has three subsets, `'train'`, `'valid'`, and `'test'`. We can select a specific subset or load all of them with the `split` parameter.
- The images are stored in `PIL.Image` format. And we can obtain a transformed version using a custom `transform` function, such as `transforms.ToTensor` and `transforms.Resize`.
- There are different types of targets we can use, including `'attributes'`, `'identity'`, and `'landmarks'`. `'attributes'` is 40 facial attributes for the person in the image, such as facial expression, makeup, hair properties, and so on; `'identity'` is the person ID for an image; and `'landmarks'` refers to the dictionary of extracted facial points, such as the position of the eyes, nose, and so on.

Next, we will call the `torchvision.datasets.CelebA` class to download the data, store it on disk in a designated folder, and load it into a `torch.utils.data.Dataset` object:

```
>>> import torchvision
>>> image_path = './'
>>> celeba_dataset = torchvision.datasets.CelebA(
...     image_path, split='train', target_type='attr', download=True
... )
1443490838/? [01:28<00:00, 6730259.81it/s]
26721026/? [00:03<00:00, 8225581.57it/s]
3424458/? [00:00<00:00, 14141274.46it/s]
6082035/? [00:00<00:00, 21695906.49it/s]
12156055/? [00:00<00:00, 12002767.35it/s]
2836386/? [00:00<00:00, 3858079.93it/s]
```

You may run into a `BadZipFile: File is not a zip file` error, or `RuntimeError: The daily quota of the file img_align_celeba.zip is exceeded and it can't be downloaded. This is a limitation of Google Drive and can only be overcome by trying again later`; it just means that Google Drive has a daily maximum quota that is exceeded by the CelebA files. To work around it, you can manually download the files from the source: `http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`. In the downloaded folder, `celeba/`, you can unzip the `img_align_celeba.zip` file. The `image_path` is the root of the downloaded folder, `celeba/`. If you have already downloaded the files once, you can simply set `download=False`. For additional information and guidance, we highly recommend to see accompanying code notebook at `https://github.com/rasbt/machine-learning-book/blob/main/ch12/ch12_part1.ipynb`.

Now that we have instantiated the datasets, let's check if the object is of the `torch.utils.data.Dataset` class:

```
>>> assert isinstance(celeba_dataset, torch.utils.data.Dataset)
```

As mentioned, the dataset is already split into train, test, and validation datasets, and we only load the train set. And we only use the `'attributes'` target. In order to see what the data examples look like, we can execute the following code:

```
>>> example = next(iter(celeba_dataset))
>>> print(example)
(<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=178x218 at
0x120C6C668>, tensor([0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1,
1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1]))
```

Note that the sample in this dataset comes in a tuple of `(PIL.Image, attributes)`. If we want to pass this dataset to a supervised deep learning model during training, we have to reformat it as a tuple of `(features tensor, label)`. For the label, we will use the `'Smiling'` category from the attributes as an example, which is the 31st element.

Finally, let's take the first 18 examples from it to visualize them with their `'Smiling'` labels:

```
>>> from itertools import islice
>>> fig = plt.figure(figsize=(12, 8))
>>> for i, (image, attributes) in islice(enumerate(celeba_dataset), 18):
...     ax = fig.add_subplot(3, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image)
...     ax.set_title(f'{attributes[31]}', size=15)
>>> plt.show()
```

The examples and their labels that are retrieved from `celeba_dataset` are shown in *Figure 12.5*:

Figure 12.5: Model predicts smiling celebrities

This was all we needed to do to fetch and use the CelebA image dataset.

Next, we will proceed with the second dataset from `torchvision.datasets.MNIST` (`https://pytorch.org/vision/stable/datasets.html#mnist`). Let's see how it can be used to fetch the MNIST digit dataset:

- The database has two partitions, `'train'` and `'test'`. We need to select a specific subset to load.
- The images are stored in `PIL.Image` format. And we can obtain a transformed version using a custom `transform` function, such as `transforms.ToTensor` and `transforms.Resize`.
- There are 10 classes for the target, from `0` to `9`.

Now, we can download the `'train'` partition, convert the elements to tuples, and visualize 10 examples:

```
>>> mnist_dataset = torchvision.datasets.MNIST(image_path, 'train',
download=True)
>>> assert isinstance(mnist_dataset, torch.utils.data.Dataset)
>>> example = next(iter(mnist_dataset))
>>> print(example)
(<PIL.Image.Image image mode=L size=28x28 at 0x126895B00>, 5)
>>> fig = plt.figure(figsize=(15, 6))
>>> for i, (image, label) in  islice(enumerate(mnist_dataset), 10):
...     ax = fig.add_subplot(2, 5, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image, cmap='gray_r')
...     ax.set_title(f'{label}', size=15)
>>> plt.show()
```

The retrieved example handwritten digits from this dataset are shown as follows:

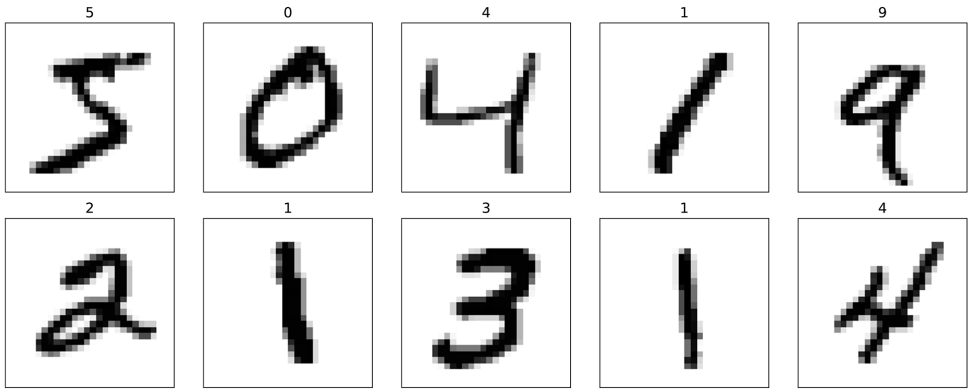


Figure 12.6: Correctly identifying handwritten digits

This concludes our coverage of building and manipulating datasets and fetching datasets from the `torchvision.datasets` library. Next, we will see how to build NN models in PyTorch.

Building an NN model in PyTorch

So far in this chapter, you have learned about the basic utility components of PyTorch for manipulating tensors and organizing data into formats that we can iterate over during training. In this section, we will finally implement our first predictive model in PyTorch. As PyTorch is a bit more flexible but also more complex than machine learning libraries such as scikit-learn, we will start with a simple linear regression model.

The PyTorch neural network module (torch.nn)

`torch.nn` is an elegantly designed module developed to help create and train NNs. It allows easy prototyping and the building of complex models in just a few lines of code.

To fully utilize the power of the module and customize it for your problem, you need to understand what it's doing. To develop this understanding, we will first train a basic linear regression model on a toy dataset without using any features from the `torch.nn` module; we will use nothing but the basic PyTorch tensor operations.

Then, we will incrementally add features from `torch.nn` and `torch.optim`. As you will see in the following subsections, these modules make building an NN model extremely easy. We will also take advantage of the dataset pipeline functionalities supported in PyTorch, such as `Dataset` and `DataLoader`, which you learned about in the previous section. In this book, we will use the `torch.nn` module to build NN models.

The most commonly used approach for building an NN in PyTorch is through `nn.Module`, which allows layers to be stacked to form a network. This gives us more control over the forward pass. We will see examples of building an NN model using the `nn.Module` class.

Finally, as you will see in the following subsections, a trained model can be saved and reloaded for future use.

Building a linear regression model

In this subsection, we will build a simple model to solve a linear regression problem. First, let's create a toy dataset in NumPy and visualize it:

```
>>> X_train = np.arange(10, dtype='float32').reshape((10, 1))
>>> y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0,
...                     6.3, 6.6,7.4, 8.0,
...                     9.0], dtype='float32')
>>> plt.plot(X_train, y_train, 'o', markersize=10)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

As a result, the training examples will be shown in a scatterplot as follows:

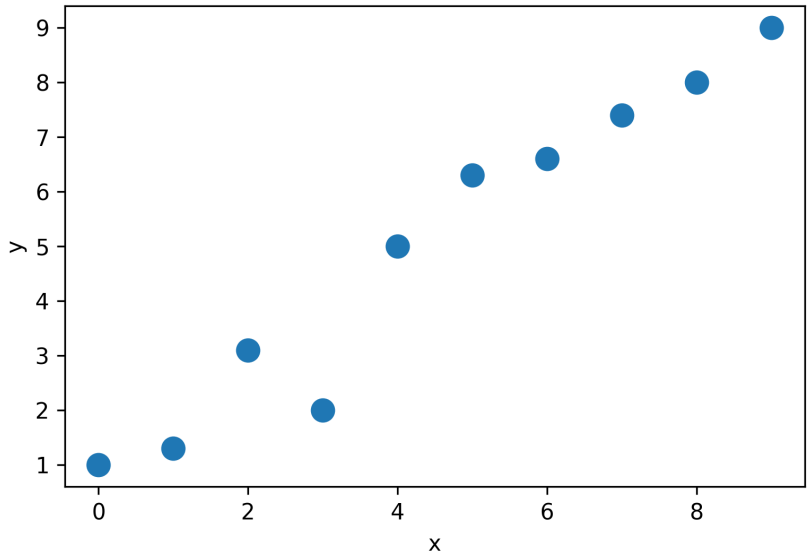


Figure 12.7: A scatterplot of the training examples

Next, we will standardize the features (mean centering and dividing by the standard deviation) and create a PyTorch `Dataset` for the training set and a corresponding `DataLoader`:

```
>>> from torch.utils.data import TensorDataset
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>>> X_train_norm = torch.from_numpy(X_train_norm)
>>> y_train = torch.from_numpy(y_train).float()
>>> train_ds = TensorDataset(X_train_norm, y_train)
>>> batch_size = 1
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Here, we set a batch size of `1` for the `DataLoader`.

Now, we can define our model for linear regression as $z = wx + b$. Here, we are going to use the `torch.nn` module. It provides predefined layers for building complex NN models, but to start, you will learn how to define a model from scratch. Later in this chapter, you will see how to use those predefined layers.

For this regression problem, we will define a linear regression model from scratch. We will define the parameters of our model, `weight` and `bias`, which correspond to the weight and the bias parameters, respectively. Finally, we will define the `model()` function to determine how this model uses the input data to generate its output:

```
>>> torch.manual_seed(1)
>>> weight = torch.randn(1)
>>> weight.requires_grad_()
>>> bias = torch.zeros(1, requires_grad=True)
>>> def model(xb):
...     return xb @ weight + bias
```

After defining the model, we can define the loss function that we want to minimize to find the optimal model weights. Here, we will choose the **mean squared error** (**MSE**) as our loss function:

```
>>> def loss_fn(input, target):
...     return (input-target).pow(2).mean()
```

Furthermore, to learn the weight parameters of the model, we will use stochastic gradient descent. In this subsection, we will implement this training via the stochastic gradient descent procedure by ourselves, but in the next subsection, we will use the `SGD` method from the optimization package, `torch.optim`, to do the same thing.

To implement the stochastic gradient descent algorithm, we need to compute the gradients. Rather than manually computing the gradients, we will use PyTorch's `torch.autograd.backward` function. We will cover `torch.autograd` and its different classes and functions for implementing automatic differentiation in *Chapter 13*, *Going Deeper – The Mechanics of PyTorch*.

Now, we can set the learning rate and train the model for 200 epochs. The code for training the model against the batched version of the dataset is as follows:

```
>>> learning_rate = 0.001
>>> num_epochs = 200
>>> log_epochs = 10
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch.long())
...         loss.backward()
...     with torch.no_grad():
...         weight -= weight.grad * learning_rate
...         bias -= bias.grad * learning_rate
...         weight.grad.zero_()
...         bias.grad.zero_()
...     if epoch % log_epochs==0:
```

```
...         print(f'Epoch {epoch}  Loss {loss.item():.4f}')
Epoch 0  Loss 5.1701
Epoch 10  Loss 30.3370
Epoch 20  Loss 26.9436
Epoch 30  Loss 0.9315
Epoch 40  Loss 3.5942
Epoch 50  Loss 5.8960
Epoch 60  Loss 3.7567
Epoch 70  Loss 1.5877
Epoch 80  Loss 0.6213
Epoch 90  Loss 1.5596
Epoch 100  Loss 0.2583
Epoch 110  Loss 0.6957
Epoch 120  Loss 0.2659
Epoch 130  Loss 0.1615
Epoch 140  Loss 0.6025
Epoch 150  Loss 0.0639
Epoch 160  Loss 0.1177
Epoch 170  Loss 0.3501
Epoch 180  Loss 0.3281
Epoch 190  Loss 0.0970
```

Let's look at the trained model and plot it. For the test data, we will create a NumPy array of values evenly spaced between 0 and 9. Since we trained our model with standardized features, we will also apply the same standardization to the test data:

```
>>> print('Final Parameters:', weight.item(), bias.item())
Final Parameters:  2.669806480407715 4.879569053649902
>>> X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> X_test_norm = torch.from_numpy(X_test_norm)
>>> y_pred = model(X_test_norm).detach().numpy()
>>> fig = plt.figure(figsize=(13, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(X_train_norm, y_train, 'o', markersize=10)
>>> plt.plot(X_test_norm, y_pred, '--', lw=3)
>>> plt.legend(['Training examples', 'Linear reg.'], fontsize=15)
>>> ax.set_xlabel('x', size=15)
>>> ax.set_ylabel('y', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

Figure 12.8 shows a scatterplot of the training examples and the trained linear regression model:

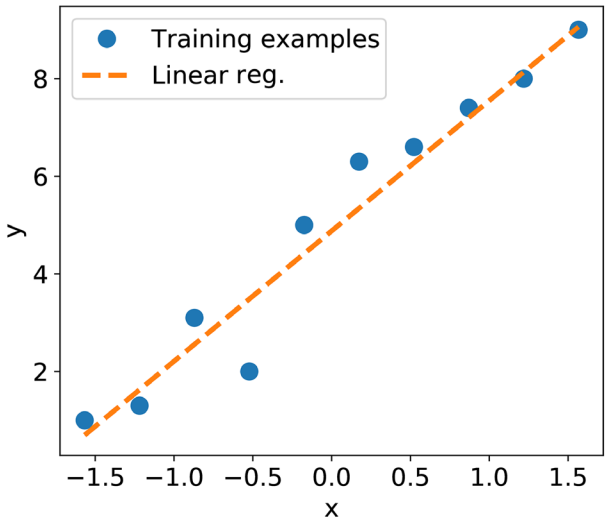


Figure 12.8: The linear regression model fits the data well

Model training via the torch.nn and torch.optim modules

In the previous example, we saw how to train a model by writing a custom loss function `loss_fn()` and applied stochastic gradient descent optimization. However, writing the loss function and gradient updates can be a repeatable task across different projects. The `torch.nn` module provides a set of loss functions, and `torch.optim` supports most commonly used optimization algorithms that can be called to update the parameters based on the computed gradients. To see how they work, let's create a new MSE loss function and a stochastic gradient descent optimizer:

```
>>> import torch.nn as nn
>>> loss_fn = nn.MSELoss(reduction='mean')
>>> input_size = 1
>>> output_size = 1
>>> model = nn.Linear(input_size, output_size)
>>> optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Note that here we use the `torch.nn.Linear` class for the linear layer instead of manually defining it.

Now, we can simply call the `step()` method of the `optimizer` to train the model. We can pass a batched dataset (such as `train_dl`, which was created in the previous example):

```
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         # 1. Generate predictions
...         pred = model(x_batch)[:, 0]
...         # 2. Calculate loss
...         loss = loss_fn(pred, y_batch)
...         # 3. Compute gradients
...         loss.backward()
...         # 4. Update parameters using gradients
...         optimizer.step()
...         # 5. Reset the gradients to zero
...         optimizer.zero_grad()
...     if epoch % log_epochs==0:
...         print(f'Epoch {epoch}  Loss {loss.item():.4f}')
```

After the model is trained, visualize the results and make sure that they are similar to the results of the previous method. To obtain the weight and bias parameters, we can do the following:

```
>>> print('Final Parameters:', model.weight.item(), model.bias.item())
Final Parameters: 2.646660089492798 4.883835315704346
```

Building a multilayer perceptron for classifying flowers in the Iris dataset

In the previous example, you saw how to build a model from scratch. We trained this model using stochastic gradient descent optimization. While we started our journey based on the simplest possible example, you can see that defining the model from scratch, even for such a simple case, is neither appealing nor good practice. PyTorch instead provides already defined layers through `torch.nn` that can be readily used as the building blocks of an NN model. In this section, you will learn how to use these layers to solve a classification task using the Iris flower dataset (identifying between three species of irises) and build a two-layer perceptron using the `torch.nn` module. First, let's get the data from `sklearn.datasets`:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()
```

```
>>> X = iris['data']
>>> y = iris['target']
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=1./3, random_state=1)
```

Here, we randomly select 100 samples (2/3) for training and 50 samples (1/3) for testing.

Next, we standardize the features (mean centering and dividing by the standard deviation) and create a PyTorch `Dataset` for the training set and a corresponding `DataLoader`:

```
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>>> X_train_norm = torch.from_numpy(X_train_norm).float()
>>> y_train = torch.from_numpy(y_train)
>>> train_ds = TensorDataset(X_train_norm, y_train)
>>> torch.manual_seed(1)
>>> batch_size = 2
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Here, we set the batch size to 2 for the `DataLoader`.

Now, we are ready to use the `torch.nn` module to build a model efficiently. In particular, using the `nn.Module` class, we can stack a few layers and build an NN. You can see the list of all the layers that are already available at `https://pytorch.org/docs/stable/nn.html`. For this problem, we are going to use the `Linear` layer, which is also known as a fully connected layer or dense layer, and can be best represented by $f(w \times x + b)$, where x represents a tensor containing the input features, w and b are the weight matrix and the bias vector, and f is the activation function.

Each layer in an NN receives its inputs from the preceding layer; therefore, its dimensionality (rank and shape) is fixed. Typically, we need to concern ourselves with the dimensionality of output only when we design an NN architecture. Here, we want to define a model with two hidden layers. The first one receives an input of four features and projects them to 16 neurons. The second layer receives the output of the previous layer (which has a size of *16*) and projects them to three output neurons, since we have three class labels. This can be done as follows:

```
>>> class Model(nn.Module):
...     def __init__(self, input_size, hidden_size, output_size):
...         super().__init__()
...         self.layer1 = nn.Linear(input_size, hidden_size)
...         self.layer2 = nn.Linear(hidden_size, output_size)
...     def forward(self, x):
...         x = self.layer1(x)
...         x = nn.Sigmoid()(x)
...         x = self.layer2(x)
...         return x
>>> input_size = X_train_norm.shape[1]
```

```
>>> hidden_size = 16
>>> output_size = 3
>>> model = Model(input_size, hidden_size, output_size)
```

Here, we used the sigmoid activation function for the first layer and softmax activation for the last (output) layer. Softmax activation in the last layer is used to support multiclass classification since we have three class labels here (which is why we have three neurons in the output layer). We will discuss the different activation functions and their applications later in this chapter.

Next, we specify the loss function as cross-entropy loss and the optimizer as Adam:

The Adam optimizer is a robust, gradient-based optimization method, which we will talk about in detail in *Chapter 14, Classifying Images with Deep Convolutional Neural Networks*.

```
>>> learning_rate = 0.001
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

Now, we can train the model. We will specify the number of epochs to be `100`. The code of training the flower classification model is as follows:

```
>>> num_epochs = 100
>>> loss_hist = [0] * num_epochs
>>> accuracy_hist = [0] * num_epochs
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist[epoch] += loss.item()*y_batch.size(0)
...         is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
...         accuracy_hist[epoch] += is_correct.sum()
...      loss_hist[epoch] /= len(train_dl.dataset)
...      accuracy_hist[epoch] /= len(train_dl.dataset)
```

The `loss_hist` and `accuracy_hist` lists keep the training loss and the training accuracy after each epoch. We can use this to visualize the learning curves as follows:

```
>>> fig = plt.figure(figsize=(12, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(loss_hist, lw=3)
```

```
>>> ax.set_title('Training loss', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(accuracy_hist, lw=3)
>>> ax.set_title('Training accuracy', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

The learning curves (training loss and training accuracy) are as follows:

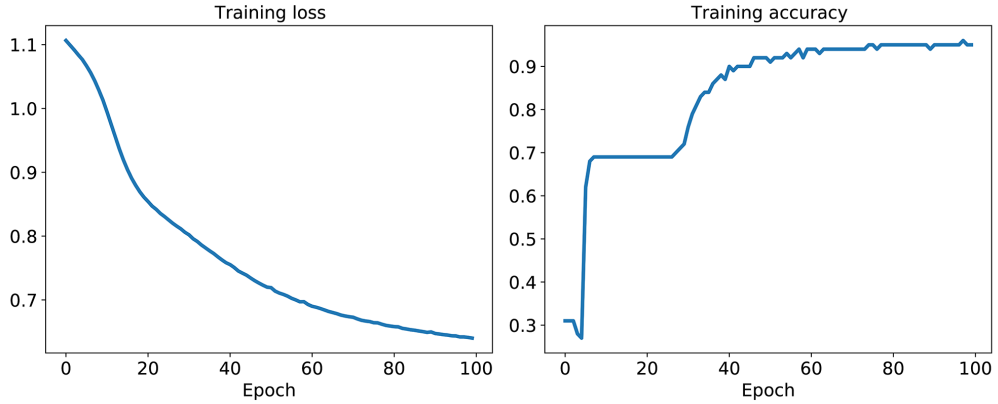


Figure 12.9: Training loss and accuracy curves

Evaluating the trained model on the test dataset

We can now evaluate the classification accuracy of the trained model on the test dataset:

```
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> X_test_norm = torch.from_numpy(X_test_norm).float()
>>> y_test = torch.from_numpy(y_test)
>>> pred_test = model(X_test_norm)
>>> correct = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> accuracy = correct.mean()
>>> print(f'Test Acc.: {accuracy:.4f}')
Test Acc.: 0.9800
```

Since we trained our model with standardized features, we also applied the same standardization to the test data. The classification accuracy is 0.98 (that is, 98 percent).

Saving and reloading the trained model

Trained models can be saved on disk for future use. This can be done as follows:

```
>>> path = 'iris_classifier.pt'
>>> torch.save(model, path)
```

Calling `save(model)` will save both the model architecture and all the learned parameters. As a common convention, we can save models using a `'pt'` or `'pth'` file extension.

Now, let's reload the saved model. Since we have saved both the model architecture and the weights, we can easily rebuild and reload the parameters in just one line:

```
>>> model_new = torch.load(path)
```

Try to verify the model architecture by calling `model_new.eval()`:

```
>>> model_new.eval()
Model(
  (layer1): Linear(in_features=4, out_features=16, bias=True)
  (layer2): Linear(in_features=16, out_features=3, bias=True)
)
```

Finally, let's evaluate this new model that is reloaded on the test dataset to verify that the results are the same as before:

```
>>> pred_test = model_new(X_test_norm)
>>> correct = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> accuracy = correct.mean()
>>> print(f'Test Acc.: {accuracy:.4f}')
Test Acc.: 0.9800
```

If you want to save only the learned parameters, you can use `save(model.state_dict())` as follows:

```
>>> path = 'iris_classifier_state.pt'
>>> torch.save(model.state_dict(), path)
```

To reload the saved parameters, we first need to construct the model as we did before, then feed the loaded parameters to the model:

```
>>> model_new = Model(input_size, hidden_size, output_size)
>>> model_new.load_state_dict(torch.load(path))
```

Choosing activation functions for multilayer neural networks

For simplicity, we have only discussed the sigmoid activation function in the context of multilayer feedforward NNs so far; we have used it in the hidden layer as well as the output layer in the MLP implementation in *Chapter 11*.

Note that in this book, the sigmoidal logistic function, $\sigma(z) = \frac{1}{1+e^{-z}}$, is referred to as the *sigmoid* function for brevity, which is common in machine learning literature. In the following subsections, you will learn more about alternative nonlinear functions that are useful for implementing multilayer NNs.

Technically, we can use any function as an activation function in multilayer NNs as long as it is differentiable. We can even use linear activation functions, such as in Adaline (*Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*). However, in practice, it would not be very useful to use linear activation functions for both hidden and output layers, since we want to introduce nonlinearity in a typical artificial NN to be able to tackle complex problems. The sum of linear functions yields a linear function after all.

The logistic (sigmoid) activation function that we used in *Chapter 11* probably mimics the concept of a neuron in a brain most closely—we can think of it as the probability of whether a neuron fires. However, the logistic (sigmoid) activation function can be problematic if we have highly negative input, since the output of the sigmoid function will be close to zero in this case. If the sigmoid function returns output that is close to zero, the NN will learn very slowly, and it will be more likely to get trapped in the local minima of the loss landscape during training. This is why people often prefer a hyperbolic tangent as an activation function in hidden layers.

Before we discuss what a hyperbolic tangent looks like, let's briefly recapitulate some of the basics of the logistic function and look at a generalization that makes it more useful for multiclass classification problems.

Logistic function recap

As was mentioned in the introduction to this section, the logistic function is, in fact, a special case of a sigmoid function. You will recall from the section on logistic regression in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, that we can use a logistic function to model the probability that sample x belongs to the positive class (class `1`) in a binary classification task.

The given net input, z, is shown in the following equation:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{i=0}^{m} w_i x_i = w^{\mathrm{T}} x$$

The logistic (sigmoid) function will compute the following:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}$$

Note that w_0 is the bias unit (y-axis intercept, which means $x_0 = 1$). To provide a more concrete example, let's take a model for a two-dimensional data point, x, and a model with the following weight coefficients assigned to the w vector:

```
>>> import numpy as np
>>> X = np.array([1, 1.4, 2.5]) ## first value must be 1
>>> w = np.array([0.4, 0.3, 0.5])
>>> def net_input(X, w):
...     return np.dot(X, w)
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)
>>> print(f'P(y=1|x) = {logistic_activation(X, w):.3f}')
P(y=1|x) = 0.888
```

If we calculate the net input (z) and use it to activate a logistic neuron with those particular feature values and weight coefficients, we get a value of `0.888`, which we can interpret as an 88.8 percent probability that this particular sample, x, belongs to the positive class.

In *Chapter 11*, we used the one-hot encoding technique to represent multiclass ground truth labels and designed the output layer consisting of multiple logistic activation units. However, as will be demonstrated by the following code example, an output layer consisting of multiple logistic activation units does not produce meaningful, interpretable probability values:

```
>>> # W : array with shape = (n_output_units, n_hidden_units+1)
>>> #     note that the first column are the bias units
>>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...               [0.2, 0.4, 1.0, 0.2],
...               [0.6, 1.5, 1.2, 0.7]])
>>> # A : data array with shape = (n_hidden_units + 1, n_samples)
>>> #     note that the first column of this array must be 1
>>> A = np.array([[1, 0.1, 0.4, 0.6]])
>>> Z = np.dot(W, A[0])
>>> y_probas = logistic(Z)
>>> print('Net Input: \n', Z)
Net Input:
[1.78  0.76  1.65]
>>> print('Output Units:\n', y_probas)
Output Units:
[ 0.85569687  0.68135373  0.83889105]
```

As you can see in the output, the resulting values cannot be interpreted as probabilities for a three-class problem. The reason for this is that they do not sum to 1. However, this is, in fact, not a big concern if we use our model to predict only the class labels and not the class membership probabilities. One way to predict the class label from the output units obtained earlier is to use the maximum value:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Predicted class label:', y_class)
Predicted class label: 0
```

In certain contexts, it can be useful to compute meaningful class probabilities for multiclass predictions. In the next section, we will take a look at a generalization of the logistic function, the `softmax` function, which can help us with this task.

Estimating class probabilities in multiclass classification via the softmax function

In the previous section, you saw how we can obtain a class label using the `argmax` function. Previously, in the *Building a multilayer perceptron for classifying flowers in the Iris dataset* section, we determined `activation='softmax'` in the last layer of the MLP model. The `softmax` function is a soft form of the `argmax` function; instead of giving a single class index, it provides the probability of each class. Therefore, it allows us to compute meaningful class probabilities in multiclass settings (multinomial logistic regression).

In `softmax`, the probability of a particular sample with net input z belonging to the ith class can be computed with a normalization term in the denominator, that is, the sum of the exponentially weighted linear functions:

$$p(z) = \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}$$

To see `softmax` in action, let's code it up in Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))
>>> y_probas = softmax(Z)
>>> print('Probabilities:\n', y_probas)
Probabilities:
[ 0.44668973  0.16107406  0.39223621]
>>> np.sum(y_probas)
1.0
```

As you can see, the predicted class probabilities now sum to 1, as we would expect. It is also notable that the predicted class label is the same as when we applied the `argmax` function to the logistic output.

It may help to think of the result of the `softmax` function as a *normalized* output that is useful for obtaining meaningful class-membership predictions in multiclass settings. Therefore, when we build a multiclass classification model in PyTorch, we can use the `torch.softmax()` function to estimate the probabilities of each class membership for an input batch of examples. To see how we can use the `torch.softmax()` activation function in PyTorch, we will convert `Z` to a tensor in the following code, with an additional dimension reserved for the batch size:

```
>>> torch.softmax(torch.from_numpy(Z), dim=0)
tensor([0.4467, 0.1611, 0.3922], dtype=torch.float64)
```

Broadening the output spectrum using a hyperbolic tangent

Another sigmoidal function that is often used in the hidden layers of artificial NNs is the **hyperbolic tangent** (commonly known as **tanh**), which can be interpreted as a rescaled version of the logistic function:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1+e^{-z}}$$

$$\sigma_{\tanh}(z) = 2 \times \sigma_{\text{logistic}}(2z) - 1 = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

The advantage of the hyperbolic tangent over the logistic function is that it has a broader output spectrum ranging in the open interval (–1, 1), which can improve the convergence of the backpropagation algorithm (*Neural Networks for Pattern Recognition*, *C. M. Bishop*, *Oxford University Press*, pages: 500-501, *1995*).

In contrast, the logistic function returns an output signal ranging in the open interval (0, 1). For a simple comparison of the logistic function and the hyperbolic tangent, let's plot the two sigmoidal functions:

```
>>> import matplotlib.pyplot as plt
>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('net input $z$')
>>> plt.ylabel('activation $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle=':')
>>> plt.axhline(0.5, color='black', linestyle=':')
>>> plt.axhline(0, color='black', linestyle=':')
```

```
>>> plt.axhline(-0.5, color='black', linestyle=':')
>>> plt.axhline(-1, color='black', linestyle=':')
>>> plt.plot(z, tanh_act,
...          linewidth=3, linestyle='--',
...          label='tanh')
>>> plt.plot(z, log_act,
...          linewidth=3,
...          label='logistic')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

As you can see, the shapes of the two sigmoidal curves look very similar; however, the `tanh` function has double the output space of the `logistic` function:

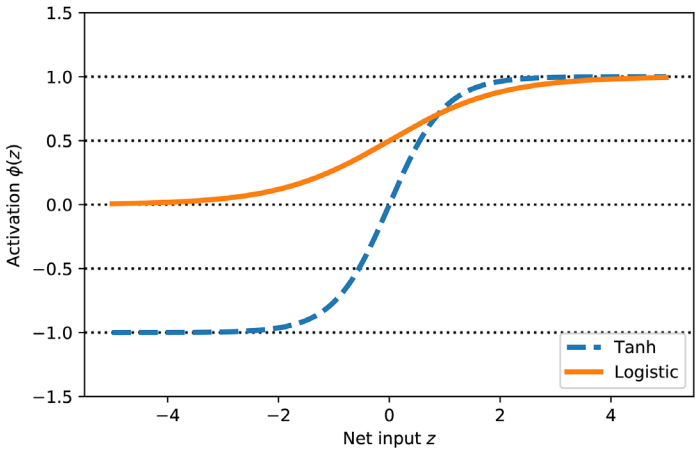


Figure 12.10: A comparison of the tanh and logistic functions

Note that we previously implemented the `logistic` and `tanh` functions verbosely for the purpose of illustration. In practice, we can use NumPy's `tanh` function.

Alternatively, when building an NN model, we can use `torch.tanh(x)` in PyTorch to achieve the same results:

```
>>> np.tanh(z)
array([-0.9999092 , -0.99990829, -0.99990737, ...,  0.99990644,
        0.99990737,  0.99990829])
>>> torch.tanh(torch.from_numpy(z))
tensor([-0.9999, -0.9999, -0.9999,  ...,  0.9999,  0.9999,  0.9999],
       dtype=torch.float64)
```

In addition, the logistic function is available in SciPy's `special` module:

```
>>> from scipy.special import expit
>>> expit(z)
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669, 0.99324034,
       0.99327383])
```

Similarly, we can use the `torch.sigmoid()` function in PyTorch to do the same computation, as follows:

```
>>> torch.sigmoid(torch.from_numpy(z))
tensor([0.0067, 0.0067, 0.0068,  ..., 0.9932, 0.9932, 0.9933],
       dtype=torch.float64)
```

Note that using `torch.sigmoid(x)` produces results that are equivalent to `torch.nn.Sigmoid()(x)`, which we used earlier. `torch.nn.Sigmoid` is a class to which you can pass in parameters to construct an object in order to control the behavior. In contrast, `torch.sigmoid` is a function.

Rectified linear unit activation

The **rectified linear unit** (**ReLU**) is another activation function that is often used in deep NNs. Before we delve into ReLU, we should step back and understand the vanishing gradient problem of tanh and logistic activations.

To understand this problem, let's assume that we initially have the net input $z_1 = 20$, which changes to $z_2 = 25$. Computing the tanh activation, we get $\sigma(z_1) = 1.0$ and $\sigma(z_2) = 1.0$, which shows no change in the output (due to the asymptotic behavior of the tanh function and numerical errors).

This means that the derivative of activations with respect to the net input diminishes as z becomes large. As a result, learning the weights during the training phase becomes very slow because the gradient terms may be very close to zero. ReLU activation addresses this issue. Mathematically, ReLU is defined as follows:

$$\sigma(z) = \max(0, z)$$

ReLU is still a nonlinear function that is good for learning complex functions with NNs. Besides this, the derivative of ReLU, with respect to its input, is always 1 for positive input values. Therefore, it solves the problem of vanishing gradients, making it suitable for deep NNs. In PyTorch, we can apply the ReLU activation `torch.relu()` as follows:

```
>>> torch.relu(torch.from_numpy(z))
tensor([0.0000, 0.0000, 0.0000,  ..., 4.9850, 4.9900, 4.9950],
       dtype=torch.float64)
```

We will use the ReLU activation function in the next chapter as an activation function for multilayer convolutional NNs.

Now that we know more about the different activation functions that are commonly used in artificial NNs, let's conclude this section with an overview of the different activation functions that we have encountered so far in this book:

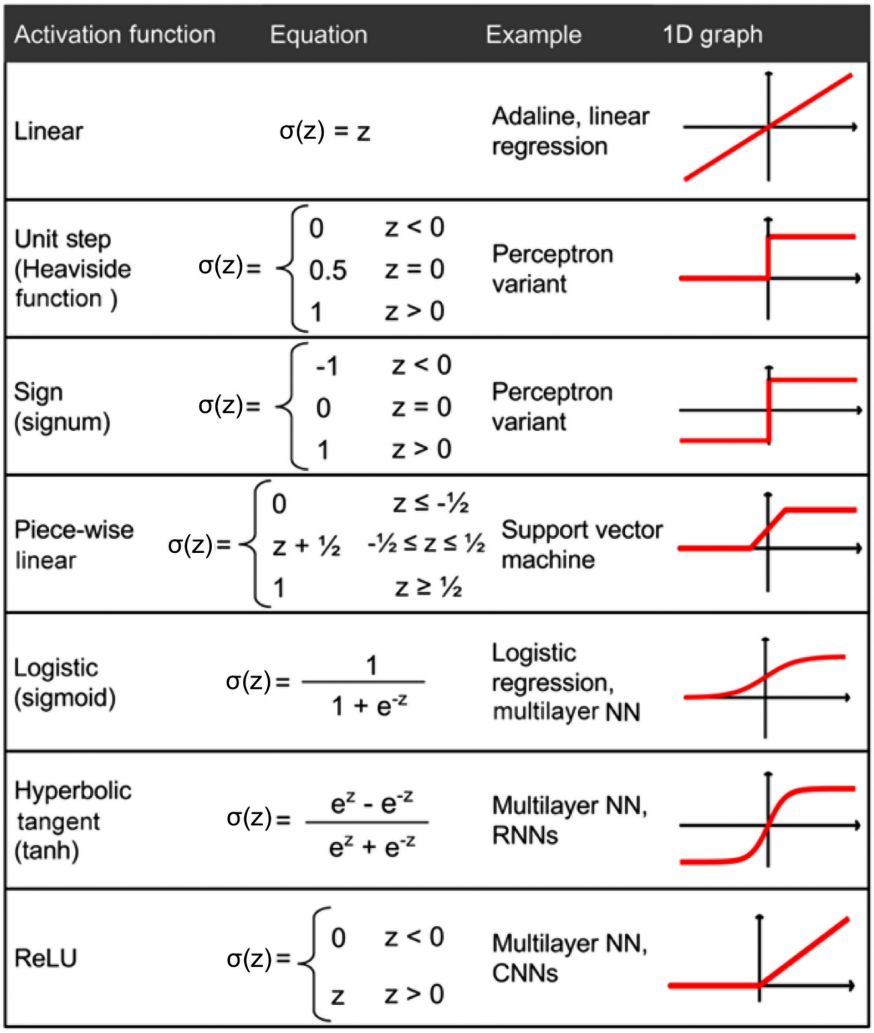


Figure 12.11: The activation functions covered in this book

You can find the list of all activation functions available in the `torch.nn` module at `https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions`.

Summary

In this chapter, you learned how to use PyTorch, an open source library for numerical computations, with a special focus on deep learning. While PyTorch is more inconvenient to use than NumPy, due to its additional complexity to support GPUs, it allows us to define and train large, multilayer NNs very efficiently.

Also, you learned about using the `torch.nn` module to build complex machine learning and NN models and run them efficiently. We explored model building in PyTorch by defining a model from scratch via the basic PyTorch tensor functionality. Implementing models can be tedious when we have to program at the level of matrix-vector multiplications and define every detail of each operation. However, the advantage is that this allows us, as developers, to combine such basic operations and build more complex models. We then explored `torch.nn`, which makes building NN models a lot easier than implementing them from scratch.

Finally, you learned about different activation functions and understood their behaviors and applications. Specifically, in this chapter, we covered tanh, softmax, and ReLU.

In the next chapter, we'll continue our journey and dive deeper into PyTorch, where we'll find ourselves working with PyTorch computation graphs and the automatic differentiation package. Along the way, you'll learn many new concepts, such as gradient computations.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`

13

Going Deeper – The Mechanics of PyTorch

In *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, we covered how to define and manipulate tensors and worked with the `torch.utils.data` module to build input pipelines. We further built and trained a multilayer perceptron to classify the Iris dataset using the PyTorch neural network module (`torch.nn`).

Now that we have some hands-on experience with PyTorch neural network training and machine learning, it's time to take a deeper dive into the PyTorch library and explore its rich set of features, which will allow us to implement more advanced deep learning models in upcoming chapters.

In this chapter, we will use different aspects of PyTorch's API to implement NNs. In particular, we will again use the `torch.nn` module, which provides multiple layers of abstraction to make the implementation of standard architectures very convenient. It also allows us to implement custom NN layers, which is very useful in research-oriented projects that require more customization. Later in this chapter, we will implement such a custom layer.

To illustrate the different ways of model building using the `torch.nn` module, we will also consider the classic **exclusive or** (**XOR**) problem. Firstly, we will build multilayer perceptrons using the `Sequential` class. Then, we will consider other methods, such as subclassing `nn.Module` for defining custom layers. Finally, we will work on two real-world projects that cover the machine learning steps from raw input to prediction.

The topics that we will cover are as follows:

- Understanding and working with PyTorch computation graphs
- Working with PyTorch tensor objects
- Solving the classic XOR problem and understanding model capacity
- Building complex NN models using PyTorch's `Sequential` class and the `nn.Module` class
- Computing gradients using automatic differentiation and `torch.autograd`

The key features of PyTorch

In the previous chapter, we saw that PyTorch provides us with a scalable, multiplatform programming interface for implementing and running machine learning algorithms. After its initial release in 2016 and its 1.0 release in 2018, PyTorch has evolved into one of the two most popular frameworks for deep learning. It uses dynamic computational graphs, which have the advantage of being more flexible compared to its static counterparts. Dynamic computational graphs are debugging friendly: PyTorch allows for interleaving the graph declaration and graph evaluation steps. You can execute the code line by line while having full access to all variables. This is a very important feature that makes the development and training of NNs very convenient.

While PyTorch is an open-source library and can be used for free by everyone, its development is funded and supported by Facebook. This involves a large team of software engineers who expand and improve the library continuously. Since PyTorch is an open-source library, it also has strong support from other developers outside of Facebook, who avidly contribute and provide user feedback. This has made the PyTorch library more useful to both academic researchers and developers. A further consequence of these factors is that PyTorch has extensive documentation and tutorials to help new users.

Another key feature of PyTorch, which was also noted in the previous chapter, is its ability to work with single or multiple **graphical processing units** (**GPUs**). This allows users to train deep learning models very efficiently on large datasets and large-scale systems.

Last but not least, PyTorch supports mobile deployment, which also makes it a very suitable tool for production.

In the next section, we will look at how a tensor and function in PyTorch are interconnected via a computation graph.

PyTorch's computation graphs

PyTorch performs its computations based on a **directed acyclic graph** (**DAG**). In this section, we will see how these graphs can be defined for a simple arithmetic computation. Then, we will see the dynamic graph paradigm, as well as how the graph is created on the fly in PyTorch.

Understanding computation graphs

PyTorch relies on building a computation graph at its core, and it uses this computation graph to derive relationships between tensors from the input all the way to the output. Let's say that we have rank 0 (scalar) tensors a, b, and c and we want to evaluate $z = 2 \times (a - b) + c$.

This evaluation can be represented as a computation graph, as shown in *Figure 13.1*:

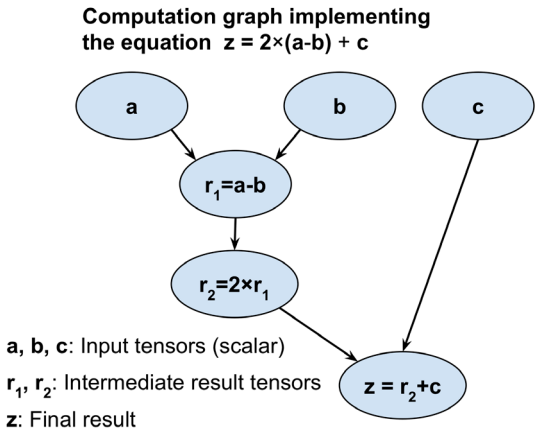


Figure 13.1: How a computation graph works

As you can see, the computation graph is simply a network of nodes. Each node resembles an operation, which applies a function to its input tensor or tensors and returns zero or more tensors as the output. PyTorch builds this computation graph and uses it to compute the gradients accordingly. In the next subsection, we will see some examples of creating a graph for this computation using PyTorch.

Creating a graph in PyTorch

Let's look at a simple example that illustrates how to create a graph in PyTorch for evaluating $z = 2 \times (a - b) + c$, as shown in the previous figure. The variables a, b, and c are scalars (single numbers), and we define these as PyTorch tensors. To create the graph, we can simply define a regular Python function with `a`, `b`, and `c` as its input arguments, for example:

```
>>> import torch
>>> def compute_z(a, b, c):
...     r1 = torch.sub(a, b)
...     r2 = torch.mul(r1, 2)
...     z = torch.add(r2, c)
...     return z
```

Now, to carry out the computation, we can simply call this function with tensor objects as function arguments. Note that PyTorch functions such as `add`, `sub` (or `subtract`), and `mul` (or `multiply`) also allow us to provide inputs of higher ranks in the form of a PyTorch tensor object. In the following code example, we provide scalar inputs (rank 0), as well as rank 1 and rank 2 inputs, as lists:

```
>>> print('Scalar Inputs:', compute_z(torch.tensor(1),
...       torch.tensor(2), torch.tensor(3)))
Scalar Inputs: tensor(1)
>>> print('Rank 1 Inputs:', compute_z(torch.tensor([1]),
...       torch.tensor([2]), torch.tensor([3])))
Rank 1 Inputs: tensor([1])
>>> print('Rank 2 Inputs:', compute_z(torch.tensor([[1]]),
...       torch.tensor([[2]]), torch.tensor([[3]])))
Rank 2 Inputs: tensor([[1]])
```

In this section, you saw how simple it is to create a computation graph in PyTorch. Next, we will look at PyTorch tensors that can be used for storing and updating model parameters.

PyTorch tensor objects for storing and updating model parameters

We covered tensor objects in *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*. In PyTorch, a special tensor object for which gradients need to be computed allows us to store and update the parameters of our models during training. Such a tensor can be created by just assigning `requires_grad` to `True` on user-specified initial values. Note that as of now (mid-2021), only tensors of floating point and complex `dtype` can require gradients. In the following code, we will generate tensor objects of type `float32`:

```
>>> a = torch.tensor(3.14, requires_grad=True)
>>> print(a)
tensor(3.1400, requires_grad=True)
>>> b = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> print(b)
tensor([1., 2., 3.], requires_grad=True)
```

Notice that `requires_grad` is set to `False` by default. This value can be efficiently set to `True` by running `requires_grad_()`.

`method_()` is an in-place method in PyTorch that is used for operations without making a copy of the input.

Let's take a look at the following example:

```
>>> w = torch.tensor([1.0, 2.0, 3.0])
>>> print(w.requires_grad)
False
>>> w.requires_grad_()
>>> print(w.requires_grad)
True
```

You will recall that for NN models, initializing model parameters with random weights is necessary to break the symmetry during backpropagation—otherwise, a multilayer NN would be no more useful than a single-layer NN like logistic regression. When creating a PyTorch tensor, we can also use a random initialization scheme. PyTorch can generate random numbers based on a variety of probability distributions (see `https://pytorch.org/docs/stable/torch.html#random-sampling`). In the following example, we will take a look at some standard initialization methods that are also available in the `torch.nn.init` module (see `https://pytorch.org/docs/stable/nn.init.html`).

So, let's look at how we can create a tensor with Glorot initialization, which is a classic random initialization scheme that was proposed by Xavier Glorot and Yoshua Bengio. For this, we first create an empty tensor and an operator called `init` as an object of class `GlorotNormal`. Then, we fill this tensor with values according to the Glorot initialization by calling the `xavier_normal_()` method. In the following example, we initialize a tensor of shape 2×3:

```
>>> import torch.nn as nn
>>> torch.manual_seed(1)
>>> w = torch.empty(2, 3)
>>> nn.init.xavier_normal_(w)
>>> print(w)
tensor([[ 0.4183,  0.1688,  0.0390],
        [ 0.3930, -0.2858, -0.1051]])
```

Xavier (or Glorot) initialization

In the early development of deep learning, it was observed that random uniform or random normal weight initialization could often result in poor model performance during training.

In 2010, Glorot and Bengio investigated the effect of initialization and proposed a novel, more robust initialization scheme to facilitate the training of deep networks. The general idea behind Xavier initialization is to roughly balance the variance of the gradients across different layers. Otherwise, some layers may get too much attention during training while the other layers lag behind.

According to the research paper by Glorot and Bengio, if we want to initialize the weights in a uniform distribution, we should choose the interval of this uniform distribution as follows:

$$W \sim Uniform\left(-\frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in}+n_{out}}}\right)$$

Here, n_{in} is the number of input neurons that are multiplied by the weights, and n_{out} is the number of output neurons that feed into the next layer. For initializing the weights from Gaussian (normal) distribution, we recommend that you choose the standard deviation of this Gaussian to be:

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{in}+n_{out}}}$$

PyTorch supports Xavier initialization in both uniform and normal distributions of weights.

For more information about Glorot and Bengio's initialization scheme, including the rationale and mathematical motivation, we recommend the original paper (*Understanding the difficulty of deep feedforward neural networks*, *Xavier Glorot* and *Yoshua Bengio*, 2010), which is freely available at `http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf`.

Now, to put this into the context of a more practical use case, let's see how we can define two `Tensor` objects inside the base `nn.Module` class:

```
>>> class MyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.w1 = torch.empty(2, 3, requires_grad=True)
...         nn.init.xavier_normal_(self.w1)
...         self.w2 = torch.empty(1, 2, requires_grad=True)
...         nn.init.xavier_normal_(self.w2)
```

These two tensors can be then used as weights whose gradients will be computed via automatic differentiation.

Computing gradients via automatic differentiation

As you already know, optimizing NNs requires computing the gradients of the loss with respect to the NN weights. This is required for optimization algorithms such as **stochastic gradient descent** (**SGD**). In addition, gradients have other applications, such as diagnosing the network to find out why an NN model is making a particular prediction for a test example. Therefore, in this section, we will cover how to compute gradients of a computation with respect to its input variables.

Computing the gradients of the loss with respect to trainable variables

PyTorch supports *automatic differentiation*, which can be thought of as an implementation of the *chain rule* for computing gradients of nested functions. Note that for the sake of simplicity, we will use the term *gradient* to refer to both partial derivatives and gradients.

Partial derivatives and gradients

A partial derivative $\frac{\partial f}{\partial x_1}$ can be understood as the rate of change of a multivariate function—a function with multiple inputs, $f(x_1, x_2, ...)$, with respect to one of its inputs (here: x_1). The gradient, ∇f, of a function is a vector composed of all the inputs' partial derivatives, $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ...\right)$.

When we define a series of operations that results in some output or even intermediate tensors, PyTorch provides a context for calculating gradients of these computed tensors with respect to its dependent nodes in the computation graph. To compute these gradients, we can call the `backward` method from the `torch.autograd` module. It computes the sum of gradients of the given tensor with regard to leaf nodes (terminal nodes) in the graph.

Let's work with a simple example where we will compute $z = wx + b$ and define the loss as the squared loss between the target y and prediction z, $Loss = (y - z)^2$. In the more general case, where we may have multiple predictions and targets, we compute the loss as the sum of the squared error, $Loss = \sum_i (y_i - z_i)^2$. In order to implement this computation in PyTorch, we will define the model parameters, w and b, as variables (tensors with the `requires_gradient` attribute set to `True`), and the input, x and y, as default tensors. We will compute the loss tensor and use it to compute the gradients of the model parameters, w and b, as follows:

```
>>> w = torch.tensor(1.0, requires_grad=True)
>>> b = torch.tensor(0.5, requires_grad=True)
>>> x = torch.tensor([1.4])
>>> y = torch.tensor([2.1])
>>> z = torch.add(torch.mul(w, x), b)
>>> loss = (y-z).pow(2).sum()
```

```
>>> loss.backward()
>>> print('dL/dw : ', w.grad)
>>> print('dL/db : ', b.grad)
dL/dw :  tensor(-0.5600)
dL/db :  tensor(-0.4000)
```

Computing the value z is a forward pass in an NN. We used the `backward` method on the `loss` tensor to compute $\frac{\partial Loss}{\partial w}$ and $\frac{\partial Loss}{\partial b}$. Since this is a very simple example, we can obtain $\frac{\partial Loss}{\partial w} = 2x(wx + b - y)$ symbolically to verify that the computed gradients match the results we obtained in the previous code example:

```
>>> # verifying the computed gradient
>>> print(2 * x * ((w * x + b) - y))
tensor([-0.5600], grad_fn=<MulBackward0>)
```

We leave the verification of b as an exercise for the reader.

Understanding automatic differentiation

Automatic differentiation represents a set of computational techniques for computing gradients of arbitrary arithmetic operations. During this process, gradients of a computation (expressed as a series of operations) are obtained by accumulating the gradients through repeated applications of the chain rule. To better understand the concept behind automatic differentiation, let's consider a series of nested computations, $y = f(g(h(x)))$, with input x and output y. This can be broken into a series of steps:

- $u_0 = x$
- $u_1 = h(x)$
- $u_2 = g(u_1)$
- $u_3 = f(u_2) = y$

The derivative $\frac{dy}{dx}$ can be computed in two different ways: forward accumulation, which starts with $\frac{du_3}{dx} = \frac{du_3}{du_2}\frac{du_2}{du_0}$, and reverse accumulation, which starts with $\frac{dy}{du_0} = \frac{dy}{du_1}\frac{du_1}{du_0}$. Note that PyTorch uses the latter, reverse accumulation, which is more efficient for implementing backpropagation.

Adversarial examples

Computing gradients of the loss with respect to the input example is used for generating *adversarial examples* (or *adversarial attacks*). In computer vision, adversarial examples are examples that are generated by adding some small, imperceptible noise (or perturbations) to the input example, which results in a deep NN misclassifying them. Covering adversarial examples is beyond the scope of this book, but if you are interested, you can find the original paper by *Christian Szegedy et al.*, *Intriguing properties of neural networks* at `https://arxiv.org/pdf/1312.6199.pdf`.

Simplifying implementations of common architectures via the torch.nn module

You have already seen some examples of building a feedforward NN model (for instance, a multilayer perceptron) and defining a sequence of layers using the `nn.Module` class. Before we take a deeper dive into `nn.Module`, let's briefly look at another approach for conjuring those layers via `nn.Sequential`.

Implementing models based on nn.Sequential

With `nn.Sequential` (`https://pytorch.org/docs/master/generated/torch.nn.Sequential.html#sequential`), the layers stored inside the model are connected in a cascaded way. In the following example, we will build a model with two densely (fully) connected layers:

```
>>> model = nn.Sequential(
...     nn.Linear(4, 16),
...     nn.ReLU(),
...     nn.Linear(16, 32),
...     nn.ReLU()
... )
>>> model
Sequential(
  (0): Linear(in_features=4, out_features=16, bias=True)
  (1): ReLU()
  (2): Linear(in_features=16, out_features=32, bias=True)
  (3): ReLU()
)
```

We specified the layers and instantiated the `model` after passing the layers to the `nn.Sequential` class. The output of the first fully connected layer is used as the input to the first ReLU layer. The output of the first ReLU layer becomes the input for the second fully connected layer. Finally, the output of the second fully connected layer is used as the input to the second ReLU layer.

We can further configure these layers, for example, by applying different activation functions, initializers, or regularization methods to the parameters. A comprehensive and complete list of available options for most of these categories can be found in the official documentation:

- Choosing activation functions: `https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity`
- Initializing the layer parameters via `nn.init`: `https://pytorch.org/docs/stable/nn.init.html`
- Applying L2 regularization to the layer parameters (to prevent overfitting) via the parameter `weight_decay` of some optimizers in `torch.optim`: `https://pytorch.org/docs/stable/optim.html`

- Applying L1 regularization to the layer parameters (to prevent overfitting) by adding the L1 penalty term to the loss tensor, which we will implement next

In the following code example, we will configure the first fully connected layer by specifying the initial value distribution for the weight. Then, we will configure the second fully connected layer by computing the L1 penalty term for the weight matrix:

```
>>> nn.init.xavier_uniform_(model[0].weight)
>>> l1_weight = 0.01
>>> l1_penalty = l1_weight * model[2].weight.abs().sum()
```

Here, we initialized the weight of the first linear layer with Xavier initialization. And we computed the L1 norm of the weight of the second linear layer.

Furthermore, we can also specify the type of optimizer and the loss function for training. Again, a comprehensive list of all available options can be found in the official documentation:

- Optimizers via `torch.optim`: `https://pytorch.org/docs/stable/optim.html#algorithms`
- Loss functions: `https://pytorch.org/docs/stable/nn.html#loss-functions`

Choosing a loss function

Regarding the choices for optimization algorithms, SGD and Adam are the most widely used methods. The choice of loss function depends on the task; for example, you might use mean square error loss for a regression problem.

The family of cross-entropy loss functions supplies the possible choices for classification tasks, which are extensively discussed in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*.

Furthermore, you can use the techniques you have learned from previous chapters (such as techniques for model evaluation from *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*) combined with the appropriate metrics for the problem. For example, precision and recall, accuracy, **area under the curve** (**AUC**), and false negative and false positive scores are appropriate metrics for evaluating classification models.

In this example, we will use the SGD optimizer, and cross-entropy loss for binary classification:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Next, we will look at a more practical example: solving the classic XOR classification problem. First, we will use the `nn.Sequential()` class to build the model. Along the way, you will also learn about the capacity of a model for handling nonlinear decision boundaries. Then, we will cover building a model via `nn.Module` that will give us more flexibility and control over the layers of the network.

Solving an XOR classification problem

The XOR classification problem is a classic problem for analyzing the capacity of a model with regard to capturing the nonlinear decision boundary between two classes. We generate a toy dataset of 200 training examples with two features (x_0, x_1) drawn from a uniform distribution between [–1, 1). Then, we assign the ground truth label for training example i according to the following rule:

$$y^{(i)} = \begin{cases} 0 & \text{if } x_0^{(i)} \times x_1^{(i)} < 0 \\ 1 & \text{otherwise} \end{cases}$$

We will use half of the data (100 training examples) for training and the remaining half for validation. The code for generating the data and splitting it into the training and validation datasets is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> torch.manual_seed(1)
>>> np.random.seed(1)
>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1]<0] = 0
>>> n_train = 100
>>> x_train = torch.tensor(x[:n_train, :], dtype=torch.float32)
>>> y_train = torch.tensor(y[:n_train], dtype=torch.float32)
>>> x_valid = torch.tensor(x[n_train:, :], dtype=torch.float32)
>>> y_valid = torch.tensor(y[n_train:], dtype=torch.float32)
>>> fig = plt.figure(figsize=(6, 6))
>>> plt.plot(x[y==0, 0], x[y==0, 1], 'o', alpha=0.75, markersize=10)
>>> plt.plot(x[y==1, 0], x[y==1, 1], '<', alpha=0.75, markersize=10)
>>> plt.xlabel(r'$x_1$', size=15)
>>> plt.ylabel(r'$x_2$', size=15)
>>> plt.show()
```

The code results in the following scatterplot of the training and validation examples, shown with different markers based on their class label:

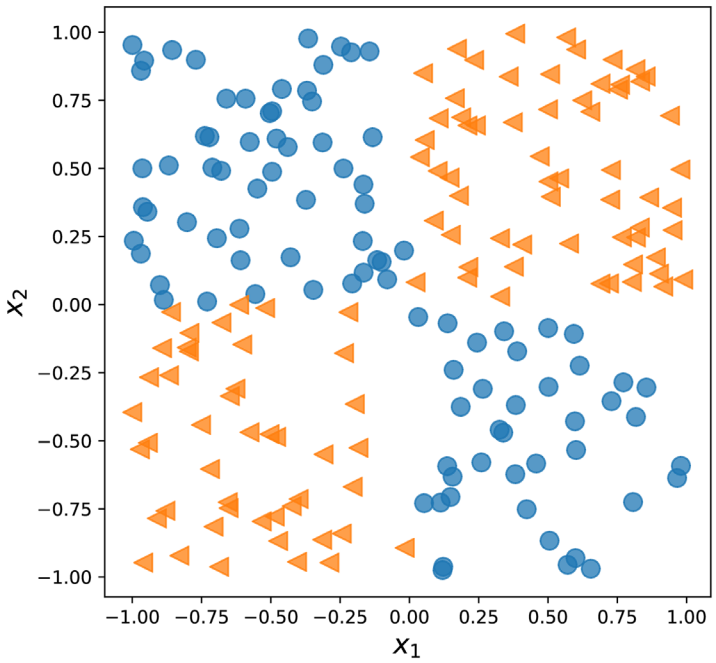


Figure 13.2: Scatterplot of training and validation examples

In the previous subsection, we covered the essential tools that we need to implement a classifier in PyTorch. We now need to decide what architecture we should choose for this task and dataset. As a general rule of thumb, the more layers we have, and the more neurons we have in each layer, the larger the capacity of the model will be. Here, the model capacity can be thought of as a measure of how readily the model can approximate complex functions. While having more parameters means the network can fit more complex functions, larger models are usually harder to train (and prone to overfitting). In practice, it is always a good idea to start with a simple model as a baseline, for example, a single-layer NN like logistic regression:

```
>>> model = nn.Sequential(
...     nn.Linear(2, 1),
...     nn.Sigmoid()
... )
>>> model
Sequential(
  (0): Linear(in_features=2, out_features=1, bias=True)
  (1): Sigmoid()
)
```

After defining the model, we will initialize the cross-entropy loss function for binary classification and the SGD optimizer:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Next, we will create a data loader that uses a batch size of 2 for the train data:

```
>>> from torch.utils.data import DataLoader, TensorDataset
>>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 2
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Now we will train the model for 200 epochs and record a history of training epochs:

```
>>> torch.manual_seed(1)
>>> num_epochs = 200
>>> def train(model, num_epochs, train_dl, x_valid, y_valid):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)[:, 0]
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item()
...             is_correct = ((pred>=0.5).float() == y_batch).float()
...             accuracy_hist_train[epoch] += is_correct.mean()
...         loss_hist_train[epoch] /= n_train/batch_size
...         accuracy_hist_train[epoch] /= n_train/batch_size
...         pred = model(x_valid)[:, 0]
...         loss = loss_fn(pred, y_valid)
...         loss_hist_valid[epoch] = loss.item()
...         is_correct = ((pred>=0.5).float() == y_valid).float()
...         accuracy_hist_valid[epoch] += is_correct.mean()
...     return loss_hist_train, loss_hist_valid, \
...            accuracy_hist_train, accuracy_hist_valid
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Notice that the history of training epochs includes the train loss and validation loss and the train accuracy and validation accuracy, which is useful for visual inspection after training. In the following code, we will plot the learning curves, including the training and validation loss, as well as their accuracies.

The following code will plot the training performance:

```
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(history[0], lw=4)
>>> plt.plot(history[1], lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> plt.plot(history[2], lw=4)
>>> plt.plot(history[3], lw=4)
>>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
```

This results in the following figure, with two separate panels for the losses and accuracies:

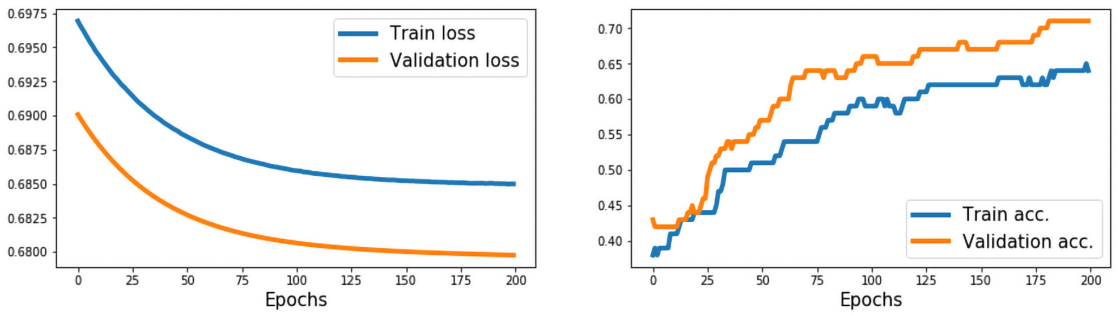


Figure 13.3: Loss and accuracy results

As you can see, a simple model with no hidden layer can only derive a linear decision boundary, which is unable to solve the XOR problem. As a consequence, we can observe that the loss terms for both the training and the validation datasets are very high, and the classification accuracy is very low.

To derive a nonlinear decision boundary, we can add one or more hidden layers connected via nonlinear activation functions. The universal approximation theorem states that a feedforward NN with a single hidden layer and a relatively large number of hidden units can approximate arbitrary continuous functions relatively well. Thus, one approach for tackling the XOR problem more satisfactorily is to add a hidden layer and compare different numbers of hidden units until we observe satisfactory results on the validation dataset. Adding more hidden units would correspond to increasing the width of a layer.

Alternatively, we can also add more hidden layers, which will make the model deeper. The advantage of making a network deeper rather than wider is that fewer parameters are required to achieve a comparable model capacity.

However, a downside of deep (versus wide) models is that deep models are prone to vanishing and exploding gradients, which make them harder to train.

As an exercise, try adding one, two, three, and four hidden layers, each with four hidden units. In the following example, we will take a look at the results of a feedforward NN with two hidden layers:

```
>>> model = nn.Sequential(
...     nn.Linear(2, 4),
...     nn.ReLU(),
...     nn.Linear(4, 4),
...     nn.ReLU(),
...     nn.Linear(4, 1),
...     nn.Sigmoid()
... )
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> model
Sequential(
  (0): Linear(in_features=2, out_features=4, bias=True)
  (1): ReLU()
  (2): Linear(in_features=4, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
  (5): Sigmoid()
)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

We can repeat the previous code for visualization, which produces the following:

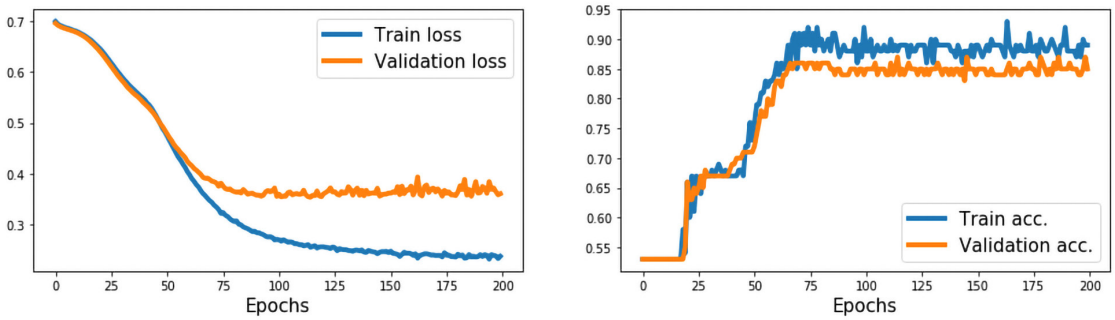


Figure 13.4: Loss and accuracy results after adding two hidden layers

Now, we can see that the model is able to derive a nonlinear decision boundary for this data, and the model reaches 100 percent accuracy on the training dataset. The validation dataset's accuracy is 95 percent, which indicates that the model is slightly overfitting.

Making model building more flexible with nn.Module

In the previous example, we used the PyTorch `Sequential` class to create a fully connected NN with multiple layers. This is a very common and convenient way of building models. However, it unfortunately doesn't allow us to create more complex models that have multiple input, output, or intermediate branches. That's where `nn.Module` comes in handy.

The alternative way to build complex models is by subclassing `nn.Module`. In this approach, we create a new class derived from `nn.Module` and define the method, `__init__()`, as a constructor. The `forward()` method is used to specify the forward pass. In the constructor function, `__init__()`, we define the layers as attributes of the class so that they can be accessed via the `self` reference attribute. Then, in the `forward()` method, we specify how these layers are to be used in the forward pass of the NN. The code for defining a new class that implements the previous model is as follows:

```
>>> class MyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         l1 = nn.Linear(2, 4)
...         a1 = nn.ReLU()
...         l2 = nn.Linear(4, 4)
...         a2 = nn.ReLU()
...         l3 = nn.Linear(4, 1)
...         a3 = nn.Sigmoid()
...         l = [l1, a1, l2, a2, l3, a3]
...         self.module_list = nn.ModuleList(l)
...
...     def forward(self, x):
...         for f in self.module_list:
...             x = f(x)
...         return x
```

Notice that we put all layers in the `nn.ModuleList` object, which is just a `list` object composed of `nn.Module` items. This makes the code more readable and easier to follow.

Once we define an instance of this new class, we can train it as we did previously:

```
>>> model = MyModule()
>>> model
MyModule(
  (module_list): ModuleList(
    (0): Linear(in_features=2, out_features=4, bias=True)
    (1): ReLU()
    (2): Linear(in_features=4, out_features=4, bias=True)
    (3): ReLU()
```

```
    (4): Linear(in_features=4, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Next, besides the train history, we will use the mlxtend library to visualize the validation data and the decision boundary.

Mlxtend can be installed via `conda` or `pip` as follows:

```
conda install mlxtend -c conda-forge
pip install mlxtend
```

To compute the decision boundary of our model, we need to add a `predict()` method in the `MyModule` class:

```
>>>     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
```

It will return the predicted class (0 or 1) for a sample.

The following code will plot the training performance along with the decision region bias:

```
>>> from mlxtend.plotting import plot_decision_regions
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history[0], lw=4)
>>> plt.plot(history[1], lw=4)
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history[2], lw=4)
>>> plt.plot(history[3], lw=4)
>>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid.numpy(),
...                       y=y_valid.numpy().astype(np.integer),
...                       clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
```

```
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

This results in *Figure 13.5*, with three separate panels for the losses, accuracies, and the scatterplot of the validation examples, along with the decision boundary:

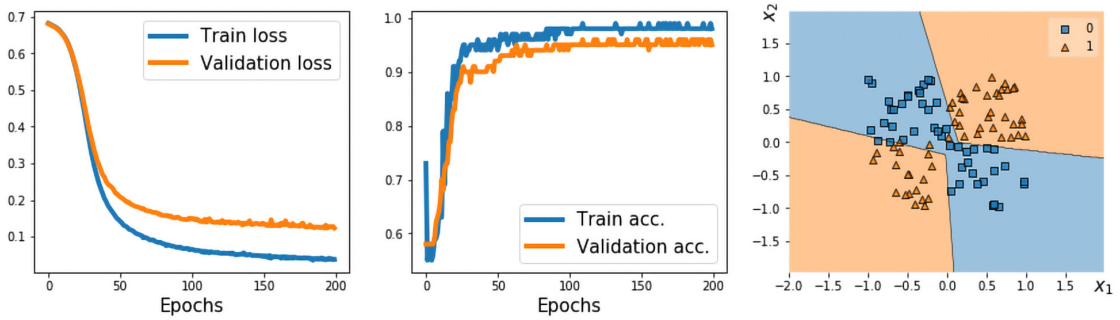


Figure 13.5: Results, including a scatterplot

Writing custom layers in PyTorch

In cases where we want to define a new layer that is not already supported by PyTorch, we can define a new class derived from the `nn.Module` class. This is especially useful when designing a new layer or customizing an existing layer.

To illustrate the concept of implementing custom layers, let's consider a simple example. Imagine we want to define a new linear layer that computes $w(x + \epsilon) + b$, where ϵ refers to a random variable as a noise variable. To implement this computation, we define a new class as a subclass of `nn.Module`. For this new class, we have to define both the constructor `__init__()` method and the `forward()` method. In the constructor, we define the variables and other required tensors for our customized layer. We can create variables and initialize them in the constructor if the `input_size` is given to the constructor. Alternatively, we can delay the variable initialization (for instance, if we do not know the exact input shape upfront) and delegate it to another method for late variable creation.

To look at a concrete example, we are going to define a new layer called `NoisyLinear`, which implements the computation $w(x + \epsilon) + b$, which was mentioned in the preceding paragraph:

```
>>> class NoisyLinear(nn.Module):
...     def __init__(self, input_size, output_size,
...                  noise_stddev=0.1):
...         super().__init__()
...         w = torch.Tensor(input_size, output_size)
...         self.w = nn.Parameter(w)  # nn.Parameter is a Tensor
...                                   # that's a module parameter.
```

```
...         nn.init.xavier_uniform_(self.w)
...         b = torch.Tensor(output_size).fill_(0)
...         self.b = nn.Parameter(b)
...         self.noise_stddev = noise_stddev
...
...     def forward(self, x, training=False):
...         if training:
...             noise = torch.normal(0.0, self.noise_stddev, x.shape)
...             x_new = torch.add(x, noise)
...         else:
...             x_new = x
...         return torch.add(torch.mm(x_new, self.w), self.b)
```

In the constructor, we have added an argument, `noise_stddev`, to specify the standard deviation for the distribution of ϵ, which is sampled from a Gaussian distribution. Furthermore, notice that in the `forward()` method, we have used an additional argument, `training=False`. We use it to distinguish whether the layer is used during training or only for prediction (this is sometimes also called *inference*) or evaluation. Also, there are certain methods that behave differently in training and prediction modes. You will encounter an example of such a method, `Dropout`, in the upcoming chapters. In the previous code snippet, we also specified that the random vector, ϵ, was to be generated and added to the input during training only and not used for inference or evaluation.

Before we go a step further and use our custom `NoisyLinear` layer in a model, let's test it in the context of a simple example.

1. In the following code, we will define a new instance of this layer, and execute it on an input tensor. Then, we will call the layer three times on the same input tensor:

```
>>> torch.manual_seed(1)
>>> noisy_layer = NoisyLinear(4, 2)
>>> x = torch.zeros((1, 4))
>>> print(noisy_layer(x, training=True))
tensor([[ 0.1154, -0.0598]], grad_fn=<AddBackward0>)
>>> print(noisy_layer(x, training=True))
tensor([[ 0.0432, -0.0375]], grad_fn=<AddBackward0>)
>>> print(noisy_layer(x, training=False))
tensor([[0., 0.]], grad_fn=<AddBackward0>)
```

Note that the outputs for the first two calls differ because the `NoisyLinear` layer added random noise to the input tensor. The third call outputs [0, 0] as we didn't add noise by specifying `training=False`.

2. Now, let's create a new model similar to the previous one for solving the XOR classification task. As before, we will use the `nn.Module` class for model building, but this time, we will use our `NoisyLinear` layer as the first hidden layer of the multilayer perceptron. The code is as follows:

```
>>> class MyNoisyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.l1 = NoisyLinear(2, 4, 0.07)
...         self.a1 = nn.ReLU()
...         self.l2 = nn.Linear(4, 4)
...         self.a2 = nn.ReLU()
...         self.l3 = nn.Linear(4, 1)
...         self.a3 = nn.Sigmoid()
...
...     def forward(self, x, training=False):
...         x = self.l1(x, training)
...         x = self.a1(x)
...         x = self.l2(x)
...         x = self.a2(x)
...         x = self.l3(x)
...         x = self.a3(x)
...         return x
...
...     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
...
>>> torch.manual_seed(1)
>>> model = MyNoisyModule()
>>> model
MyNoisyModule(
  (l1): NoisyLinear()
  (a1): ReLU()
  (l2): Linear(in_features=4, out_features=4, bias=True)
```

```
  (a2): ReLU()
  (l3): Linear(in_features=4, out_features=1, bias=True)
  (a3): Sigmoid()
)
```

3. Similarly, we will train the model as we did previously. At this time, to compute the prediction on the training batch, we use `pred = model(x_batch, True)[:, 0]` instead of `pred = model(x_batch)[:, 0]`:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> torch.manual_seed(1)
>>> loss_hist_train = [0] * num_epochs
>>> accuracy_hist_train = [0] * num_epochs
>>> loss_hist_valid = [0] * num_epochs
>>> accuracy_hist_valid = [0] * num_epochs
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch, True)[:, 0]
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train[epoch] += loss.item()
...         is_correct = (
...             (pred>=0.5).float() == y_batch
...         ).float()
...         accuracy_hist_train[epoch] += is_correct.mean()
...     loss_hist_train[epoch] /= n_train/batch_size
...     accuracy_hist_train[epoch] /= n_train/batch_size
...     pred = model(x_valid)[:, 0]
...     loss = loss_fn(pred, y_valid)
...     loss_hist_valid[epoch] = loss.item()
...     is_correct = ((pred>=0.5).float() == y_valid).float()
...     accuracy_hist_valid[epoch] += is_correct.mean()
```

4. After the model is trained, we can plot the losses, accuracies, and the decision boundary:

```
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(loss_hist_train, lw=4)
>>> plt.plot(loss_hist_valid, lw=4)
```

```
>>> plt.legend(['Train loss', 'Validation loss'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(accuracy_hist_train, lw=4)
>>> plt.plot(accuracy_hist_valid, lw=4)
>>> plt.legend(['Train acc.', 'Validation acc.'], fontsize=15)
>>> ax.set_xlabel('Epochs', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(
...     X=x_valid.numpy(),
...     y=y_valid.numpy().astype(np.integer),
...     clf=model
... )
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

5. The resulting figure will be as follows:

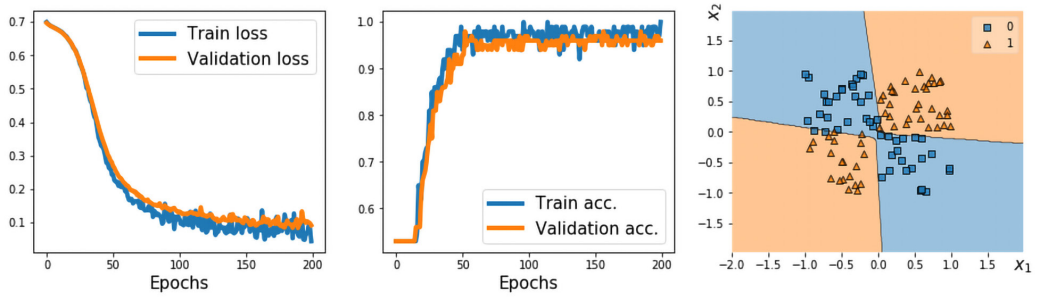


Figure 13.6: Results using NoisyLinear as the first hidden layer

Here, our goal was to learn how to define a new custom layer subclassed from `nn.Module` and to use it as we would use any other standard `torch.nn` layer. Although, with this particular example, `NoisyLinear` did not help to improve the performance, please keep in mind that our objective was to mainly learn how to write a customized layer from scratch. In general, writing a new customized layer can be useful in other applications, for example, if you develop a new algorithm that depends on a new layer beyond the existing ones.

Project one – predicting the fuel efficiency of a car

So far, in this chapter, we have mostly focused on the `torch.nn` module. We used `nn.Sequential` to construct the models for simplicity. Then, we made model building more flexible with `nn.Module` and implemented feedforward NNs, to which we added customized layers. In this section, we will work on a real-world project of predicting the fuel efficiency of a car in miles per gallon (MPG). We will cover the underlying steps in machine learning tasks, such as data preprocessing, feature engineering, training, prediction (inference), and evaluation.

Working with feature columns

In machine learning and deep learning applications, we can encounter various different types of features: continuous, unordered categorical (nominal), and ordered categorical (ordinal). You will recall that in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, we covered different types of features and learned how to handle each type. Note that while numeric data can be either continuous or discrete, in the context of machine learning with PyTorch, "numeric" data specifically refers to continuous data of floating point type.

Sometimes, feature sets are comprised of a mixture of different feature types. For example, consider a scenario with a set of seven different features, as shown in *Figure 13.7*:

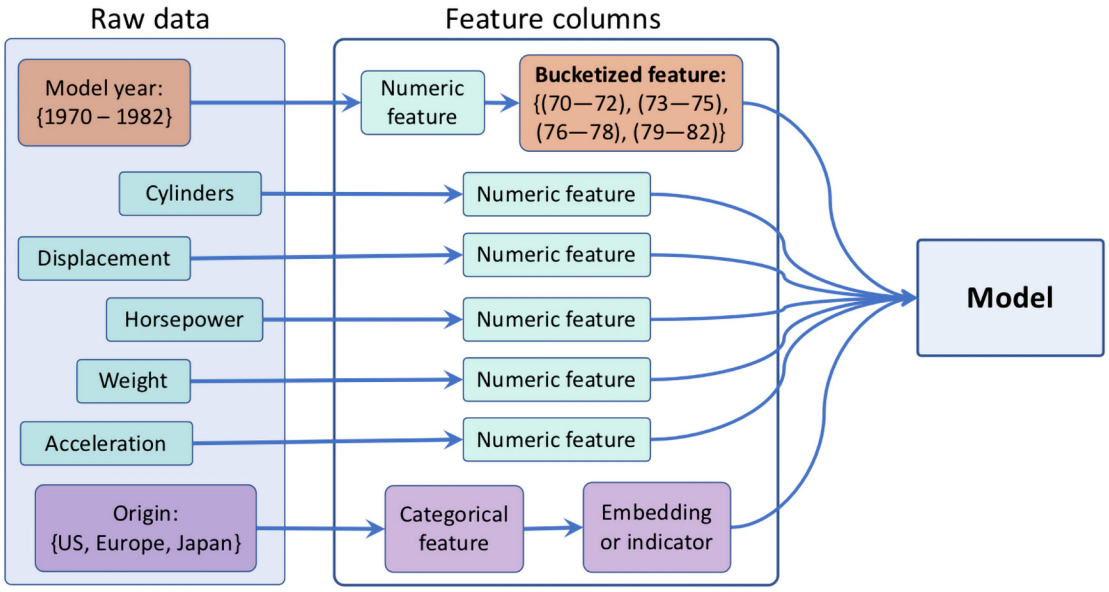


Figure 13.7: Auto MPG data structure

The features shown in the figure (model year, cylinders, displacement, horsepower, weight, acceleration, and origin) were obtained from the Auto MPG dataset, which is a common machine learning benchmark dataset for predicting the fuel efficiency of a car in MPG. The full dataset and its description are available from UCI's machine learning repository at `https://archive.ics.uci.edu/ml/datasets/auto+mpg`.

We are going to treat five features from the Auto MPG dataset (number of cylinders, displacement, horsepower, weight, and acceleration) as "numeric" (here, continuous) features. The model year can be regarded as an ordered categorical (ordinal) feature. Lastly, the manufacturing origin can be regarded as an unordered categorical (nominal) feature with three possible discrete values, 1, 2, and 3, which correspond to the US, Europe, and Japan, respectively.

Let's first load the data and apply the necessary preprocessing steps, including dropping the incomplete rows, partitioning the dataset into training and test datasets, as well as standardizing the continuous features:

```
>>> import pandas as pd
>>> url = 'http://archive.ics.uci.edu/ml/' \
...       'machine-learning-databases/auto-mpg/auto-mpg.data'
>>> column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower',
...                 'Weight', 'Acceleration', 'Model Year', 'Origin']
>>> df = pd.read_csv(url, names=column_names,
...                  na_values = "?", comment='\t',
...                  sep=" ", skipinitialspace=True)
>>>
>>> ## drop the NA rows
>>> df = df.dropna()
>>> df = df.reset_index(drop=True)
>>>
>>> ## train/test splits:
>>> import sklearn
>>> import sklearn.model_selection
>>> df_train, df_test = sklearn.model_selection.train_test_split(
...     df, train_size=0.8, random_state=1
... )
>>> train_stats = df_train.describe().transpose()
>>>
>>> numeric_column_names = [
...     'Cylinders', 'Displacement',
...     'Horsepower', 'Weight',
...     'Acceleration'
... ]
```

```
>>> df_train_norm, df_test_norm = df_train.copy(), df_test.copy()
>>> for col_name in numeric_column_names:
...     mean = train_stats.loc[col_name, 'mean']
...     std  = train_stats.loc[col_name, 'std']
...     df_train_norm.loc[:, col_name] = \
...         (df_train_norm.loc[:, col_name] - mean)/std
...     df_test_norm.loc[:, col_name] = \
...         (df_test_norm.loc[:, col_name] - mean)/std
>>> df_train_norm.tail()
```

This results in the following:

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	ModelYear	Origin
203	28.0	-0.824303	-0.901020	-0.736562	-0.950031	0.255202	76	3
255	19.4	0.351127	0.413800	-0.340982	0.293190	0.548737	78	1
72	13.0	1.526556	1.144256	0.713897	1.339617	-0.625403	72	1
235	30.5	-0.824303	-0.891280	-1.053025	-1.072585	0.475353	77	1
37	14.0	1.526556	1.563051	1.636916	1.470420	-1.359240	71	1

Figure 13.8: Preprocessed Auto MG data

The pandas `DataFrame` that we created via the previous code snippet contains five columns with values of the type `float`. These columns will constitute the continuous features.

Next, let's group the rather fine-grained model year (`ModelYear`) information into buckets to simplify the learning task for the model that we are going to train later. Concretely, we are going to assign each car into one of four *year* buckets, as follows:

$$\text{bucket} = \begin{cases} 0 & \text{if year} < 73 \\ 1 & \text{if } 73 \leq \text{year} < 76 \\ 2 & \text{if } 76 \leq \text{year} < 79 \\ 3 & \text{if year} \geq 79 \end{cases}$$

Note that the chosen intervals were selected arbitrarily to illustrate the concepts of "bucketing." In order to group the cars into these buckets, we will first define three cut-off values: [73, 76, 79] for the model year feature. These cut-off values are used to specify half-closed intervals, for instance, (–∞, 73), [73, 76), [76, 79), and [76, ∞). Then, the original numeric features will be passed to the `torch.bucketize` function (`https://pytorch.org/docs/stable/generated/torch.bucketize.html`) to generate the indices of the buckets. The code is as follows:

```
>>> boundaries = torch.tensor([73, 76, 79])
>>> v = torch.tensor(df_train_norm['Model Year'].values)
>>> df_train_norm['Model Year Bucketed'] = torch.bucketize(
```

```
...     v, boundaries, right=True
... )
>>> v = torch.tensor(df_test_norm['Model Year'].values)
>>> df_test_norm['Model Year Bucketed'] = torch.bucketize(
...     v, boundaries, right=True
... )
>>> numeric_column_names.append('Model Year Bucketed')
```

We added this bucketized feature column to the Python list `numeric_column_names`.

Next, we will proceed with defining a list for the unordered categorical feature, `Origin`. In PyTorch, There are two ways to work with a categorical feature: using an embedding layer via `nn.Embedding` (`https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html`), or using one-hot-encoded vectors (also called *indicator*). In the encoding approach, for example, index 0 will be encoded as [1, 0, 0], index 1 will be encoded as [0, 1, 0], and so on. On the other hand, the embedding layer maps each index to a vector of random numbers of the type `float`, which can be trained. (You can think of the embedding layer as a more efficient implementation of a one-hot encoding multiplied with a trainable weight matrix.)

When the number of categories is large, using the embedding layer with fewer dimensions than the number of categories can improve the performance.

In the following code snippet, we will use the one-hot-encoding approach on the categorical feature in order to convert it into the dense format:

```
>>> from torch.nn.functional import one_hot
>>> total_origin = len(set(df_train_norm['Origin']))
>>> origin_encoded = one_hot(torch.from_numpy(
...     df_train_norm['Origin'].values) % total_origin)
>>> x_train_numeric = torch.tensor(
...     df_train_norm[numeric_column_names].values)
>>> x_train = torch.cat([x_train_numeric, origin_encoded], 1).float()
>>> origin_encoded = one_hot(torch.from_numpy(
...     df_test_norm['Origin'].values) % total_origin)
>>> x_test_numeric = torch.tensor(
...     df_test_norm[numeric_column_names].values)
>>> x_test = torch.cat([x_test_numeric, origin_encoded], 1).float()
```

After encoding the categorical feature into a three-dimensional dense feature, we concatenated it with the numeric features we processed in the previous step. Finally, we will create the label tensors from the ground truth MPG values as follows:

```
>>> y_train = torch.tensor(df_train_norm['MPG'].values).float()
>>> y_test = torch.tensor(df_test_norm['MPG'].values).float()
```

In this section, we have covered the most common approaches for preprocessing and creating features in PyTorch.

Training a DNN regression model

Now, after constructing the mandatory features and labels, we will create a data loader that uses a batch size of 8 for the train data:

```
>>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 8
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Next, we will build a model with two fully connected layers where one has 8 hidden units and another has 4:

```
>>> hidden_units = [8, 4]
>>> input_size = x_train.shape[1]
>>> all_layers = []
>>> for hidden_unit in hidden_units:
...     layer = nn.Linear(input_size, hidden_unit)
...     all_layers.append(layer)
...     all_layers.append(nn.ReLU())
...     input_size = hidden_unit
>>> all_layers.append(nn.Linear(hidden_units[-1], 1))
>>> model = nn.Sequential(*all_layers)
>>> model
Sequential(
  (0): Linear(in_features=9, out_features=8, bias=True)
  (1): ReLU()
  (2): Linear(in_features=8, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
)
```

After defining the model, we will define the MSE loss function for regression and use stochastic gradient descent for optimization:

```
>>> loss_fn = nn.MSELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Now we will train the model for 200 epochs and display the train loss for every 20 epochs:

```
>>> torch.manual_seed(1)
>>> num_epochs = 200
>>> log_epochs = 20
```

```
>>> for epoch in range(num_epochs):
...     loss_hist_train = 0
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)[:, 0]
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train += loss.item()
...     if epoch % log_epochs==0:
...         print(f'Epoch {epoch}  Loss '
...               f'{loss_hist_train/len(train_dl):.4f}')

Epoch 0  Loss 536.1047
Epoch 20  Loss 8.4361
Epoch 40  Loss 7.8695
Epoch 60  Loss 7.1891
Epoch 80  Loss 6.7062
Epoch 100  Loss 6.7599
Epoch 120  Loss 6.3124
Epoch 140  Loss 6.6864
Epoch 160  Loss 6.7648
Epoch 180  Loss 6.2156
```

After 200 epochs, the train loss was around 5. We can now evaluate the regression performance of the trained model on the test dataset. To predict the target values on new data points, we can feed their features to the model:

```
>>> with torch.no_grad():
...     pred = model(x_test.float())[:, 0]
...     loss = loss_fn(pred, y_test)
...     print(f'Test MSE: {loss.item():.4f}')
...     print(f'Test MAE: {nn.L1Loss()(pred, y_test).item():.4f}')
Test MSE: 9.6130
Test MAE: 2.1211
```

The MSE on the test set is 9.6, and the **mean absolute error** (**MAE**) is 2.1. After this regression project, we will work on a classification project in the next section.

Project two – classifying MNIST handwritten digits

For this classification project, we are going to categorize MNIST handwritten digits. In the previous section, we covered the four essential steps for machine learning in PyTorch in detail, which we will need to repeat in this section.

You will recall that in *Chapter 12* you learned the way of loading available datasets from the `torchvision` module. First, we are going to load the MNIST dataset using the `torchvision` module.

1. The setup step includes loading the dataset and specifying hyperparameters (the size of the train set and test set, and the size of mini-batches):

```
>>> import torchvision
>>> from torchvision import transforms
>>> image_path = './'
>>> transform = transforms.Compose([
...     transforms.ToTensor()
... ])
>>> mnist_train_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=True,
...     transform=transform, download=False
... )
>>> mnist_test_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=False,
...     transform=transform, download=False
... )
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(mnist_train_dataset,
...                       batch_size, shuffle=True)
```

   Here, we constructed a data loader with batches of 64 samples. Next, we will preprocess the loaded datasets.

2. We preprocess the input features and the labels. The features in this project are the pixels of the images we read from **Step 1**. We defined a custom transformation using `torchvision.transforms.Compose`. In this simple case, our transformation consisted only of one method, `ToTensor()`. The `ToTensor()` method converts the pixel features into a floating type tensor and also normalizes the pixels from the [0, 255] to [0, 1] range. In *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*, we will see some additional data transformation methods when we work with more complex image datasets. The labels are integers from 0 to 9 representing ten digits. Hence, we don't need to do any scaling or further conversion. Note that we can access the raw pixels using the `data` attribute, and don't forget to scale them to the range [0, 1].

   We will construct the model in the next step once the data is preprocessed.

3. Construct the NN model:

```
>>> hidden_units = [32, 16]
>>> image_size = mnist_train_dataset[0][0].shape
>>> input_size = image_size[0] * image_size[1] * image_size[2]
>>> all_layers = [nn.Flatten()]
>>> for hidden_unit in hidden_units:
...     layer = nn.Linear(input_size, hidden_unit)
...     all_layers.append(layer)
...     all_layers.append(nn.ReLU())
...     input_size = hidden_unit
>>> all_layers.append(nn.Linear(hidden_units[-1], 10))
>>> model = nn.Sequential(*all_layers)
>>> model
Sequential(
  (0): Flatten(start_dim=1, end_dim=-1)
  (1): Linear(in_features=784, out_features=32, bias=True)
  (2): ReLU()
  (3): Linear(in_features=32, out_features=16, bias=True)
  (4): ReLU()
  (5): Linear(in_features=16, out_features=10, bias=True)
)
```

Note that the model starts with a flatten layer that flattens an input image into a one-dimensional tensor. This is because the input images are in the shape of [1, 28, 28]. The model has two hidden layers, with 32 and 16 units respectively. And it ends with an output layer of ten units representing ten classes, activated by a softmax function. In the next step, we will train the model on the train set and evaluate it on the test set.

4. Use the model for training, evaluation, and prediction:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
>>> torch.manual_seed(1)
>>> num_epochs = 20
>>> for epoch in range(num_epochs):
...     accuracy_hist_train = 0
...     for x_batch, y_batch in train_dl:
```

```
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         is_correct = (
...             torch.argmax(pred, dim=1) == y_batch
...         ).float()
...         accuracy_hist_train += is_correct.sum()
...     accuracy_hist_train /= len(train_dl.dataset)
...     print(f'Epoch {epoch}  Accuracy '
...           f'{accuracy_hist_train:.4f}')
Epoch 0  Accuracy 0.8531
...
Epoch 9  Accuracy 0.9691
...
Epoch 19  Accuracy 0.9813
```

We used the cross-entropy loss function for multiclass classification and the Adam optimizer for gradient descent. We will talk about the Adam optimizer in *Chapter 14*. We trained the model for 20 epochs and displayed the train accuracy for every epoch. The trained model reached an accuracy of 96.3 percent on the training set and we will evaluate it on the testing set:

```
>>> pred = model(mnist_test_dataset.data / 255.)
>>> is_correct = (
...     torch.argmax(pred, dim=1) ==
...     mnist_test_dataset.targets
... ).float()
>>> print(f'Test accuracy: {is_correct.mean():.4f}')
Test accuracy: 0.9645
```

The test accuracy is 95.6 percent. You have learned how to solve a classification problem using PyTorch.

Higher-level PyTorch APIs: a short introduction to PyTorch-Lightning

In recent years, the PyTorch community developed several different libraries and APIs on top of PyTorch. Notable examples include fastai (`https://docs.fast.ai/`), Catalyst (`https://github.com/catalyst-team/catalyst`), PyTorch Lightning (`https://www.pytorchlightning.ai`), (`https://lightning-flash.readthedocs.io/en/latest/quickstart.html`), and PyTorch-Ignite (`https://github.com/pytorch/ignite`).

In this section, we will explore PyTorch Lightning (Lightning for short), which is a widely used PyTorch library that makes training deep neural networks simpler by removing much of the boilerplate code. However, while Lightning's focus lies in simplicity and flexibility, it also allows us to use many advanced features such as multi-GPU support and fast low-precision training, which you can learn about in the official documentation at `https://pytorch-lightning.rtfd.io/en/latest/`.

There is also a bonus introduction to PyTorch-Ignite at `https://github.com/rasbt/machine-learning-book/blob/main/ch13/ch13_part4_ignite.ipynb`.

In an earlier section, *Project two – classifying MNIST handwritten digits*, we implemented a multilayer perceptron for classifying handwritten digits in the MNIST dataset. In the next subsections, we will reimplement this classifier using Lightning.

Installing PyTorch Lightning

Lightning can be installed via pip or conda, depending on your preference. For instance, the command for installing Lightning via pip is as follows:

```
pip install pytorch-lightning
```

The following is the command for installing Lightning via conda:

```
conda install pytorch-lightning -c conda-forge
```

The code in the following subsections is based on PyTorch Lightning version 1.5, which you can install by replacing `pytorch-lightning` with `pytorch-lightning==1.5` in these commands.

Setting up the PyTorch Lightning model

We start by implementing the model, which we will train in the next subsections. Defining a model for Lightning is relatively straightforward as it is based on regular Python and PyTorch code. All that is required to implement a Lightning model is to use `LightningModule` instead of the regular PyTorch module. To take advantage of PyTorch's convenience functions, such as the trainer API and automatic logging, we just define a few specifically named methods, which we will see in the following code:

```
import pytorch_lightning as pl
import torch
import torch.nn as nn

from torchmetrics import Accuracy

class MultiLayerPerceptron(pl.LightningModule):
    def __init__(self, image_shape=(1, 28, 28), hidden_units=(32, 16)):
        super().__init__()
```

```
        # new PL attributes:
        self.train_acc = Accuracy()
        self.valid_acc = Accuracy()
        self.test_acc = Accuracy()

        # Model similar to previous section:
        input_size = image_shape[0] * image_shape[1] * image_shape[2]
        all_layers = [nn.Flatten()]
        for hidden_unit in hidden_units:
            layer = nn.Linear(input_size, hidden_unit)
            all_layers.append(layer)
            all_layers.append(nn.ReLU())
            input_size = hidden_unit

        all_layers.append(nn.Linear(hidden_units[-1], 10))
        self.model = nn.Sequential(*all_layers)

    def forward(self, x):
        x = self.model(x)
        return x

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.train_acc.update(preds, y)
        self.log("train_loss", loss, prog_bar=True)
        return loss

    def training_epoch_end(self, outs):
        self.log("train_acc", self.train_acc.compute())

    def validation_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.valid_acc.update(preds, y)
        self.log("valid_loss", loss, prog_bar=True)
        self.log("valid_acc", self.valid_acc.compute(), prog_bar=True)
```

```
        return loss

    def test_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.test_acc.update(preds, y)
        self.log("test_loss", loss, prog_bar=True)
        self.log("test_acc", self.test_acc.compute(), prog_bar=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
        return optimizer
```

Let's now discuss the different methods one by one. As you can see, the `__init__` constructor contains the same model code that we used in a previous subsection. What's new is that we added the accuracy attributes such as `self.train_acc = Accuracy()`. These will allow us to track the accuracy during training. `Accuracy` was imported from the `torchmetrics` module, which should be automatically installed with Lightning. If you cannot import `torchmetrics`, you can try to install it via `pip install torchmetrics`. More information can be found at `https://torchmetrics.readthedocs.io/en/latest/pages/quickstart.html`.

The `forward` method implements a simple forward pass that returns the logits (outputs of the last fully connected layer of our network before the softmax layer) when we call our model on the input data. The logits, computed via the `forward` method by calling `self(x)`, are used for the training, validation, and test steps, which we'll describe next.

The `training_step`, `training_epoch_end`, `validation_step`, `test_step`, and `configure_optimizers` methods are methods that are specifically recognized by Lightning. For instance, `training_step` defines a single forward pass during training, where we also keep track of the accuracy and loss so that we can analyze these later. Note that we compute the accuracy via `self.train_acc.update(preds, y)` but don't log it yet. The `training_step` method is executed on each individual batch during training, and via the `training_epoch_end` method, which is executed at the end of each training epoch, we compute the training set accuracy from the accuracy values we accumulated via training.

The `validation_step` and `test_step` methods define, analogous to the `training_step` method, how the validation and test evaluation process should be computed. Similar to `training_step`, each `validation_step` and `test_step` receives a single batch, which is why we log the accuracy via respective accuracy attributes derived from `Accuracy` of `torchmetric`. However, note that `validation_step` is only called in certain intervals, for example, after each training epoch. This is why we log the validation accuracy inside the validation step, whereas with the training accuracy, we log it after each training epoch, otherwise, the accuracy plot that we inspect later will look too noisy.

Finally, via the `configure_optimizers` method, we specify the optimizer used for training. The following two subsections will discuss how we can set up the dataset and how we can train the model.

Setting up the data loaders for Lightning

There are three main ways in which we can prepare the dataset for Lightning. We can:

- Make the dataset part of the model
- Set up the data loaders as usual and feed them to the `fit` method of a Lightning Trainer—the Trainer is introduced in the next subsection
- Create a `LightningDataModule`

Here, we are going to use a `LightningDataModule`, which is the most organized approach. The `LightningDataModule` consists of five main methods, as we can see in the following:

```
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torchvision.datasets import MNIST
from torchvision import transforms

class MnistDataModule(pl.LightningDataModule):
    def __init__(self, data_path='./'):
        super().__init__()
        self.data_path = data_path
        self.transform = transforms.Compose([transforms.ToTensor()])

    def prepare_data(self):
        MNIST(root=self.data_path, download=True)

    def setup(self, stage=None):
        # stage is either 'fit', 'validate', 'test', or 'predict'
        # here note relevant
        mnist_all = MNIST(
            root=self.data_path,
            train=True,
            transform=self.transform,
            download=False
        )

        self.train, self.val = random_split(
            mnist_all, [55000, 5000], generator=torch.Generator().manual_
seed(1)
        )
```

```
        self.test = MNIST(
            root=self.data_path,
            train=False,
            transform=self.transform,
            download=False
        )

    def train_dataloader(self):
        return DataLoader(self.train, batch_size=64, num_workers=4)

    def val_dataloader(self):
        return DataLoader(self.val, batch_size=64, num_workers=4)

    def test_dataloader(self):
        return DataLoader(self.test, batch_size=64, num_workers=4)
```

In the `prepare_data` method, we define general steps, such as downloading the dataset. In the `setup` method, we define the datasets used for training, validation, and testing. Note that MNIST does not have a dedicated validation split, which is why we use the `random_split` function to divide the 60,000-example training set into 55,000 examples for training and 5,000 examples for validation.

The data loader methods are self-explanatory and define how the respective datasets are loaded. Now, we can initialize the data module and use it for training, validation, and testing in the next subsections:

```
torch.manual_seed(1)
mnist_dm = MnistDataModule()
```

Training the model using the PyTorch Lightning Trainer class

Now we can reap the rewards from setting up the model with the specifically named methods, as well as the Lightning data module. Lightning implements a `Trainer` class that makes the training model super convenient by taking care of all the intermediate steps, such as calling `zero_grad()`, `backward()`, and `optimizer.step()` for us. Also, as a bonus, it lets us easily specify one or more GPUs to use (if available):

```
mnistclassifier = MultiLayerPerceptron()

if torch.cuda.is_available(): # if you have GPUs
    trainer = pl.Trainer(max_epochs=10, gpus=1)
else:
    trainer = pl.Trainer(max_epochs=10)

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

Via the preceding code, we train our multilayer perceptron for 10 epochs. During training, we see a handy progress bar that keeps track of the epoch and core metrics such as the training and validation losses:

```
Epoch 9: 100% 939/939 [00:07<00:00, 130.42it/s, loss=0.1, v_num=0, train_
loss=0.260, valid_loss=0.166, valid_acc=0.949]
```

After the training has finished, we can also inspect the metrics we logged in more detail, as we will see in the next subsection.

Evaluating the model using TensorBoard

In the previous section, we experienced the convenience of the `Trainer` class. Another nice feature of Lightning is its logging capabilities. Recall that we specified several `self.log` steps in our Lightning model earlier. After, and even during training, we can visualize them in TensorBoard. (Note that Lightning supports other loggers as well; for more information, please see the official documentation at `https://pytorch-lightning.readthedocs.io/en/latest/common/loggers.html`.)

Installing TensorBoard

TensorBoard can be installed via pip or conda, depending on your preference. For instance, the command for installing TensorBoard via pip is as follows:

```
pip install tensorboard
```

The following is the command for installing Lightning via conda:

```
conda install tensorboard -c conda-forge
```

The code in the following subsection is based on TensorBoard version 2.4, which you can install by replacing `tensorboard` with `tensorboard==2.4` in these commands.

By default, Lightning tracks the training in a subfolder named `lightning_logs`. To visualize the training runs, you can execute the following code in the command-line terminal, which will open TensorBoard in your browser:

```
tensorboard --logdir lightning_logs/
```

Alternatively, if you are running the code in a Jupyter notebook, you can add the following code to a Jupyter notebook cell to show the TensorBoard dashboard in the notebook directly:

```
%load_ext tensorboard
%tensorboard --logdir lightning_logs/
```

Figure 13.9 shows the TensorBoard dashboard with the logged training and validation accuracy. Note that there is a `version_0` toggle shown in the lower-left corner. If you run the training code multiple times, Lightning will track them as separate subfolders: `version_0`, `version_1`, `version_2`, and so forth:

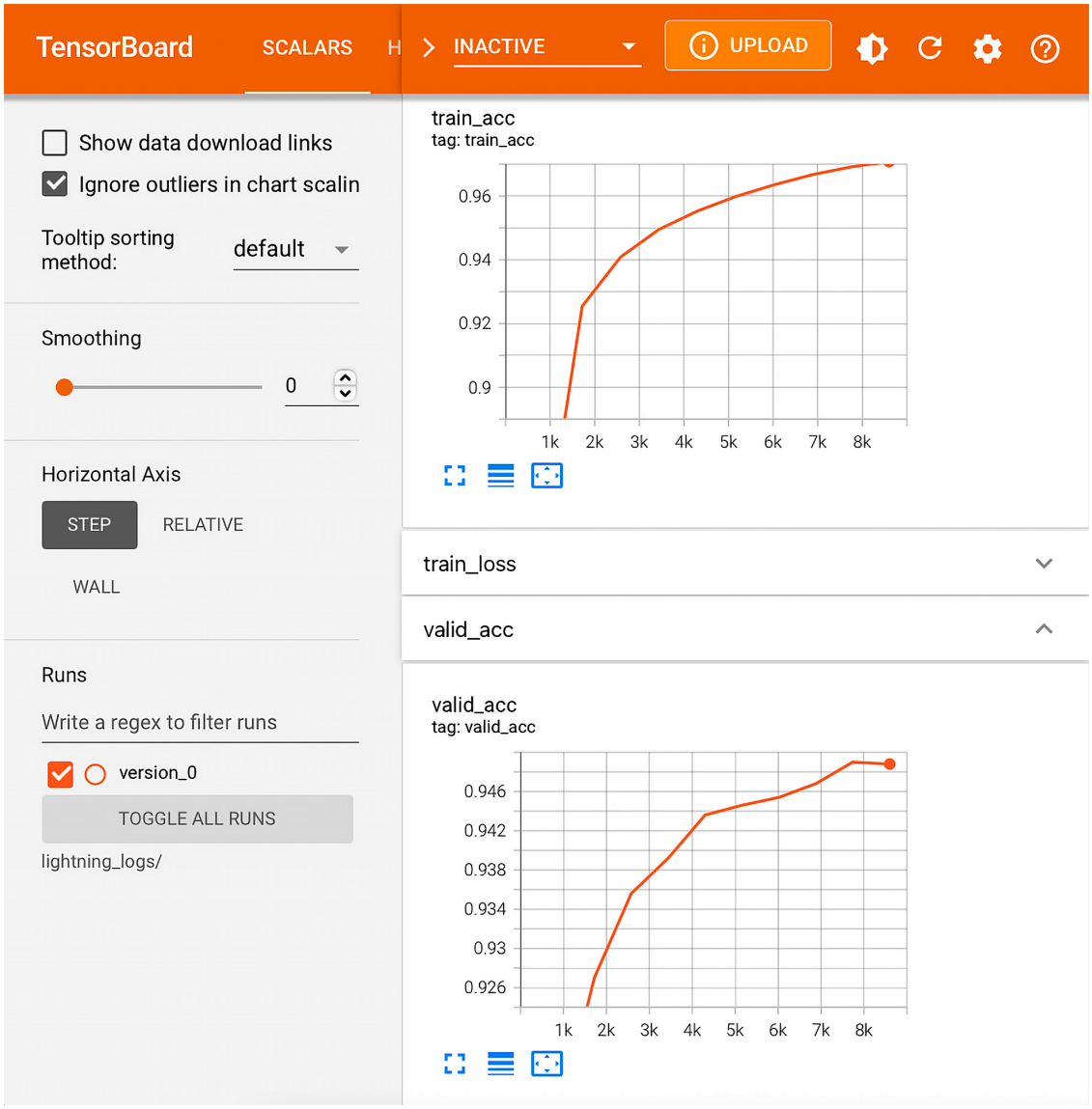


Figure 13.9: TensorBoard dashboard

By looking at the training and validation accuracies in *Figure 13.9*, we can hypothesize that training the model for a few additional epochs can improve performance.

Lightning allows us to load a trained model and train it for additional epochs conveniently. As mentioned previously, Lightning tracks the individual training runs via subfolders. In *Figure 13.10*, we see the contents of the `version_0` subfolder, which contains log files and a model checkpoint for reloading the model:

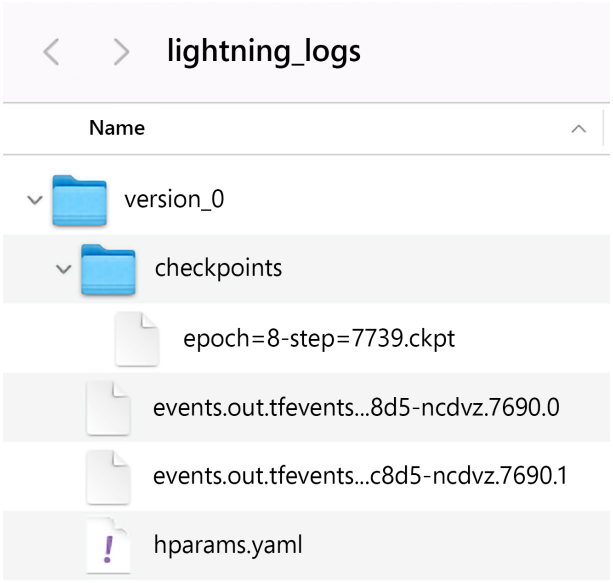


Figure 13.10: PyTorch Lightning log files

For instance, we can use the following code to load the latest model checkpoint from this folder and train the model via `fit`:

```
if torch.cuda.is_available(): # if you have GPUs
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
logs/version_0/checkpoints/epoch=8-step=7739.ckpt', gpus=1)
else:
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
logs/version_0/checkpoints/epoch=8-step=7739.ckpt')

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

Here, we set `max_epochs` to `15`, which trained the model for 5 additional epochs (previously, we trained it for 10 epochs).

Now, let's take a look at the TensorBoard dashboard in *Figure 13.11* and see whether training the model for a few additional epochs was worthwhile:

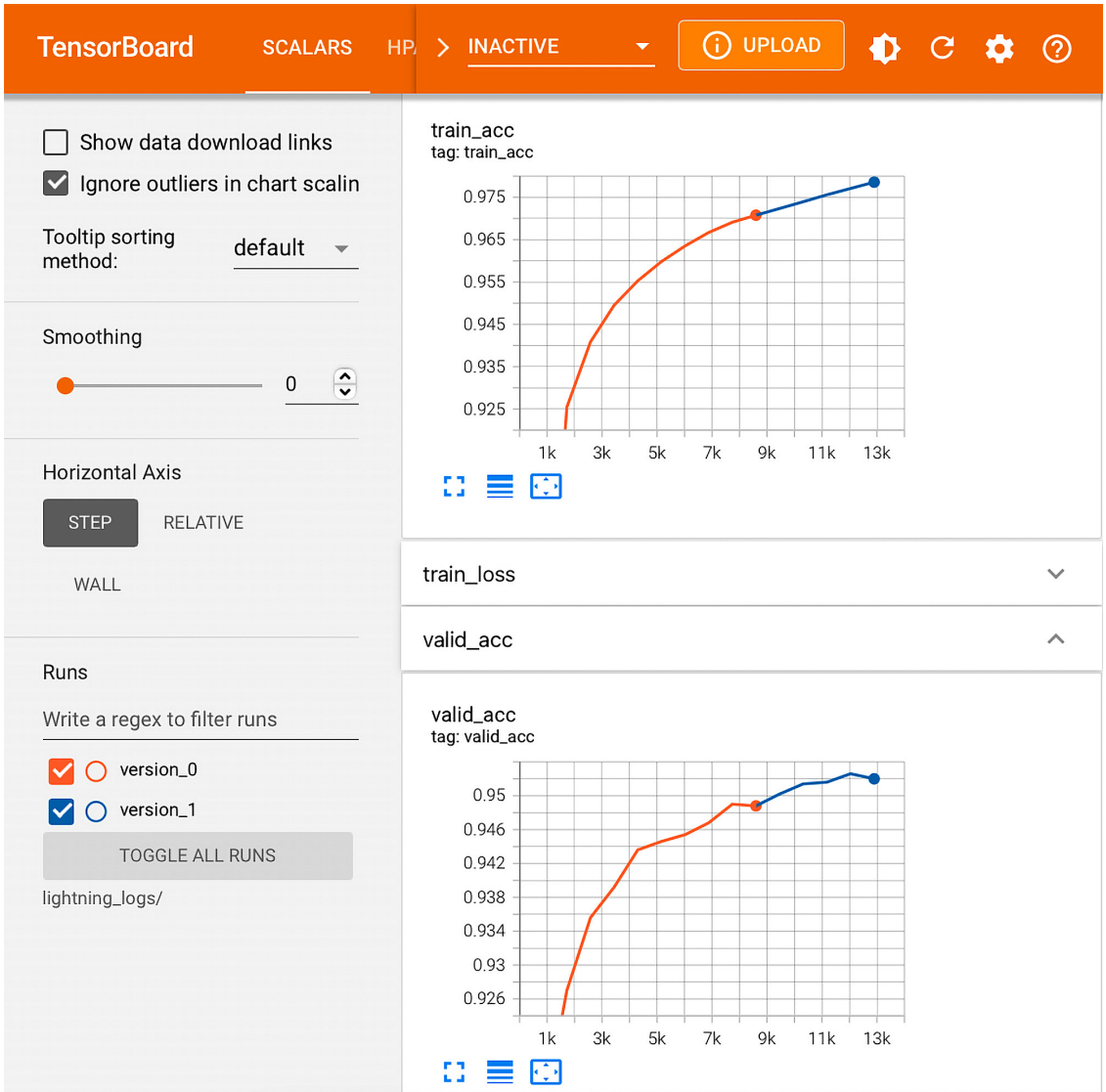


Figure 13.11: TensorBoard dashboard after training for five more epochs

As we can see in *Figure 13.11*, TensorBoard allows us to show the results from the additional training epochs (`version_1`) next to the previous ones (`version_0`), which is very convenient. Indeed, we can see that training for five more epochs improved the validation accuracy. At this point, we may decide to train the model for more epochs, which we leave as an exercise to you.

Once we are finished with training, we can evaluate the model on the test set using the following code:

```
trainer.test(model=mnistclassifier, datamodule=mnist_dm)
```

The resulting test set performance, after training for 15 epochs in total, is approximately 95 percent:

```
[{'test_loss': 0.14912301301956177, 'test_acc': 0.9499600529670715}]
```

Note that PyTorch Lightning also saves the model automatically for us. If you want to reuse the model later, you can conveniently load it via the following code:

```
model = MultiLayerPerceptron.load_from_checkpoint("path/to/checkpoint.ckpt")
```

Learn more about PyTorch Lightning

To learn more about Lightning, please visit the official website, which contains tutorials and examples, at `https://pytorch-lightning.readthedocs.io`.

Lightning also has an active community on Slack that welcomes new users and contributors. To find out more, please visit the official Lightning website at `https://www.pytorchlightning.ai`.

Summary

In this chapter, we covered PyTorch's most essential and useful features. We started by discussing PyTorch's dynamic computation graph, which makes implementing computations very convenient. We also covered the semantics of defining PyTorch tensor objects as model parameters.

After we considered the concept of computing partial derivatives and gradients of arbitrary functions, we covered the `torch.nn` module in more detail. It provides us with a user-friendly interface for building more complex deep NN models. Finally, we concluded this chapter by solving a regression and classification problem using what we have discussed so far.

Now that we have covered the core mechanics of PyTorch, the next chapter will introduce the concept behind **convolutional neural network** (**CNN**) architectures for deep learning. CNNs are powerful models and have shown great performance in the field of computer vision.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:

`https://packt.link/MLwPyTorch`

14

Classifying Images with Deep Convolutional Neural Networks

In the previous chapter, we looked in depth at different aspects of the PyTorch neural network and automatic differentiation modules, you became familiar with tensors and decorating functions, and you learned how to work with `torch.nn`. In this chapter, you will now learn about **convolutional neural networks** (**CNNs**) for image classification. We will start by discussing the basic building blocks of CNNs, using a bottom-up approach. Then, we will take a deeper dive into the CNN architecture and explore how to implement CNNs in PyTorch. In this chapter, we will cover the following topics:

- Convolution operations in one and two dimensions
- The building blocks of CNN architectures
- Implementing deep CNNs in PyTorch
- Data augmentation techniques for improving the generalization performance
- Implementing a facial CNN classifier for recognizing if someone is smiling or not

The building blocks of CNNs

CNNs are a family of models that were originally inspired by how the visual cortex of the human brain works when recognizing objects. The development of CNNs goes back to the 1990s, when Yann LeCun and his colleagues proposed a novel NN architecture for classifying handwritten digits from images (*Handwritten Digit Recognition with a Back-Propagation Network* by *Y. LeCun*, and colleagues, 1989, published at the *Neural Information Processing Systems (NeurIPS)* conference).

The human visual cortex

The original discovery of how the visual cortex of our brain functions was made by David H. Hubel and Torsten Wiesel in 1959, when they inserted a microelectrode into the primary visual cortex of an anesthetized cat. They observed that neurons respond differently after projecting different patterns of light in front of the cat. This eventually led to the discovery of the different layers of the visual cortex. While the primary layer mainly detects edges and straight lines, higher-order layers focus more on extracting complex shapes and patterns.

Due to the outstanding performance of CNNs for image classification tasks, this particular type of feedforward NN gained a lot of attention and led to tremendous improvements in machine learning for computer vision. Several years later, in 2019, Yann LeCun received the Turing award (the most prestigious award in computer science) for his contributions to the field of **artificial intelligence** (**AI**), along with two other researchers, Yoshua Bengio and Geoffrey Hinton, whose names you encountered in previous chapters.

In the following sections, we will discuss the broader concepts of CNNs and why convolutional architectures are often described as "feature extraction layers." Then, we will delve into the theoretical definition of the type of convolution operation that is commonly used in CNNs and walk through examples of computing convolutions in one and two dimensions.

Understanding CNNs and feature hierarchies

Successfully extracting **salient** (**relevant**) **features** is key to the performance of any machine learning algorithm, and traditional machine learning models rely on input features that may come from a domain expert or are based on computational feature extraction techniques.

Certain types of NNs, such as CNNs, can automatically learn the features from raw data that are most useful for a particular task. For this reason, it's common to consider CNN layers as feature extractors: the early layers (those right after the input layer) extract **low-level features** from raw data, and the later layers (often **fully connected layers**, as in a **multilayer perceptron** (**MLP**)) use these features to predict a continuous target value or class label.

Certain types of multilayer NNs, and in particular, deep CNNs, construct a so-called **feature hierarchy** by combining the low-level features in a layer-wise fashion to form high-level features. For example, if we're dealing with images, then low-level features, such as edges and blobs, are extracted from the earlier layers, which are combined to form high-level features. These high-level features can form more complex shapes, such as the general contours of objects like buildings, cats, or dogs.

As you can see in *Figure 14.1*, a CNN computes **feature maps** from an input image, where each element comes from a local patch of pixels in the input image:

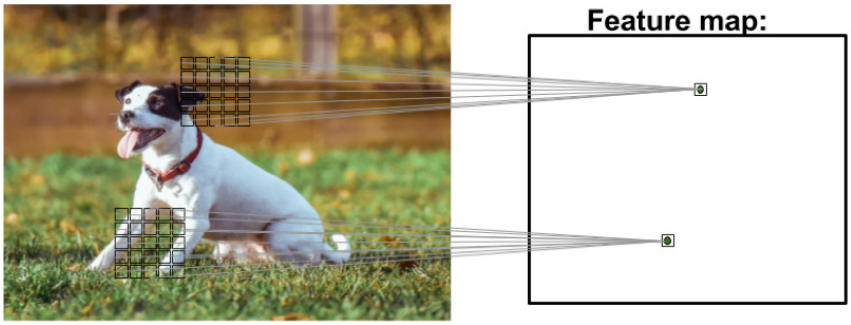


Figure 14.1: Creating feature maps from an image (photo by Alexander Dummer on Unsplash)

This local patch of pixels is referred to as the **local receptive field**. CNNs will usually perform very well on image-related tasks, and that's largely due to two important ideas:

- **Sparse connectivity**: A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image, as in the case of MLPs. You may find it useful to look back and compare how we implemented a fully connected network that connected to the whole image in *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*.)
- **Parameter sharing**: The same weights are used for different patches of the input image.

As a direct consequence of these two ideas, replacing a conventional, fully connected MLP with a convolution layer substantially decreases the number of weights (parameters) in the network, and we will see an improvement in the ability to capture *salient* features. In the context of image data, it makes sense to assume that nearby pixels are typically more relevant to each other than pixels that are far away from each other.

Typically, CNNs are composed of several **convolutional** and subsampling layers that are followed by one or more fully connected layers at the end. The fully connected layers are essentially an MLP, where every input unit, i, is connected to every output unit, j, with weight w_{ij} (which we covered in more detail in *Chapter 11*).

Please note that subsampling layers, commonly known as **pooling layers**, do not have any learnable parameters; for instance, there are no weights or bias units in pooling layers. However, both the convolutional and fully connected layers have weights and biases that are optimized during training.

In the following sections, we will study convolutional and pooling layers in more detail and see how they work. To understand how convolution operations work, let's start with a convolution in one dimension, which is sometimes used for working with certain types of sequence data, such as text. After discussing one-dimensional convolutions, we will work through the typical two-dimensional ones that are commonly applied to two-dimensional images.

Performing discrete convolutions

A **discrete convolution** (or simply **convolution**) is a fundamental operation in a CNN. Therefore, it's important to understand how this operation works. In this section, we will cover the mathematical definition and discuss some of the **naive** algorithms to compute convolutions of one-dimensional tensors (vectors) and two-dimensional tensors (matrices).

Please note that the formulas and descriptions in this section are solely for understanding how convolution operations in CNNs work. Indeed, much more efficient implementations of convolutional operations already exist in packages such as PyTorch, as you will see later in this chapter.

Mathematical notation

In this chapter, we will use subscript to denote the size of a multidimensional array (tensor); for example, $A_{n_1 \times n_2}$ is a two-dimensional array of size $n_1 \times n_2$. We use brackets, [ ], to denote the indexing of a multidimensional array. For example, $A[i, j]$ refers to the element at index i, j of matrix A. Furthermore, note that we use a special symbol, $*$, to denote the convolution operation between two vectors or matrices, which is not to be confused with the multiplication operator, *, in Python.

Discrete convolutions in one dimension

Let's start with some basic definitions and notations that we are going to use. A discrete convolution for two vectors, $\boldsymbol{x}$ and $\boldsymbol{w}$, is denoted by $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, in which vector $\boldsymbol{x}$ is our input (sometimes called **signal**) and $\boldsymbol{w}$ is called the **filter** or **kernel**. A discrete convolution is mathematically defined as follows:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]\, w[k]$$

As mentioned earlier, the brackets, [ ], are used to denote the indexing for vector elements. The index, i, runs through each element of the output vector, $\boldsymbol{y}$. There are two odd things in the preceding formula that we need to clarify: $-\infty$ to $+\infty$ indices and negative indexing for $\boldsymbol{x}$.

The fact that the sum runs through indices from −∞ to +∞ seems odd, mainly because in machine learning applications, we always deal with finite feature vectors. For example, if $\boldsymbol{x}$ has 10 features with indices 0, 1, 2, ..., 8, 9, then indices −∞: −1 and 10: +∞ are out of bounds for $\boldsymbol{x}$. Therefore, to correctly compute the summation shown in the preceding formula, it is assumed that $\boldsymbol{x}$ and $\boldsymbol{w}$ are filled with zeros. This will result in an output vector, $\boldsymbol{y}$, that also has infinite size, with lots of zeros as well. Since this is not useful in practical situations, $\boldsymbol{x}$ is padded only with a finite number of zeros.

This process is called **zero-padding** or simply **padding**. Here, the number of zeros padded on each side is denoted by p. An example padding of a one-dimensional vector, $\boldsymbol{x}$, is shown in *Figure 14.2*:

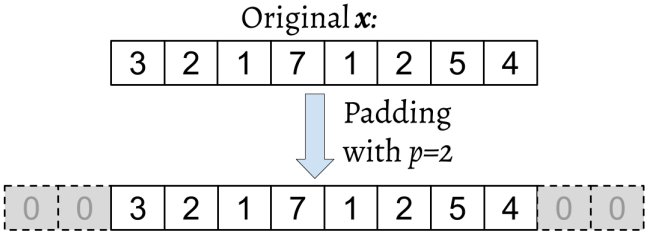


Figure 14.2: An example of padding

Let's assume that the original input, $\boldsymbol{x}$, and filter, $\boldsymbol{w}$, have n and m elements, respectively, where $m \leq n$. Therefore, the padded vector, x^p, has size $n + 2p$. The practical formula for computing a discrete convolution will change to the following:

$$\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w} \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i + m - k]\, w[k]$$

Now that we have solved the infinite index issue, the second issue is indexing $\boldsymbol{x}$ with $i + m - k$. The important point to notice here is that $\boldsymbol{x}$ and $\boldsymbol{w}$ are indexed in different directions in this summation. Computing the sum with one index going in the reverse direction is equivalent to computing the sum with both indices in the forward direction after flipping one of those vectors, $\boldsymbol{x}$ or $\boldsymbol{w}$, after they are padded. Then, we can simply compute their dot product. Let's assume we flip (rotate) the filter, $\boldsymbol{w}$, to get the rotated filter, $\boldsymbol{w}^r$. Then, the dot product, $x[i: i + m].\boldsymbol{w}^r$, is computed to get one element, $y[i]$, where $\boldsymbol{x}[i: i + m]$ is a patch of $\boldsymbol{x}$ with size m. This operation is repeated like in a sliding window approach to get all the output elements.

The following figure provides an example with $\boldsymbol{x}$ = [3 2 1 7 1 2 5 4] and $\boldsymbol{w} = \left[\frac{1}{2}\ \frac{3}{4}\ 1\ \frac{1}{4}\right]$ so that the first three output elements are computed:

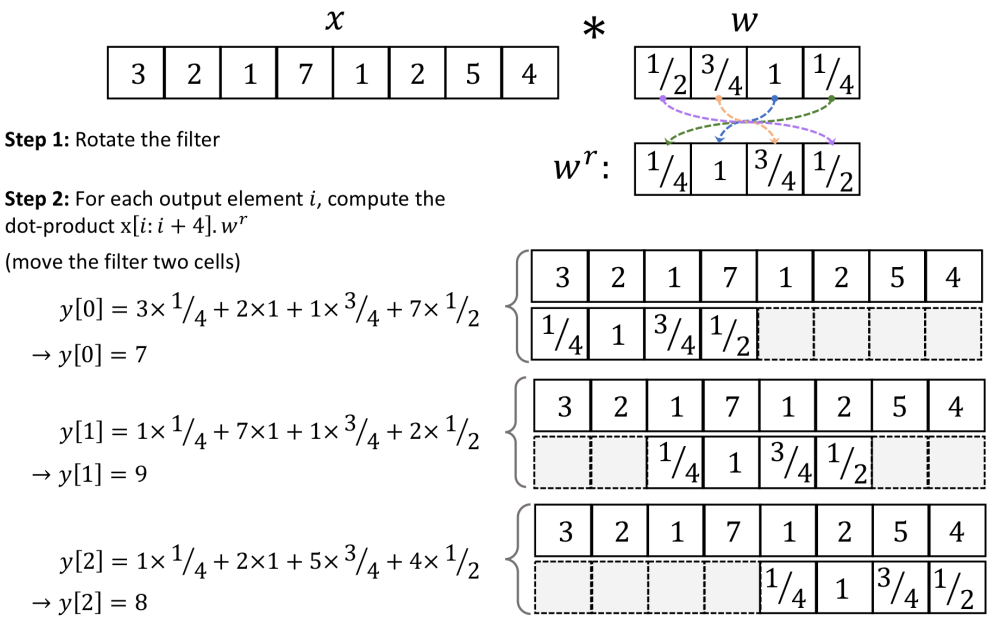


Figure 14.3: The steps for computing a discrete convolution

You can see in the preceding example that the padding size is zero ($p = 0$). Notice that the rotated filter, $\boldsymbol{w}^r$, is shifted by two cells each time we **shift**. This shift is another hyperparameter of a convolution, the **stride**, s. In this example, the stride is two, $s = 2$. Note that the stride has to be a positive number smaller than the size of the input vector. We will talk more about padding and strides in the next section.

Cross-correlation

Cross-correlation (or simply correlation) between an input vector and a filter is denoted by $\boldsymbol{y} = \boldsymbol{x} \star \boldsymbol{w}$ and is very much like a sibling of a convolution, with a small difference: in cross-correlation, the multiplication is performed in the same direction. Therefore, it is not a requirement to rotate the filter matrix, w, in each dimension. Mathematically, cross-correlation is defined as follows:

$$\boldsymbol{y} = \boldsymbol{x} \star \boldsymbol{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i+k]\, w[k]$$

The same rules for padding and stride may be applied to cross-correlation as well. Note that most deep learning frameworks (including PyTorch) implement cross-correlation but refer to it as convolution, which is a common convention in the deep learning field.

Padding inputs to control the size of the output feature maps

So far, we've only used zero-padding in convolutions to compute finite-sized output vectors. Technically, padding can be applied with any $p \geq 0$. Depending on the choice of p, boundary cells may be treated differently than the cells located in the middle of $\boldsymbol{x}$.

Now, consider an example where $n = 5$ and $m = 3$. Then, with $p = 0$, $x[0]$ is only used in computing one output element (for instance, $y[0]$), while $x[1]$ is used in the computation of two output elements (for instance, $y[0]$ and $y[1]$). So, you can see that this different treatment of elements of $\boldsymbol{x}$ can artificially put more emphasis on the middle element, $x[2]$, since it has appeared in most computations. We can avoid this issue if we choose $p = 2$, in which case, each element of $\boldsymbol{x}$ will be involved in computing three elements of $\boldsymbol{y}$.

Furthermore, the size of the output, $\boldsymbol{y}$, also depends on the choice of the padding strategy we use.

There are three modes of padding that are commonly used in practice: *full*, *same*, and *valid*.

In full mode, the padding parameter, p, is set to $p = m - 1$. Full padding increases the dimensions of the output; thus, it is rarely used in CNN architectures.

The same padding mode is usually used to ensure that the output vector has the same size as the input vector, $\boldsymbol{x}$. In this case, the padding parameter, p, is computed according to the filter size, along with the requirement that the input size and output size are the same.

Finally, computing a convolution in valid mode refers to the case where $p = 0$ (no padding).

Figure 14.4 illustrates the three different padding modes for a simple 5×5 pixel input with a kernel size of 3×3 and a stride of 1:

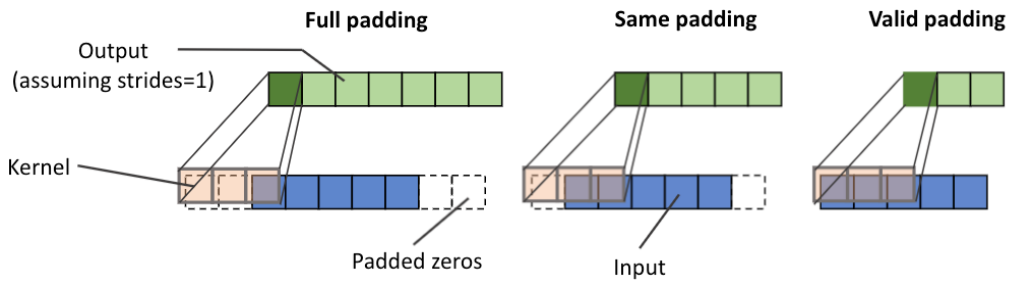


Figure 14.4: The three modes of padding

The most commonly used padding mode in CNNs is same padding. One of its advantages over the other padding modes is that same padding preserves the size of the vector—or the height and width of the input images when we are working on image-related tasks in computer vision—which makes designing a network architecture more convenient.

One big disadvantage of valid padding versus full and same padding is that the volume of the tensors will decrease substantially in NNs with many layers, which can be detrimental to the network's performance. In practice, you should preserve the spatial size using same padding for the convolutional layers and decrease the spatial size via pooling layers or convolutional layers with stride 2 instead, as described in *Striving for Simplicity: The All Convolutional Net* ICLR (workshop track), by *Jost Tobias Springenberg*, *Alexey Dosovitskiy*, and others, 2015 (`https://arxiv.org/abs/1412.6806`).

As for full padding, its size results in an output larger than the input size. Full padding is usually used in signal processing applications where it is important to minimize boundary effects. However, in a deep learning context, boundary effects are usually not an issue, so we rarely see full padding being used in practice.

Determining the size of the convolution output

The output size of a convolution is determined by the total number of times that we shift the filter, $\boldsymbol{w}$, along the input vector. Let's assume that the input vector is of size n and the filter is of size m. Then, the size of the output resulting from $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, with padding p and stride s, would be determined as follows:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

Here, $\lfloor \cdot \rfloor$ denotes the *floor* operation.

The floor operation

The floor operation returns the largest integer that is equal to or smaller than the input, for example:

$$\text{floor}(1.77) = \lfloor 1.77 \rfloor = 1$$

Consider the following two cases:

- Compute the output size for an input vector of size 10 with a convolution kernel of size 5, padding 2, and stride 1:

  $$n = 10, m = 5, \quad p = 2, \quad s = 1 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 5}{1} \right\rfloor + 1 = 10$$

  (Note that in this case, the output size turns out to be the same as the input; therefore, we can conclude this to be same padding mode.)

- How does the output size change for the same input vector when we have a kernel of size 3 and stride 2?

  $$n = 10, m = 3, \quad p = 2, \quad s = 2 \rightarrow o = \left\lfloor \frac{10 + 2 \times 2 - 3}{2} \right\rfloor + 1 = 6$$

If you are interested in learning more about the size of the convolution output, we recommend the manuscript *A guide to convolution arithmetic for deep learning* by *Vincent Dumoulin* and *Francesco Visin*, which is freely available at `https://arxiv.org/abs/1603.07285`.

Finally, in order to learn how to compute convolutions in one dimension, a naive implementation is shown in the following code block, and the results are compared with the `numpy.convolve` function. The code is as follows:

```
>>> import numpy as np
>>> def conv1d(x, w, p=0, s=1):
...     w_rot = np.array(w[::-1])
...     x_padded = np.array(x)
...     if p > 0:
...         zero_pad = np.zeros(shape=p)
...         x_padded = np.concatenate([
...             zero_pad, x_padded, zero_pad
...         ])
...     res = []
...     for i in range(0, int((len(x_padded) - len(w_rot))) + 1, s):
...         res.append(np.sum(x_padded[i:i+w_rot.shape[0]] * w_rot))
...     return np.array(res)
>>> ## Testing:
>>> x = [1, 3, 2, 4, 5, 6, 1, 3]
>>> w = [1, 0, 3, 1, 2]
>>> print('Conv1d Implementation:',
...       conv1d(x, w, p=2, s=1))
Conv1d Implementation: [ 5. 14. 16. 26. 24. 34. 19. 22.]
>>> print('NumPy Results:',
...       np.convolve(x, w, mode='same'))
NumPy Results: [ 5 14 16 26 24 34 19 22]
```

So far, we have mostly focused on convolutions for vectors (1D convolutions). We started with the 1D case to make the concepts easier to understand. In the next section, we will cover 2D convolutions in more detail, which are the building blocks of CNNs for image-related tasks.

Performing a discrete convolution in 2D

The concepts you learned in the previous sections are easily extendible to 2D. When we deal with 2D inputs, such as a matrix, $\boldsymbol{X}_{n_1 \times n_2}$, and the filter matrix, $\boldsymbol{W}_{m_1 \times m_2}$, where $m_1 \leq n_1$ and $m_2 \leq n_2$, then the matrix $\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W}$ is the result of a 2D convolution between $\boldsymbol{X}$ and $\boldsymbol{W}$. This is defined mathematically as follows:

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W} \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2]\, W[k_1, k_2]$$

Notice that if you omit one of the dimensions, the remaining formula is exactly the same as the one we used previously to compute the convolution in 1D. In fact, all the previously mentioned techniques, such as zero padding, rotating the filter matrix, and the use of strides, are also applicable to 2D convolutions, provided that they are extended to both dimensions independently. *Figure 14.5* demonstrates the 2D convolution of an input matrix of size 8×8, using a kernel of size 3×3. The input matrix is padded with zeros with $p = 1$. As a result, the output of the 2D convolution will have a size of 8×8:

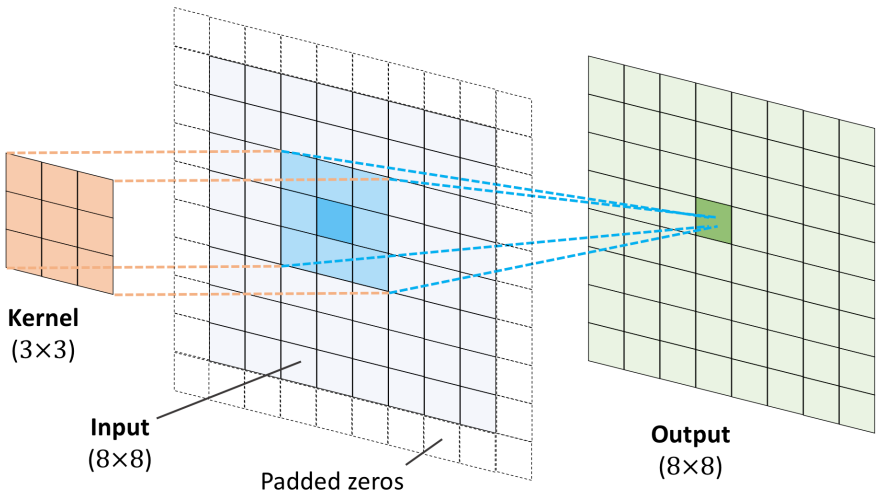


Figure 14.5: The output of a 2D convolution

The following example illustrates the computation of a 2D convolution between an input matrix, $\boldsymbol{X}_{3\times3}$, and a kernel matrix, $\boldsymbol{W}_{3\times3}$, using padding $p = (1, 1)$ and stride $s = (2, 2)$. According to the specified padding, one layer of zeros is added on each side of the input matrix, which results in the padded matrix $\boldsymbol{X}^{\text{padded}}_{5\times5}$, as follows:

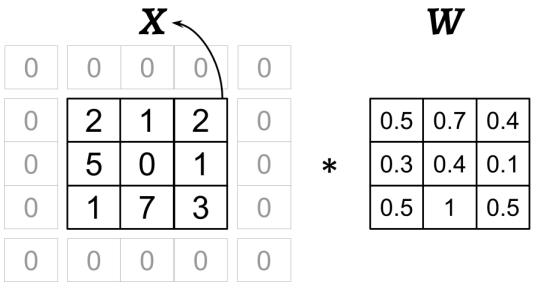


Figure 14.6: Computing a 2D convolution between an input and kernel matrix

With the preceding filter, the rotated filter will be:

$$\boldsymbol{W}^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Note that this rotation is not the same as the transpose matrix. To get the rotated filter in NumPy, we can write `W_rot=W[::-1,::-1]`. Next, we can shift the rotated filter matrix along the padded input matrix, X^{padded}, like a sliding window, and compute the sum of the element-wise product, which is denoted by the $\odot$ operator in *Figure 14.7*:

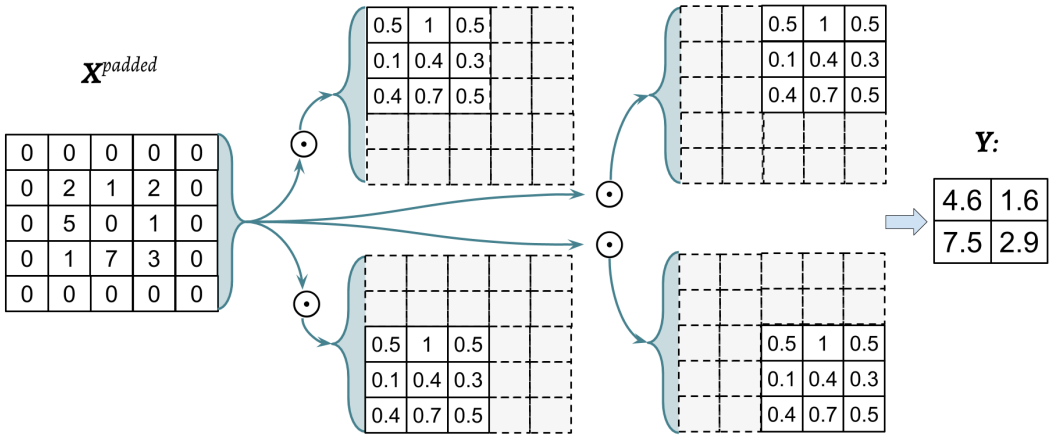


Figure 14.7: Computing the sum of the element-wise product

The result will be the 2×2 matrix, Y.

Let's also implement the 2D convolution according to the *naive* algorithm described. The `scipy.signal` package provides a way to compute 2D convolution via the `scipy.signal.convolve2d` function:

```
>>> import numpy as np
>>> import scipy.signal
>>> def conv2d(X, W, p=(0, 0), s=(1, 1)):
...     W_rot = np.array(W)[::-1,::-1]
...     X_orig = np.array(X)
...     n1 = X_orig.shape[0] + 2*p[0]
...     n2 = X_orig.shape[1] + 2*p[1]
...     X_padded = np.zeros(shape=(n1, n2))
...     X_padded[p[0]:p[0]+X_orig.shape[0],
...              p[1]:p[1]+X_orig.shape[1]] = X_orig
...
...     res = []
...     for i in range(0,
...             int((X_padded.shape[0] - \
...             W_rot.shape[0])/s[0])+1, s[0]):
...         res.append([])
```

```
...         for j in range(0,
...                 int((X_padded.shape[1] - \
...                 W_rot.shape[1])/s[1])+1, s[1]):
...             X_sub = X_padded[i:i+W_rot.shape[0],
...                              j:j+W_rot.shape[1]]
...             res[-1].append(np.sum(X_sub * W_rot))
...     return(np.array(res))
>>> X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
>>> W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
>>> print('Conv2d Implementation:\n',
...       conv2d(X, W, p=(1, 1), s=(1, 1)))
Conv2d Implementation:
[[ 11.  25.  32.  13.]
 [ 19.  25.  24.  13.]
 [ 13.  28.  25.  17.]
 [ 11.  17.  14.   9.]]
>>> print('SciPy Results:\n',
...       scipy.signal.convolve2d(X, W, mode='same'))
SciPy Results:
[[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]
```

Efficient algorithms for computing convolution

We provided a naive implementation to compute a 2D convolution for the purpose of understanding the concepts. However, this implementation is very inefficient in terms of memory requirements and computational complexity. Therefore, it should not be used in real-world NN applications.

One aspect is that the filter matrix is actually not rotated in most tools like PyTorch. Moreover, in recent years, much more efficient algorithms have been developed that use the Fourier transform to compute convolutions. It is also important to note that in the context of NNs, the size of a convolution kernel is usually much smaller than the size of the input image.

For example, modern CNNs usually use kernel sizes such as 1×1, 3×3, or 5×5, for which efficient algorithms have been designed that can carry out the convolutional operations much more efficiently, such as Winograd's minimal filtering algorithm. These algorithms are beyond the scope of this book, but if you are interested in learning more, you can read the manuscript *Fast Algorithms for Convolutional Neural Networks* by *Andrew Lavin* and *Scott Gray*, 2015, which is freely available at `https://arxiv.org/abs/1509.09308`.

In the next section, we will discuss subsampling or pooling, which is another important operation often used in CNNs.

Subsampling layers

Subsampling is typically applied in two forms of pooling operations in CNNs: **max-pooling** and **mean-pooling** (also known as **average-pooling**). The pooling layer is usually denoted by $P_{n_1 \times n_2}$. Here, the subscript determines the size of the neighborhood (the number of adjacent pixels in each dimension) where the max or mean operation is performed. We refer to such a neighborhood as the **pooling size**.

The operation is described in *Figure 14.8*. Here, max-pooling takes the maximum value from a neighborhood of pixels, and mean-pooling computes their average:

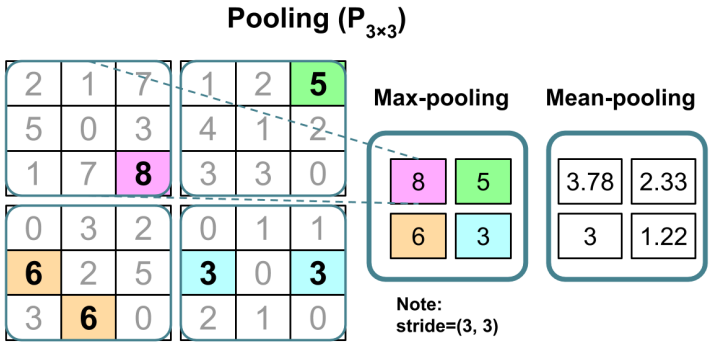


Figure 14.8: An example of max-pooling and mean-pooling

The advantage of pooling is twofold:

- Pooling (max-pooling) introduces a local invariance. This means that small changes in a local neighborhood do not change the result of max-pooling. Therefore, it helps with generating features that are more robust to noise in the input data. Refer to the following example, which shows that the max-pooling of two different input matrices, X_1 and X_2, results in the same output:

$$\boldsymbol{X}_1 = \begin{bmatrix} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{bmatrix}$$

$$\boldsymbol{X}_2 = \begin{bmatrix} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{bmatrix}$$

$$\xrightarrow{\text{max pooling } P_{2\times 2}} \begin{bmatrix} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{bmatrix}$$

- Pooling decreases the size of features, which results in higher computational efficiency. Furthermore, reducing the number of features may reduce the degree of overfitting as well.

Overlapping versus non-overlapping pooling

Traditionally, pooling is assumed to be non-overlapping. Pooling is typically performed on non-overlapping neighborhoods, which can be done by setting the stride parameter equal to the pooling size. For example, a non-overlapping pooling layer, $P_{n_1 \times n_2}$, requires a stride parameter $s = (n_1, n_2)$. On the other hand, overlapping pooling occurs if the stride is smaller than the pooling size. An example where overlapping pooling is used in a convolutional network is described in *ImageNet Classification with Deep Convolutional Neural Networks* by *A. Krizhevsky*, *I. Sutskever*, and *G. Hinton*, 2012, which is freely available as a manuscript at `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks`.

While pooling is still an essential part of many CNN architectures, several CNN architectures have also been developed without using pooling layers. Instead of using pooling layers to reduce the feature size, researchers use convolutional layers with a stride of 2.

In a sense, you can think of a convolutional layer with stride 2 as a pooling layer with learnable weights. If you are interested in an empirical comparison of different CNN architectures developed with and without pooling layers, we recommend reading the research article *Striving for Simplicity: The All Convolutional Net* by *Jost Tobias Springenberg*, *Alexey Dosovitskiy*, *Thomas Brox*, and *Martin Riedmiller*. This article is freely available at `https://arxiv.org/abs/1412.6806`.

Putting everything together — implementing a CNN

So far, you have learned about the basic building blocks of CNNs. The concepts illustrated in this chapter are not really more difficult than traditional multilayer NNs. We can say that the most important operation in a traditional NN is matrix multiplication. For instance, we use matrix multiplications to compute the pre-activations (or net inputs), as in $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + b$. Here, $\boldsymbol{x}$ is a column vector ($\mathbb{R}^{n \times 1}$ matrix) representing pixels, and $\boldsymbol{W}$ is the weight matrix connecting the pixel inputs to each hidden unit.

In a CNN, this operation is replaced by a convolution operation, as in $\boldsymbol{Z} = \boldsymbol{W} * \boldsymbol{X} + \boldsymbol{b}$, where $\boldsymbol{X}$ is a matrix representing the pixels in a *height×width* arrangement. In both cases, the pre-activations are passed to an activation function to obtain the activation of a hidden unit, $\boldsymbol{A} = \sigma(\boldsymbol{Z})$, where σ is the activation function. Furthermore, you will recall that subsampling is another building block of a CNN, which may appear in the form of pooling, as was described in the previous section.

Working with multiple input or color channels

An input to a convolutional layer may contain one or more 2D arrays or matrices with dimensions $N_1 \times N_2$ (for example, the image height and width in pixels). These $N_1 \times N_2$ matrices are called *channels*. Conventional implementations of convolutional layers expect a rank-3 tensor representation as an input, for example, a three-dimensional array, $\boldsymbol{X}_{N_1 \times N_2 \times C_{in}}$, where C_{in} is the number of input channels. For example, let's consider images as input to the first layer of a CNN. If the image is colored and uses the RGB color mode, then $C_{in} = 3$ (for the red, green, and blue color channels in RGB). However, if the image is in grayscale, then we have $C_{in} = 1$, because there is only one channel with the grayscale pixel intensity values.

Reading an image file

When we work with images, we can read images into NumPy arrays using the `uint8` (unsigned 8-bit integer) data type to reduce memory usage compared to 16-bit, 32-bit, or 64-bit integer types, for example.

Unsigned 8-bit integers take values in the range [0, 255], which are sufficient to store the pixel information in RGB images, which also take values in the same range.

In *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, you saw that PyTorch provides a module for loading/storing and manipulating images via `torchvision`. Let's recap how to read an image (this example RGB image is located in the code bundle folder that is provided with this chapter):

```
>>> import torch
>>> from torchvision.io import read_image
>>> img = read_image('example-image.png')
>>> print('Image shape:', img.shape)
Image shape: torch.Size([3, 252, 221])
>>> print('Number of channels:', img.shape[0])
Number of channels: 3
>>> print('Image data type:', img.dtype)
Image data type: torch.uint8
>>> print(img[:, 100:102, 100:102])
tensor([[[179, 182],
         [180, 182]],

        [[134, 136],
         [135, 137]],

        [[110, 112],
         [111, 113]]], dtype=torch.uint8)
```

Note that with `torchvision`, the input and output image tensors are in the format of `Tensor[channels, image_height, image_width]`.

Now that you are familiar with the structure of input data, the next question is, how can we incorporate multiple input channels in the convolution operation that we discussed in the previous sections? The answer is very simple: we perform the convolution operation for each channel separately and then add the results together using the matrix summation. The convolution associated with each channel (c) has its own kernel matrix as $W[:, :, c]$.

The total pre-activation result is computed in the following formula:

$$\begin{array}{l}\text{Given an example } \boldsymbol{X}_{n_1 \times n_2 \times C_{in}},\\ \text{a kernel matrix } \boldsymbol{W}_{m_1 \times m_2 \times C_{in}},\\ \text{and a bias value } b\end{array} \quad \Rightarrow \quad \begin{cases} \boldsymbol{Z}^{Conv} = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c] * \boldsymbol{X}[:,:,c] \\ \text{Pre-activation:} \quad \boldsymbol{Z} = \boldsymbol{Z}^{Conv} + b_c \\ \text{Feature map:} \quad \boldsymbol{A} = \sigma(\boldsymbol{Z}) \end{cases}$$

The final result, $\boldsymbol{A}$, is a feature map. Usually, a convolutional layer of a CNN has more than one feature map. If we use multiple feature maps, the kernel tensor becomes four-dimensional: $width \times height \times C_{in} \times C_{out}$. Here, $width \times height$ is the kernel size, C_{in} is the number of input channels, and C_{out} is the number of output feature maps. So, now let's include the number of output feature maps in the preceding formula and update it, as follows:

$$\begin{array}{l}\text{Given an example } \boldsymbol{X}_{n_1 \times n_2 \times C_{in}},\\ \text{a kernel matrix } \boldsymbol{W}_{m_1 \times m_2 \times C_{in} \times C_{out}},\\ \text{and a bias vector } \boldsymbol{b}_{C_{out}}\end{array} \quad \Rightarrow \quad \begin{cases} \boldsymbol{Z}^{Conv}[:,:,k] = \sum_{c=1}^{C_{in}} \boldsymbol{W}[:,:,c,k] * \boldsymbol{X}[:,:,c] \\ \boldsymbol{Z}[:,:,k] = \boldsymbol{Z}^{Conv}[:,:,k] + b[k] \\ \boldsymbol{A}[:,:,k] = \sigma(\boldsymbol{Z}[:,:,k]) \end{cases}$$

To conclude our discussion of computing convolutions in the context of NNs, let's look at the example in *Figure 14.9*, which shows a convolutional layer, followed by a pooling layer. In this example, there are three input channels. The kernel tensor is four-dimensional. Each kernel matrix is denoted as $m_1 \times m_2$, and there are three of them, one for each input channel. Furthermore, there are five such kernels, accounting for five output feature maps. Finally, there is a pooling layer for subsampling the feature maps:

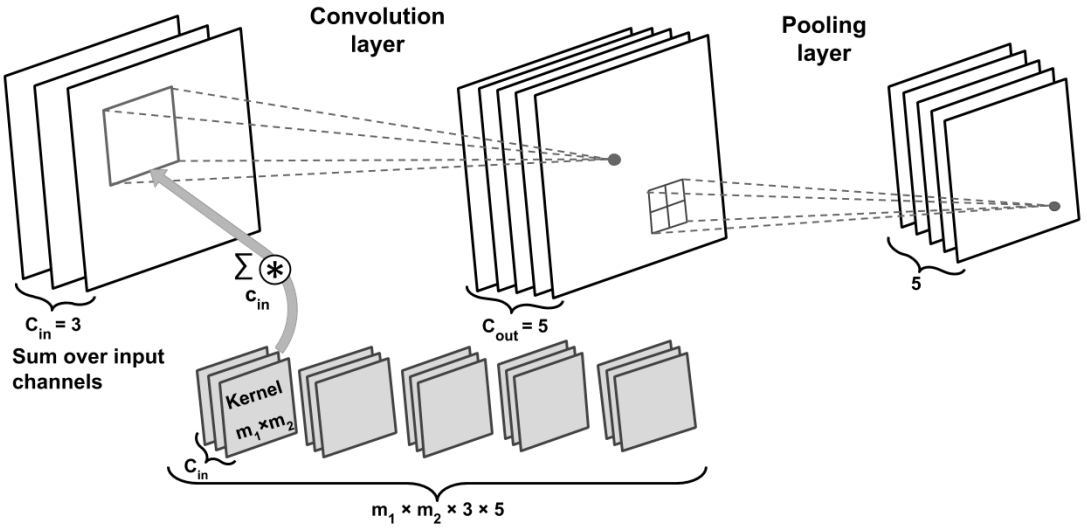


Figure 14.9: Implementing a CNN

How many trainable parameters exist in the preceding example?

To illustrate the advantages of convolution, parameter sharing, and sparse connectivity, let's work through an example. The convolutional layer in the network shown in *Figure 14.9* is a four-dimensional tensor. So, there are $m_1 \times m_2 \times 3 \times 5$ parameters associated with the kernel. Furthermore, there is a bias vector for each output feature map of the convolutional layer. Thus, the size of the bias vector is 5. Pooling layers do not have any (trainable) parameters; therefore, we can write the following:

$$m_1 \times m_2 \times 3 \times 5 + 5$$

If the input tensor is of size $n_1 \times n_2 \times 3$, assuming that the convolution is performed with the same-padding mode, then the size of the output feature maps would be $n_1 \times n_2 \times 5$.

Note that if we use a fully connected layer instead of a convolutional layer, this number will be much larger. In the case of a fully connected layer, the number of parameters for the weight matrix to reach the same number of output units would have been as follows:

$$(n_1 \times n_2 \times 3) \times (n_1 \times n_2 \times 5) = (n_1 \times n_2)^2 \times 3 \times 5$$

In addition, the size of the bias vector is $n_1 \times n_2 \times 5$ (one bias element for each output unit). Given that $m_1 < n_1$ and $m_2 < n_2$, we can see that the difference in the number of trainable parameters is significant.

Lastly, as was already mentioned, the convolution operations typically are carried out by treating an input image with multiple color channels as a stack of matrices; that is, we perform the convolution on each matrix separately and then add the results, as was illustrated in the previous figure. However, convolutions can also be extended to 3D volumes if you are working with 3D datasets, for example, as shown in the paper *VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition* by *Daniel Maturana* and *Sebastian Scherer*, 2015, which can be accessed at `https://www.ri.cmu.edu/pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf`.

In the next section, we will talk about how to regularize an NN.

Regularizing an NN with L2 regularization and dropout

Choosing the size of a network, whether we are dealing with a traditional (fully connected) NN or a CNN, has always been a challenging problem. For instance, the size of a weight matrix and the number of layers need to be tuned to achieve a reasonably good performance.

You will recall from *Chapter 13*, *Going Deeper – The Mechanics of PyTorch*, that a simple network without a hidden layer could only capture a linear decision boundary, which is not sufficient for dealing with an exclusive or (or XOR) or similar problem. The *capacity* of a network refers to the level of complexity of the function that it can learn to approximate. Small networks, or networks with a relatively small number of parameters, have a low capacity and are therefore likely to *underfit*, resulting in poor performance, since they cannot learn the underlying structure of complex datasets. However, very large networks may result in *overfitting*, where the network will memorize the training data and do extremely well on the training dataset while achieving a poor performance on the held-out test dataset. When we deal with real-world machine learning problems, we do not know how large the network should be *a priori*.

One way to address this problem is to build a network with a relatively large capacity (in practice, we want to choose a capacity that is slightly larger than necessary) to do well on the training dataset. Then, to prevent overfitting, we can apply one or multiple regularization schemes to achieve good generalization performance on new data, such as the held-out test dataset.

In *Chapters 3* and *4*, we covered L1 and L2 regularization. Both techniques can prevent or reduce the effect of overfitting by adding a penalty to the loss that results in shrinking the weight parameters during training. While both L1 and L2 regularization can be used for NNs as well, with L2 being the more common choice of the two, there are other methods for regularizing NNs, such as dropout, which we discuss in this section. But before we move on to discussing dropout, to use L2 regularization within a convolutional or fully connected network (recall, fully connected layers are implemented via `torch.nn.Linear` in PyTorch), you can simply add the L2 penalty of a particular layer to the loss function in PyTorch, as follows:

```
>>> import torch.nn as nn
>>> loss_func = nn.BCELoss()
>>> loss = loss_func(torch.tensor([0.9]), torch.tensor([1.0]))
>>> l2_lambda = 0.001
>>> conv_layer = nn.Conv2d(in_channels=3,
...                        out_channels=5,
...                        kernel_size=5)
>>> l2_penalty = l2_lambda * sum(
...     [(p**2).sum() for p in conv_layer.parameters()]
... )
>>> loss_with_penalty = loss + l2_penalty
>>> linear_layer = nn.Linear(10, 16)
```

```
>>> l2_penalty = l2_lambda * sum(
...     [(p**2).sum() for p in linear_layer.parameters()]
... )
>>> loss_with_penalty = loss + l2_penalty
```

Weight decay versus L2 regularization

An alternative way to use L2 regularization is by setting the `weight_decay` parameter in a PyTorch optimizer to a positive value, for example:

```
optimizer = torch.optim.SGD(
    model.parameters(),
    weight_decay=l2_lambda,
    ...
)
```

While L2 regularization and `weight_decay` are not strictly identical, it can be shown that they are equivalent when using **stochastic gradient descent** (**SGD**) optimizers. Interested readers can find more information in the article *Decoupled Weight Decay Regularization* by *Ilya Loshchilov* and *Frank Hutter*, 2019, which is freely available at `https://arxiv.org/abs/1711.05101`.

In recent years, **dropout** has emerged as a popular technique for regularizing (deep) NNs to avoid overfitting, thus improving the generalization performance (*Dropout: A Simple Way to Prevent Neural Networks from Overfitting* by *N. Srivastava*, *G. Hinton*, *A. Krizhevsky*, *I. Sutskever*, and *R. Salakhutdinov*, *Journal of Machine Learning Research 15.1*, pages 1929-1958, 2014, `http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`). Dropout is usually applied to the hidden units of higher layers and works as follows: during the training phase of an NN, a fraction of the hidden units is randomly dropped at every iteration with probability p_{drop} (or keep probability $p_{keep} = 1 - p_{drop}$). This dropout probability is determined by the user and the common choice is $p = 0.5$, as discussed in the previously mentioned article by *Nitish Srivastava* and others, 2014. When dropping a certain fraction of input neurons, the weights associated with the remaining neurons are rescaled to account for the missing (dropped) neurons.

The effect of this random dropout is that the network is forced to learn a redundant representation of the data. Therefore, the network cannot rely on the activation of any set of hidden units, since they may be turned off at any time during training, and is forced to learn more general and robust patterns from the data.

This random dropout can effectively prevent overfitting. *Figure 14.10* shows an example of applying dropout with probability $p = 0.5$ during the training phase, whereby half of the neurons will become inactive randomly (dropped units are selected randomly in each forward pass of training). However, during prediction, all neurons will contribute to computing the pre-activations of the next layer:

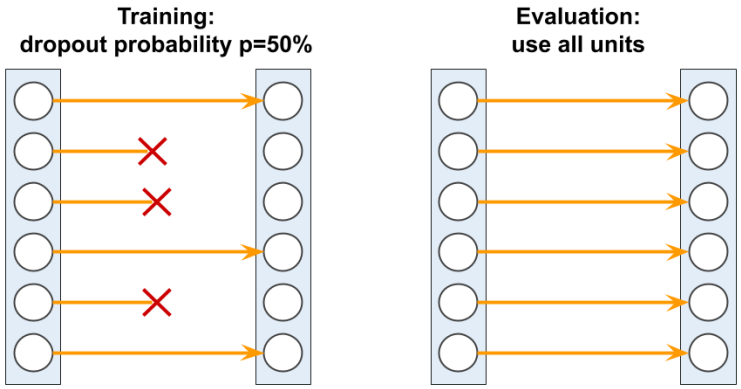


Figure 14.10: Applying dropout during the training phase

As shown here, one important point to remember is that units may drop randomly during training only, whereas for the evaluation (inference) phase, all the hidden units must be active (for instance, $p_{drop} = 0$ or $p_{keep} = 1$). To ensure that the overall activations are on the same scale during training and prediction, the activations of the active neurons have to be scaled appropriately (for example, by halving the activation if the dropout probability was set to $p = 0.5$).

However, since it is inconvenient to always scale activations when making predictions, PyTorch and other tools scale the activations during training (for example, by doubling the activations if the dropout probability was set to $p = 0.5$). This approach is commonly referred to as *inverse dropout*.

While the relationship is not immediately obvious, dropout can be interpreted as the consensus (averaging) of an ensemble of models. As discussed in *Chapter 7*, *Combining Different Models for Ensemble Learning*, in ensemble learning, we train several models independently. During prediction, we then use the consensus of all the trained models. We already know that model ensembles are known to perform better than single models. In deep learning, however, both training several models and collecting and averaging the output of multiple models is computationally expensive. Here, dropout offers a workaround, with an efficient way to train many models at once and compute their average predictions at test or prediction time.

As mentioned previously, the relationship between model ensembles and dropout is not immediately obvious. However, consider that in dropout, we have a different model for each mini-batch (due to setting the weights to zero randomly during each forward pass).

Then, via iterating over the mini-batches, we essentially sample over $M = 2^h$ models, where h is the number of hidden units.

The restriction and aspect that distinguishes dropout from regular ensembling, however, is that we share the weights over these "different models," which can be seen as a form of regularization. Then, during "inference" (for instance, predicting the labels in the test dataset), we can average over all these different models that we sampled over during training. This is very expensive, though.

Then, averaging the models, that is, computing the geometric mean of the class-membership probability that is returned by a model, i, can be computed as follows:

$$p_{Ensemble} = \left[\prod_{j=1}^{M} p^{\{i\}}\right]^{\frac{1}{M}}$$

Now, the trick behind dropout is that this geometric mean of the model ensembles (here, M models) can be approximated by scaling the predictions of the last (or final) model sampled during training by a factor of $1/(1 - p)$, which is much cheaper than computing the geometric mean explicitly using the previous equation. (In fact, the approximation is exactly equivalent to the true geometric mean if we consider linear models.)

Loss functions for classification

In *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, we saw different activation functions, such as ReLU, sigmoid, and tanh. Some of these activation functions, like ReLU, are mainly used in the intermediate (hidden) layers of an NN to add non-linearities to our model. But others, like sigmoid (for binary) and softmax (for multiclass), are added at the last (output) layer, which results in class-membership probabilities as the output of the model. If the sigmoid or softmax activations are not included at the output layer, then the model will compute the logits instead of the class-membership probabilities.

Focusing on classification problems here, depending on the type of problem (binary versus multiclass) and the type of output (logits versus probabilities), we should choose the appropriate loss function to train our model. **Binary cross-entropy** is the loss function for a binary classification (with a single output unit), and **categorical cross-entropy** is the loss function for multiclass classification. In the `torch.nn` module, the categorical cross-entropy loss takes in ground truth labels as integers (for example, y=2, out of three classes, 0, 1, and 2).

Figure 14.11 describes two loss functions available in `torch.nn` for dealing with both cases: binary classification and multiclass with integer labels. Each one of these two loss functions also has the option to receive the predictions in the form of logits or class-membership probabilities:

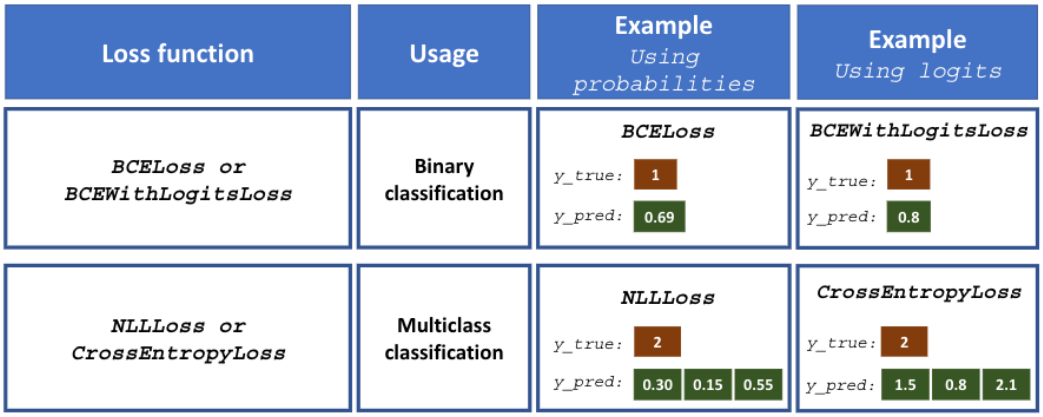


Figure 14.11: Two examples of loss functions in PyTorch

Please note that computing the cross-entropy loss by providing the logits, and not the class-membership probabilities, is usually preferred due to numerical stability reasons. For binary classification, we can either provide logits as inputs to the loss function `nn.BCEWithLogitsLoss()`, or compute the probabilities based on the logits and feed them to the loss function `nn.BCELoss()`. For multiclass classification, we can either provide logits as inputs to the loss function `nn.CrossEntropyLoss()`, or compute the log probabilities based on the logits and feed them to the negative log-likelihood loss function `nn.NLLLoss()`.

The following code will show you how to use these loss functions with two different formats, where either the logits or class-membership probabilities are given as inputs to the loss functions:

```
>>> ####### Binary Cross-entropy
>>> logits = torch.tensor([0.8])
>>> probas = torch.sigmoid(logits)
>>> target = torch.tensor([1.0])
>>> bce_loss_fn = nn.BCELoss()
>>> bce_logits_loss_fn = nn.BCEWithLogitsLoss()
>>> print(f'BCE (w Probas): {bce_loss_fn(probas, target):.4f}')
BCE (w Probas): 0.3711
>>> print(f'BCE (w Logits): '
...       f'{bce_logits_loss_fn(logits, target):.4f}')
BCE (w Logits): 0.3711
```

```
>>> ####### Categorical Cross-entropy
>>> logits = torch.tensor([[1.5, 0.8, 2.1]])
>>> probas = torch.softmax(logits, dim=1)
>>> target = torch.tensor([2])
>>> cce_loss_fn = nn.NLLLoss()
>>> cce_logits_loss_fn = nn.CrossEntropyLoss()
>>> print(f'CCE (w Logits): '
...       f'{cce_logits_loss_fn(logits, target):.4f}')
CCE (w Probas): 0.5996
>>> print(f'CCE (w Probas): '
...       f'{cce_loss_fn(torch.log(probas), target):.4f}')
CCE (w Logits): 0.5996
```

Note that sometimes, you may come across an implementation where a categorical cross-entropy loss is used for binary classification. Typically, when we have a binary classification task, the model returns a single output value for each example. We interpret this single model output as the probability of the positive class (for example, class 1), $P(\text{class} = 1|\boldsymbol{x})$. In a binary classification problem, it is implied that $P(\text{class} = 0|\boldsymbol{x}) = 1 - P(\text{class} = 1|\boldsymbol{x})$; hence, we do not need a second output unit in order to obtain the probability of the negative class. However, sometimes practitioners choose to return two outputs for each training example and interpret them as probabilities of each class: $P(\text{class} = 0|\boldsymbol{x})$ versus $P(\text{class} = 1|\boldsymbol{x})$. Then, in such a case, using a softmax function (instead of the logistic sigmoid) to normalize the outputs (so that they sum to 1) is recommended, and categorical cross-entropy is the appropriate loss function.

Implementing a deep CNN using PyTorch

In *Chapter 13*, as you may recall, we solved the handwritten digit recognition problem using the `torch.nn` module. You may also recall that we achieved about 95.6 percent accuracy using an NN with two linear hidden layers.

Now, let's implement a CNN and see whether it can achieve a better predictive performance compared to the previous model for classifying handwritten digits. Note that the fully connected layers that we saw in *Chapter 13* were able to perform well on this problem. However, in some applications, such as reading bank account numbers from handwritten digits, even tiny mistakes can be very costly. Therefore, it is crucial to reduce this error as much as possible.

The multilayer CNN architecture

The architecture of the network that we are going to implement is shown in *Figure 14.12*. The inputs are 28×28 grayscale images. Considering the number of channels (which is 1 for grayscale images) and a batch of input images, the input tensor's dimensions will be *batchsize*×28×28×1.

The input data goes through two convolutional layers that have a kernel size of 5×5. The first convolution has 32 output feature maps, and the second one has 64 output feature maps. Each convolution layer is followed by a subsampling layer in the form of a max-pooling operation, $P_{2\times2}$. Then a fully connected layer passes the output to a second fully connected layer, which acts as the final *softmax* output layer. The architecture of the network that we are going to implement is shown in *Figure 14.12*:

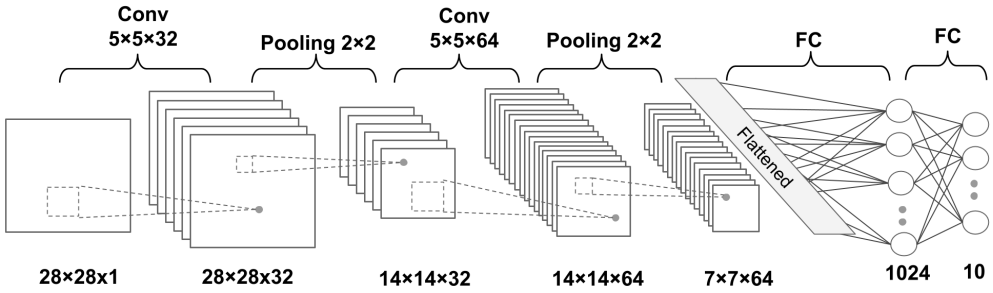


Figure 14.12: A deep CNN

The dimensions of the tensors in each layer are as follows:

- Input: [*batchsize*×28×28×1]
- Conv_1: [*batchsize*×28×28×32]
- Pooling_1: [*batchsize*×14×14×32]
- Conv_2: [*batchsize*×14×14×64]
- Pooling_2: [*batchsize*×7×7×64]
- FC_1: [*batchsize*×1024]
- FC_2 and softmax layer: [*batchsize*×10]

For the convolutional kernels, we are using `stride=1` such that the input dimensions are preserved in the resulting feature maps. For the pooling layers, we are using `kernel_size=2` to subsample the image and shrink the size of the output feature maps. We will implement this network using the PyTorch NN module.

Loading and preprocessing the data

First, we will load the MNIST dataset using the `torchvision` module and construct the training and test sets, as we did in *Chapter 13*:

```
>>> import torchvision
>>> from torchvision import transforms
>>> image_path = './'
>>> transform = transforms.Compose([
...     transforms.ToTensor()
... ])
```

```
>>> mnist_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=True,
...     transform=transform, download=True
... )
>>> from torch.utils.data import Subset
>>> mnist_valid_dataset = Subset(mnist_dataset,
...                              torch.arange(10000))
>>> mnist_train_dataset = Subset(mnist_dataset,
...                              torch.arange(
...                                  10000, len(mnist_dataset)
...                              ))
>>> mnist_test_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=False,
...     transform=transform, download=False
... )
```

The MNIST dataset comes with a pre-specified training and test dataset partitioning scheme, but we also want to create a validation split from the train partition. Hence, we used the first 10,000 training examples for validation. Note that the images are not sorted by class label, so we do not have to worry about whether those validation set images are from the same classes.

Next, we will construct the data loader with batches of 64 images for the training set and validation set, respectively:

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(mnist_train_dataset,
...                       batch_size,
...                       shuffle=True)
>>> valid_dl = DataLoader(mnist_valid_dataset,
...                       batch_size,
...                       shuffle=False)
```

The features we read are of values in the range [0, 1]. Also, we already converted the images to tensors. The labels are integers from 0 to 9, representing ten digits. Hence, we don't need to do any scaling or further conversion.

Now, after preparing the dataset, we are ready to implement the CNN we just described.

Implementing a CNN using the torch.nn module

For implementing a CNN in PyTorch, we use the `torch.nn` `Sequential` class to stack different layers, such as convolution, pooling, and dropout, as well as the fully connected layers. The `torch.nn` module provides classes for each one: `nn.Conv2d` for a two-dimensional convolution layer; `nn.MaxPool2d` and `nn.AvgPool2d` for subsampling (max-pooling and average-pooling); and `nn.Dropout` for regularization using dropout. We will go over each of these classes in more detail.

Configuring CNN layers in PyTorch

Constructing a layer with the `Conv2d` class requires us to specify the number of output channels (which is equivalent to the number of output feature maps, or the number of output filters) and kernel sizes.

In addition, there are optional parameters that we can use to configure a convolutional layer. The most commonly used ones are the strides (with a default value of 1 in both x, y dimensions) and padding, which controls the amount of implicit padding on both dimensions. Additional configuration parameters are listed in the official documentation: `https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html`.

It is worth mentioning that usually, when we read an image, the default dimension for the channels is the first dimension of the tensor array (or the second dimension considering the batch dimension). This is called the NCHW format, where N stands for the number of images within the batch, C stands for channels, and H and W stand for height and width, respectively.

Note that the `Conv2D` class assumes that inputs are in NCHW format by default. (Other tools, such as TensorFlow, use NHWC format.) However, if you come across some data whose channels are placed at the last dimension, you would need to swap the axes in your data to move the channels to the first dimension (or the second dimension considering the batch dimension). After the layer is constructed, it can be called by providing a four-dimensional tensor, with the first dimension reserved for a batch of examples; the second dimension corresponds to the channel; and the other two dimensions are the spatial dimensions.

As shown in the architecture of the CNN model that we want to build, each convolution layer is followed by a pooling layer for subsampling (reducing the size of feature maps). The `MaxPool2d` and `AvgPool2d` classes construct the max-pooling and average-pooling layers, respectively. The `kernel_size` argument determines the size of the window (or neighborhood) that will be used to compute the max or mean operations. Furthermore, the `stride` parameter can be used to configure the pooling layer, as we discussed earlier.

Finally, the `Dropout` class will construct the dropout layer for regularization, with the argument `p` that denotes the drop probability p_{drop}, which is used to determine the probability of dropping the input units during training, as we discussed earlier. When calling this layer, its behavior can be controlled via `model.train()` and `model.eval()`, to specify whether this call will be made during training or during the inference. When using dropout, alternating between these two modes is crucial to ensure that it behaves correctly; for instance, nodes are only randomly dropped during training, not evaluation or inference.

Constructing a CNN in PyTorch

Now that you have learned about these classes, we can construct the CNN model that was shown in the previous figure. In the following code, we will use the `Sequential` class and add the convolution and pooling layers:

```
>>> model = nn.Sequential()
>>> model.add_module(
...     'conv1',
...     nn.Conv2d(
...         in_channels=1, out_channels=32,
...         kernel_size=5, padding=2
...     )
... )
>>> model.add_module('relu1', nn.ReLU())
>>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
>>> model.add_module(
...     'conv2',
...     nn.Conv2d(
...         in_channels=32, out_channels=64,
...         kernel_size=5, padding=2
...     )
... )
>>> model.add_module('relu2', nn.ReLU())
>>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
```

So far, we have added two convolution layers to the model. For each convolutional layer, we used a kernel of size 5×5 and `padding=2`. As discussed earlier, using same padding mode preserves the spatial dimensions (vertical and horizontal dimensions) of the feature maps such that the inputs and outputs have the same height and width (and the number of channels may only differ in terms of the number of filters used). As mentioned before, the spatial dimension of the output feature map is calculated by:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

where n is the spatial dimension of the input feature map, and p, m, and s denote the padding, kernel size, and stride, respectively. We obtain $p = 2$ in order to achieve $o = i$.

The max-pooling layers with pooling size 2×2 and stride of 2 will reduce the spatial dimensions by half. (Note that if the `stride` parameter is not specified in `MaxPool2D`, by default, it is set equal to the pooling kernel size.)

While we can calculate the size of the feature maps at this stage manually, PyTorch provides a convenient method to compute this for us:

```
>>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 64, 7, 7])
```

By providing the input shape as a tuple `(4, 1, 28, 28)` (4 images within the batch, 1 channel, and image size 28×28), specified in this example, we calculated the output to have a shape `(4, 64, 7, 7)`, indicating feature maps with 64 channels and a spatial size of 7×7. The first dimension corresponds to the batch dimension, for which we used 4 arbitrarily.

The next layer that we want to add is a fully connected layer for implementing a classifier on top of our convolutional and pooling layers. The input to this layer must have rank 2, that is, shape [*batchsize* × *input_units*]. Thus, we need to flatten the output of the previous layers to meet this requirement for the fully connected layer:

```
>>> model.add_module('flatten', nn.Flatten())
>>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 3136])
```

As the output shape indicates, the input dimensions for the fully connected layer are correctly set up. Next, we will add two fully connected layers with a dropout layer in between:

```
>>> model.add_module('fc1', nn.Linear(3136, 1024))
>>> model.add_module('relu3', nn.ReLU())
>>> model.add_module('dropout', nn.Dropout(p=0.5))
>>> model.add_module('fc2', nn.Linear(1024, 10))
```

The last fully connected layer, named `'fc2'`, has 10 output units for the 10 class labels in the MNIST dataset. In practice, we usually use the sofmax activation to obtain the class-membership probabilities of each input example, assuming that the classes are mutually exclusive, so the probabilities for each example sum to 1. However, the softmax function is already used internally inside PyTorch's `CrossEntropyLoss` implementation, which is why don't have to explicitly add it as a layer after the output layer above. The following code will create the loss function and optimizer for the model:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

The Adam optimizer

Note that in this implementation, we used the `torch.optim.Adam` class for training the CNN model. The Adam optimizer is a robust, gradient-based optimization method suited to nonconvex optimization and machine learning problems. Two popular optimization methods inspired Adam: `RMSProp` and `AdaGrad`.

The key advantage of Adam is in the choice of update step size derived from the running average of gradient moments. Please feel free to read more about the Adam optimizer in the manuscript, *Adam: A Method for Stochastic Optimization* by *Diederik P. Kingma* and *Jimmy Lei Ba*, 2014. The article is freely available at `https://arxiv.org/abs/1412.6980`.

Now we can train the model by defining the following function:

```
>>> def train(model, num_epochs, train_dl, valid_dl):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         model.train()
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item()*y_batch.size(0)
...             is_correct = (
...                 torch.argmax(pred, dim=1) == y_batch
...             ).float()
...             accuracy_hist_train[epoch] += is_correct.sum()
...         loss_hist_train[epoch] /= len(train_dl.dataset)
...         accuracy_hist_train[epoch] /= len(train_dl.dataset)
...
...         model.eval()
```

```
...         with torch.no_grad():
...             for x_batch, y_batch in valid_dl:
...                 pred = model(x_batch)
...                 loss = loss_fn(pred, y_batch)
...                 loss_hist_valid[epoch] += \
...                     loss.item()*y_batch.size(0)
...                 is_correct = (
...                     torch.argmax(pred, dim=1) == y_batch
...                 ).float()
...                 accuracy_hist_valid[epoch] += is_correct.sum()
...         loss_hist_valid[epoch] /= len(valid_dl.dataset)
...         accuracy_hist_valid[epoch] /= len(valid_dl.dataset)
...
...         print(f'Epoch {epoch+1} accuracy: '
...               f'{accuracy_hist_train[epoch]:.4f} val_accuracy: '
...               f'{accuracy_hist_valid[epoch]:.4f}')
...     return loss_hist_train, loss_hist_valid, \
...            accuracy_hist_train, accuracy_hist_valid
```

Note that using the designated settings for training `model.train()` and evaluation `model.eval()` will automatically set the mode for the dropout layer and rescale the hidden units appropriately so that we do not have to worry about that at all. Next, we will train this CNN model and use the validation dataset that we created for monitoring the learning progress:

```
>>> torch.manual_seed(1)
>>> num_epochs = 20
>>> hist = train(model, num_epochs, train_dl, valid_dl)
Epoch 1 accuracy: 0.9503 val_accuracy: 0.9802
...
Epoch 9 accuracy: 0.9968 val_accuracy: 0.9892
...
Epoch 20 accuracy: 0.9979 val_accuracy: 0.9907
```

Once the 20 epochs of training are finished, we can visualize the learning curves:

```
>>> import matplotlib.pyplot as plt
>>> x_arr = np.arange(len(hist[0])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist[0], '-o', label='Train loss')
>>> ax.plot(x_arr, hist[1], '--<', label='Validation loss')
```

```
>>> ax.legend(fontsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist[2], '-o', label='Train acc.')
>>> ax.plot(x_arr, hist[3], '--<',
...         label='Validation acc.')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.set_ylabel('Accuracy', size=15)
>>> plt.show()
```

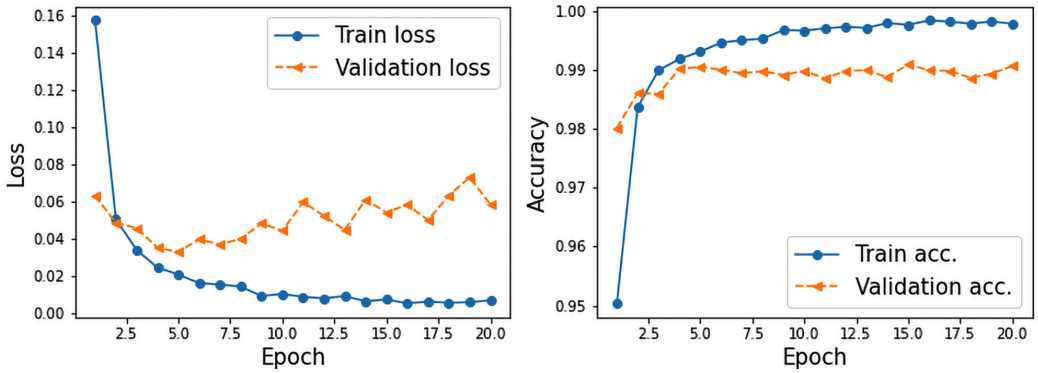


Figure 14.13: Loss and accuracy graphs for the training and validation data

Now, we evaluate the trained model on the test dataset:

```
>>> pred = model(mnist_test_dataset.data.unsqueeze(1) / 255.)
>>> is_correct = (
...     torch.argmax(pred, dim=1) == mnist_test_dataset.targets
... ).float()
>>> print(f'Test accuracy: {is_correct.mean():.4f}')
Test accuracy: 0.9914
```

The CNN model achieves an accuracy of 99.07 percent. Remember that in *Chapter 13*, we got approximately 95 percent accuracy using only fully connected (instead of convolutional) layers.

Finally, we can get the prediction results in the form of class-membership probabilities and convert them to predicted labels by using the `torch.argmax` function to find the element with the maximum probability. We will do this for a batch of 12 examples and visualize the input and predicted labels:

```
>>> fig = plt.figure(figsize=(12, 4))
>>> for i in range(12):
...     ax = fig.add_subplot(2, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
```

```
...     img = mnist_test_dataset[i][0][0, :, :]
...     pred = model(img.unsqueeze(0).unsqueeze(1))
...     y_pred = torch.argmax(pred)
...     ax.imshow(img, cmap='gray_r')
...     ax.text(0.9, 0.1, y_pred.item(),
...             size=15, color='blue',
...             horizontalalignment='center',
...             verticalalignment='center',
...             transform=ax.transAxes)
>>> plt.show()
```

Figure 14.14 shows the handwritten inputs and their predicted labels:

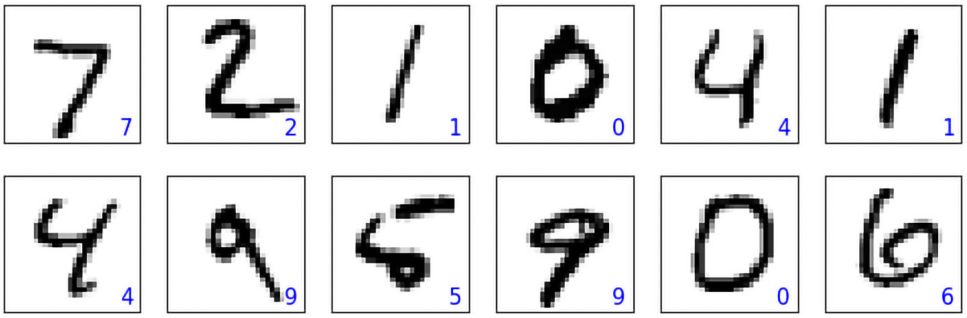


Figure 14.14: Predicted labels for handwritten digits

In this set of plotted examples, all the predicted labels are correct.

We leave the task of showing some of the misclassified digits, as we did in *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*, as an exercise for the reader.

Smile classification from face images using a CNN

In this section, we are going to implement a CNN for smile classification from face images using the CelebA dataset. As you saw in *Chapter 12*, the CelebA dataset contains 202,599 images of celebrities' faces. In addition, 40 binary facial attributes are available for each image, including whether a celebrity is smiling (or not) and their age (young or old).

Based on what you have learned so far, the goal of this section is to build and train a CNN model for predicting the smile attribute from these face images. Here, for simplicity, we will only be using a small portion of the training data (16,000 training examples) to speed up the training process. However, to improve the generalization performance and reduce overfitting on such a small dataset, we will use a technique called **data augmentation**.

Loading the CelebA dataset

First, let's load the data similarly to how we did in the previous section for the MNIST dataset. CelebA data comes in three partitions: a training dataset, a validation dataset, and a test dataset. Next, we will count the number of examples in each partition:

```
>>> image_path = './'
>>> celeba_train_dataset = torchvision.datasets.CelebA(
...     image_path, split='train',
...     target_type='attr', download=True
... )
>>> celeba_valid_dataset = torchvision.datasets.CelebA(
...     image_path, split='valid',
...     target_type='attr', download=True
... )
>>> celeba_test_dataset = torchvision.datasets.CelebA(
...     image_path, split='test',
...     target_type='attr', download=True
... )
>>>
>>> print('Train set:', len(celeba_train_dataset))
Train set:  162770
>>> print('Validation set:', len(celeba_valid_dataset))
Validation: 19867
>>> print('Test set:', len(celeba_test_dataset))
Test set:   19962
```

Alternative ways to download the CelebA dataset

The CelebA dataset is relatively large (approximately 1.5 GB) and the `torchvision` download link is notoriously unstable. If you encounter problems executing the previous code, you can download the files from the official CelebA website manually (`https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html`) or use our download link: `https://drive.google.com/file/d/1m8-EBPgi5MRubrm6iQjafK2QMHDBMSfJ/view?usp=sharing`. If you use our download link, it will download a `celeba.zip` file, which you need to unpack in the current directory where you are running the code. Also, after downloading and unzipping the `celeba` folder, you need to rerun the code above with the setting `download=False` instead of `download=True`. In case you are encountering problems with this approach, please do not hesitate to open a new issue or start a discussion at `https://github.com/rasbt/machine-learning-book` so that we can provide you with additional information.

Next, we will discuss data augmentation as a technique for boosting the performance of deep NNs.

Image transformation and data augmentation

Data augmentation summarizes a broad set of techniques for dealing with cases where the train-

ing data is limited. For instance, certain data augmentation techniques allow us to modify or even artificially synthesize more data and thereby boost the performance of a machine or deep learning model by reducing overfitting. While data augmentation is not only for image data, there is a set of transformations uniquely applicable to image data, such as cropping parts of an image, flipping, and changing the contrast, brightness, and saturation. Let's see some of these transformations that are available via the `torchvision.transforms` module. In the following code block, we will first get five examples from the `celeba_train_dataset` dataset and apply five different types of transformation: 1) cropping an image to a bounding box, 2) flipping an image horizontally, 3) adjusting the contrast, 4) adjusting the brightness, and 5) center-cropping an image and resizing the resulting image back to its original size, (218, 178). In the following code, we will visualize the results of these transformations, showing each one in a separate column for comparison:

```
>>> fig = plt.figure(figsize=(16, 8.5))
>>> ## Column 1: cropping to a bounding-box
>>> ax = fig.add_subplot(2, 5, 1)
>>> img, attr = celeba_train_dataset[0]
>>> ax.set_title('Crop to a \nbounding-box', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 6)
>>> img_cropped = transforms.functional.crop(img, 50, 20, 128, 128)
>>> ax.imshow(img_cropped)
>>>
>>> ## Column 2: flipping (horizontally)
>>> ax = fig.add_subplot(2, 5, 2)
>>> img, attr = celeba_train_dataset[1]
>>> ax.set_title('Flip (horizontal)', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 7)
>>> img_flipped = transforms.functional.hflip(img)
>>> ax.imshow(img_flipped)
>>>
>>> ## Column 3: adjust contrast
>>> ax = fig.add_subplot(2, 5, 3)
>>> img, attr = celeba_train_dataset[2]
>>> ax.set_title('Adjust constrast', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 8)
>>> img_adj_contrast = transforms.functional.adjust_contrast(
...     img, contrast_factor=2
... )
>>> ax.imshow(img_adj_contrast)
>>>
>>> ## Column 4: adjust brightness
>>> ax = fig.add_subplot(2, 5, 4)
```

```
>>> img, attr = celeba_train_dataset[3]
>>> ax.set_title('Adjust brightness', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 9)
>>> img_adj_brightness = transforms.functional.adjust_brightness(
...     img, brightness_factor=1.3
... )
>>> ax.imshow(img_adj_brightness)
>>>
>>> ## Column 5: cropping from image center
>>> ax = fig.add_subplot(2, 5, 5)
>>> img, attr = celeba_train_dataset[4]
>>> ax.set_title('Center crop\nand resize', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 10)
>>> img_center_crop = transforms.functional.center_crop(
...     img, [0.7*218, 0.7*178]
... )
>>> img_resized = transforms.functional.resize(
...     img_center_crop, size=(218, 178)
... )
>>> ax.imshow(img_resized)
>>> plt.show()
```

Figure 14.15 shows the results:

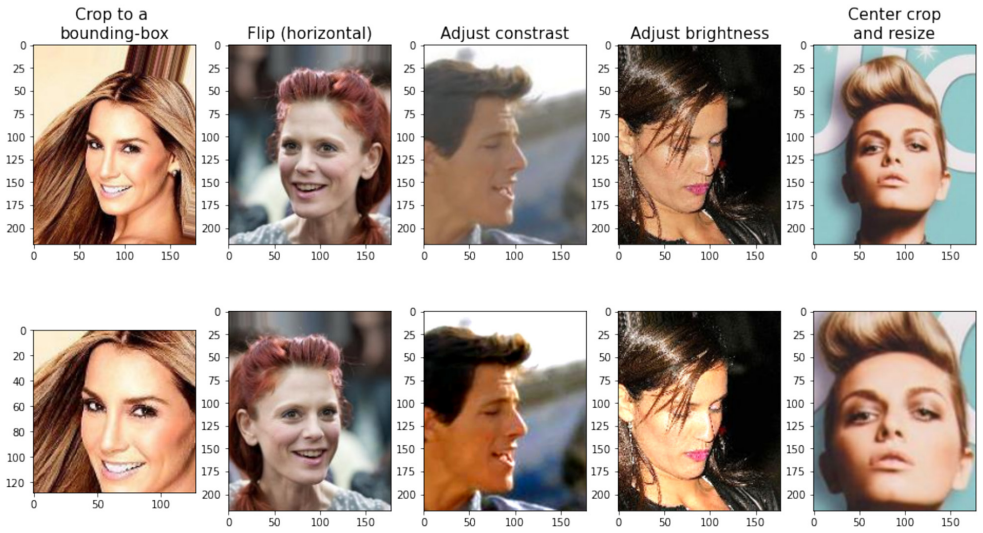


Figure 14.15: Different image transformations

In *Figure 14.15*, the original images are shown in the first row and their transformed versions in the

second row. Note that for the first transformation (leftmost column), the bounding box is specified by four numbers: the coordinate of the upper-left corner of the bounding box (here x=20, y=50), and the width and height of the box (width=128, height=128). Also note that the origin (the coordinates at the location denoted as (0, 0)) for images loaded by PyTorch (as well as other packages such as `imageio`) is the upper-left corner of the image.

The transformations in the previous code block are deterministic. However, all such transformations can also be randomized, which is recommended for data augmentation during model training. For example, a random bounding box (where the coordinates of the upper-left corner are selected randomly) can be cropped from an image, an image can be randomly flipped along either the horizontal or vertical axes with a probability of 0.5, or the contrast of an image can be changed randomly, where the `contrast_factor` is selected at random, but with uniform distribution, from a range of values. In addition, we can create a pipeline of these transformations.

For example, we can first randomly crop an image, then flip it randomly, and finally, resize it to the desired size. The code is as follows (since we have random elements, we set the random seed for reproducibility):

```
>>> torch.manual_seed(1)
>>> fig = plt.figure(figsize=(14, 12))
>>> for i, (img, attr) in enumerate(celeba_train_dataset):
...     ax = fig.add_subplot(3, 4, i*4+1)
...     ax.imshow(img)
...     if i == 0:
...         ax.set_title('Orig.', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+2)
...     img_transform = transforms.Compose([
...         transforms.RandomCrop([178, 178])
...     ])
...     img_cropped = img_transform(img)
...     ax.imshow(img_cropped)
...     if i == 0:
...         ax.set_title('Step 1: Random crop', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+3)
...     img_transform = transforms.Compose([
...         transforms.RandomHorizontalFlip()
...     ])
...     img_flip = img_transform(img_cropped)
...     ax.imshow(img_flip)
...     if i == 0:
...         ax.set_title('Step 2: Random flip', size=15)
```