

CITS5508 Machine Learning

Lectures for Semester 1, 2021

Lecture Week 4: Book Chapter 5, SVMs

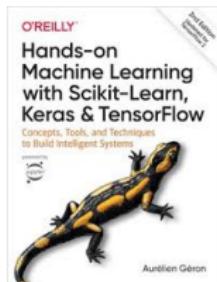
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 5.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Chapter Five

Support Vector Machines

In this chapter, we will explain the core concepts of SVMs, how to use them, and how they work.

Here are the main topics we will go cover:

- Linear SVM Classification
- Nonlinear SVM Classification
- SVM Regression
- Under the Hood

SVMs

A Support Vector Machine (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex but small- or medium-sized datasets.

Linear SVM Classification

You can think of an **SVM classifier** as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

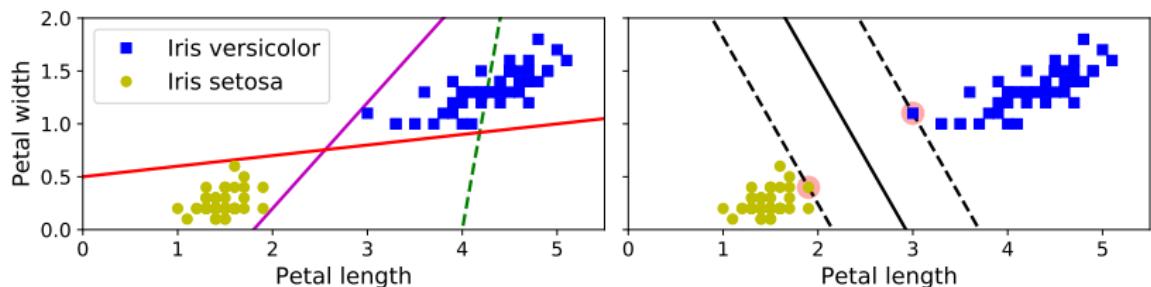


Figure 5-1. Large margin classification

Linear SVM Classification

Notice that adding more training instances “off the street” will not affect the decision boundary at all: it is fully determined (or “supported”) by the instances located on the edge of the street. These instances are called the *support vectors*.

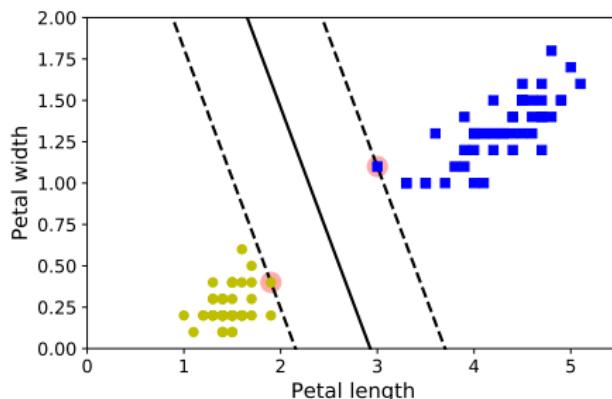


Figure 5-1. Large margin classification

Sensitive to Scale

SVMs are sensitive to the feature scales. On the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After [feature scaling](#) (e.g., using Scikit-Learn's [StandardScaler](#)), the decision boundary looks much better (on the right plot).

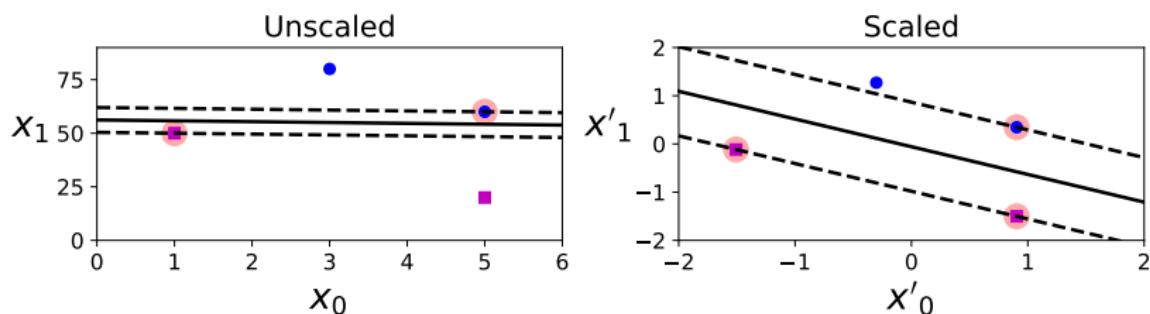
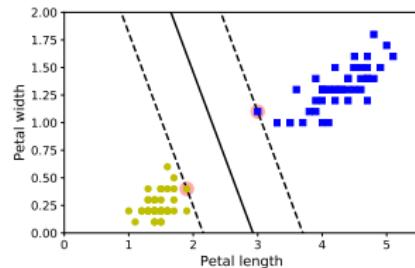


Figure 5-2. Sensitivity to feature scales

Hard Margin Classification

If we strictly impose that all instances be off the street and on the correct side, this is called *hard margin classification*.

The hard margin linear SVM classifier objective can be expressed as a *constrained optimization problem*:



$$\underset{\mathbf{w}, b}{\text{minimize}} \frac{1}{2} \mathbf{w}^\top \mathbf{w}$$

$$\text{subject to: } t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1, \text{ for } i = 1, 2, \dots, m$$

where

$$t^{(i)} = \begin{cases} +1 & \text{if training data } i \text{ is a positive instance} \\ -1 & \text{if training data } i \text{ is a negative instance} \end{cases}$$

Hard Margin Classification (cont.)

- However, *hard margin classification* only works if the data is linearly separable.
- Furthermore, it is quite sensitive to outliers.

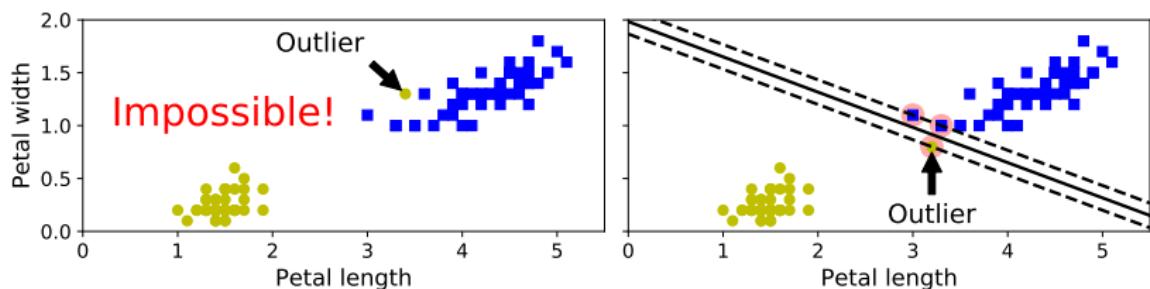


Figure 5-3. Hard margin sensitivity to outliers

Soft Margin Classification

- To avoid the issues mentioned on the previous slide, we can use a more flexible model – soft margin classification.
- The objective of *soft margin classification* is to find a good balance between keeping the street as large as possible and limiting the margin violations:

$$\underset{\mathbf{w}, b, \zeta}{\text{minimize}} \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)}$$

subject to: $t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)}$ and $\zeta^{(i)} \geq 0$, for $i = 1, 2, \dots, m$

where $t^{(i)} = \begin{cases} +1 & \text{if training data } i \text{ is a positive instance} \\ -1 & \text{if training data } i \text{ is a negative instance} \end{cases}$

$\zeta^{(i)}$ is a *slack variable*, which measures how much the i^{th} instance is allowed to violate the margin.

NOTE: C is the hyperparameter which allows us to define the trade-off between the two objectives $\mathbf{w}^\top \mathbf{w}$ and $\sum_{i=1}^m \zeta^{(i)}$.

Soft Margin Classification – the C hyperparameter

Use the C hyperparameter to adjust the balance between the two objectives.

- A small C value penalizes less on the points falling onto the wrong side of the street, leading to “wider street” and more *margin violations*.
- A large C value penalizes more on points falling onto the wrong side of the street, leading to “narrower street” and less margin violations.

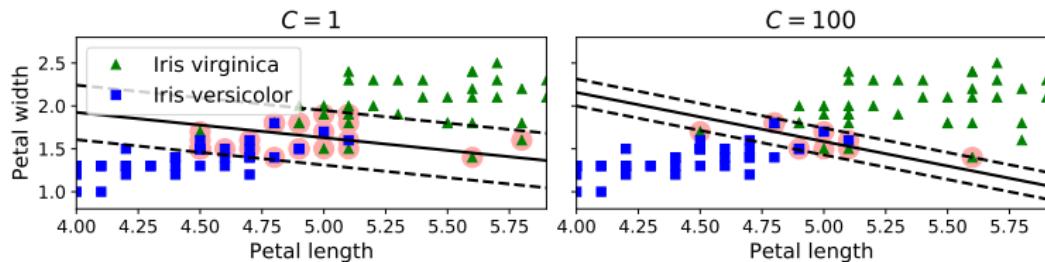


Figure 5-4. Large margin (left) versus fewer margin violations (right)

Soft Margin Classification – An example

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]      # petal length, petal width
y = (iris["target"] == 2).astype(np.float64) # Iris virginica

scaler = StandardScaler()
X = scaler.fit_transform(X)      # normalize X

clf = LinearSVC(C=1, loss=hinge) # try a small C value
clf.fit(X,y)                  # train the SVM classifier
```

The *hinge loss* function is a loss function used in maximum-margin classification, typically SVMs. It is defined as $\text{hinge_loss}(y) = \max(0, 1 - ty)$, where $t = \pm 1$ is the ground truth label and y is the classification score.

```
>>> svm_clf.predict([[5.5, 1.7]])
array([1.])
```

Nonlinear SVM Classification – using Polynomial Features

Many datasets are not even close to being linearly separable.
One approach to handling nonlinear datasets is to add more features, such as **polynomial features**.
In some cases this can result in a linearly separable dataset.

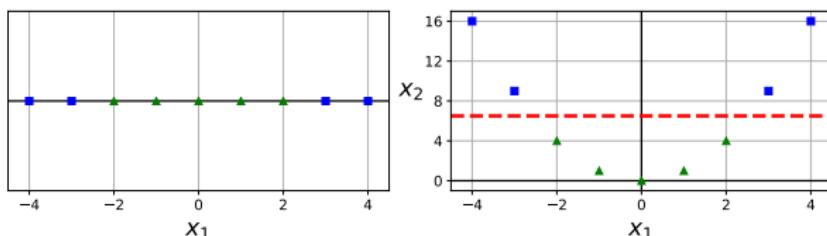


Figure 5-5. Adding features to make a dataset linearly separable

Another example

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])
polynomial_svm_clf.fit(X, y)
```

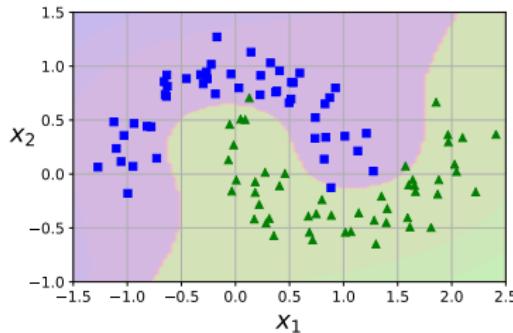


Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

Using polynomial features can make the model training very slow. Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick*. It makes it possible to get the same result as if you added many polynomial features, even with very high-degree polynomials, without actually having to add them.

For example, to use a polynomial kernel (of degree 3):

```
SVC(kernel="poly", degree=3, coef0=1, C=5)
```

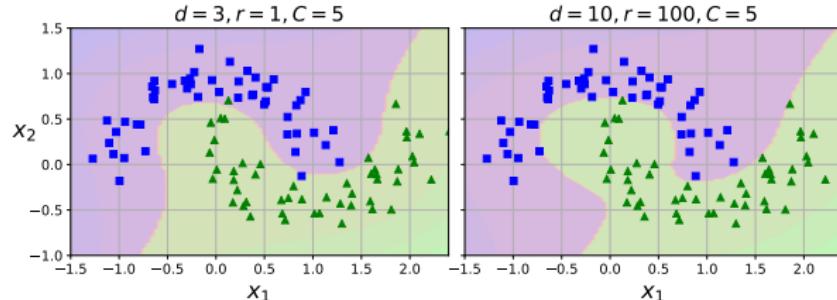


Figure 5-7. SVM classifiers with a polynomial kernel

Adding similarity functions

Another technique to tackle nonlinear problems is to add features computed using a similarity function that measures how much each instance resembles a particular landmark. For example, let's take the one-dimensional dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (left plot). Next, let's define the similarity function to be the *Gaussian Radial Basis Function (RBF)* with $\gamma = 0.3$.

$$\phi_\gamma(x, \ell) = \exp(-\gamma \|x - \ell\|^2)$$

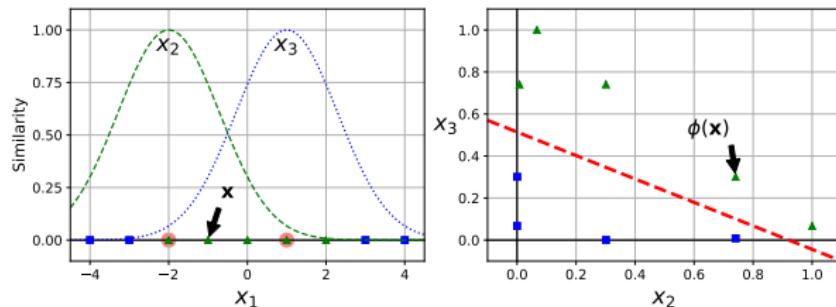


Figure 5-8. Similarity features using the Gaussian RBF

Gaussian RBF Kernel

The kernel trick again makes it possible to obtain a similar result as if you had added many similarity features, without actually having to add them.

Try the Gaussian RBF kernel using the **SVC** class.

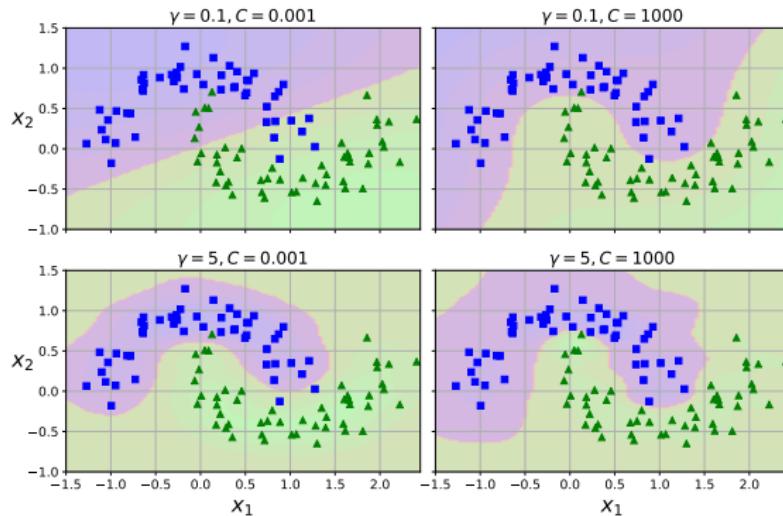


Figure 5-9. SVM classifiers using an RBF kernel

Computational Complexity

- `LinearSVC` implements an optimized algorithm for *linear SVMs*. No kernel trick, but it scales almost linearly with the number of training instances and the number of features.
- `SVC` implements an algorithm that supports the kernel trick. Unfortunately, high training time complexity means that it gets dreadfully slow when the number of training instances gets large.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
<code>LinearSVC</code>	$O(m \times n)$	No	Yes	No
<code>SGDClassifier</code>	$O(m \times n)$	Yes	Yes	No
<code>svc</code>	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Regression

Instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street).

```
from sklearn.svm import LinearSVR  
svm_reg = LinearSVR(epsilon=1.5)  
svm_reg.fit(X, y)
```

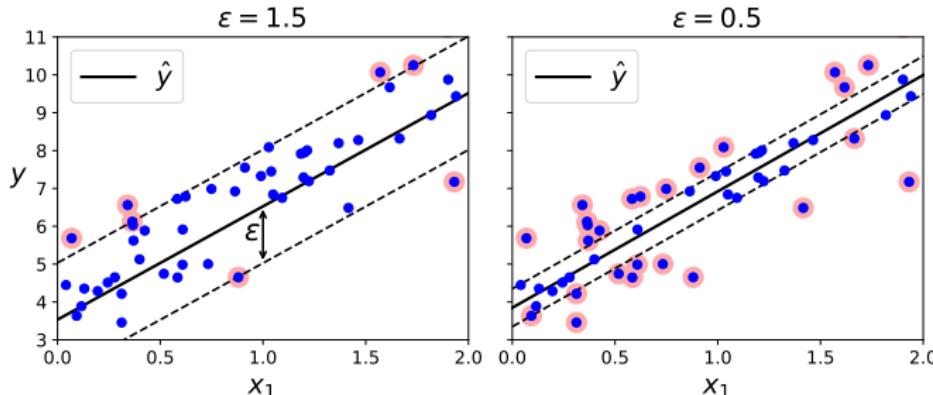


Figure 5-10. SVM Regression

Non-linear Regression

For nonlinear regression use a kernelized SVM model. E.g SVM Regression on a random quadratic training set, using a 2nd-degree polynomial kernel. There is little regularization on the left plot (i.e., a large C value), and much more regularization on the right plot (i.e., a small C value).

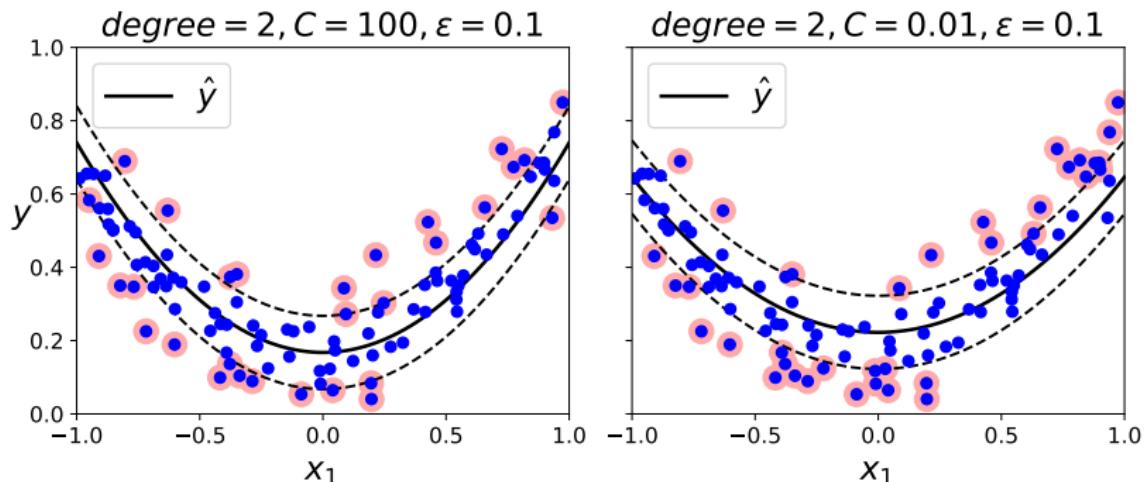


Figure 5-11. SVM Regression using a second-degree polynomial kernel

Under the Hood

Rest of chapter explains how SVMs make predictions and how their training algorithms work.

This has some heavy maths, beyond the scope of the course. We just take a high level overview.

Note: In this chapter, the bias term will be called b and the feature weights vector will be called w . No bias feature ($x_0 = 1$) will be added to the input feature vectors.

Decision Function and Predictions

The linear SVM classifier model predicts the class of a new instance x by simply computing the decision function $\mathbf{w}^T \mathbf{x} + b = w_1x_1 + \dots + w_nx_n + b$ and predict the class label \hat{y} as follows:

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

Training Objective

We want to choose \mathbf{w} and ζ to

$$\underset{\mathbf{w}, b, \zeta}{\text{minimize}} \frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)}$$

subject to: $t^{(i)} (\mathbf{w}^\top \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)}$ and $\zeta^{(i)} \geq 0$, for $i = 1, 2, \dots, m$

where the $t^{(i)}$ are ± 1 to indicate positive or negative training instances.

We observe that smaller \mathbf{w} gives wider roads:

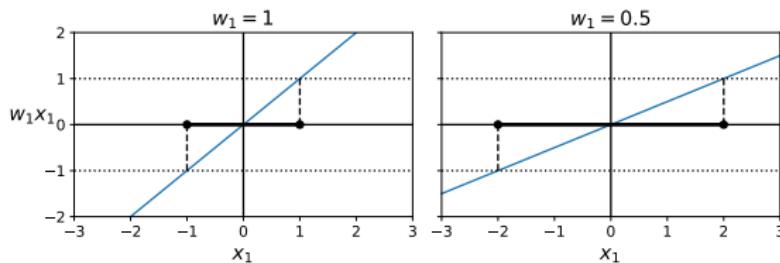


Figure 5-13. A smaller weight vector results in a larger margin

So we minimize $\|\mathbf{w}\|$ to get a larger margin.

Quadratic Programming

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *Quadratic Programming (QP) problems*.

Many off-the-shelf solvers are available to solve QP problems using a variety of techniques that are outside the scope of this unit.

Text book shows how to write the hard margin linear SVM classifier problem in QP form, so you can just use an off-the-shelf QP solver. (Soft margin problem is an exercise).

The Dual Problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*.

The dual form of the linear SVM objective is:

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

subject to all $\alpha^{(i)} > 0$, for $i = 1, \dots, n$.

The dual problem is faster to solve than the primal when the number of training instances is smaller than the number of features. More importantly, it makes the *kernel trick* possible, while the primal does not.

Kernalized SVM, “The Kernel Trick”

Suppose that we want to transform our features $(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)})$ using the function ϕ (e.g., ϕ can be a 2nd degree polynomial) to a higher dimensional space. We then train a linear SVM classifier on the transformed training set.

Using the dual form on the previous slide, in the new transformed space, it seems that we will need to compute the dot product $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$. However, if ϕ is the 2nd-degree polynomial transformation above, then you can simply replace this dot product of the transformed vectors by $(\mathbf{x}^{(i)T} \mathbf{x}^{(j)})^2$.

The function $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ is called a 2nd-degree polynomial kernel. In Machine Learning, a kernel is a function capable of computing the dot product $\phi(\mathbf{a})^T \phi(\mathbf{b})$ based only on the original vectors \mathbf{a} and \mathbf{b} , without having to compute (or even to know about) the transformation ϕ .

Common kernels used in SVM

Linear:	$K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$
Polynomial of degree d:	$K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \mathbf{b} + r)^d$
Gaussian RBF:	$K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \ \mathbf{a} - \mathbf{b}\ ^2 + r)$
Sigmoid:	$K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \mathbf{b} + r)$

where γ , d , and r are hyperparameters.

(The hyperparameter r is the `coef0` parameter in the `SVC` and `SVR` classes of Scikit-learn)

Summary for Chapter 5

- Linear SVM Classification: hard margin vs soft margin
- Nonlinear SVM Classification: polynomial kernels, similarity features, Gaussian RBF, Complexity
- SVM Regression
- Under the Hood: decision function, training objectives, the kernel trick

For next week

Work through the lab sheet and attend the supervised lab. The Unit Coordinator and a casual Teaching Assistant will be there to help most of that time.

Prepare for the Mid-semester test by studying Chapters 1 to 5.

Read Chapter 6 on Decision Trees.

And that's all for the fourth lecture.

Have a good week.

CITS5508 Machine Learning

Lectures for Semester 1, 2021

Lecture Week 5: Book Chapter 6, Decision Trees

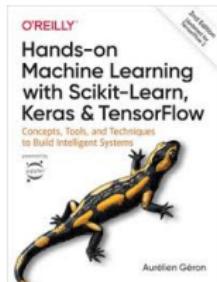
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 6.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Decision Trees

In this chapter, we will explain the core concepts of Decision Trees (DTs), how to use them, and how they work.

Like SVMs, DTs are versatile ML algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are very powerful algorithms, capable of fitting complex datasets. E.g., recall the housing data example from Ch2.

DTs are also the fundamental components of Random Forests (RF, Ch7), which are among the most powerful ML algorithms available today.

Topics

Here are the main topics we will go cover:

- Training and Visualising a Decision Tree
- Making Predictions
- Estimating Class Probabilities
- The CART Training Algorithm
- Computational Complexity
- Gini Impurity or Entropy
- Regularization Hyperparameters
- Regression
- Instability

Training and Visualising a Decision Tree

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target
tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Then can use `export_graphviz()` method to output a graph definition file and display using graphviz package.

Decision Tree Visualized

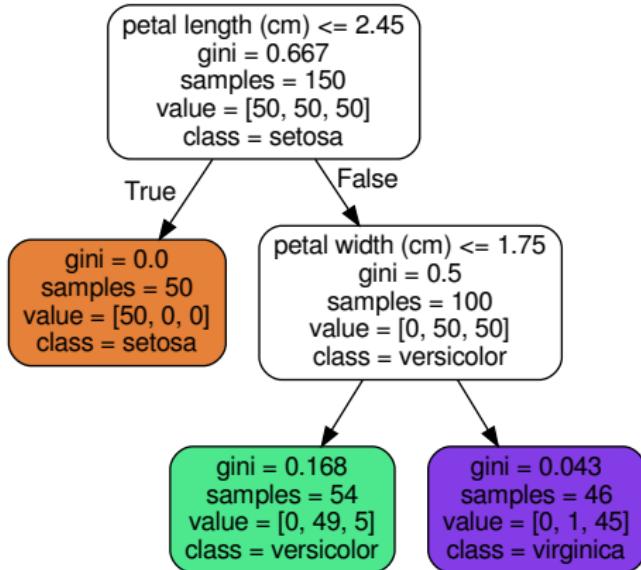


Figure 6-1. Iris Decision Tree

(Note: To run this chapter's Python code of the author, you must install an additional package using the command: `conda install python-graphviz`.)

Making Predictions

- Use the tree to make a decision for a single instance by following the sequence of questions and answers down from the node, left or right depending on the answer.
- Class of leaf node is the output class.
- Tree nodes also tell us number of samples allocated to each node, and broken down by class.
- *Gini impurity* (low if most instances from just one class):

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

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

A “White-Box” Classifier: (i.e. human understandable)

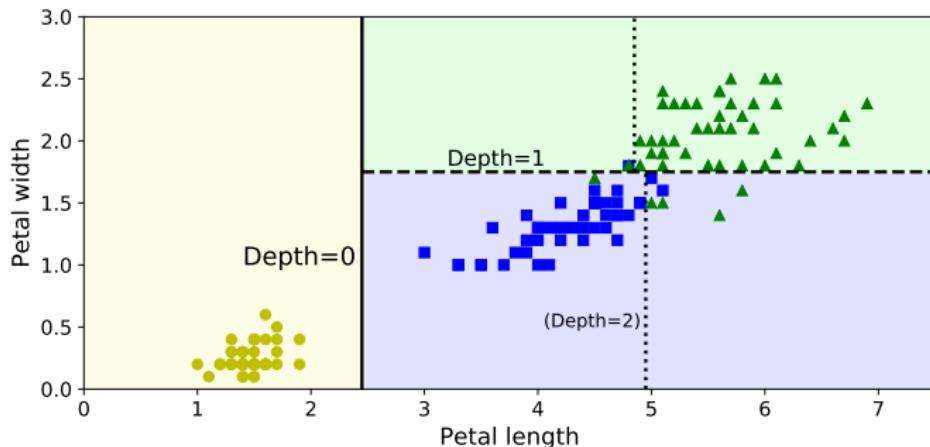


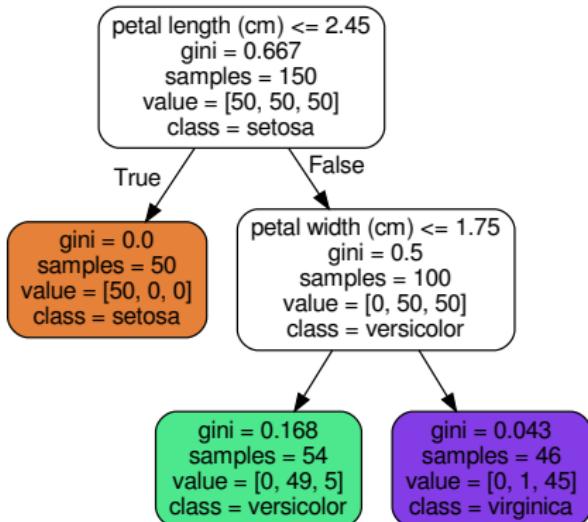
Figure 6-2. Decision Tree decision boundaries

(Refer to Figure 6-1)

- *White box models* – e.g., decision trees
- *Black box models* – e.g., random forests, neural networks

Estimating Class Probabilities

You can use the class breakdowns in each leaf node to get estimates of the probabilities of class membership for each instance that ends up at a certain leaf.



```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. , 0.90740741, 0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

The CART Training Algorithm

Scikit-Learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees (also called “growing” trees). The idea is really quite simple: the algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (e.g., “petal length” ≤ 2.45 cm?). How does it choose k and t_k ? It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). The cost function that the algorithm tries to minimize is given by

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

Note: we use a *Greedy Algorithm* to get a reasonable solution, not necessarily optimal (which would take too long to find).

Computational Complexity

Making predictions requires traversing the Decision Tree from the root to a leaf. Decision Trees are generally approximately balanced, so traversing the Decision Tree requires going through roughly $O(\log_2(m))$ nodes. Since each node only requires checking the value of one feature, the overall prediction complexity is just $O(\log_2(m))$, independent of the number of features. So predictions are very fast, even when dealing with large training sets. However, the training algorithm compares all features (or less if `max_features` is set) on all samples at each node. This results in a training complexity of $O(nm \log_2(m))$. For small training sets (less than a few thousand instances), Scikit-Learn can speed up training by presorting the data (set `presort=True`), but this slows down training considerably for larger training sets.

(m = number of leaf nodes; n = number of features)

Gini Impurity or Entropy

- By default, the Gini impurity measure is used
- *entropy* is an alternative (by setting the `criterion` hyperparameter to “`entropy`”)
- From thermodynamics: *entropy* is a measure of molecular disorder
- Later more widespread use, e.g., in *Shannon's information theory* it measures the average information content of a message
- frequently used as an impurity measure in ML: a set's entropy is zero when it contains instances of only one class.

Gini Impurity or Entropy

- Definition of Entropy H_i of node i :

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

- Most of the time it does not make a big difference which one you use: they lead to similar trees.
- Gini impurity is slightly faster to compute, so good default.
- But when they differ, entropy tends to produce slightly more balanced trees.

Regularization Hyperparameters

- DTs make few assumptions about the training data: they are thus referred to as *nonparametric models*, very (maybe too) adaptive.
- To avoid overfitting, we need to restrict DTs' freedom during training, e.g., using `regularization` or set the `max_depth` hyperparameter smaller.
- `DecisionTreeClassifier` has other similar parameters:
`min_samples_split`, `min_samples_leaf`,
`min_weight_fraction_leaf`, `max_leaf_nodes`.
- Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters regularizes the model.

Example Regularization of a Decision Tree

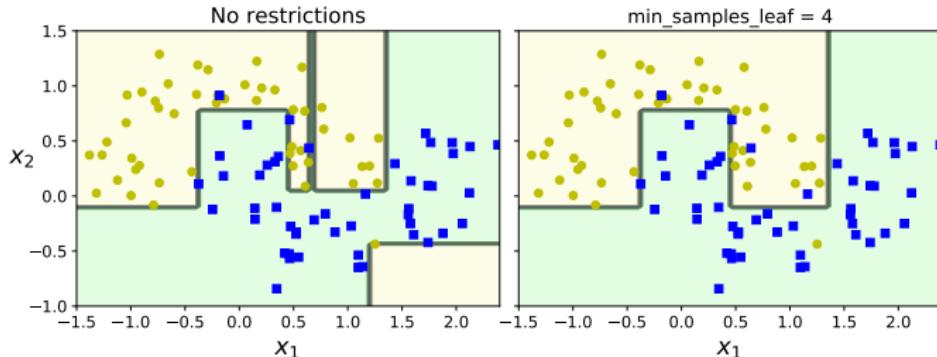


Figure 6-3. Regularization using `min_samples_leaf`

Two Decision Trees trained on the moons dataset are shown above.

- Left: Decision tree trained with the default hyperparameters (i.e., no restrictions)
- Right: Decision tree trained with `min_samples_leaf=4`.

It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.

Regression

Example tree trained on a noisy quadratic (`max_depth=2`).

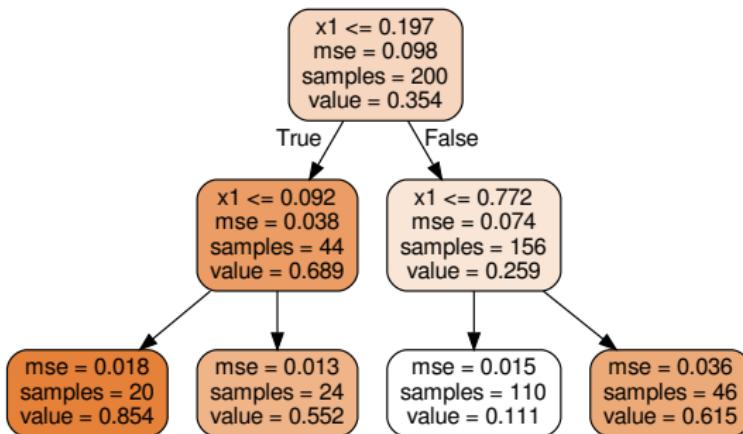


Figure 6-4. A Decision Tree for regression

```
from sklearn.tree import DecisionTreeRegressor  
  
tree_reg = DecisionTreeRegressor(max_depth=2)  
tree_reg.fit(X, y)
```

... Or with depth 3 ...

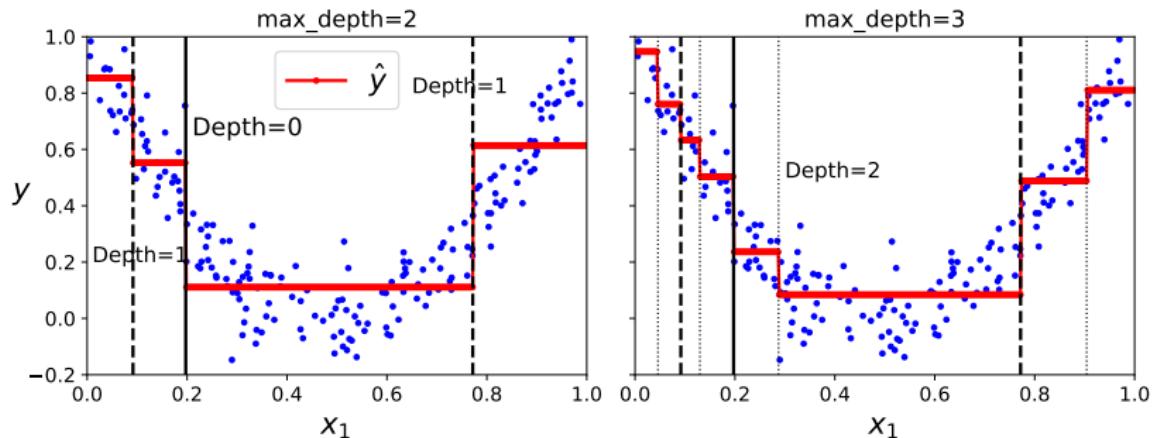


Figure 6-5. Predictions of two Decision Tree regression models

Training consists of trying to minimize the MSE.

Prone to overfitting so use regularization

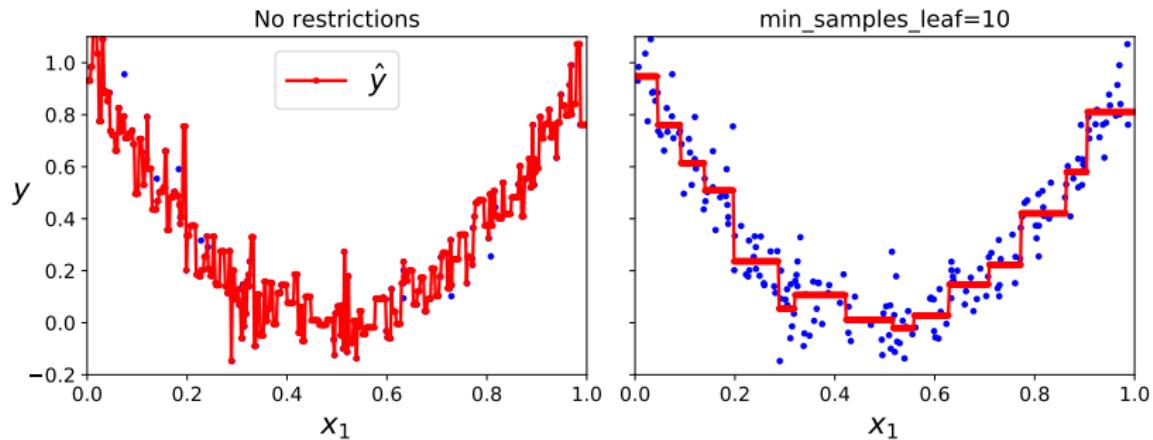


Figure 6-6. Regularizing a Decision Tree regressor

- Left: without any regularization (i.e., using the default hyperparameters). Overfitting the training data is obvious.
- Right: minimum number of samples per leaf node is set to 10, resulting in a much more reasonable model.

Instability: e.g., sensitivity to rotation

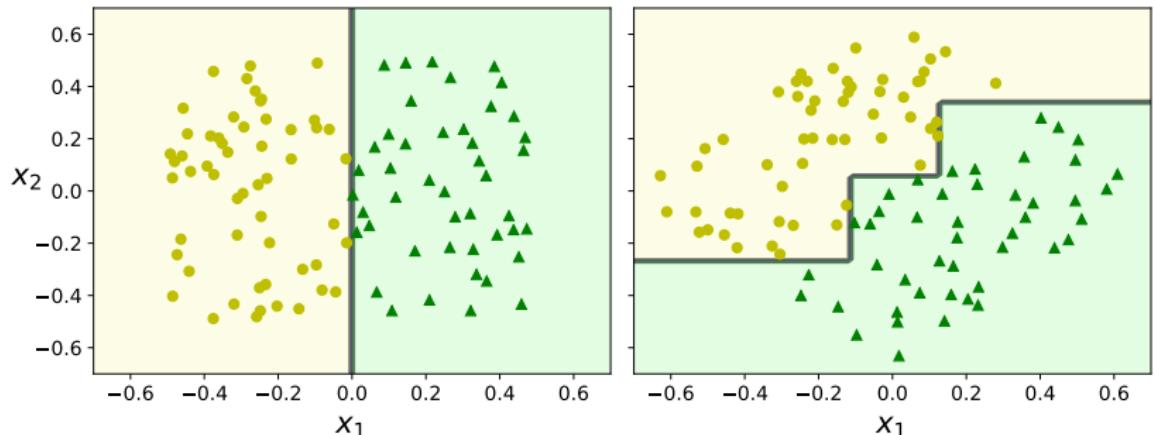


Figure 6-7. Sensitivity to training set rotation

(later in the unit we see PCA which can help)

Instability: DTs also sensitive to small changes in data

E.g., retrain DT with just one instance

($petal_length=4.8\text{cm}$, $petal_width=1.8\text{cm}$) removed.

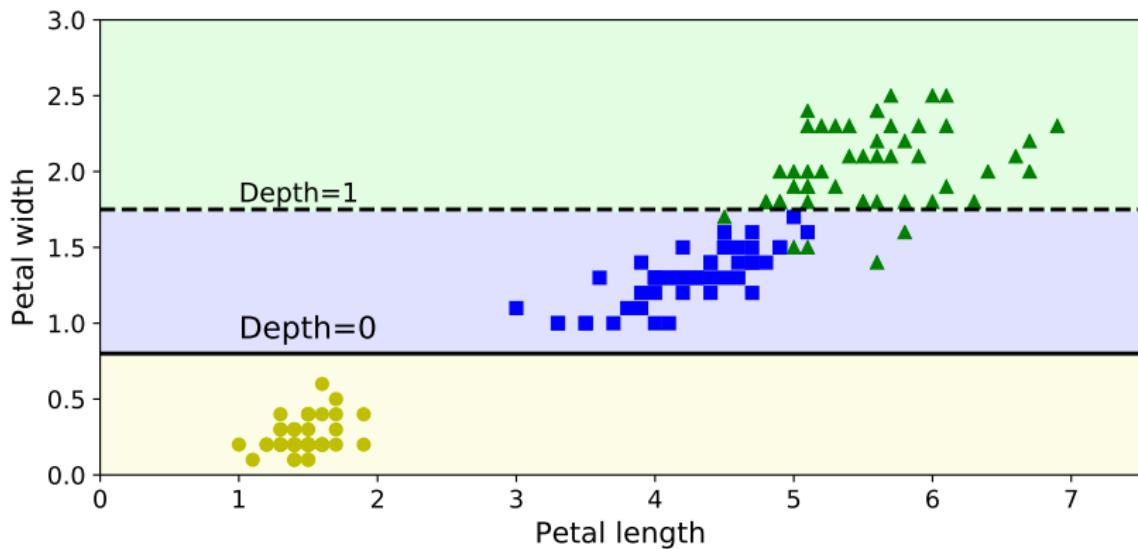


Figure 6-8. Sensitivity to training set details

(see [random forests \(RF\)](#) to overcome this)

Summary for Chapter 6

- Training and Visualising a Decision Tree
- Making Predictions
- Estimating Class Probabilities
- The CART Training Algorithm
- Computational Complexity
- Gini Impurity or Entropy
- Regularization Hyperparameters
- Regression
- Instability

For next week

Work through the lab sheet and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read Chapter 7 on Ensemble Learning and Random Forests.

And that's all for the fifth lecture.

Have a good week.

CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Weeks 5&6: Book Chapter 7

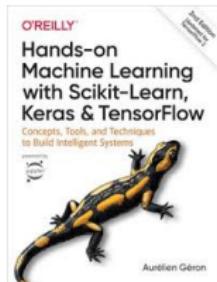
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 7.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Ensemble Learning and Random Forests

In this chapter, we will look at how different classifiers can be combined to improve the performance of the algorithm. Here are the main topics we will cover:

- Voting Classifiers
- Bagging and Pasting
- Random Patches and Random Subspaces
- Random Forests
- Boosting
- Stacking

Ensemble Learning

Suppose that you are seeking the answer to a complex question. You could ask the question to thousands of random people and then aggregate their answers. In many cases you will find that this aggregated answer is better than the answer from a single expert.

This is called the *wisdom of the crowd*.

Ensemble learning adopts this similar concept.

Ensemble Learning

Given that you have implemented a group of predictors (such as classifiers or regressors), if you aggregate the predictions of all the predictors together, you will often get better predictions than with the best individual predictor.

- A group of predictors is called an *ensemble*.
- This technique is therefore called *Ensemble Learning*.
- An Ensemble Learning algorithm is called an *Ensemble method*.
e.g., a *Random Forest* is an ensemble of Decision Trees.

Voting Classifiers

Suppose that you have trained a number of classifiers, each one achieving about 80% accuracy.

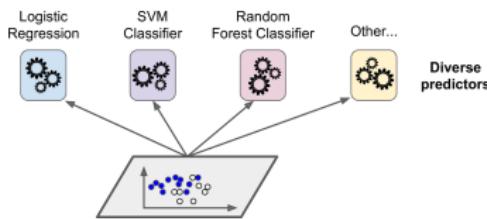


Figure 7-1. Training diverse classifiers

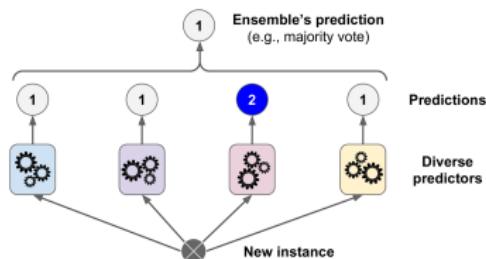


Figure 7-2. Hard voting classifier predictions

You can aggregate the predictions of the classifiers and predict the class that gets the most votes.
This *majority-vote classifier* is called a *hard voting classifier*.

Voting Classifiers

Surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a **weak learner**.

That is, the ensemble can be a **strong learner**, provided that there are enough weak learners and they are sufficiently diverse.

Here, **diversity** means that the classifiers are perfectly independent and making uncorrelated errors.

One way to get diverse classifiers is to train them using very different algorithms.

A Voting Classifier Example

Instead of a *hard voting classifier*, we can implement a *soft voting classifier* if all the classifiers are able to estimate the class probabilities (i.e., they have a `predict_proba()` method). We can tell Scikit-Learn to predict the class with the highest class probability, averaged over all the classifiers.

- **Training a voting classifier:**

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC(probability=True)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
voting_clf.fit(X_train, y_train)
```

A Voting Classifier Example (cont.)

- **Testing individual classifier's accuracy versus the voting classifier's accuracy:**

```
from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

- **Output:**

```
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.912
```

Hard Voting vs Soft Voting

- **Hard voting** – Ensemble's predicted class is the class that receives the highest vote.
- **Soft voting** – Ensemble's predicted class is the class that receives the highest average class probability.

(See lecture recording)

Bagging and Pasting

We mentioned earlier that to get a diverse set of classifiers we should use very different training algorithms. Another approach is to **use the same training algorithm** for every classifier but to train each classifier on **different random subsets of the training set**. This means that we need to draw random samples from the training set.

There are two sampling methods:

- ① **Sampling with replacement** – this method is called *bagging*¹ (short for **bootstrap aggregating**)
- ② **Sampling without replacement** – this method is called *pasting*

(Note that we use the terms “predictor” and “classifier” interchangeably)

¹ “Bagging Predictors,” L. Breiman (1996)

Bagging and Pasting

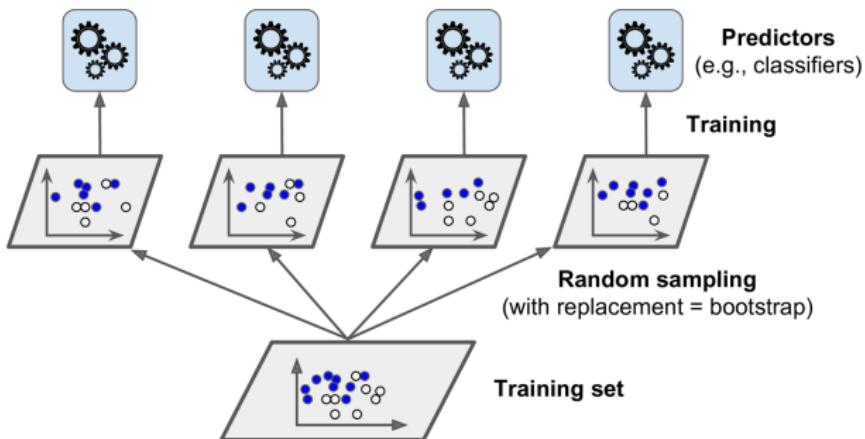


Figure 7-4. Bagging and pasting involves training several predictors on different random samples of the training set

Once all the predictors are trained, the ensemble can make a prediction for a new instance by aggregating the predictions from all the predictors. Typically, the *statistical mode* (i.e., the most frequent prediction) is used as the aggregation function.

Bagging and Pasting

Each individual predictor has a higher **bias** than if it were trained on the original training set, but aggregation reduces both **bias** and **variance**.

In general, the net result is: the ensemble has a similar bias but a lower variance than a single predictor trained on the original whole training set.

Bagging and pasting are popular methods because they scale very well:

- the predictors can be trained in parallel, using different CPU cores or even on different servers.
- the predictors can also be tested in parallel.

Bagging and Pasting in Scikit-Learn

The `BaggingClassifier` class (or `BaggingRegressor` for regression) of Scikit-Learn implements bagging and pasting.

In the example code below, we design an ensemble to have 500 predictors, each trained on 100 randomly drawn samples with replacement (i.e., bagging):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

To use sampling without replacement (i.e., pasting), set `bootstrap=False`.

By default, `BaggingClassifier` automatically performs soft voting if the base classifier can estimate class probabilities.

Decision Boundary

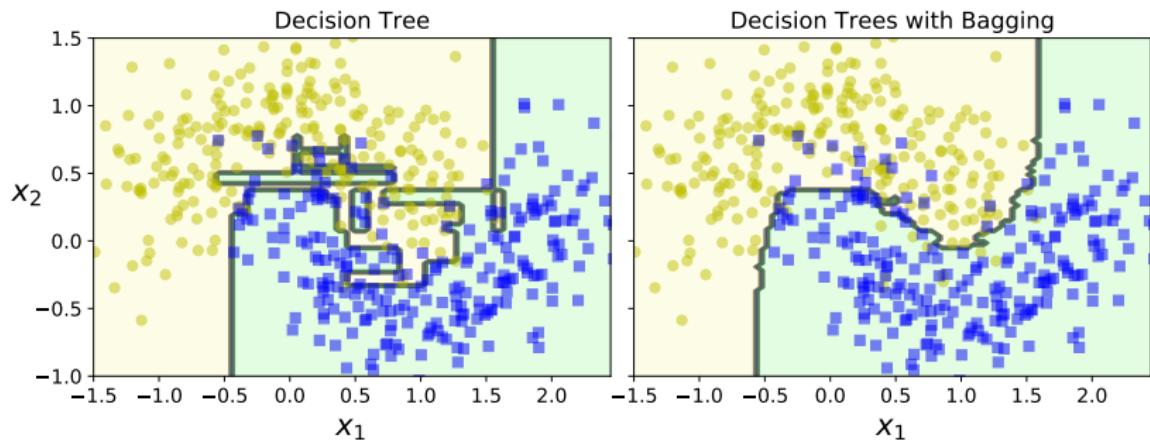


Figure 7-5. A single Decision Tree (left) versus a bagging ensemble of 500 trees (right)

Comparison of the decision boundary. Left: A single decision tree. Right: A bagging ensemble of 500 trees. Both were trained on the moons dataset.

Comparing Bagging and Pasting

- Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so *bagging* ends up with a slightly higher bias than *pasting*.
- However, the point above also means that the predictors of *bagging* are less correlated, so the ensemble has lower variance if *bagging* is used.
- Overall, *bagging* often results in better models, so it is generally more preferred than *pasting*.

One good exercise is to use cross validation to evaluate both *bagging* and *pasting* on your own problem data and select the one that works best.

Out-of-Bag Evaluation

By default, a `BaggingClassifier` samples training instances with replacement (i.e., option `bootstrap=True`). With bagging, some instances may be sampled several times while others may not be sampled at all. On average, only about 63% of the training instances are sampled for each predictor. The remaining 37% that are not sampled are called *out-of-bag (oob)* instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, we can use these instances to form our validation set for cross validation.

Out-of-Bag Evaluation in Scikit-Learn

To request an automatic oob evaluation after training, we can set the option `oob_score=True` when creating a `BaggingClassifier`:

```
>>> bag_clf = BaggingClassifier(  
... DecisionTreeClassifier(), n_estimators=500,  
... bootstrap=True, n_jobs=-1, oob_score=True)  
...  
>>> bag_clf.fit(X_train, y_train)
```

We can inspect the oob score via

```
>>> bag_clf.oob_score_  
0.9013333333333332
```

and the oob decision function via

```
>>> bag_clf.oob_decision_function_  
array([[ 0.31746032,  0.68253968],  
[ 0.34117647,  0.65882353],  
...  
[ 0.57291667,  0.42708333]])
```

Random Patches and Random Subspaces

The BaggingClassifier class has two hyperparameters:

- `max_samples`, for controlling the maximum number of samples for each classifier in the ensemble; and
- `bootstrap`, for specifying whether to use bagging or pasting.

The BaggingClassifier class also supports `sampling the features`. This is controlled by two hyperparameters `max_features` and `bootstrap_features`. This is useful when you are dealing with high-dimensional inputs (such as images).

Random Patches and Random Subspaces (cont.)

Random Patches method and *Random Subspaces* method are ways of sampling the features.

- **Sampling both training instances and features** is called the **Random Patches** method.
- **Keeping all training instances** (`bootstrap=False` and `max_samples=1.0`) but **sampling features** (`bootstrap_features=True` and/or `max_features < 1.0`) is called the **Random Subspaces** method.

Random Patches and Random Subspaces (cont.)

Random Patches method – each predictor samples both instances and features

feat. 1	feat. 2	...	feat. <i>i</i>	...	feat. <i>j</i>	...	feat. <i>n</i>
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx

Hyperparameter settings: `bootstrap = True`, `max_samples < 1.0`,
`bootstrap_features = True`, and `max_features < 1.0`.

Random Patches and Random Subspaces (cont.)

Random Subspaces method – each predictor samples only features

feat. 1	feat. 2	...	feat. <i>i</i>	...	feat. <i>j</i>	...	feat. <i>n</i>
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
...
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx
xxx	xxx	xxx	xxx	xxx	xxx	xxx	xxx

Hyperparameter settings: `bootstrap = False`, `max_samples = 1.0`,
`bootstrap_features = True`, and `max_features < 1.0`.

Random Patches and Random Subspaces (cont.)

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

(See lecture recording)

Which one to use? Random Patches method? or Random Subspaces method?

Rule of thumb: large dataset having many (sufficient) instances, then try Random Patches; otherwise, try Random Subspaces.

Random Forests

A *Random Forest*² (RF) is **an ensemble of Decision Trees**, generally trained via the bagging method (or sometimes pasting), typically with `max_samples=1.0` (i.e., the entire training set).

We can build a *random forest classifier* by building a `DecisionTreeClassifier` and pass it to a `BaggingClassifier`. However, it is more convenient and optimized for Decision Trees by using the `RandomForestClassifier` class available in Scikit-Learn. Similarly, there is a `RandomForestRegressor` class for regression.

Example:

```
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                  n_jobs=-1)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

² "Random Decision Forest," T. Ho (1995)

Random Forests (cont.)

A `RandomForestClassifier` has almost all the hyperparameters of a `DecisionTreeClassifier` plus all the hyperparameters of a `BaggingClassifier`.

The RF algorithm introduces extra randomness when growing trees: instead of searching for the best feature when splitting a node (see previous topic), it searches for the best feature among the random subset of features.

- ⇒ greater tree diversity (which trades a higher bias for a lower variance)
- ⇒ generally yielding an overall better model.

The following `BaggingClassifier` is roughly equivalent to the previous `RandomForestClassifier`:

```
bag_clf = BaggingClassifier(  
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),  
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

Extra Trees

It is possible to make a RF tree even more random by using random thresholds for each feature when splitting the node (rather than searching for the best possible thresholds like other Decision Trees do).

A forest with such an extremely random trees is called an *Extremely Randomized Trees*³ (or *Extra-Trees* for short). Again, this trades more bias for a lower variance. It also makes Extra-Trees much faster to train than regular RFs.

Scikit-Learn has the `ExtraTreesClassifier` class to support the creation of Extra-Trees classifiers. Similarly, the `ExtraTreesRegressor` class is for regression tasks.

³ "Extremely Randomized Trees." P. Geurts, D. Ernst, L. Wehenkel (2005)

Feature Importance

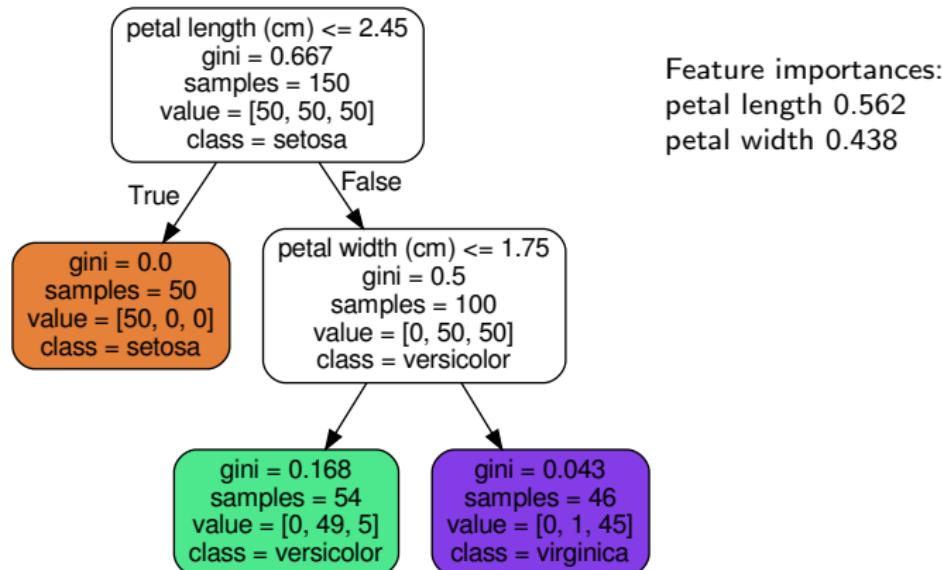
Scikit-Learn measures a feature's (say x_i) importance as follows:

- collects all the nodes (in all the trees) that use x_i for node splitting;
- computes the impurity reduction at each of these nodes, weighted by the number of training instances associated with the node;
- computes the average impurity reduction over all these nodes.

Scikit-Learn computes this score automatically for each feature x_i after training, then it scales the results so that the sum of all importances is equal to 1.

We can access this result via the `feature_importances_` variable.

Feature Importance (cont.)



A decision tree of `max_depth=2` on the Iris dataset.

Feature Importance (cont.)

Example:

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"], rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

The result shows that the most important features are the petal length (44%) and petal width (42%) while the sepal length and width are rather unimportant.

*** RFs are very handy to get a quick understanding of what features actually matter, particularly if you need to perform feature selection and/or feature pruning.

Feature Importance (cont.)

Similarly, we can train a RF classifier on the MNIST dataset and visualize the importance of each pixel:

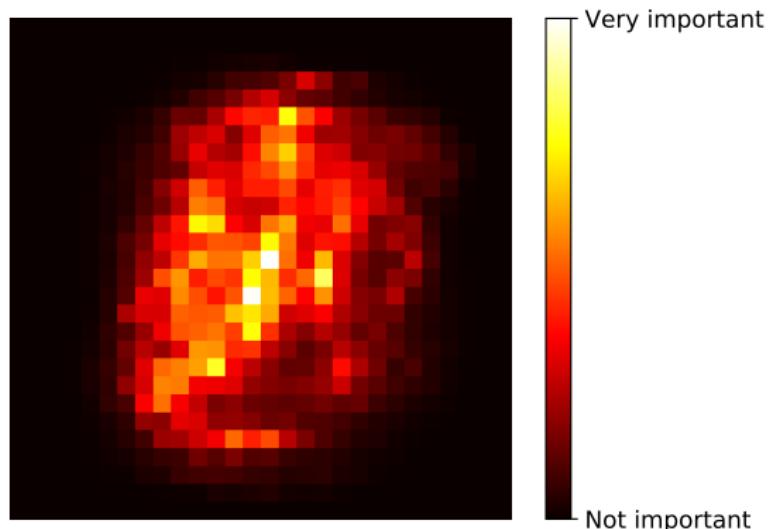


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

Boosting

Boosting (originally called *hypothesis boosting*) refers to any Ensemble method that can combine several weak learners into a strong learner.

The general idea of most boosting methods is to train the predictors **sequentially**, each trying to correct the predecessor.

Many boosting methods are available. By far the most popular are *Adaptive Boosting*⁴ (*AdaBoost* for short) and *Gradient Boosting*.

⁴ "A Decision-Theoretic Generalization of On-Line Learning and an Application in Boosting," Yoav Freund, Robert E. Schapire (1997)

AdaBoost

For example, a first base classifier is trained and used to make predictions on the training set. The relative weights of the misclassified training instances are then increased. A second classifier is trained using the updated weights and then makes predictions and so on.

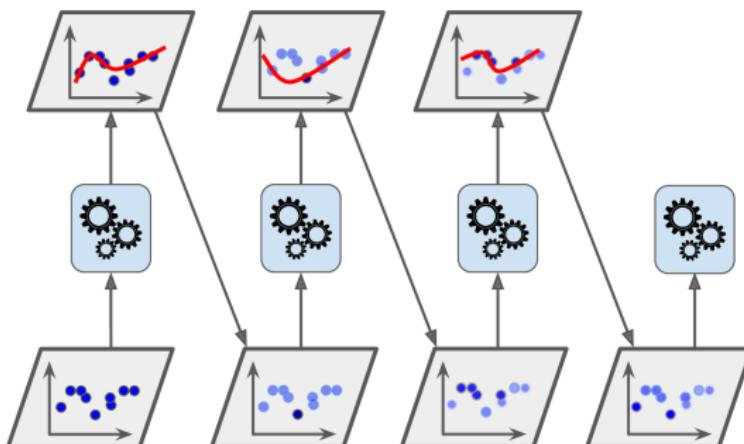


Figure 7-7. AdaBoost sequential training with instance weight updates

AdaBoost (cont.)

The example below shows the decision boundary of consecutive predictors on the *moons* dataset. Here, each predictor is a highly regularized SVM classifier with an RBF (stands for *radial basis function*) kernel.

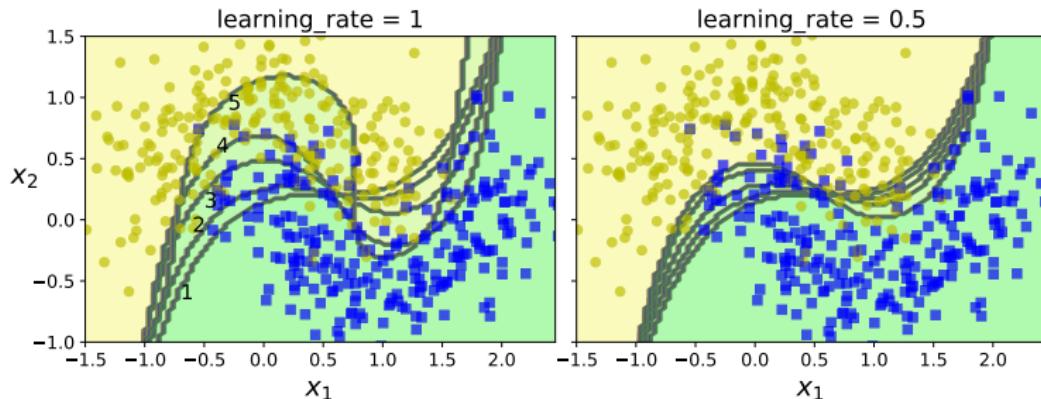


Figure 7-8. Decision boundaries of consecutive predictors

Exercise: Look at the sample code of the author and replace it by using the `sklearn.ensemble.AdaBoostClassifier` class.

AdaBoost (cont.)

- Scikit-Learn uses a multiclass version of AdaBoost called *SAMME*⁵ (stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*).
- When there are just two classes, SAMME is equivalent to AdaBoost.
- A variant of SAMME is *SAMME.R* (the *R* stands for “Real”), which relies on class probabilities rather than predictions and generally performs better.

⁵ “Multi-Class AdaBoost,” J. Zhu et al (2006).

AdaBoost (cont.)

Example: Train an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn's `AdaBoostClassifier` class:

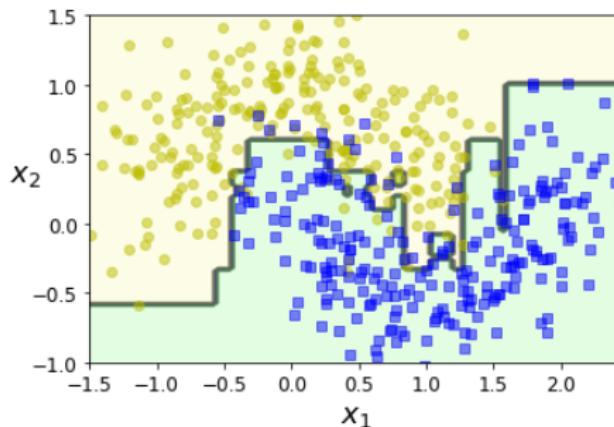
```
from sklearn.ensemble import AdaBoostClassifier  
ada_clf = AdaBoostClassifier(  
    DecisionTreeClassifier(max_depth=1), n_estimators=200,  
    algorithm="SAMME.R", learning_rate=0.5)  
ada_clf.fit(X_train, y_train)
```

- A *Decision Stump* is a Decision Tree with `max_depth=1` – a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class.
- There is also an `AdaBoostRegressor` class for regression problems.

Tip: If your AdaBoost ensemble is overfitting the training set, you can try: (i) reducing the number of estimators or (ii) more strongly regularizing the base estimator.

AdaBoost (cont.)

```
from sklearn.ensemble import AdaBoostClassifier  
ada_clf = AdaBoostClassifier(  
    DecisionTreeClassifier(max_depth=1), n_estimators=200,  
    algorithm="SAMME.R", learning_rate=0.5)  
ada_clf.fit(X_train, y_train)  
  
plot_decision_boundary(ada_clf, X, y) # see the author's code
```



Gradient Boosting

Just like AdaBoost, *Gradient Boosting*⁶ works by **sequentially** adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration, it tries to fit the new predictor to the **residual errors** made by the previous predictor.

Unlike the AdaBoost classifiers, which can take any base estimator, e.g., SVC, SGDClassifier, etc, Gradient Boosting classifiers can only have **DecisionTreeClassifier** as the base estimator.
(similarly for Gradient Boosting regressors)

Using Decision Trees as the base predictors with gradient boosting, we get what is called *Gradient Tree Boosting* (or *Gradient Boosted Regression Trees (GBRT)* for regression).

⁶First introduced in “Arcing the Edge,” L. Breiman (1997).

Gradient Boosting (cont.)

Example:

```
from sklearn.tree import DecisionTreeRegressor

# define and train the first regressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)

# define and train the second regressor on the residual errors
# made by the first predictor
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
# repeat this for the third regressor
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)

# testing for a prediction
y_pred = sum(tree.predict(X_new) for tree in (tree_reg1,tree_reg2,tree_reg3))
```

Gradient Boosting (cont.)

Left column: Predictions of the three trees from the code on the previous slide
Right column: the ensemble's predictions.

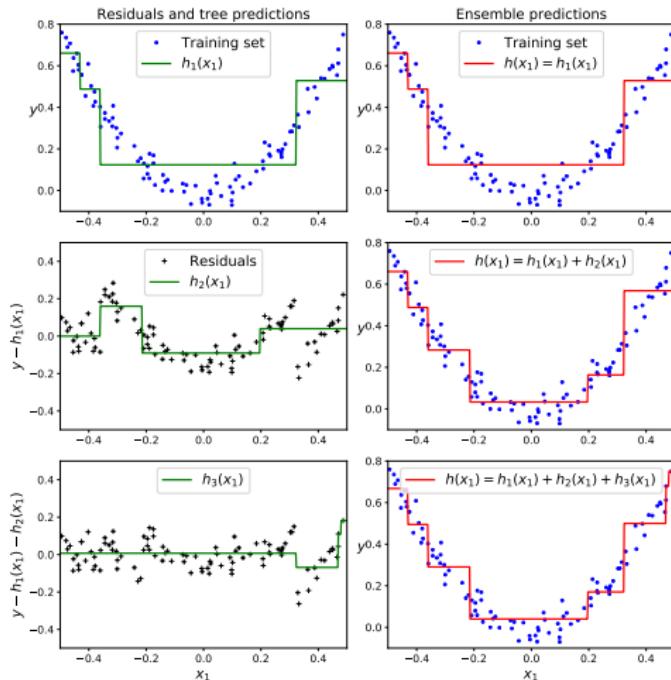


Figure 7-9

Gradient Boosting (cont.)

- Rather than building the base regressor one by one, a simpler way to train GBRT ensembles is to use Scikit-Learn's `GradientBoostingRegressor` class, which has hyperparameters to control the growth the Decision Trees (e.g., `max_depth`, `min_samples_leaf`, and so on), and hyperparameters to control the ensemble training (e.g., `n_estimators`).
- Example:

Train a GBRT on the training data X and training values y:

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                  learning_rate=1.0)
gbrt.fit(X, y)
```

Gradient Boosting (cont.)

- There are other hyperparameters as well. For example, the hyperparameter **learning rate** can be used to scale the contribution of each tree. For a low learning rate (e.g., 0.1), more trees in the ensemble would be needed to fit the training set but the predictions will usually generalize better. This is a regularization technique known as **shrinkage**.
- Example: Two GBRT ensembles trained with a low learning rate. Left column: underfitting due to insufficient #trees; right column: overfitting due to too many trees.

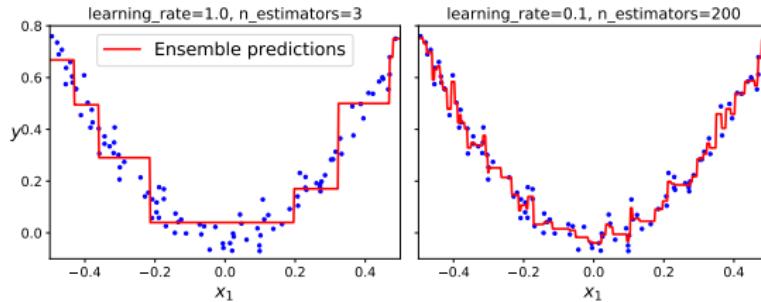


Figure 7-10.

Gradient Boosting (cont.)

- It seems that it is important to find the optimal number of trees to avoid underfitting and overfitting. To do so, we can use `early stopping` by calling the method `staged_predict()` in the `GradientBoostingRegressor` class. Example:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
X_train, X_val, y_train, y_val = train_test_split(X, y)
# train a GBRT ensemble using 120 trees
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
          for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors) + 1

gbrt_best = GradientBoostingRegressor(max_depth=2,
                                       n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

Gradient Boosting (cont.)

The evaluation errors and the best model's prediction for the code on the previous slide:

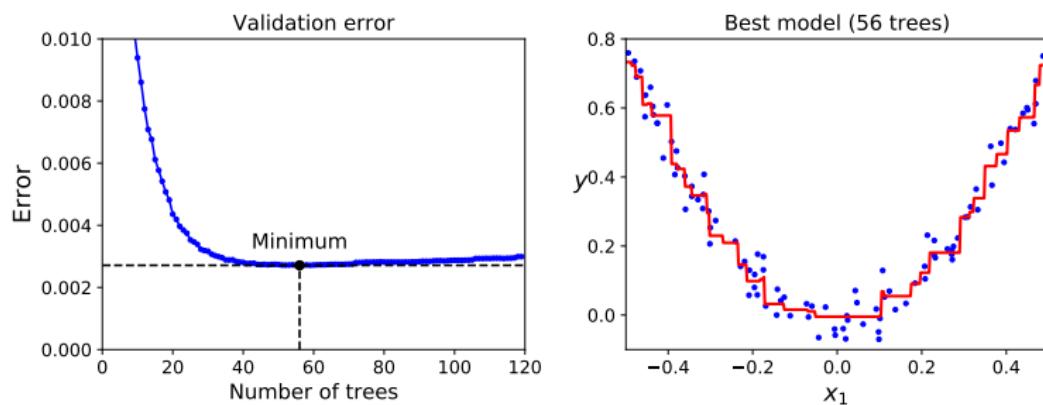


Figure 7-11. Tuning the number of trees using early stopping

Gradient Boosting (cont.)

- It is possible to implement early stopping by actually stopping training early (instead of training a large number of trees first and then looking back to find the optimal number). We can do so by setting `warm_start=True`, which makes Scikit-Learn keep existing trees when the `fit()` method is called, allowing incremental training.
- We can also specify the fraction of training instances for training each tree by setting `subsamples` appropriately, e.g., `subsamples=0.25` means that each tree is trained on 25% of the training instances randomly selected. Again, this trades a higher bias for a lower variance. It also speeds up the training considerably. This technique is called *Stochastic Gradient Boosting*.

Gradient Boosting (cont.)

Example: stop training when the validation error does not improve for 5 iterations in a row:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)

min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break # early stopping
```

Stacking

This is the last Ensemble method that we will cover in this chapter. *Stacking* is short for *stacked generalization*⁷. It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation?

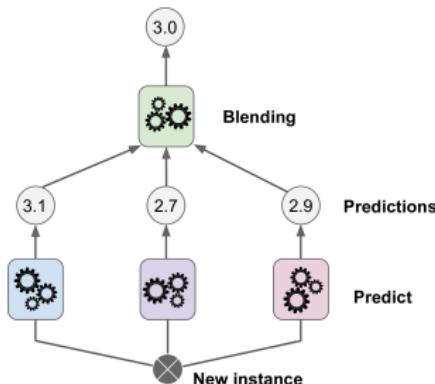
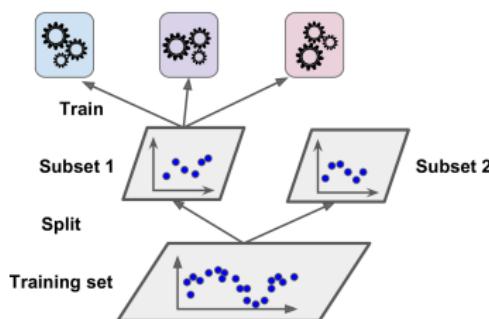


Figure 7-12. Aggregating predictions using a blending predictor

⁷ "Stacked Generalization," D. Wolpert (1992).

Stacking (cont.)

To train the blender, a common approach is to use a **hold-out set**.
The main steps for implementing *Stacking* are:



1. Split the training data into 2 subsets: **subset 1** and **subset 2**.
2. Use subset 1 to train the base predictors.

Figure 7-13. Training the first layer

Stacking (cont.)

The main steps for implementing *Stacking* are:

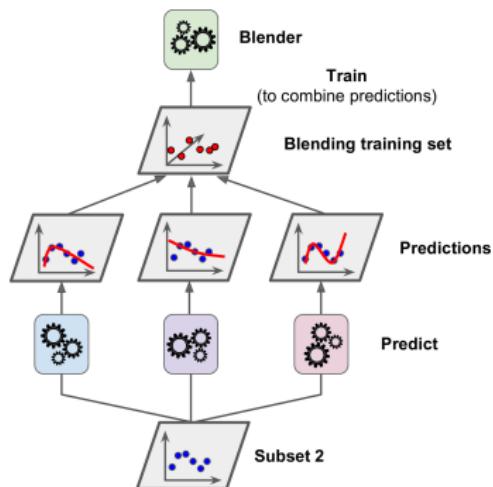


Figure 7-14. Training the blender

3. Use subset 2 to test the predictions produced by the predictors.
4. Create a new 3-dimensional (since we have 3 predictors in this example) training set using these predicted values as input features and keep the target values (see next slide).
5. Train the *blender* on this new training set to learn to predict the target values.

Stacking (cont.)

For the example shown in Figure 7-14, the [blending training set](#) created from subset 2 looks like:

predictor 1's output	predictor 2's output	predictor 3's output	ground truth
xxx	xxx	xxx	yyy
xxx	xxx	xxx	yyy
...
xxx	xxx	xxx	yyy

This blending training set is used for training the blender.

The first three columns can be considered as the feature columns (3D features in this case); the last column is the class label (or ground truth value).

Stacking (cont.)

See `sklearn.ensemble.StackingClassifier` and
`sklearn.ensemble.StackingRegressor`

Some parameters from the `StackingClassifier` class:

- `estimators` – list of base estimators (strings or objects)
- `final_estimator` that will be used to combine the base estimators.
- `stack_method` – method to called for each base estimator for the stacking.

Similarly for the `StackingRegressor` class, except for the parameter `stack_method` which is not used.

Work through the lab sheet and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read up to Chapter 8 on Dimensionality Reduction.

And that's all for the lecture.

Have a good week.

CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Week 8: Book Chapter 8

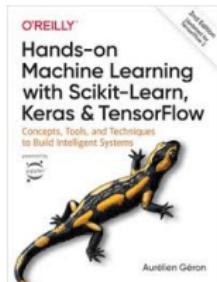
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 8.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Chapter Eight

Dimensionality Reduction

Here are the main topics we will cover:

- Main Approaches in Dimensionality Reduction
- Principal Component Analysis (PCA)
- Kernel PCA
- Locally Linear Embedding (LLE)
- Other Dimensionality Reduction Techniques

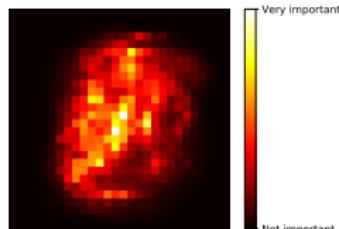
The Curse of Dimensionality

Training instances in machine learning often have thousands or even millions of features. This means that

- it takes a long time to train the ML algorithm, and
- it is difficult to find a good solution.

This problem is referred to as the *curse of dimensionality*

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably.



For example, consider the MNIST images, we could completely drop the boundary pixels from the training set without losing much information for classification.

Notes on Dimensionality Reduction

- Some information is lost after reducing the dimensionality of your data.
- Although dimensionality reduction will speed up training, it may make your system perform slightly worse, and your pipeline more complex and harder to maintain.

Apart from speeding up training, dimensionality reduction is also useful for data visualization (or *DataVis*). e.g., plotting a high dimensional training set on a graph helps you gain some insight about the data.

The Curse of Dimensionality

Consider a unit square and a circle inside the square.

- The area of the square is 1.
- The area of the circle is $\pi r^2 = \pi 0.5^2 = 0.7854$
- The area difference is 0.2146. \Rightarrow points have 21% probability to be outside the circle but inside the square.

Now consider a unit cube and a sphere inside it. The volumes of the cube and sphere are, respectively, 1 and 0.5236. The probability increases to 48%.

As we increase the dimension, say to a 10,000-dimensional unit hypercube, this probability increases to > 99.999999%.

\Rightarrow Most points in a high-dimensional hypercube are very close to the border.

The Curse of Dimensionality (cont.)

A more troublesome difference is the average distance between points in high-dimensional space: in 2D, it is roughly 0.52; in 3D, it is 0.66, in 1,000,000D, it is 408.25.

This fact implies that high dimensional datasets are at risk of being very sparse:

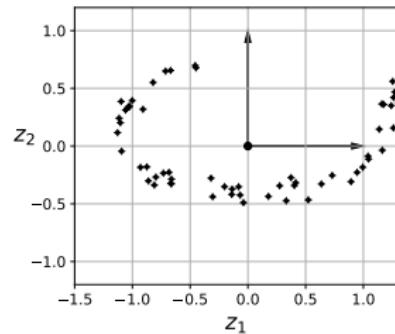
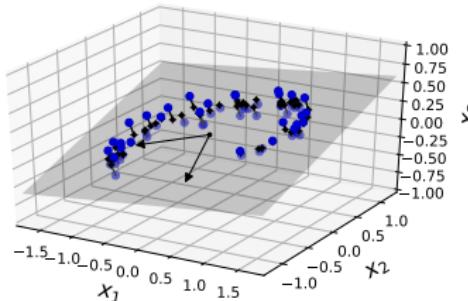
- most training instances are likely to be far away from each other;
- new instance is likely to be far away from any training instance, making prediction much less reliable.

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

Main Approaches for Dimensionality Reduction

Projection

Inspect the data to see if they lie in a subspace of the high dimensional space.

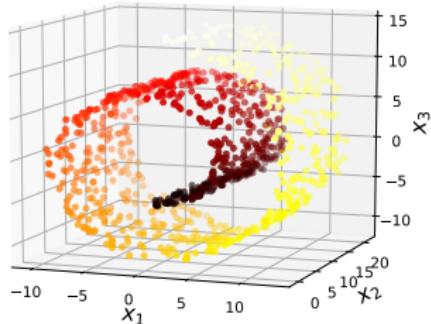


Left: before projection. Right: after projection.

Main Approaches for Dimensionality Reduction

Manifold Learning

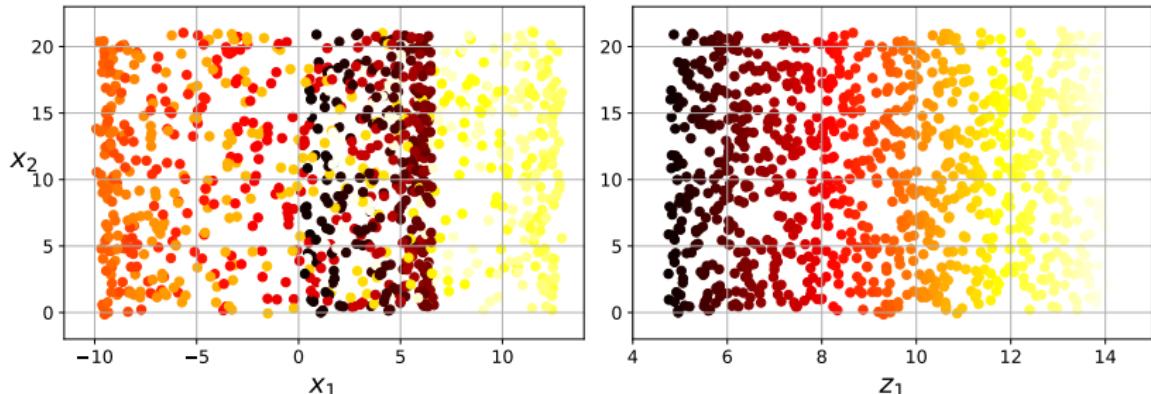
Projection is not always the best approach to dimensionality reduction, when the subspace may twist and turn, e.g., the famous *Swiss roll* toy dataset:



Main Approaches for Dimensionality Reduction

Manifold Learning

Simply projecting onto a plane (e.g., by dropping x_3) would squash different layers of the Swiss roll together, as shown on the left diagram. However, what you really want is to unroll the Swiss roll to obtain the 2D dataset on the right.



Main Approaches for Dimensionality Reduction

Manifold Learning

More generally, a d -dimensional manifold is a part of an n -dimensional space (where $d < n$) that locally resembles a d -dimensional hyperplane. In the case of the Swiss roll, $d = 2$ and $n = 3$.

Many DR¹ algorithms work by modelling the *manifold* that the training instances lie on. This is called *manifold learning*. It relies on *manifold assumption* (a.k.a. *manifold hypothesis*).

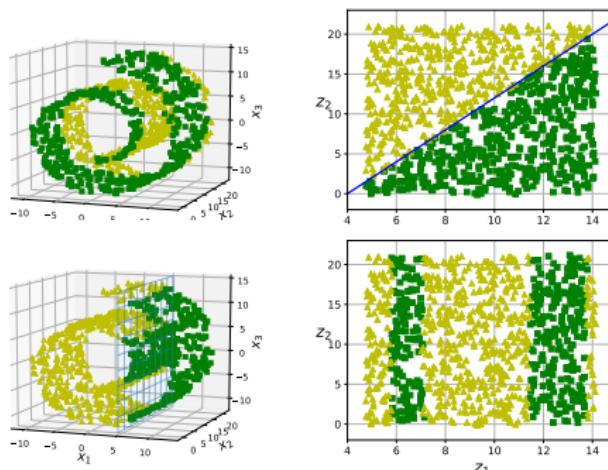
Also accompanied with this assumption is another implicit assumption that the classification or regression task will be simpler if the data are expressed in a lower-dimensional manifold.

¹DR = dimensionality reduction

Main Approaches for Dimensionality Reduction

Manifold Learning

However, the decision boundary may not always be simpler with lower dimensions. \Rightarrow DR will definitely speed up training but it may not always lead to a better or simpler solution – it all depends on the dataset.



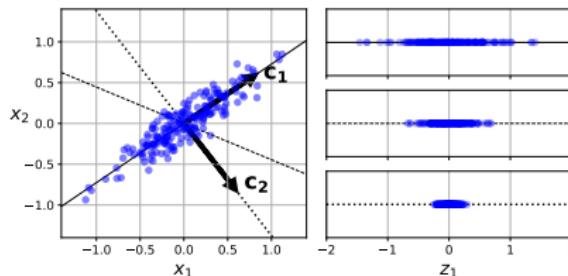
Main Approaches for Dimensionality Reduction

Principal Component Analysis (PCA)

PCA is by far the most popular DR algorithms. It is a linear dimensionality reduction technique. Firstly, it identifies the hyperplane that lies closest to the data. It then projects the data onto it.

- **Preserving the variance**

The key idea behind PCA is finding the projection matrix that would **preserve the variance of the data as much as possible**.



Main Approaches for Dimensionality Reduction

PCA (cont.)

- **Principal axes and principal components**

For a given n -dimensional dataset, PCA identifies

- the first axis that accounts for the largest variance in the dataset,
- the second axis that is orthogonal to the first axis and that has the second largest variance,
- the third axis that is orthogonal to the first two axes and that has the third largest variance,
- and so on.

The unit vector that defines the i^{th} axis is called the i^{th} *principal component axis* (or just *principal axis*).

When a data vector is projected onto a principal axis, we obtain a *principal component* (PC) along that axis.

Main Approaches for Dimensionality Reduction

PCA (cont.)

- **Principal axes and principal components**

Thus, in dimensionality reduction, to determine the number of principal axes to keep is the same as to determine how many principal components to have after data projection.

Note: the direction of each PC axis is not unique (i.e., either direction is correct).

Main Approaches for Dimensionality Reduction

PCA (cont.)

- How to find the principal axes?

There is a standard matrix factorization technique called *Singular Value Decomposition (SVD)* that can decompose the training set \mathbf{X} into three matrices \mathbf{U} , Σ , and \mathbf{V} :

$$\mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T.$$

If each row of your \mathbf{X} matrix stores one data point, then each column of \mathbf{V} stores a principal axis:

$$\mathbf{V} = \begin{bmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{bmatrix}$$

Note: \mathbf{X} is $m \times n$; \mathbf{U} is $m \times n$, Σ is $n \times n$; \mathbf{V} is $n \times n$ (m is the total number of data points; $m \gg n$).

Main Approaches for Dimensionality Reduction

PCA (cont.)

- How to find the principal axes?

Numpy has a `svd()` function which can be used to obtain all the principal axes from the training set.

Example:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```

Notes:

- You must centre the data so that the origin is at the mean.
- As `np.linalg.svd` outputs the V^T matrix (rather than V), to extract a column from V , we must do `Vt.T` to transpose the matrix first.

Main Approaches for Dimensionality Reduction

PCA (cont.)

- How to project the data?

In general, you would want to reduce the dimension from n to d by projecting the data onto the hyperplane defined by the first d principal axes while preserving as much variance of the data as possible.

To do so, define an $n \times d$ matrix \mathbf{W}_d to contain the first d columns of matrix \mathbf{V} . The projection is then simply:

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

(\mathbf{X} is $m \times n$, \mathbf{W}_d is $n \times d$, $\mathbf{X}_{d\text{-proj}}$ is $m \times d$)

In Python:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

Main Approaches for Dimensionality Reduction

PCA (cont.)

- How to find the principal axes using Scikit-Learn?

Instead of Numpy's `svd()` function, you can also use Scikit-Learn's `PCA` class which implements the PCA and dimensionality reduction:

```
from sklearn.decomposition import PCA  
pca = PCA(n_components = 2)  
X2D = pca.fit_transform(X)
```

You can access the PC axes using the `components_` variable.
e.g., `pca.components_.T[:,0]` gives you the first PC axis.

The line `X2D = pca.fit_transform(X)` above handles the projection. The output `X2D` is the 2D projection of the original 3D data `X`.

Main Approaches for Dimensionality Reduction

PCA (cont.)

- **Explained variance ratio**

It is important to find out the proportion of the dataset's variance that lies along each principal component axis.

This is the *explained variance ratio*, available via the `explained_variance_ratio_` variable:

```
>>> pca.explained_variance_ratio_
array([ 0.84248607,  0.14631839])
```

Adding up the above numbers shows that the third PC that is removed from dimensionality reduction takes up only around 1.2% of the variance.

Main Approaches for Dimensionality Reduction

PCA (cont.)

- **Choosing the right number of dimensions**

It is generally preferable to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%).

Example:

```
pca = PCA()  
pca.fit(X)  
cumsum = np.cumsum(pca.explained_variance_ratio_)  
d = np.argmax(cumsum >= 0.95) + 1  
pca = PCA(n_components=d)  
X_reduced = pca.fit_transform(X)
```

A simpler and better option to the above is:

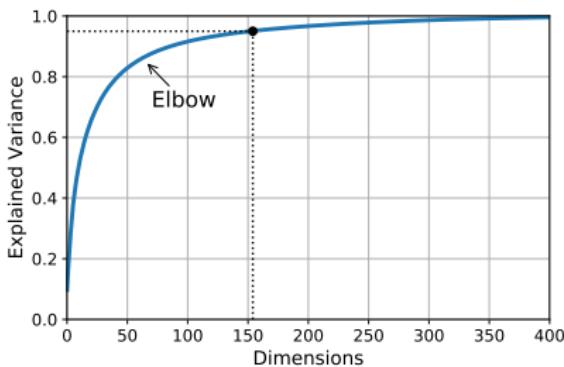
```
pca = PCA(n_components=0.95)  
X_reduced = pca.fit_transform(X)
```

Main Approaches for Dimensionality Reduction

PCA (cont.)

- Choosing the right number of dimensions

Another option is to plot the explained variance as a function of dimensions (simply plot the `cumsum`). There will usually be an **elbow** in the curve, where the explained variance stops growing fast.



Main Approaches for Dimensionality Reduction

PCA (cont.)

- **PCA for compression**

For the MNIST dataset, sacrificing 5% of the variance allows the 784-dimensional features to reduce down to 154 dimensions. This data compression ratio can speed up a classification algorithm (e.g. SVM) tremendously.

It is also possible to decompress the data back to 784 dimensions by applying the inverse transformation of the PCA projection to reconstruct the data. The mean squared distance between the original data and the reconstructed data is called the *reconstruction error*.

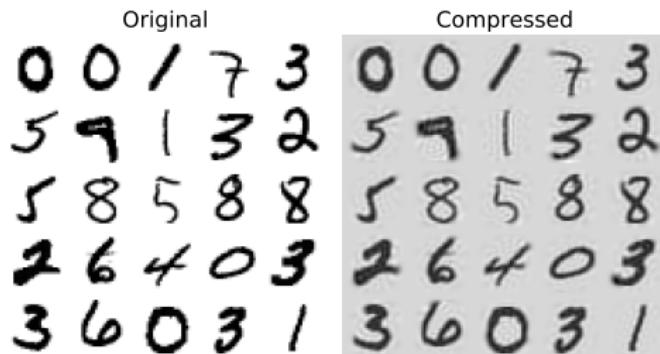
```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X)
X_recovered = pca.inverse_transform(X_reduced)
```

Main Approaches for Dimensionality Reduction

PCA (cont.)

- **PCA for compression**

Below is a sample original and reconstructed example of MNIST compression which preserves 95% of the variance. There is a slight image quality loss, but the digits are still mostly intact. Mathematically: $\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d\text{-proj}} \mathbf{W}_d^T$



Main Approaches for Dimensionality Reduction

PCA (cont.)

- **Incremental PCA**

The preceding implementation of PCA requires the whole training set to fit in memory in order for the SVD algorithm to run. For the *Incremental PCA (IPCA)* algorithm, mini-batches can be fed to the algorithm one at a time. This also allows PCA to be applied online. e.g., using Scikit-Learn's *IncrementalPCA* class:

```
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_train)
```

An alternative is Numpy's *memmap* class.

Main Approaches for Dimensionality Reduction

PCA (cont.)

- **Randomized PCA**

Randomized PCA is a stochastic algorithm which quickly finds an approximation of the first d principal component axes. Its computational complexity is $O(md^2) + O(d^3)$, instead of $O(mn^2) + O(n^3)$, so it is dramatically faster than the previous algorithms when d is much smaller than n .

Example: using Scikit-Learn's Randomized PCA:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

Main Approaches for Dimensionality Reduction

Kernel PCA

Recall that the *kernel trick* is a mathematical technique that implicitly maps instances into a very high-dimensional space called the *feature space*, allowing non-linear classification and regression to be done by linear techniques such as SVM.

This same trick can be applied to PCA, making it possible to perform complex *nonlinear* projections for dimensionality reduction. This is called *Kernel PCA (kPCA)*.

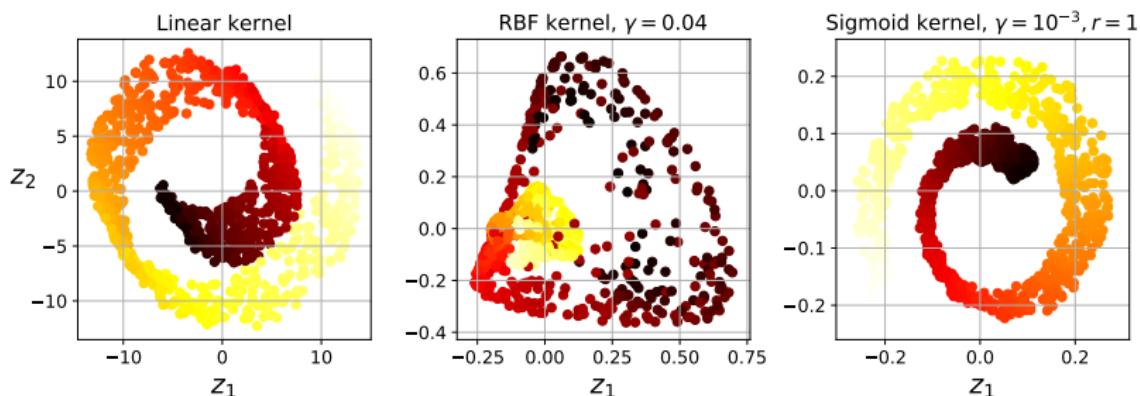
Example: Using Scikit-Learns *KernelPCA* class to perform kPCA with an RBF kernel:

```
from sklearn.decomposition import KernelPCA  
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)  
X_reduced = rbf_pca.fit_transform(X)
```

Main Approaches for Dimensionality Reduction

Kernel PCA (cont.)

Example: Swiss roll reduced to 2D using kPCA with various kernels



Main Approaches for Dimensionality Reduction

Kernel PCA (cont.)

- Selecting a kernel and tuning hyperparameters

As kPCA is an unsupervised learning algorithm, there is no obvious performance measure to help you select the best kernel and hyperparameter values. To select the optimal kernel and hyperparameters, we can perform a grid search. Example:

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
clf = Pipeline([
    ("kpca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())
])
param_grid = [
    {"kpca__gamma": np.linspace(0.03, 0.05, 10),
     "kpca__kernel": ["rbf", "sigmoid"]}
]
```

Main Approaches for Dimensionality Reduction

Kernel PCA (cont.)

- Selecting a kernel and tuning hyperparameters

```
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

The argument `cv=3` means that we want 3-fold cross validation.

The best kernel and hyperparameters are then available through the `best_params_` variable:

```
>>> print(grid_search.best_params_)
{'kpca__gamma': 0.04333333333333335, 'kpca__kernel': 'rbf'}
```

Main Approaches for Dimensionality Reduction

Kernel PCA (cont.)

- **Selecting a kernel and tuning hyperparameters**

Another approach, this time **entirely unsupervised**, is to select the kernel and hyperparameters that yield the **lowest reconstruction error**. However, reconstruction using kernel PCA is more complicated than using linear PCA.

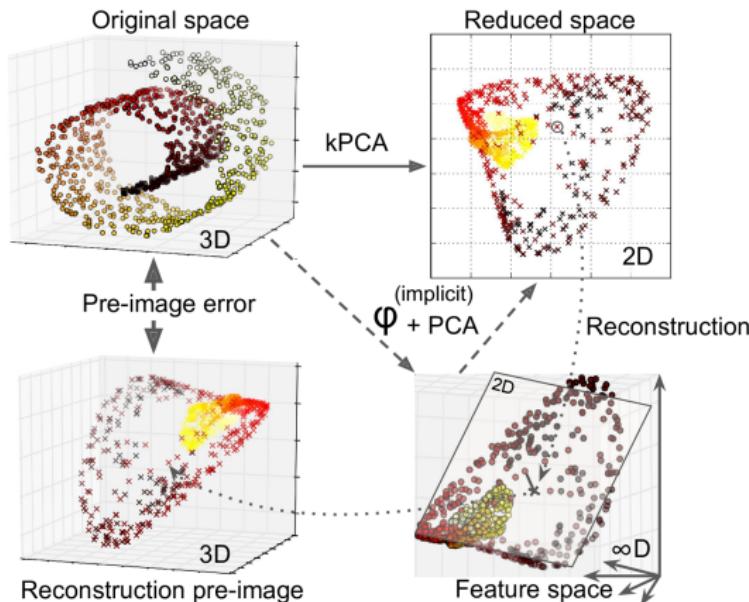
- The dimension of the feature space can be infinite (if we use, e.g., a **rbf** kernel)

Fortunately, it is possible to reconstruct the ***pre-image*** from the feature space. We can then compute the reconstruction error between the data points in the original space and in the pre-image. This is known as the ***pre-image error***.

Main Approaches for Dimensionality Reduction

Kernel PCA (cont.)

- Selecting a kernel and tuning hyperparameters



Kernel PCA and the reconstruction pre-image error.

Main Approaches for Dimensionality Reduction

Locally Linear Embedding (LLE)

LLE² is another powerful *nonlinear dimensionality reduction (NLDR)* technique. It is a Manifold Learning technique that does not rely on projections like the previous algorithms.

In a nutshell, LLE works by

- first measuring how each training instance linearly relates to its closest neighbours, and then
- looking for a low-dimensional representation of the training set where these local relationships are best preserved.

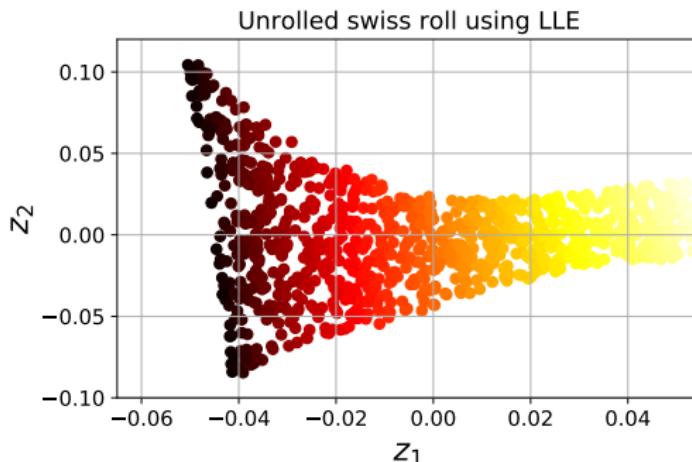
² "Nonlinear Dimensionality Reduction by Locally Linear Embedding," S. Roweis, L. Saul (2000).

Main Approaches for Dimensionality Reduction

Locally Linear Embedding (LLE)

Scikit-Learns `LocallyLinearEmbedding` class implements LLE:

```
from sklearn.manifold import LocallyLinearEmbedding  
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)  
X_reduced = lle.fit_transform(X)
```



Main Approaches for Dimensionality Reduction

Locally Linear Embedding (LLE)

Scikit-Learns LLE implementation has the following computational complexity:

- $O(m \log(m) n \log(k))$ for finding the k nearest neighbours,
- $O(mnk^3)$ for optimizing the weights, and
- $O(dm^2)$ for constructing the low-dimensional representations.

(Recall that n is the dimension of the data points in the original space, m is the total number of data points, and d is the dimension that we want to reduce to.)

Unfortunately, the m^2 in the last term makes this algorithm scale poorly to very large datasets.

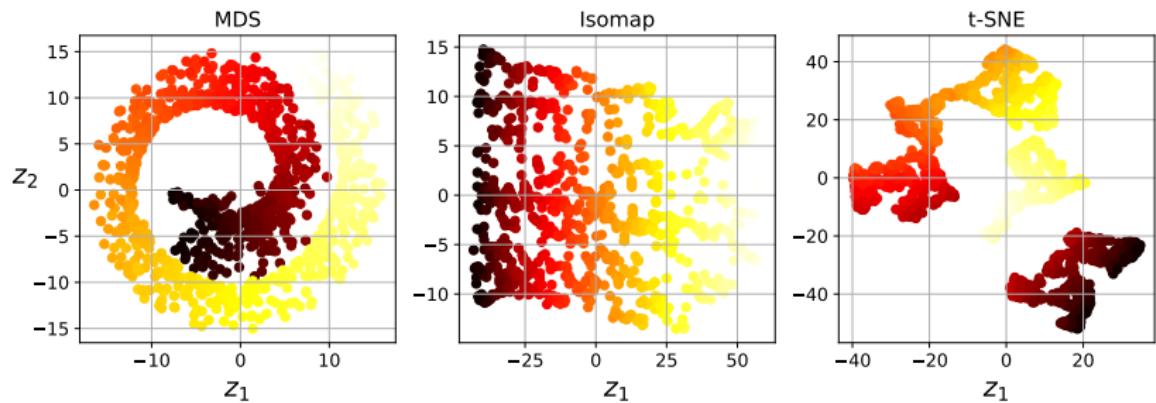
Other Dimensionality Reduction Techniques

Some other popular dimensionality reduction techniques are:

- **Multidimensional Scaling (MDS)** reduces dimensionality while trying to preserve the distances between instances.
- **Isomap** creates a graph by connecting each instance to its nearest neighbours, then reduces dimensionality while trying to preserve the *geodesic distances* between instances.
- ***t*-Distributed Stochastic Neighbour Embedding³** (*t-SNE*) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space.

³van der Maaten, L.J.P.; Hinton, G.E. "Visualizing High-Dimensional Data Using t-SNE," Journal of Machine Learning Research, 2008.

Other Dimensionality Reduction Techniques



Reducing the Swiss roll to 2D using various techniques.

Other Dimensionality Reduction Techniques

- **Linear Discriminant Analysis (LDA)** is actually a classification algorithm, but during training it learns the most discriminative axes between the classes, and these axes can then be used to define a hyperplane onto which to project the data.
As the projection will keep classes as far apart as possible, LDA is a good technique to reduce dimensionality before running another classification algorithm such as an SVM classifier.

Summary

- Understand the main motivation for dimensionality reduction.
- Understand the curse of dimensionality.
- Understand how PCA works, its merit and limitation, and variants of PCA, e.g., incremental PCA, randomized PCA, and kernel PCA.
- Understand manifold learning and LLE.
- Understand the concept behind a few popular DR techniques, e.g., MDS, isomap, t-SNE, and LDA.
- Understand your data so you can pick the best DR technique for your problem. Data visualization is one way to get more insight about your data.

For next week

Work through the lab sheet and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there.

Read up to Chapter 10 on [Intro. to ANNs with Keras](#).

And that's all for the eighth lecture.

Have a good week.

CITS5508 Machine Learning

Lectures for Semester 1, 2021

Lecture Week 9: Book Chapters 10

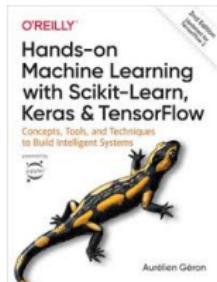
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapters 10.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Introduction to Artificial Neural Networks

Here are the main topics we will cover:

- Perceptrons and *threshold logic unit* (TLU)
- Training of a Perceptron and *Hebb's rule*
- Perceptrons and the XOR classification problem
- Multi-Layer Perceptrons (MLP) and backpropagation
- Commonly used activation functions
- Training a deep neural network (DNN) using the Sequential API
- Fine-tuning neural network hyperparameters

The Perceptron

- The *Perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a slightly different artificial neuron called a *threshold logic unit* (TLU), where the inputs and output are numbers (instead of binary on/off values) and each input connection is associated with a *weight*.
- The TLU computes a weighted sum of its inputs:
$$z = w_1x_1 + \dots + w_nx_n = \mathbf{x}^T \mathbf{w}$$
then applies a *step function* to that sum and outputs the result:
$$h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{x}^T \mathbf{w}).$$

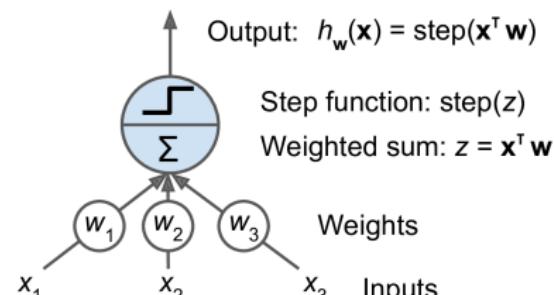


Figure 10-4.
Threshold logic unit

The Perception (cont.)

Step Functions

The most common step function used in Perceptrons is the *Heaviside step function*:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$

Sometimes the *sign function* is used instead:

$$\text{sign}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

The Perceptron (cont.)

A single TLU can be used for [simple linear binary classification](#): It computes a linear combination of the inputs and if the result exceeds a threshold, it outputs the positive class or else outputs the negative class (just like a [Logistic Regression classifier](#) or a [linear SVM](#)).

A Perceptron is simply composed of [a single layer of TLUs](#), with

- special passthrough neurons called [*input neurons*](#), which just output whatever input they are fed; and
- an extra bias feature ($x_0 = 1$), which is typically represented using a special type of neuron called a [*bias neuron*](#) (it outputs 1 all the time).

Perceptron: An example

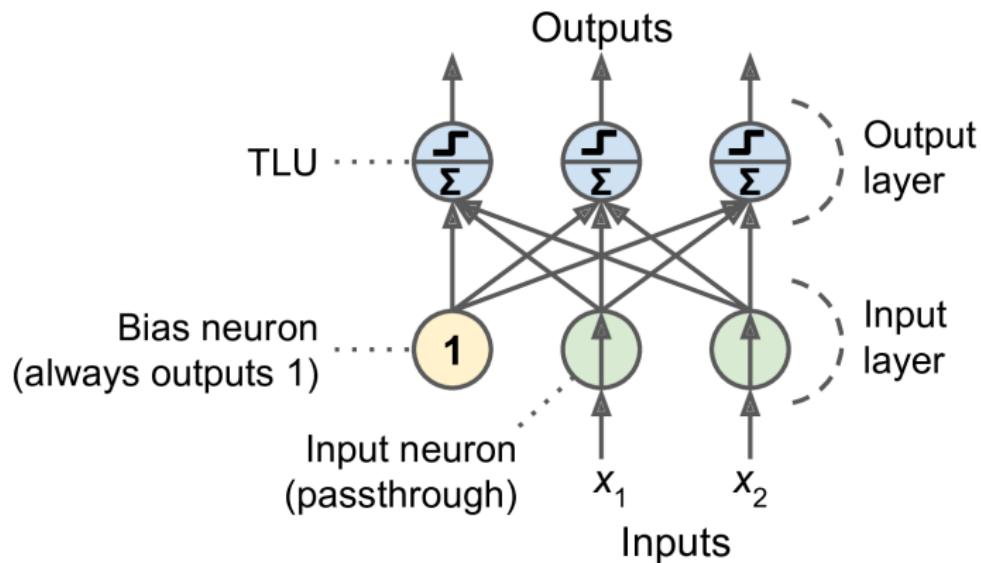


Figure 10-5. A Perceptron with two input, one bias, and three output neurons.

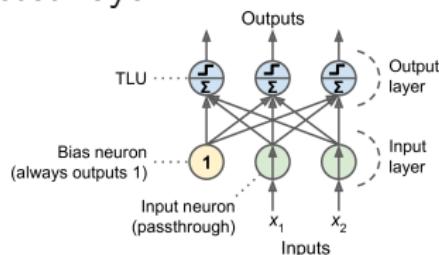
Perceptron: Computing the outputs

The magic of linear algebra makes it possible to efficiently compute the outputs of a fully connected layer:

$$h_{W,b}(x) = \phi(x^T W + b^T)$$

where

- x represents the input feature
- W represents the unknown weight matrix which contains all the connection weights except for the ones from the bias neuron
- b is the bias vector containing all the connection weights between the bias neuron and the output neurons.
- ϕ is called the *activation function*.



How is a Perceptron trained?

The Perceptron training algorithm proposed by Frank Rosenblatt was largely inspired by *Hebb's rule* or *Hebbian learning* (name after Donald Hebb) which can be summarized as “cells that fire together, wire together”.

More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

How is a Perceptron trained? (cont.)

Perceptron learning rule (weight update):

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

where

- $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
- x_i is the i^{th} input value of the current training instance.
- \hat{y}_j is the output of the j^{th} output neuron for the current training instance.
- y_j is the target output of the j^{th} output neuron for the current training instance.
- η is the learning rate.

Perceptron: Decision Boundary

The **decision boundary** of each output neuron is **linear**, so Perceptrons (just like Logistic Regression classifiers) are incapable of learning complex patterns.

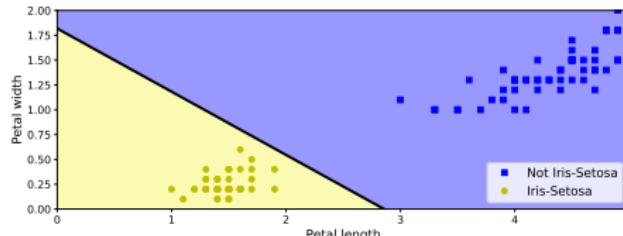
However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm would converge to a solution.¹ This is called the ***Perceptron convergence theorem***.

¹Note that this solution is generally not unique.

Implementing Perceptron in Scikit-Learn

Scikit-Learn provides a **Perceptron** class that implements a single TLU network:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron
iris = load_iris()
X = iris.data[:, (2, 3)]          # petal length, petal width
y = (iris.target == 0).astype(np.int) # Iris Setosa?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)
y_pred = per_clf.predict([[2, 0.5]])
```



Decision boundary output by the code above.

Implementing Perceptron in Scikit-Learn (cont.)

The `Perceptron` learning algorithm strongly resembles `Stochastic Gradient Descent`. In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters:

```
loss="perceptron",
learning_rate="constant",
eta0=1 (learning rate), and
penalty=None (no regularization).
```

Comparing Perceptron with Logistic Regression Classifiers

Note that contrary to Logistic Regression classifiers, Perceptrons do not output a class probability; rather, they just make predictions based on a hard threshold.

This is one of the good reasons to prefer Logistic Regression over Perceptrons.

Weaknesses of Perceptrons

In their 1969 monograph titled “*Perceptrons*”, Marvin Minsky and Seymour Papert highlighted some serious weaknesses of Perceptrons, in particular, their inability of solving some trivial problems (e.g., the *Exclusive OR (XOR)* classification problem).

Of course this is true of any other linear classification model as well (such as *Logistic Regression classifiers*), but researchers had expected much more from Perceptrons. As a result, many researchers dropped *connectionism* altogether.

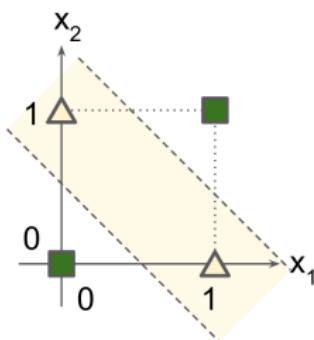


Figure 10-6a. XOR classification problem

Weaknesses of Perceptrons (cont.)

However, it turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called a *Multi-Layer Perceptron (MLP)*.

In particular, an MLP can solve the XOR problem, as you can verify by computing the output of the MLP for each combination of inputs: with inputs $(0, 0)$ or $(1, 1)$ the network outputs 0, and with inputs $(0, 1)$ or $(1, 0)$ it outputs 1.

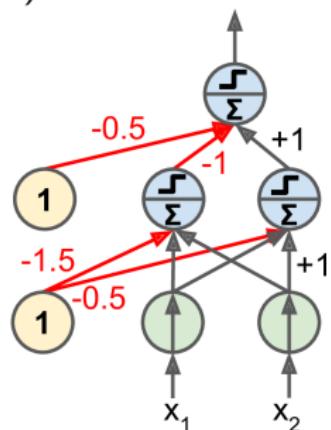


Figure 10-6b. An MLP that solves the XOR classification problem.

Multi-Layer Perceptron and Backpropagation

An **MLP** is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer*.

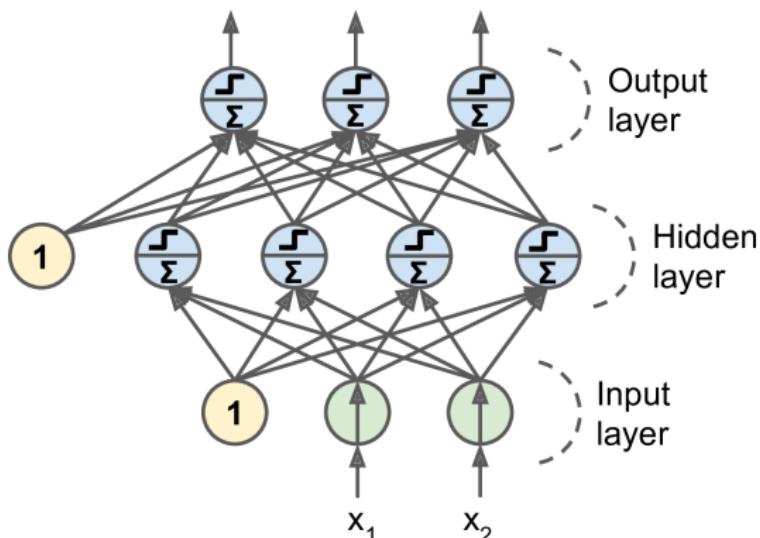


Figure 10-7. Multi-Layer Perceptron.

Multi-Layer Perceptron and Backpropagation (cont.)

We can describe the *backpropagation training algorithm*² as Gradient Descent using reverse-mode autodiff.

For each training instance, the algorithm

- feeds it to the network and computes the output of every neuron in each layer (this is the *forward pass*),
- measures the network's output error (i.e., the difference between the desired output and the actual output of the network),
- goes through each layer in reverse order to measure the error contribution from each connection (this is the *reverse pass*) until the input layer is reached, and
- finally slightly tweaks the connection weights to reduce the error (this is the *Gradient Descent step*).

² "Learning Internal Representations by Error Propagation," D. Rumelhart, G. Hinton, R. Williams (1986). This algorithm was actually invented several times, starting with P. Werbos in 1974.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

In order for the backpropagation training algorithm to work, Rumelhart et al made a key change to the MLP's architecture: they replaced the step function with the **logistic function**, $\sigma(z) = 1/(1 + \exp(-z))$. This was essential because the step function has no gradient to work with, while the logistic function has a well-defined nonzero derivative everywhere.

Such a function which defines the output of a neuron (or node) given an input or set of inputs is known as the ***activation function***.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

Apart from the logistic function, two other popular activation functions are:

- *The hyperbolic tangent function:* $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1$. This function has a similar S shape as the logistic function, is continuous and differentiable, but its values range from -1 to $+1$.
- *The ReLU function:* $\text{ReLU}(z) = \max(0, z)$. This function is continuous but unfortunately not differentiable everywhere. However, in practice it works very well and has the advantage of being fast to compute. Most importantly, the fact that it does not have a maximum output value also helps reduce some issues during Gradient Descent.

Multi-Layer Perceptron and Backpropagation (cont.)

Activation functions

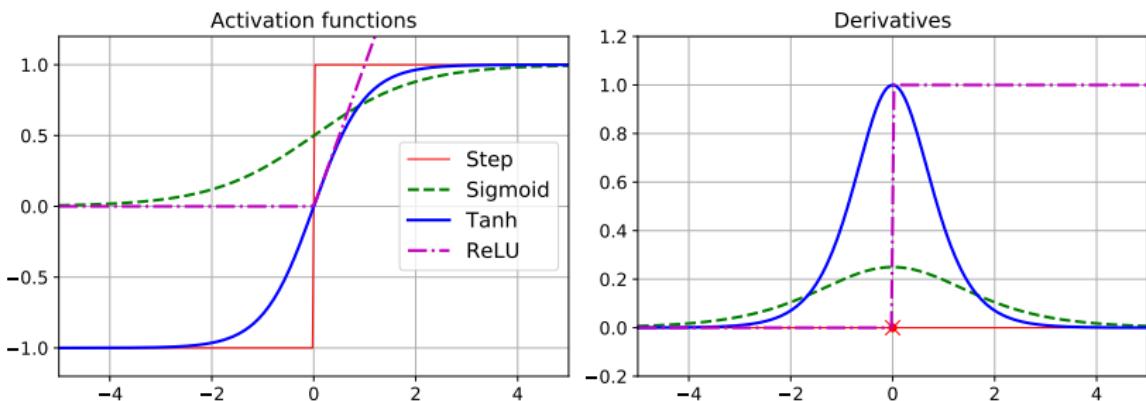


Figure 10-8. Activation functions and their derivatives

Regression MLPs

MLPs can be used for regression tasks. For example, to predict the price of a house given many of its features, you need one output neuron. For multivariate regression, you need multiple neurons.

For regression, the output neurons usually do not need an activation function. Only when the output z needs to be

- always positive, then use the ReLU or $\text{softplus}(z) = \log(1 + \exp(z))$ activation function;
- in a certain range of values, then use logistic or hyperbolic tangent and then scale the value to the required range.

Regression MLPs (cont.)

The loss function to use during training is typically the [mean squared error](#) or [mean absolute error](#) (the latter is more robust against outliers). Alternatively, the [Huber loss](#), which is a combination of MSE and MAE, can also be used.

Typical regression MLP architecture:

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., $28 \times 28 = 784$ for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

Classification MLPs

MLPs can also be used for classification tasks.

- For binary classification problems, one single output neuron with a logistic activation function would be sufficient.
- For multilabel classification problems, you can have multiple output neurons, each with a logistic activation function.
- For multiclass classification problems (e.g., MNIST dataset), you can have multiple output neurons, with the softmax activation function. This will ensure that all the output values are probabilities and they sum to 1.

For the loss function, since we are predicting probability distributions, the *cross-entropy* loss (also called the *log loss*) is generally a good choice:

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

where K denotes the total number of classes (e.g., $K = 10$ for MNIST).

Classification MLPs (cont.)

Typical classification MLP architecture:

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
Input and hidden layers	Same as regression	Same as regression	Same as regression
# output neurons	1	1 per label	1 per class
Output layer activation	Logistic	Logistic	Softmax
Loss function	Cross entropy	Cross entropy	Cross entropy

Implementing MLPs with Keras

Keras (<https://keras.io/>) is a high-level Deep Learning API that allows you to easily build, train, evaluate, and execute all sorts of neural networks.

It was developed by François Chollet and was released as an open source project in March 2015. It quickly gained popularity and, at present, is available in three popular open source Deep Learning libraries: *TensorFlow*, *Microsoft Cognitive Toolkit (CNTK)*, and *Theano*. We will refer to this reference implementation as *multibackend Keras*.

Implementing MLPs with Keras (cont.)

TensorFlow now comes bundled with its own Keras implementation, `tf.keras`, which supports TensorFlow's Data API.

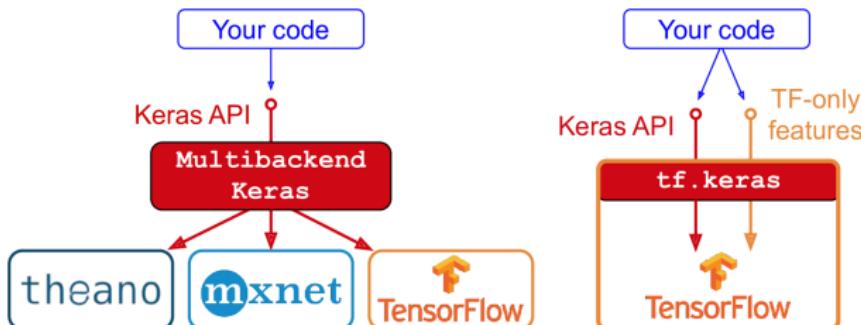


Figure 10-10. Two implementations of the Keras API:
multibackend Keras (left) and `tf.keras` (right)

Another popular Deep Learning library, after Keras and TensorFlow, is Facebook's *PyTorch* library.

Implementing MLPs with Keras (cont.)

After installation, check that you have the correct version of TensorFlow:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

Example: Building an Image Classifier Using the Sequential API

A linear model can reach about 92% on MNIST, but only about 83% on *Fashion MNIST*, which also has 10 classes with input being 28×28 images:



Figure 10-11. Samples from Fashion MNIST.

The 10 classes are: "T-shirt/top", "Trouser", "Pullover", "Dress", "Coat", "Sandal", "Shirt", "Sneaker", "Bag", and "Ankle boot".

Example: Building an Image Classifier Using the Sequential API (cont.)

Loading the data and creating the model

```
import tensorflow as tf
from tensorflow import keras

fashion_mnist = keras.datasets.fashion_mnist
(X_train_full, y_train_full), (X_test, y_test) = fashion_mnist.load_data()      #60,000 and 10,000
# split training set to form a validation set
X_valid, X_train = X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

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

The last few lines above can be replaced by

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

Example: Building an Image Classifier Using the Sequential API (cont.)

Inspecting the model

```
>>> model.summary()
Model: "sequential"
-----
Layer (type)          Output Shape       Param #
=====
flatten (Flatten)    (None, 784)        0
-----
dense (Dense)         (None, 300)        235500
-----
dense_1 (Dense)       (None, 100)        30100
-----
dense_2 (Dense)       (None, 10)         1010
=====
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0

>>> hidden1 = model.layers[1]
>>> weights, biases = hidden1.get_weights()
.....
>>> weights.shape
(784, 300)
>>> biases.shape
(300,)
```

Example: Building an Image Classifier Using the Sequential API (cont.)

Compiling the model

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

Training and evaluating the model

```
>>> history = model.fit(X_train, y_train, epochs=30,
... validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [=====] - 3s 49us/sample - loss: 0.7218 - accuracy: 0.7660
- val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840 - accuracy: 0.8327
- val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252 - accuracy: 0.9192
- val_loss: 0.2999 - val_accuracy: 0.8926
```

Useful options to incorporate: `class_weight`, `sample_weight`,
`batch_size` (default=32)

Example: Building an Image Classifier Using the Sequential API (cont.)

Inspecting the learning curves

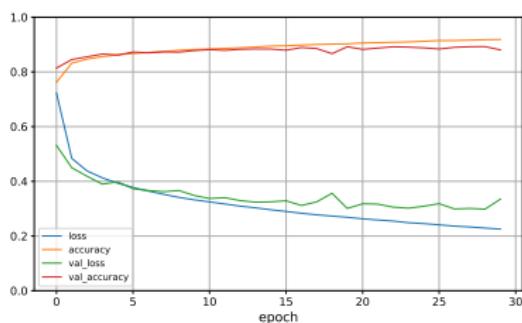


Figure 10-12. Learning curves.

Applying the trained model on the test set

```
>>> model.evaluate(X_test, y_test)
10000/10000 [=====] - 0s 29us/sample - loss: 0.3340 - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

Example: Building an Image Classifier Using the Sequential API (cont.)

Using the model to make predictions

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0.03, 0. , 0.01, 0. , 0.96],
       [0. , 0. , 0.98, 0. , 0.02, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')

# compare with the ground truth
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

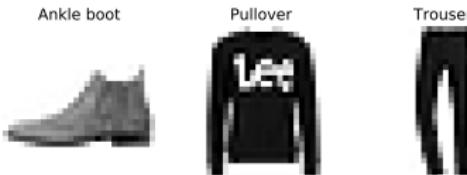


Figure 10-13. Correctly classified Fashion MNIST images.

Example: Building a Regression MLP Using the Sequential API

The code below is for the California housing problem.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_full, y_train_full, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))

# evaluate the model on the test set
mse_test = model.evaluate(X_test, y_test)

X_new = X_test[:3]
y_pred = model.predict(X_new)
```

Example: Building a Regression MLP Using the Sequential API (cont.)

A close inspection on the output shows that this simple MLP does not produce very good predictions. Sometimes it is useful to build neural networks with more complex topologies, such as using the *Functional API*.

Both *Functional API* and *Subclassing API* are outside the scope of the unit.

Saving and Restoring a Model

For both the Sequential API and the Functional API, we start by declaring the layers we want to use and how they should be connected. The actual feeding of data for training is done after the neural network has been constructed. This has the advantages that the model can be saved, cloned, and shared.

Saving a trained Keras model is very simple:

```
model = keras.models.Sequential([...]) # or keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")    # HDF5 format
```

Restoring a model from disk is equally simple:

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

If your model takes a long time to train, it is useful to save **checkpoints** at regular intervals during training.

Saving and Restoring a Model (cont.)

Using Callbacks If your model takes a long time to train, it is useful to save **checkpoints** at regular intervals during training. To do so, we use **callbacks** to tell the **fit()** method to save checkpoints.

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",
                                                save_best_only=True)
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb])
model = keras.models.load_model("my_keras_model.h5") # roll back to best model
```

By default, a model checkpoint is saved at the end of each epoch. Another way to implement early stopping is to simply use the **EarlyStopping** callback to interrupt training when no progress is observed on the validation set for a number of epochs (defined by the **patience** argument):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10, restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100, validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb, early_stopping_cb])
```

Using TensorBoard for Visualization

TensorBoard is a great interactive visualization tool for viewing the learning curves during training, compare learning curves between multiple runs, etc. It comes with the installation of TensorFlow.

To use it, your program needs to output the data that you want to visualize to special binary log files called *event files*.

We need to specify a root log directory for storing our TensorBoard logs. We can also include extra information in the log directory name, such as hyperparameter values that you are testing for inspection in TensorBoard:

```
import os
root_logdir = os.path.join(os.curdir, "my_logs")

def get_run_logdir():
    import time
    run_id = time.strftime("run_%Y_%m_%d-%H_%M_%S")
    return os.path.join(root_logdir, run_id)

run_logdir = get_run_logdir() # e.g., './my_logs/run_2020_04_17-15_15_22'
```

Using TensorBoard for Visualization (cont.)

```
[...] # Build and compile your model  
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)  
history = model.fit(X_train, y_train, epochs=30,  
                     validation_data=(X_valid, y_valid),  
                     callbacks=[tensorboard_cb])
```

To start TensorBoard:

```
$ tensorboard --logdir=./my_logs --port=6006
```

Once the server has been started, open a web browser and go to
<http://localhost:6006>.

Alternatively, start TensorBoard inside Jupyter:

```
%load_ext tensorboard  
%tensorboard --logdir=./my_logs --port=6006
```

Using TensorBoard for Visualization (cont.)

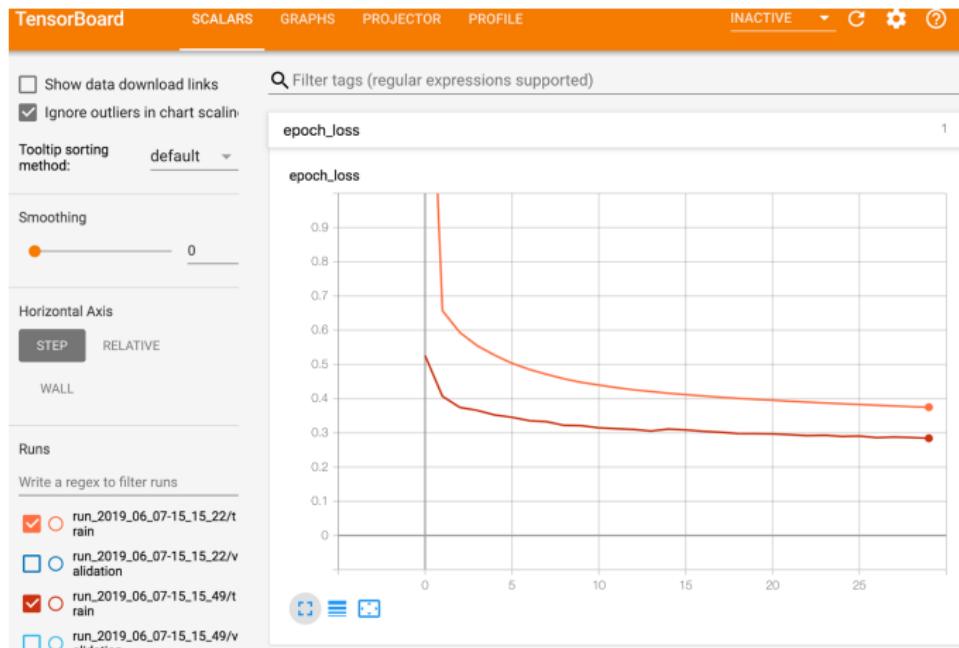


Figure 10-17. Visualizing learning curves with TensorBoard.

Using TensorBoard for Visualization (cont.)

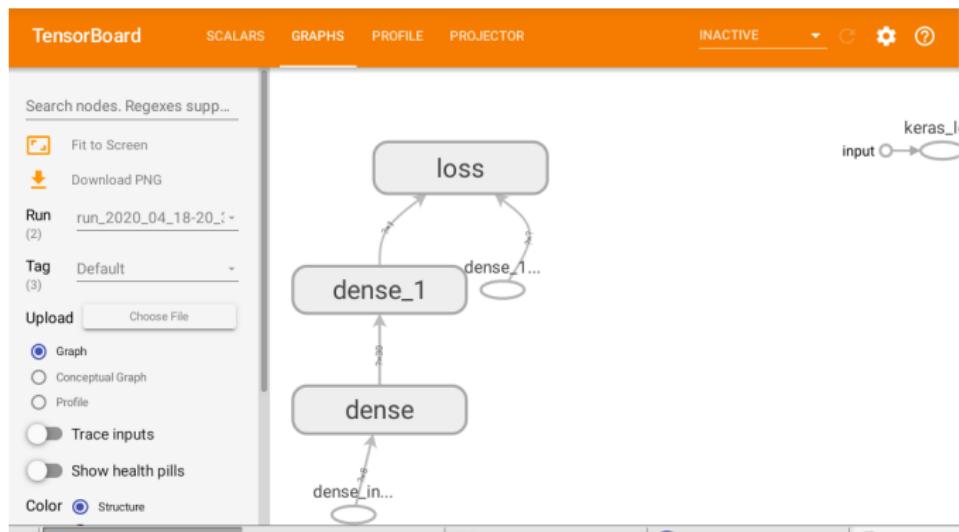


Figure 10-17. Visualizing the Network Graph with TensorBoard.

You can also visualize the learned weights (projected to 3D) and the profiling traces.

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. We can use [GridSearchCV](#) or [RandomizedSearchCV](#) to explore the hyperparameter space.

There are also many Python libraries you can use to optimize hyperparameters: [Keras Tuner](#), [Scikit-Optimize \(skopt\)](#), [Spearmint](#), etc.

Despite all this exciting progress and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter so that you can build a quick prototype and restrict the search space. The remaining slides provide guidelines for choosing some of the main hyperparameters.

Fine-Tuning Neural Network Hyperparameters (cont.)

Number of Hidden Layers

- For many problems you can start with just one or two hidden layers and the neural network will work just fine, e.g., 97% (or above 98%) accuracy can be reached on the MNIST dataset using just one (or two) hidden layer(s) with a few hundred neurons.
- For more complex problems, you can increase the number of hidden layers until you start overfitting the training set. Typically, for complex image recognition or speech recognition tasks, dozens (maybe hundreds) of hidden layers (but not fully connected) are required.
- Can also reduce the number hidden layers by training a network that has been pre-trained on a similar problem. This is known as *transfer learning*.

Fine-Tuning Neural Network Hyperparameters (cont.)

Number of Neurons per Hidden Layer

- Obviously the number of neurons in the input and output layers is determined by the type of input and output your task requires.
- As for the hidden layers, a common practice is to size them to form a **pyramid**, with fewer and fewer neurons at each layer – the rationale being that many low-level features can coalesce into far fewer high-level features. However, a common practice today is to have the same number of neurons in each hidden layer. This also helps to reduce the number of parameters to tune.
- Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting, then use early stopping and other regularization techniques to prevent it from overfitting.

Fine-Tuning Neural Network Hyperparameters (cont.)

Activation Functions

- In most cases you can use the **ReLU** activation function in the hidden layers (or one of its variants). It is a bit faster to compute than other activation functions, and Gradient Descent does not get stuck as much on plateaus (as opposed to the logistic function or the hyperbolic tangent function, which saturate at 1).
- For the output layer, the **softmax** activation function is generally a good choice for **classification tasks** when the classes are mutually exclusive. When they are not mutually exclusive (or when there are just two classes), you generally want to use the **logistic function**. For **regression tasks**, you can simply use **no activation function** at all for the output layer.

Fine-Tuning Neural Network Hyperparameters (cont.)

Optimizer

- Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will see several advanced optimizers in the next lecture.

Batch size

- The common recommendation is to use larger batch size that can fit in the GPU RAM; however, in practice, large batch sizes often lead to training instabilities, especially at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. Currently, small batch sizes (2 to 32) are preferred by some researchers while large batch sizes are recommended by others. *Learning rate warmup* has been suggested to use with large batch sizes.

Fine-Tuning Neural Network Hyperparameters (cont.)

Number of iterations

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

Summary

- Understand the architecture of Perceptrons
- Know how to train a Perceptron and understand its weakness
- Understand multi-layer Perceptrons and backpropagation
- Understand what activation functions are and know a few commonly used activation functions
- Know how to train a DNN using Keras in TensorFlow
- Understand how to use TensorBoard
- Understand some of the important hyperparameters in a DNN and how to fine tune them; in particular, the number of hidden layers, the number of neurons per layer, and the activation functions.

For next week

Work through the lab sheet and attend the supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read up to Chapters 11&12 on [Training Deep Neural Networks with TensorFlow](#)

And that's all for this week's lecture.

Have a good week.

CITS5508 Machine Learning

Lectures for Semester 1, 2021

Lecture Week 10: Book Chapter 11

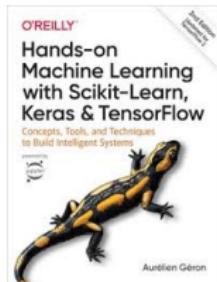
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 11.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Chapter Eleven

Training Deep Neural Networks

Here are the main topics we will briefly cover:

- Vanishing/Exploding Gradients Problems
 - Glorot and He initialization
 - Nonsaturating activation functions
 - Batch normalization
 - Gradient clipping
- Faster Optimizers
 - Momentum optimization
 - Nesterov Accelerated Gradient
 - RMSProp
 - Adam and Nadam optimization
 - Learning rate scheduling
- Avoiding overfitting
 - ℓ_1 and ℓ_2 regularization
 - Dropout

Problems in training DNNs

Training a DNN isn't a walk in the park. Here are some of the problems you could run into:

- the tricky *vanishing gradients* or *exploding gradients* problems;
- insufficient training data or insufficient labelled data to train the large network;
- training may be extremely slow;
- risk of overfitting the training data for a model with millions of parameters especially if there are not enough training instances or if they are too noisy.

Vanishing/Exploding Gradients Problems

- The *vanishing gradients* problem occurs when gradients become smaller and smaller as the algorithm progresses down to the lower layers. As a result, the Gradient Descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution or the convergence is extremely slow. In other cases, we have the opposite problem, i.e., gradients become larger and larger, leading to the *exploding gradients* problem.
- This *vanishing gradients* problem was found¹ to be the combination of the popular **logistic sigmoid activation function** and the **weight initialization technique** that was most popular at the time.

¹Xavier Glorot and Yoshua Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (2010): 249-256.

Vanishing/Exploding Gradients Problems (cont.)

- weight initialization techniques (usually drawn from $\mathcal{N}(0, 1)$);
- the variance of the outputs of each layer is much greater than the variance of its inputs;
- the logistic activation function has mean 0.5 rather than 0. Also, when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0.

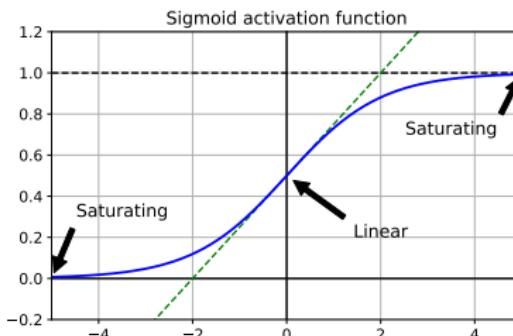


Figure 11-1. Logistic activation function saturation.

Vanishing/Exploding Gradients Problems (cont.)

Xavier or Glorot Initialization

Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights between neurons must be initialized randomly as described below:

Glorot initialization (when using the logistic activation function)

- use the normal distribution with mean 0 and variance $\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$, or
- use a uniform distribution over the interval $[-r, +r]$ with $r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$

where² fan_{in} and fan_{out} denote the numbers of input and output units of a layer and $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}})/2$

²Gluseppe Bonacorso, "Mastering Machine Learning Algorithms: Expert Techniques to Implement Popular Machine Learning Algorithms and Fine-tuning your Models", 2nd ed., 2020.

Vanishing/Exploding Gradients Problems (cont.)

He Initialization

He et al³ provide similar weight initialization strategies for the ReLU activation function and its variants (including ELU):

He initialization

Activation function	Uniform distr. $\mathcal{U}([-r, +r])$	Normal distr. $\mathcal{N}(0, \sigma^2)$
Logistic	$r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$	$\sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$
Tanh	$r = 4\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$	$\sigma^2 = \frac{16}{\text{fan}_{\text{avg}}}$
ReLU (& variants)	$r = \sqrt{2}\sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$	$\sigma^2 = \frac{2}{\text{fan}_{\text{avg}}}$

³Kaiming He et al., Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, CVPR 2015, 1026-1034.

Vanishing/Exploding Gradients Problems (cont.)

Summary of initialization parameters for each type of activation function

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

Table 11-1. Initialization parameters for each type of activation function Initialization Activation

(He's σ^2 here is given in terms of fan_{in} . If $fan_{in} = fan_{out} = fan_{avg}$, then the formula for σ^2 is consistent with that on the previous slide)

Vanishing/Exploding Gradients Problems (cont.)

Weight initialization examples in Keras

Example: Use He's initialization strategy with the Normal distribution:

```
keras.layers.Dense(10, activation="relu", kernel_initializer="he_normal")
```

Example: Use the `VarianceScaling` initializer based on `fanavg`:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2., mode='fan_avg',
                                                 distribution='uniform')
keras.layers.Dense(10, activation="sigmoid", kernel_initializer=he_avg_init)
```

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- The ReLU⁴ activation function is commonly used because it does not saturate for positive values. Unfortunately, like the logistic activation function, it suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively “die,” meaning they stop outputting anything other than 0. This happens when the input to ReLU is negative.

⁴Recall that the ReLU (rectified linear unit) activation function is defined as: $\text{ReLU}(z) = \max(0, z)$.

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- To overcome this problem, variants of ReLU have been proposed, including
 - leaky ReLU*, defined as $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$, where α is a small constant, e.g., $\alpha = 0.01$. Researchers found that a larger leak, e.g., $\alpha = 0.2$, tends to give better performance.

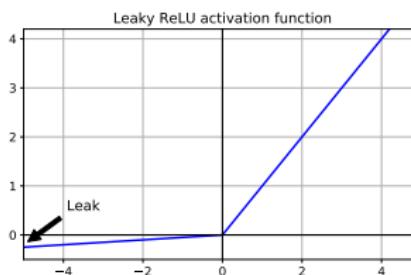


Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values.

- Randomized leaky ReLU (RReLU)*, where α is picked randomly in a given range during training and is fixed to an average value during testing.

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- Variants of ReLU...
 - *Parametric leaky ReLU (PReLU)* – same as RReLU, except that α is learned during training together with the connection weights.
 - *Exponential linear unit⁵ (ELU)* is defined as

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha \exp(z) - 1 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

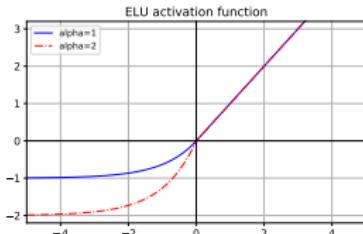


Figure 11-3. ELU activation function.

When $\alpha = 1$, the derivative at $z = 0$ is continuous also.

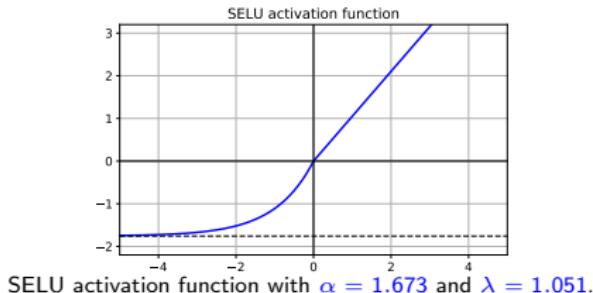
⁵Djork-Arné Clevert et al., "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," International Conference on Learning Representations, 2016.

Vanishing/Exploding Gradients Problems (cont.)

Nonsaturating Activation Functions

- Variants of ReLU...

- *Scaled ELU*⁶ (*SELU*) – as the name suggests, this is a scaled version of ELU, i.e., $\text{SELU}_{\alpha,\lambda}(z) = \lambda \text{ELU}_\alpha(z)$, with an extra scaling parameter λ .



SELU activation function with $\alpha = 1.673$ and $\lambda = 1.051$.

SELU has the self-normalization property, i.e., preserving the mean and variance of each layer's output during training. Unfortunately, this property is easily broken when used together with ℓ_1 or ℓ_2 , dropout, and non-sequential topology networks.

⁶Günter Klambauer et al., "Self-Normalizing Neural Networks," Proceedings of the 31st International Conference on Neural Information Processing Systems (2017): 972-981.

Vanishing/Exploding Gradients Problems (cont.)

Which activation function should you use?

- In general, it is recommended to follow this order: SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic.
- If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at $z = 0$).
- If runtime latency is an issue, then you may prefer leaky ReLU.
- If you don't want to tweak yet another hyperparameter, you may use the default α values used by Keras (e.g., 0.3 for leaky ReLU).
- If you have spare time and computing power, you can try RReLU or PReLU.
- As ReLU is optimized in many libraries and hardware accelerators, if speed is your priority, ReLU might still be the best choice.

Vanishing/Exploding Gradients Problems (cont.)

Batch Normalization

Although using He initialization along with ELU (or any variant of ReLU) can mitigate the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training. To overcome these problems, the *batch normalization*⁷ algorithm was proposed:

- An extra operation is inserted before or after the activation function in each hidden layer to normalize the mean to 0 and the variance to 1.
- This introduces two trainable parameters, γ and β , that the training process must learn together with the connection weights.
- The mean and variance of each layer's outputs for entire training set are estimated using the moving average of the minibatches. They are used in the predictions on the testing set.

⁷ Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ICML (2015): 448-456.

Vanishing/Exploding Gradients Problems (cont.)

Batch Normalization (cont.)

Summary of the algorithm:

$$\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2$$

$$\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$$

where $\boldsymbol{\mu}_B$ and σ_B^2 are the mean and variance vectors computed from the minibatch B ; $\hat{\mathbf{x}}^{(i)}$ is the zero-centred and normalized input vector used internally in the BN layer; γ and β are the trainable scale and shift vectors (one parameter per input unit); and \otimes represents element-wise multiplication.

Vanishing/Exploding Gradients Problems (cont.)

Batch Normalization (cont.)

- Batch Normalization (BN) has been found to also act like a regularizer, reducing the need for other regularization techniques (such as *dropout*);
- BN helps to remove the need for normalizing the input data (e.g., using `StandardScaler`);
- There is a runtime penalty though, due to the extra computations required at each layer; however, this is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance.

Vanishing/Exploding Gradients Problems (cont.)

Implementing Batch Normalization with Keras

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

Vanishing/Exploding Gradients Problems (cont.)

Implementing Batch Normalization with Keras

```
>>> model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
batch_normalization (BatchNo	(None, 784)	3136
dense (Dense)	(None, 300)	235500
batch_normalization_1 (Batch	(None, 300)	1200
dense_1 (Dense)	(None, 100)	30100
batch_normalization_2 (Batch	(None, 100)	400
dense_2 (Dense)	(None, 10)	1010

```
Total params: 271,346
```

```
Trainable params: 268,978
```

```
Non-trainable params: 2,368
```

Vanishing/Exploding Gradients Problems (cont.)

Gradient Clipping

- Another popular technique to lessen the exploding gradients problem is to simply clip the gradients during backpropagation so that they never exceed some threshold (this is mostly useful for *recurrent neural networks* (RNNs) as BN is tricky to use in RNNs. This is called *Gradient Clipping*⁸.
- For example,

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

will clip every component of the gradient vector to a value between `-1.0` and `1.0`. An alternative is to use `clipnorm=1.0` if you want to retain the gradient vector orientation.

⁸Razvan Pascanu et al., "On the Difficulty of Training Recurrent Neural Networks," ICML (2013): 1310-1318.

Faster Optimizers

Huge speed boost comes from using a faster optimizer than the regular Gradient Descent optimizer. We will look at 4-5 of the most popular algorithms:

- *Momentum optimization*,
- *Nesterov accelerated gradient*,
- *RMSProp*, and
- *Adam and Nadam optimization*

Faster Optimizers (cont.)

Momentum optimization

- Recall that Gradient Descent simply updates the weights θ as follows: $\theta \leftarrow \theta - \nabla_{\theta} J(\theta)$. It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.
- In momentum optimization, a *momentum vector* m is used to keep track of gradients in the previous iterations. The updates of the weights involve using only m , which contains information about the past momentum and the current error gradient. That is, the gradient is used as an acceleration term, rather than a speed term of the update. The algorithm can also help roll past local optima. It introduces a new hyperparameter $\beta < 1$ (usually set to 0.9). The update equations are:

$$\begin{aligned}m &\leftarrow \beta m - \eta \nabla_{\theta} J(\theta) \\ \theta &\leftarrow \theta + m\end{aligned}$$

Note that past momentum vector m exponentially decays over iterations.

Faster Optimizers (cont.)

Momentum optimization (cont.)

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

Faster Optimizers (cont.)

Nesterov Accelerated Gradient

Proposed by Nesterov⁹, this is a small variant of the momentum optimizer which measures the gradient of the loss function not at the local position θ but slightly ahead in the direction of the momentum, at $\theta + \beta m$.

$$\begin{aligned}m &\leftarrow \beta m - \eta \nabla_{\theta} J(\theta + \beta m) \\ \theta &\leftarrow \theta + m\end{aligned}$$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so the update ends up slightly closer to the optimum.

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

⁹Yurii Nesterov, "A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O(1/k²)," Doklady AN USSR 269 (1983): 543-547.

Faster Optimizers (cont.)

RMSProp

In the *RMSProp* algorithm¹⁰, instead of the moment vector \mathbf{m} , a vector \mathbf{s} is used. The update equations are:

$$\begin{aligned}\mathbf{s} &\leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ \theta &\leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \epsilon}\end{aligned}$$

where $\beta < 1$ (default value 0.9 works well) and η are hyperparameters; ϵ is a small constant to avoid division by 0; operators \otimes and \oslash denote element-wise multiplication and division.

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

RMSProp was the preferred optimization algorithm of many researchers until Adam optimization came around.

¹⁰created by Tijmen Tieleman and Geoffrey Hinton in 2012.

Faster Optimizers

Adam Optimization

Adam (<https://goo.gl/Un8Axa>),¹¹ which stands for *adaptive moment estimation*, combines the ideas of Momentum optimization and RMSProp. So the algorithm involves both the \mathbf{m} and \mathbf{s} vectors. Instead of β , it has hyperparameters β_1 and β_2 . The update equations are:

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\hat{\mathbf{m}} \leftarrow \mathbf{m} / (1 - \beta_1^t)$
4. $\hat{\mathbf{s}} \leftarrow \mathbf{s} / (1 - \beta_2^t)$
5. $\theta \leftarrow \theta - \eta \hat{\mathbf{m}} \oslash \sqrt{\hat{\mathbf{s}}} + \epsilon$

where steps 3 and 4 are normalization to help increase the magnitude \mathbf{m} and \mathbf{s} as both vectors are initialized at around zero (so may be a bit small); t is the iteration number.

¹¹ "Adam: A Method for Stochastic Optimization," D. Kingma, J. Ba (2015).

Faster Optimizers (cont.)

Adam Optimization (cont.)

Here is how to create an Adam optimizer using Keras:

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

The momentum decay hyperparameter β_1 is typically initialized to 0.9, while the scaling decay hyperparameter β_2 is often initialized to 0.999.

Faster Optimizers (cont.)

Nadam Optimization

*Nadam optimization*¹² is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In Dozat's report, he compares many different optimizers on various tasks and finds that Nadam generally outperforms Adam but is sometimes outperformed by RMSProp.

¹²Timothy Dozat, "Incorporating Nesterov Momentum into Adam" (2016)

Faster Optimizers (cont.)

Learning rate scheduling

Finding a good learning rate is very important. If you set it too high, training may diverge; if you set it too low, training will eventually converge to the optimum, but it will take a very long time.

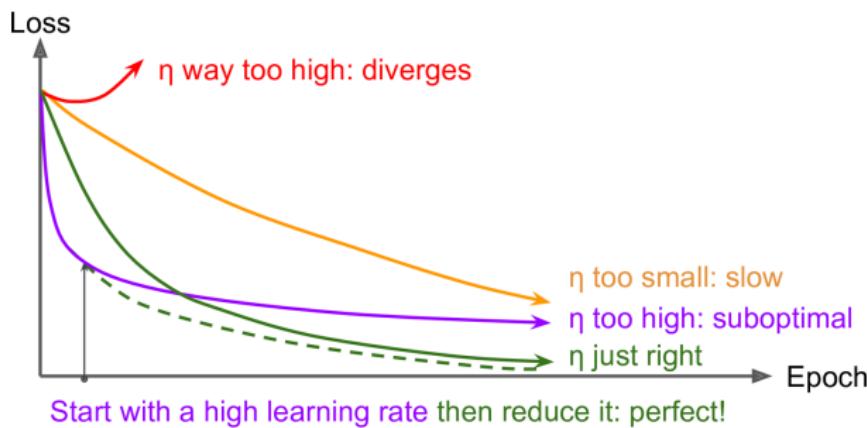


Figure 11-8. Learning curves for various learning rates η .

Faster Optimizers (cont.)

Learning rate scheduling (cont.)

If you start with a large learning rate and then reduce it once training stops making fast progress, you can reach a good solution faster than with the optimal constant learning rate.

There are many different strategies to reduce the learning rate during training. It can also be beneficial to start with a low learning rate, increase it, then drop it again. These strategies are called *learning schedules*.

Faster Optimizers (cont.)

Learning rate scheduling (cont.)

Some commonly used learning schedules are listed below.

Power scheduling: Setting the learning rate as a function of the iteration number t , as follows:

$$\eta(t) = \frac{\eta_0}{(1 + t/s)^c}.$$

where η_0 is the initial learning rate, the power c (typically set to 1) and the step s are hyperparameters. After s steps, it is down to $\eta_0/2$. After s more steps, it is down to $\eta_0/3$, and so on.

Faster Optimizers (cont.)

Learning rate scheduling (cont.)

Exponential scheduling: Setting the learning rate to:

$$\eta(t) = \eta_0 0.1^{t/s}.$$

where the learning rate gradually drops by a factor of 10 every s steps.

Performance scheduling: Measure the validation error every N steps (just like for early stopping), and reduce the learning rate by a factor of λ when the error stops dropping.

Faster Optimizers (cont.)

Learning rate scheduling (cont.)

1Cycle scheduling¹³ Contrary to the other approaches, 1Cycle starts by increasing η_0 , growing linearly up to η_1 halfway through training, then it decreases the learning rate linearly down to η_0 again during the second half of training, finishing the last few epochs by dropping the learning rate down by several orders of magnitude (still linearly). The author reported that good validation accuracy was achieved with fewer training epochs on the CIFAR10 image dataset.

¹³ Leslie N. Smith, "A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 Learning Rate, Batch Size, Momentum, and Weight Decay," arXiv preprint arXiv:1803.09820 (2018).

Faster Optimizers (cont.)

Learning rate scheduling (cont.)

Examples:

1. Implementing power scheduling in Keras:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

The `decay` parameter is the same as $1/s$ in the formula. Keras assumes that $c = 1$.

2. Implementing your own exponential schedule:

```
def exponential_decay(lr0, s):  
    def exp_decay_fn(epoch):  
        return lr0 * 0.1** (epoch / s)  
    return exp_decay_fn  
  
my_exp_decay_fn = exponential_decay(lr0=0.01, s=20)  
  
lr_scheduler = keras.callbacks.LearningRateScheduler(my_exp_decay_fn)  
  
history = model.fit(X_train, y_train, ..., callbacks=[lr_scheduler])
```

Avoiding Overfitting

ℓ_1 and ℓ_2 Regularization

Just like you did for simple linear models, you can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights.

Example: To apply ℓ_2 regularization to a Keras layer's connection weights, using a regularization factor of 0.01:

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

Avoiding Overfitting (cont.)

Dropout

Dropout is one of the most popular regularization techniques for DNNs is arguably *dropout*.¹⁴ It has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

¹⁴ “Improving neural networks by preventing co-adaptation of feature detectors,” G. Hinton et al. (2012).

Avoiding Overfitting (cont.)

Dropout (cont.)

It is a fairly simple algorithm:

- At each training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being **temporarily “dropped out,”** meaning it will be entirely ignored during this training step; however, it may be active during the next step (see Figure 11-9).
- The hyperparameter p is called the *dropout rate*, and it is typically set to 50%.
- After training, neurons don't get dropped anymore. That is, at testing, you simply use the connection weights that have been learned from the training process.

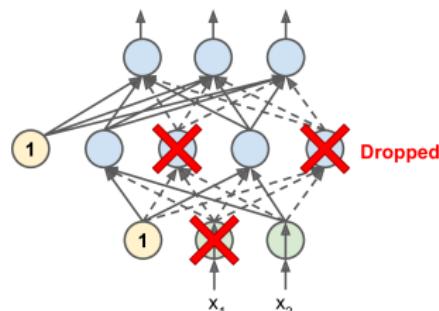


Figure 11-9. Dropout regularization

Avoiding Overfitting (cont.)

Dropout (cont.)

Example: The code below applies dropout regularization before every Dense layer, using a dropout rate of 0.2:

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

Summary

- Understand the vanishing and exploding gradients problems
- Understand the importance of properly initializing connection weights
- Know a few unsaturating activation functions
- Understand when to use batch normalization and how it works
- Know a few fast optimizers and their associated hyperparameters
- Understand learning rate scheduling.
- Understand ℓ_1 and ℓ_2 regularization
- Understand the concept of dropout

For next week

Work through the lab sheet and attend a supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read up to Chapters 14 on [Deep Computer Vision Using Convolutional Neural Networks](#)

And that's all for this week's lecture.

Have a good week.

CITS5508 Machine Learning
Lectures for Semester 1, 2021
Lecture Week 11: Book Chapters 14

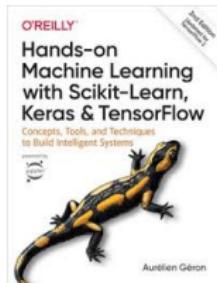
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapters 14.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Chapter Fourteen

Deep Computer Vision Using Convolutional Neural Networks (CNNs)

Here are the main topics we will cover:

- Convolutional layers
- Filters and feature maps
- Stacking multiple feature maps
- Pooling layers
- TensorFlow implementation for CNNs
- Memory requirements of CNNs
- CNN architectures

Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) emerged from the study^{1,2} of the brain's visual cortex.

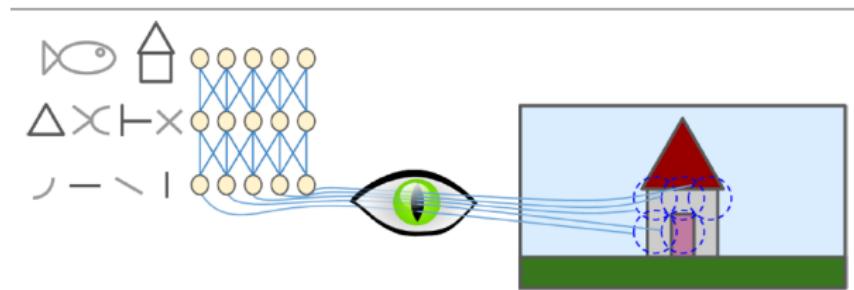


Figure 14-1. Biological neurons in the visual cortex respond to specific patterns in small regions of the visual field called *receptive fields*; as the visual signal makes its way through consecutive brain modules, neurons respond to more complex patterns in larger receptive fields.

¹David H. Hubel, "Single Unit Activity in Striate Cortex of Unrestrained Cats," *The Journal of Physiology* 147 (1959): 226-238.

²David H. Hubel and Torsten N. Wiesel, "Receptive Fields of Single Neurons in the Cat's Striate Cortex," *The Journal of Physiology* 148 (1959): 574-591.

Convolutional Neural Networks (CNNs)

These studies of the visual cortex inspired the *neocognition*³ (introduced in 1980), which gradually evolved into what we now call *convolutional neural networks*.

In the last few years, thanks to the increase in computational power, the amount of available training data, and the tricks presented in Chapter 11 for training deep nets, CNNs have managed to achieve superhuman performance on some complex visual tasks.

CNNs are not restricted to *visual perception*: they are also successful at other tasks, such as *voice recognition* or *natural language processing* (NLP).

³Kuniyuki Fukushima, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position," *Biological Cybernetics* 36 (1980): 193-202.

Convolution: A Brief Introduction

Given an image \mathcal{I} and a 3×3 filter f , the convolution output of the input image \mathcal{I} by f at pixel (x, y) is given by

$$\mathcal{I}'(x, y) = \sum_{i=-1}^1 \sum_{j=-1}^1 \mathcal{I}(x + i, y + j) f(i, j),$$

where \mathcal{I}' is the output image.

It is easier to understand the convolution operation through an animation, e.g.

<https://www.youtube.com/watch?v=u1KbLD6BRJA> (1D convolution)

https://commons.wikimedia.org/wiki/File:3D_Convolution_Animation.gif

Some common 3×3 filters used in Computer Vision:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

$\frac{1}{9}$	1	1	1
1	1	1	1
1	1	1	1

0	1	0
1	-4	1
0	1	0

(From left to right: Sobel-x (for detecting vertical edges); Sobel-y (for detecting horizontal edges); uniform averaging filter (for smoothing); Laplacian operator (2nd derivative filter)).

Larger filters can also be defined. Odd size filters (5×5 , 7×7 , ...) are usually preferred.

Convolutional Layers

The most important building block of a CNN is the *convolutional layer*.

- Neurons in the first convolutional layer are not connected to every single pixel in the input image, but only to pixels in their *receptive fields*.
- In turn, each neuron in the second convolutional layer is connected only to neurons located within a small rectangle in the first layer.

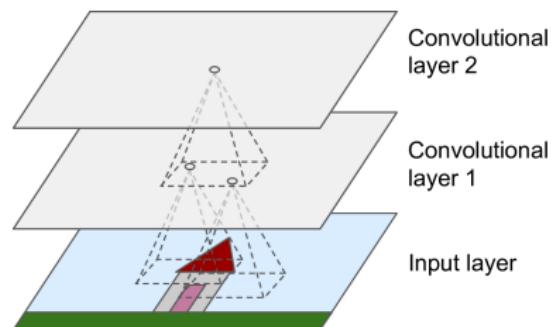
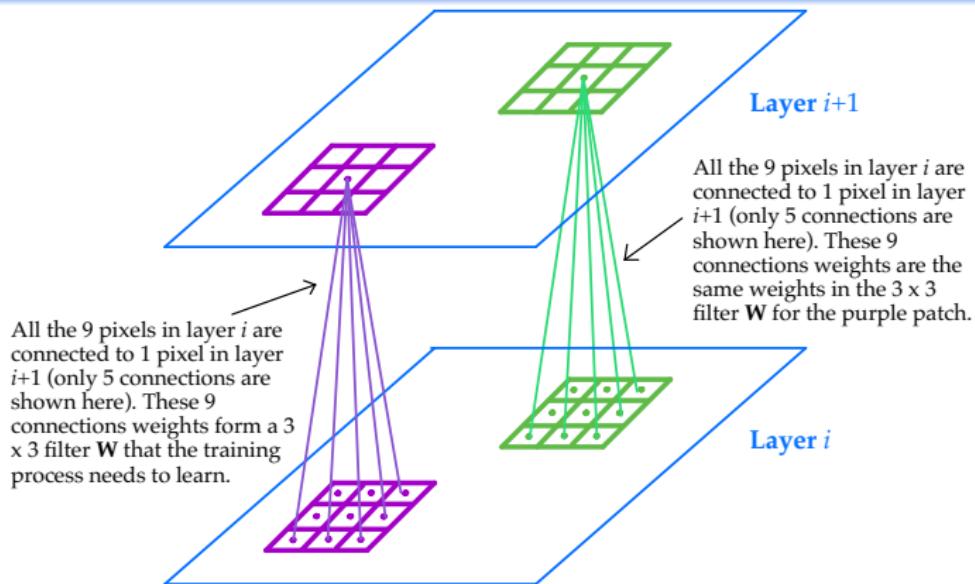


Figure 14-2. CNN layers with rectangular local receptive fields.

This architecture allows the network to concentrate on low-level features in the first hidden layer, then assemble them into higher-level features in the next hidden layer, and so on.

Convolutional Layers (cont.)



If only one 3×3 filter \mathbf{W} is used for the convolution in layer i , then only 9 parameters from \mathbf{W} plus a bias parameter need to be estimated, regardless of the numbers of rows and columns of the feature map in layer i .

Connections Between Layers

It is also possible to connect a large input layer to a much smaller layer by spacing out the receptive fields. The distance between two consecutive receptive fields is called the *stride*.

In the diagram, a 5×7 input layer (plus zero padding) is connected to a 3×4 layer, using 3×3 receptive fields and a stride of 2 in both directions, but it does not have to be so. A neuron located in row i , column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i s_h$ to $i s_h + f_h - 1$, columns $j s_w$ to $j s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

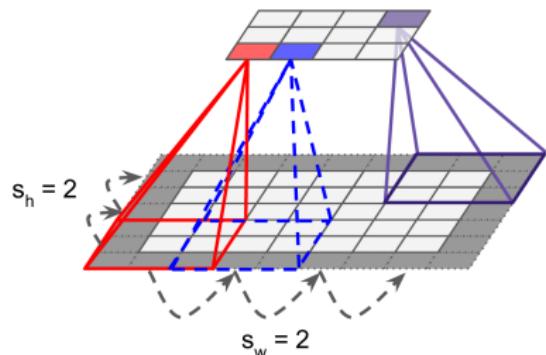


Figure 14-4. Reducing dimensionality using a stride of 2 in both directions ($s_w = s_h = 2$).

Filters

A **neuron's weights** can be represented as a **small image** the size of the receptive field. For example, Figure 14-5 shows two possible sets of weights, called ***filters*** (or ***convolution kernels***).

- The first one is represented as a black square with a vertical white line in the middle (it is a 7×7 matrix full of 0s except for the central column, which is full of 1s); neurons using these weights will ignore everything in their receptive field except for the central vertical line (since all inputs will get multiplied by 0, except for the ones located in the central vertical line).
- The second filter is a black square with a horizontal white line in the middle. Once again, neurons using these weights will ignore everything in their receptive field except for the central horizontal line.



Filters (cont.)

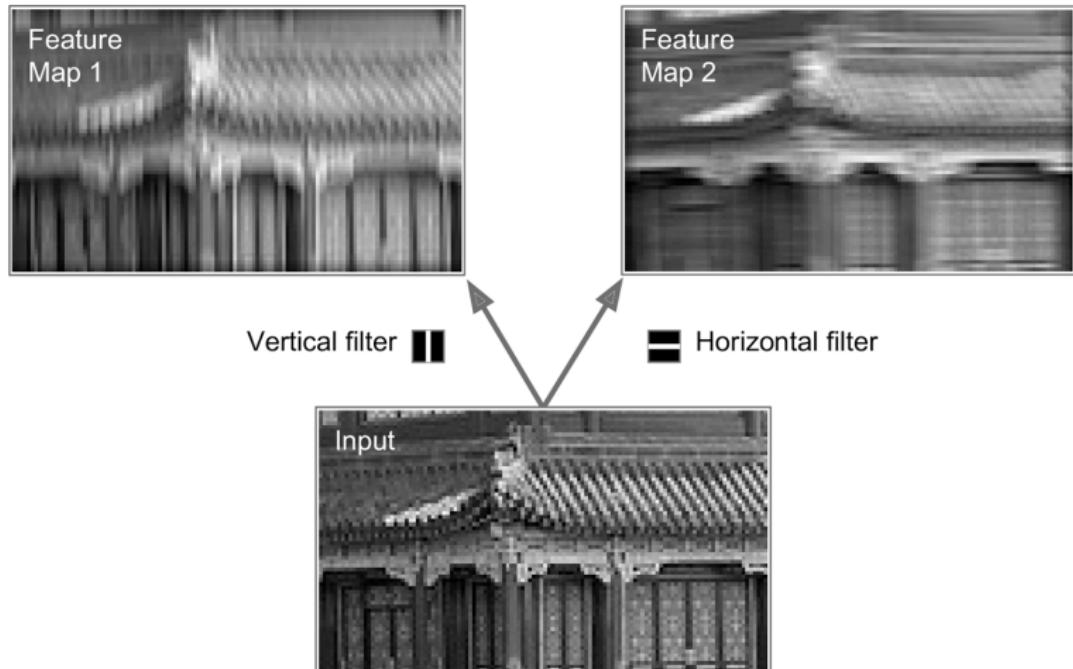


Figure 14-5. Applying two different filters to get two feature maps.

Stacking Multiple Feature Maps

So far, we have represented each convolutional layer as a thin 2D layer, but in reality it is composed of several feature maps stacked together, so it is more accurately represented in 3D.

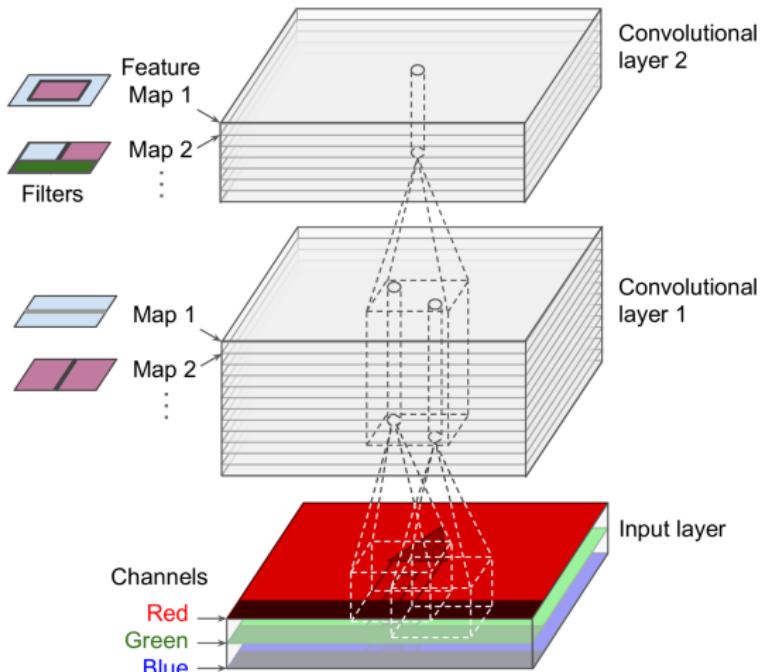


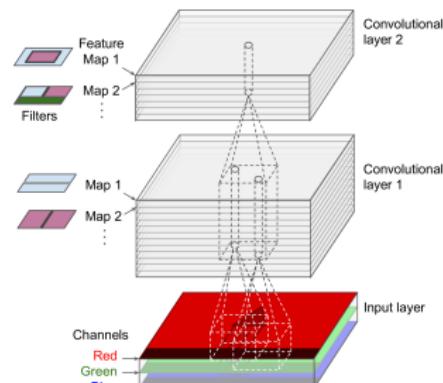
Figure 14-6. Convolution layers with multiple feature maps, and images with three channels

Stacking Multiple Feature Maps (cont.)

- Within one **feature map**, all neurons share the same parameters (weights and bias term of the filter). Neurons in different feature maps use different parameters. In fact, each feature map in a specific layer is generated by **convolving** a filter with the input of that layer.
- A neuron's receptive field is the same as described earlier, but it extends across all the previous layers' feature maps. In short, a convolutional layer simultaneously applies multiple filters to its inputs, making it capable of detecting multiple features anywhere in its inputs.
- Specifically, if a convolutional layer has **n** filters, then there are **n** feature maps generated for that layer. i.e., the output of that layer has **n channels**.

Stacking Multiple Feature Maps (cont.)

- A neuron located in row i , column j of the feature map k in a given convolutional layer ℓ is connected to the outputs of the neurons in the previous layer $\ell - 1$, located in rows $i s_h$ to $i s_h + f_h - 1$ and columns $j s_w$ to $j s_w + f_w - 1$, across all feature maps (in layer $\ell - 1$).
- Note that all neurons located in the same row i and column j but in different feature maps are connected to the outputs of the exact same neurons in the previous layer.



Stacking Multiple Feature Maps (cont.)

How the convolution is done...

Suppose that the dimension of the input image \mathbf{x} is $N_{\text{height}} \times N_{\text{width}} \times N_{\text{channels}}$.

There are K $n \times n$ filters altogether. Each filter f_k (for $k = 1, \dots, K$) must have N_{channels} channels in order for the convolution to be legal.

That is, the dimension of f_k is $n \times n \times N_{\text{channels}}$.

The k^{th} output feature map, denoted by \mathbf{z}_k , is produced as follows:

$$\mathbf{z}_k = \sum_{c=0}^{N_{\text{channels}}-1} \mathbf{x}_c * f_{k,c}$$

where $*$ denotes the convolution operation;

\mathbf{x}_c (whose dimension is $N_{\text{height}} \times N_{\text{width}}$) denotes channel c of \mathbf{x} ;

$f_{k,c}$ (a $n \times n$ weight matrix) denotes channel c of the filter f_k .

So each filter produces one feature map. A bank of K filters produce K feature maps, each of which has 1 channel.

We can also consider the K output feature maps as a single feature map of K channels.

TensorFlow Implementation

In TensorFlow,

- Each **input image** is typically represented as a **3D tensor** of shape `[height, width, channels]`.
- A **mini-batch** is represented as a **4D tensor** of shape `[minibatch_size, height, width, channels]`.
- The **weights** of a convolutional layer (i.e., all the **filters**) are represented as a 4D tensor of shape $[f_h, f_w, f_{n'}, f_n]$, where f_h and f_w are the height and width of the filters, f'_n is the number of channels (must be the same as the input image) of the filters, f_n is the total number of filters.
- The **bias terms** of a convolutional layer are simply represented as a **1D tensor** of shape $[f_n]$.

Padding Options

In the `tf.nn.conv2d()`, padding must be either “**VALID**” or “**SAME**”.

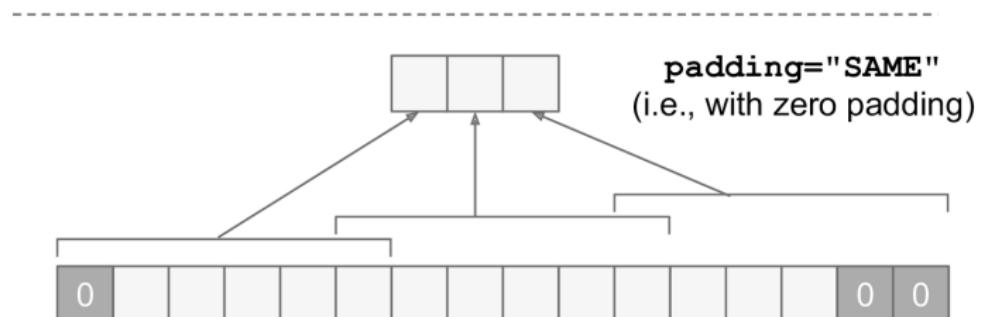
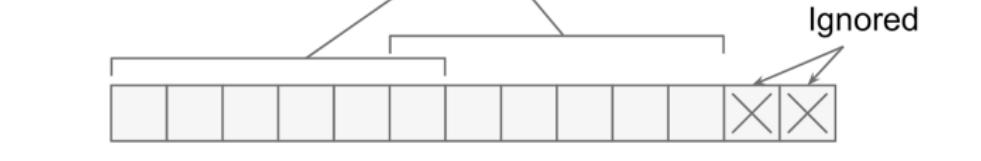


Figure 14-7. Padding options – input width: 13, filter width: 6, stride: 5

TensorFlow Implementation of a CNN Example

```
import numpy as np
from sklearn.datasets import load_sample_images

# Load sample images
china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")

images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Create 2 filters
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1 # vertical line
filters[3, :, :, 1] = 1 # horizontal line
outputs = tf.nn.conv2d(images, filters, strides=1, padding="SAME")
plt.imshow(outputs[0, :, :, 1], cmap="gray") # plot 1st image's 2nd feature map
plt.show()
```

TensorFlow Implementation of a CNN Example (cont.)

In the example on the previous slide, we manually created the filters, but in a real CNN you would let the training algorithm learn and discover the best filters automatically.

For example, we can create a `keras.layers.Conv2D` layer with 32 3×3 filters, using a stride of 1 (both horizontally and vertically) and “same” padding, and applying the ReLU activation function to its outputs:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3,  
                          strides=1, padding="same",  
                          activation="relu")
```

Memory Requirements of CNNs

A problem with CNNs is that the convolutional layers require a huge amount of RAM, especially during training, because the reverse pass of backpropagation requires all the intermediate values computed during the forward pass.

For example, consider a convolutional layer with 5×5 filters, outputting 200 feature maps of size 150×100 , with stride 1 and "same" padding. Suppose the input is a 150×100 RGB image (i.e., 3 channels). **Question: How many parameters need to be trained?** The answer is: $(5 \times 5 \times 3 + 1) \times 200 = 15,200$ (Note that the no. of parameters is independent of the height and width of the input image.)

Compared to a dense (fully connected) layer⁴, the above number is actually very small.

⁴A fully connected layer with 150×100 neurons, each connected to all $150 \times 100 \times 3$ inputs, would have $150^2 \times 100^2 \times 3 = 675$ million parameters!

Memory Requirements of CNNs (cont.)

Each of the 200 feature maps contains 150×100 neurons, and each of these neurons needs to compute a weighted sum of its $5 \times 5 \times 3 = 75$ inputs: that's a total of **225 million** float multiplications.

Not as bad as a fully connected layer, but still quite computationally intensive. This is for one training instance!

If we use single precision floats, i.e., 4 bytes (or 32 bits) per number, to store the feature maps, the memory requirement is: $150 \times 100 \times 200 \times 4 = 12$ MB RAM. This is just for 1 instance. If there are 100 instances, then it would be **1.2 GB of RAM!**

Memory Requirements of CNNs (cont.)

During testing (making prediction), the RAM occupied by one layer can be released as soon as the next layer has been computed, so you only need as much RAM as required by two consecutive layers. However, during training, everything computed during the forward pass needs to be preserved for the reverse pass, so the amount of RAM needed is (at least) the total amount of RAM required by all layers.

Pooling Layers

Pooling layers are the second common building block of CNNs.

- The pooling layers are quite easy to grasp. Their goal is to subsample (i.e., shrink) the input image in order to reduce the computational load, the memory usage, and the number of parameters (thereby limiting the risk of overfitting).
- Just like in convolutional layers, each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer, located within a small rectangular receptive field.
- You must define the pooling layer's `size`, `stride`, and the `padding type`, just like before. However, a pooling neuron has no weights; all it does is aggregate the inputs using an `aggregation function` such as the `max` or `mean`.

Pooling Layers (cont.)

In this example, we use a 2×2 max pooling kernel, a stride of 2, and no padding. Note that only the max input value in each receptive field makes it to the next layer, while the other inputs are dropped.

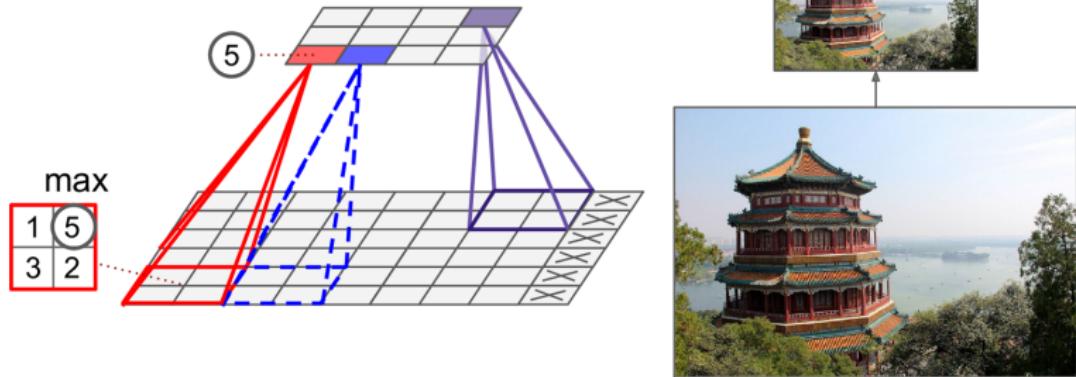


Figure 14-8. Max pooling layer (2×2 pooling kernel, stride 2, no padding).

Pooling Layers (cont.)

- A pooling layer typically works on every input channel independently, so the output depth is the same as the input depth.
- Unlike the convolutional layers where kernels have weights that need to be learned, kernels in pooling layers do not – they are just stateless sliding windows.
- Alternatively, we can pool over the depth dimension, in which case the image's spatial dimensions (height and width) remain unchanged, but the number of channels is reduced.

Pooling Layers (cont.)

Max pooling vs average pooling

- Average pooling allows more information to be preserved, while max pooling preserves only the strongest features, getting rid of all the meaningless ones, so the next layer gets a cleaner signal to work with.
- Max pooling offers stronger *translation invariance* than average pooling, and it requires slightly less work to compute.

Average pooling layers used to be very popular, but people mostly use max pooling layers now, as they generally perform better. This may seem surprising, since computing the mean generally loses less information than computing the max.

Pooling Layers (cont.)

Implementing a Pooling Layer in TensorFlow

Implementing a [max pooling](#) layer or an [average pooling](#) layer in TensorFlow is quite easy. By default, the strides default to the kernel size and “valid” padding (i.e., no padding at all) is assumed.

Example 1: create a max pooling layer using a 2×2 kernel:

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

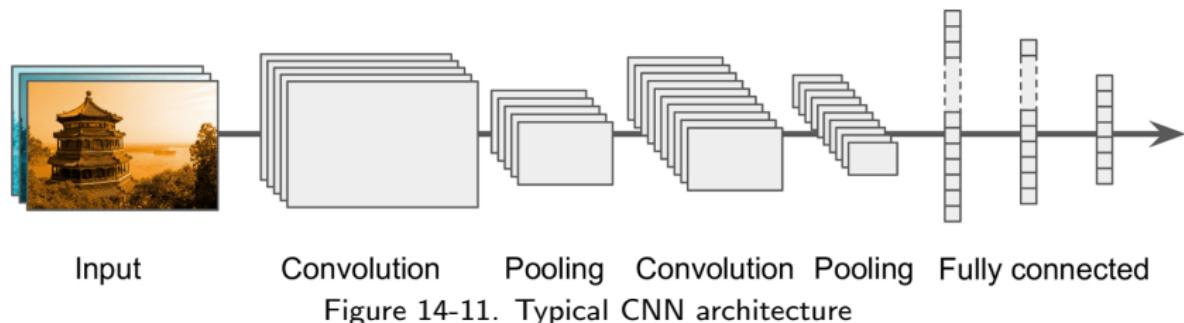
Example 2: create an average pooling layer using a 2×2 kernel:

```
max_pool = keras.layers.AvgPool2D(pool_size=2)
```

CNN Architectures

- Typical CNN architectures stack a few convolutional layers (each one generally followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then another pooling layer, and so on.
- The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps).
- Near the output layer, a regular feedforward neural network is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities).

CNN Architectures (cont.)



A common mistake is to use convolution kernels that are too large, e.g., by stacking two convolutional layers with 3×3 kernels instead of using a layer with a 5×5 kernel, it will use fewer parameters (also less computations) and will usually perform better.

One exception is for the first convolutional layer: it can typically have a large kernel (e.g., 5×5), usually with a stride of 2 or more: this will reduce the spatial dimension of the image without losing too much information.

CNN Architectures (cont.)

Example: implement a simple CNN to tackle the Fashion MNIST dataset

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

The CNN reaches over 92% accuracy on the test set. It's not state of the art, but it is pretty good, and clearly much better than what we achieved with the MLP in a previous chapter.

CNN Architectures (cont.)

Over the years, variants of this fundamental architecture have been developed, leading to amazing advances in the field. A good measure of this progress is the **error rate** in competitions such as the [ILSVRC⁵](#) [ImageNet Challenge](#). In this competition the **top-5 error rate⁶** for image classification fell from over 26% to less than 3% in just six years.

⁵ ILSVRC stands for *ImageNet Large Scale Visual Recognition Competition*. Also referred to as [ImageNet Challenge](#)

⁶ The top-five error rate is the number of test images for which the system's top five predictions did not include the correct answer.

CNN Architectures (cont.)

The LeNet-5 Architecture (1998)

The LeNet-5 architecture is perhaps the most widely known CNN architecture. It was created by Yann LeCun in 1998 and widely used for handwritten digit recognition (MNIST). It is composed of the following layers:

Table 14-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully connected	–	10	–	–	RBF
F6	Fully connected	–	84	–	–	tanh
C5	Convolution	120	1×1	5×5	1	tanh
S4	Avg pooling	16	5×5	2×2	2	tanh
C3	Convolution	16	10×10	5×5	1	tanh
S2	Avg pooling	6	14×14	2×2	2	tanh
C1	Convolution	6	28×28	5×5	1	tanh
In	Input	1	32×32	–	–	–

For the output layer, the cross entropy cost function is now preferred, as it penalizes bad predictions much more, producing larger gradients and thus converging faster.

CNN Architectures (cont.)

The AlexNet Architecture (2012)

- The AlexNet CNN architecture won the 2012 ImageNet ILSVRC challenge by a large margin: it achieved 17% top-5 error rate while the second best achieved only 26%!
- It was developed by Alex Krizhevsky (hence the name), Ilya Sutskever, and Geoffrey Hinton.
- It is quite similar to LeNet-5, only much larger and deeper, and it was the first to stack convolutional layers directly on top of each other, instead of stacking a pooling layer on top of each convolutional layer.

CNN Architectures (cont.)

The AlexNet Architecture (2012)

Table 14-2. AlexNet architecture

To reduce overfitting, the authors used two regularization techniques:

(1) **dropout** (with a 50% dropout rate) during training to the outputs of layers F9 and F10.

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	valid	–
C7	Convolution	256	13×13	3×3	1	same	ReLU
C6	Convolution	384	13×13	3×3	1	same	ReLU
C5	Convolution	384	13×13	3×3	1	same	ReLU
S4	Max pooling	256	13×13	3×3	2	valid	–
C3	Convolution	256	27×27	5×5	1	same	ReLU
S2	Max pooling	96	27×27	3×3	2	valid	–
C1	Convolution	96	55×55	11×11	4	valid	ReLU
In	Input	3 (RGB)	227×227	–	–	–	–

(2) **data augmentation** by randomly shifting the training images by various offsets, flipping them horizontally, and changing the lighting conditions.

CNN Architectures (cont.)

Data augmentation artificially increases the size of the training set by generating many realistic variants of each training instance. As it helps reduce overfitting, it is considered as a regularization technique.

For example, we can augment the training set by slightly shifting, rotating, and resizing every picture in the training set by various amounts and add the resultant pictures to the training set.

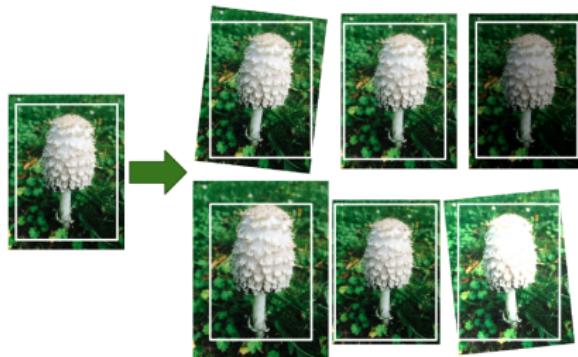


Figure 14-12. Generating new training instances from existing ones.

CNN Architectures (cont.)

The GoogLeNet Architecture (2014)

- The GoogLeNet architecture was developed by Christian Szegedy et al⁷ from Google Research.
- It won the ILSVRC 2014 challenge by pushing the top-5 error rate below 7%. This great performance came in large part from the fact that the network was **much deeper than previous CNNs** (see Figure 14-14). This was made possible by sub-networks called ***inception modules***, which allow GoogLeNet to use parameters much more efficiently than previous architectures: GoogLeNet actually has 10 times fewer parameters than AlexNet (roughly 6 million instead of 60 million).

⁷Christian Szegedy et al., "Going Deeper with Convolutions," CVPR 2015.

CNN Architectures (cont.)

The VGGNet Architecture (2014)

The runner-up in the ILSVRC 2014 challenge was VGGNet⁸ developed by Karen Simonyan and Andrew Zisserman from the Visual Geometry Group (VGG) research lab at Oxford University.

It had a very simple and classical architecture, with 2 or 3 convolutional layers and a pooling layer, then again 2 or 3 convolutional layers and a pooling layer, and so on (reaching a total of just 16 or 19 convolutional layers, depending on the VGG variant), plus a final dense network with 2 hidden layers and the output layer. It used only 3×3 filters, but many filters.

⁸Karen Simonyan and Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv preprint arXiv:1409.1556 (2014).

CNN Architectures (cont.)

More recent CNN architectures include:

Xception⁹ (2016). This is a variant of GoogLeNet proposed in 2016 by Francois Chollet (the author of Keras), and it significantly outperformed Inception-v3 on a huge vision task (350 million images and 17,000 classes).

SENet¹⁰ (2017). This is the winning architecture in the ILSVRC 2017 challenge. SENet stands for

Squeeze-and-Excitation Network. This architecture extends existing architectures such as inception networks and ResNets, and boosts their performance. This allowed SENet to win the competition with an astonishing **2.25%** top-five error rate!

⁹Francois Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," arXiv preprint arXiv: 1610.02357 (2016).

¹⁰Jie Hu et al., "Squeeze-and-Excitation Networks," CVPR 2018, pp. 7132-7141.

Summary

- Understand how the input layer and different convolutional layers are connected
- Understand the purpose of filters and what they generate (the feature maps)
- Know how to stack multiple feature maps
- Understand the role of pooling layers in a CNN
- Know how to implement a simply CNN using TensorFlow
- Understand the large memory requirements of CNNs
- Know some basic CNN architectures

For next week

Work through the lab sheet and attend a supervised lab. The Unit Coordinator or a casual Teaching Assistant will be there to help.

Read up to Chapter 17 on [Autoencoders](#)

And that's all for this week's lecture.

Have a good week.

CITS5508 Machine Learning

Lectures for Semester 1, 2021

Lecture Week 12: Book Chapter 17

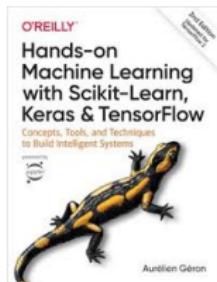
Dr Du Huynh (Unit Coordinator and Lecturer)
UWA

2021

Today

Chapter 17.

Hands-on Machine Learning with Scikit-Learn & TensorFlow



Chapter Seventeen

Representation Learning and Generative Learning Using Autoencoders and GANs

Here are the main topics we will cover:

- What are Autoencoders?
- Efficient data representations
- Encoder and decoder in an autoencoder
- Performing PCA using an *undercomplete* linear autoencoder
- Stacked autoencoders and how to implement one using Keras in TensorFlow
- Tying the weights in a stacked autoencoder and visualizing the reconstruction
- Unsupervised pre-training using stacked autoencoders
- Convolutional Autoencoders and Denoising autoencoders

What Are Autoencoders?

- Autoencoders are artificial neural networks capable of learning efficient representations of the input data, called *codings*, without any supervision (i.e., the training set is unlabelled). These codings typically have a **much lower dimensionality** than the input data, making autoencoders useful for **dimensionality reduction** (see Chapter 8).
- More importantly, autoencoders act as **powerful feature detectors**, and they can be used for unsupervised pre-training of deep neural networks.
- Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. However, the generated images are usually fuzzy and not entirely realistic.

Autoencoders

To prevent the autoencoders from simply learning to copy their inputs to their outputs, we can constrain the network in various ways to make the task more difficult. For example,

- you can limit the size of the internal representation, or
- add noise to the inputs and train the network to recover the original inputs.

These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data.

In short, the codings are byproducts of the autoencoder's attempt to learn the identity function under some constraints.

Efficient Data Representations

- Just like the chess players who can easily see chess patterns to help them quickly memorize the positions of all chess pieces in a game, an autoencoder looks at the inputs, converts them to an **efficient internal representation**, and then spits out something that (hopefully) looks very close to the inputs.
- An autoencoder is always composed of **two parts**:
 - an **encoder** (or **recognition network**) that converts the inputs to an internal representation, followed by
 - a **decoder** (or **generative network**) that converts the internal representation to the outputs.

Efficient Data Representations (cont.)

- An autoencoder typically has the same architecture as a **Multi-Layer Perceptron (MLP)**; see Chapter 10), except that the number of neurons in the output layer must be equal to the number of inputs.

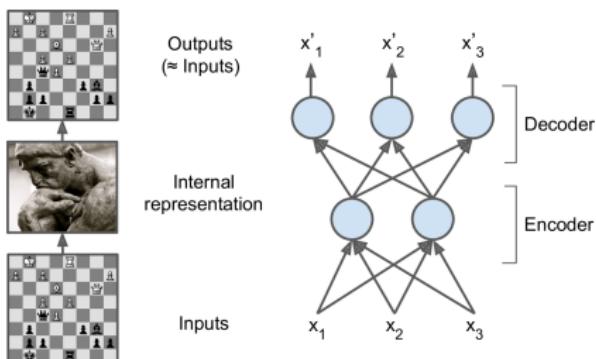


Figure 17-1. The chess memory experiment (left) and a simple autoencoder (right)

- In the example here, there is just one hidden layer composed of two neurons (the **encoder**). The outputs are often called the **reconstructions** since the autoencoder tries to reconstruct the inputs, and the cost function contains a **reconstruction loss** that penalizes the model when the reconstructions are different from the inputs.

Efficient Data Representations (cont.)

- Because the internal representation in Figure 17-1 has a lower dimensionality than the input data (it is 2D instead of 3D), the autoencoder is said to be *undercomplete*.
- An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to learn the most important features in the input data (and drop the unimportant ones).
Let's see how to implement a very simple undercomplete autoencoder for *dimensionality reduction*.

Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only [linear activations](#) and the cost function is the [Mean Squared Error \(MSE\)](#), then it can be shown that it ends up performing [Principal Component Analysis](#) (see Chapter 8). Example: Build a simple undercomplete autoencoder to perform PCA:

```
from tensorflow import keras

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

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

Performing PCA with an Undercomplete Linear Autoencoder (cont.)

This code is very similar to the MLP code in a previous chapter, but there are a few things to note:

- We organized the autoencoder into two sub-components: the *encoder* and the *decoder*. Both are regular **Sequential** models with a single **Dense** layer, and the *autoencoder* is a **Sequential** model containing the encoder followed by the decoder.
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform simple PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE.

Performing PCA with an Undercomplete Linear Autoencoder (cont.)

Now let's generate the same 3D data used in a previous chapter:

```
def generate_3d_data(m, w1=0.1, w2=0.3, noise=0.1):
    angles = np.random.rand(m) * 3 * np.pi / 2 - 0.5
    data = np.empty((m, 3))
    data[:, 0] = np.cos(angles) + np.sin(angles)/2 + noise * np.random.randn(m)
    data[:, 1] = np.sin(angles) * 0.7 + noise * np.random.randn(m) / 2
    data[:, 2] = data[:, 0] * w1 + data[:, 1] * w2 + noise * np.random.randn(m)
    return data

X_train = generate_3d_data(60)
X_train = X_train - X_train.mean(axis=0, keepdims=1)
```

and train the model, and use it for decoding (i.e., project it to 2D):

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Performing PCA with an Undercomplete Linear Autoencoder

Figure 17-2 shows the original 3D dataset (at the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, at the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

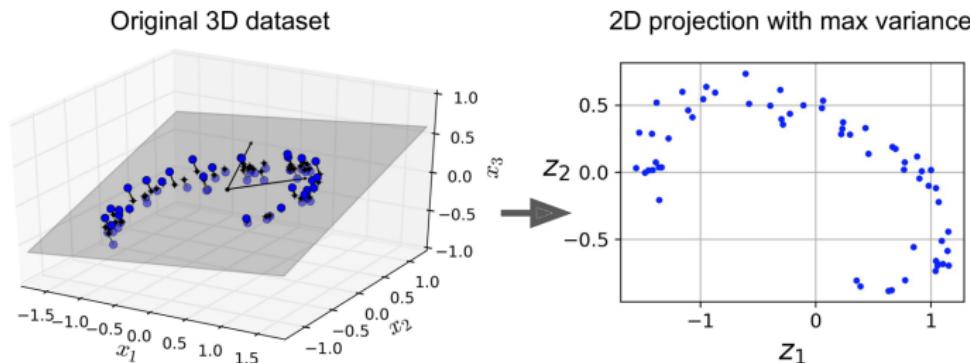


Figure 17-2. PCA performed by an undercomplete linear autoencoder

Stacked Autoencoders

- Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*).
- Adding more layers helps the autoencoder learn more complex codings. However, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process (and it is unlikely to generalize well to new instances).

Stacked Autoencoders

- The architecture of a stacked autoencoder is typically symmetrical with regards to the **central hidden layer (the coding layer)**. To put it simply, it looks like a sandwich, e.g., an autoencoder for MNIST may have 784 inputs, then a hidden layer with 300 neurons, a central hidden layer of 150 neurons, then another hidden layer with 300 neurons, and an output layer with 784 neurons (Figure 17-3).

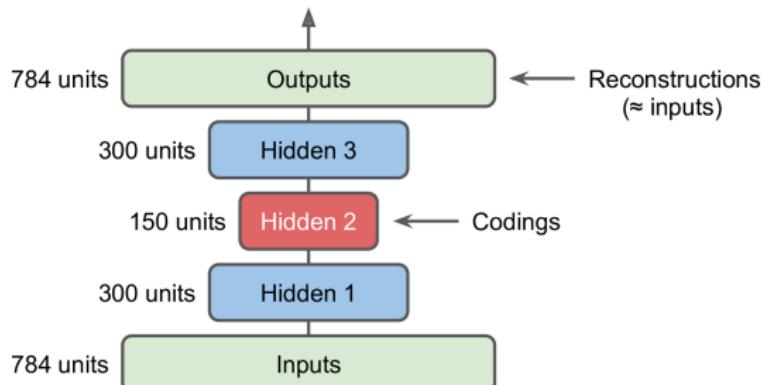


Figure 17-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using Keras

You can implement a stacked autoencoder very much like a regular deep MLP. The same techniques we used for training deep nets can be applied.

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                    optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                          validation_data=[X_valid, X_valid])
```

Implementing a Stacked Autoencoder Using Keras (cont.)

Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant.



Figure 17-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder deeper, or make the codings larger.

Implementing a Stacked Autoencoder Using Keras (cont.)

Visualizing Using T-SNE

Once we have reduced the dataset's dimensionality, we can use another dimensionality reduction algorithm to see how well the lower-dimensional data is for the downstream classification task. T-SNE is a suitable algorithm as it reduces the data down to 2D for visualization.

```
from sklearn.manifold import TSNE
X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Implementing a Stacked Autoencoder Using Keras (cont.)

Visualizing Using T-SNE (cont.)

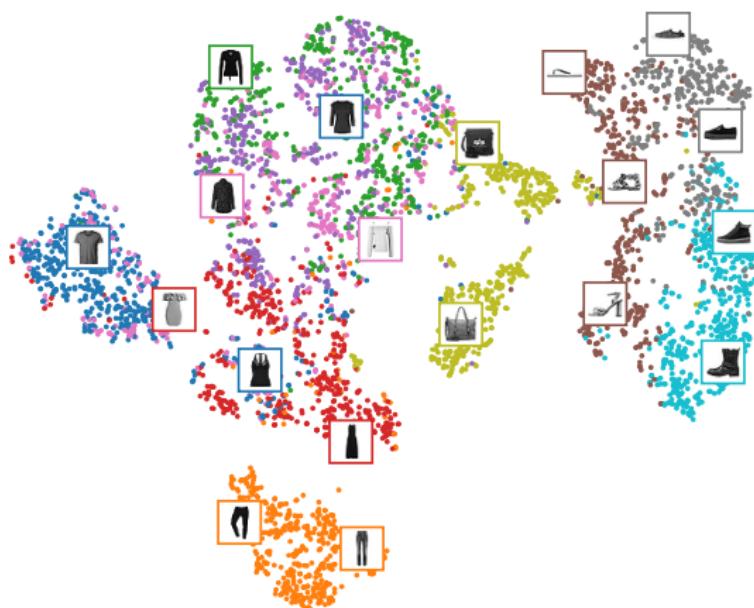


Figure 17-5. Fashion MNIST visualization using an autoencoder followed by t-SNE

Unsupervised Pretraining Using Stacked Autoencoders

If you are tackling a complex supervised task but you do not have a lot of labelled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, we can firstly train a stacked autoencoder using all the data. This training process does not require the data to be labelled. Once the stacked autoencoder is trained, we can reuse its lower layers in our neural network for the actual task. As our neural network has fewer layers that require training, it can cope with fewer labelled training data.

Unsupervised Pretraining Using Stacked Autoencoders (cont.)

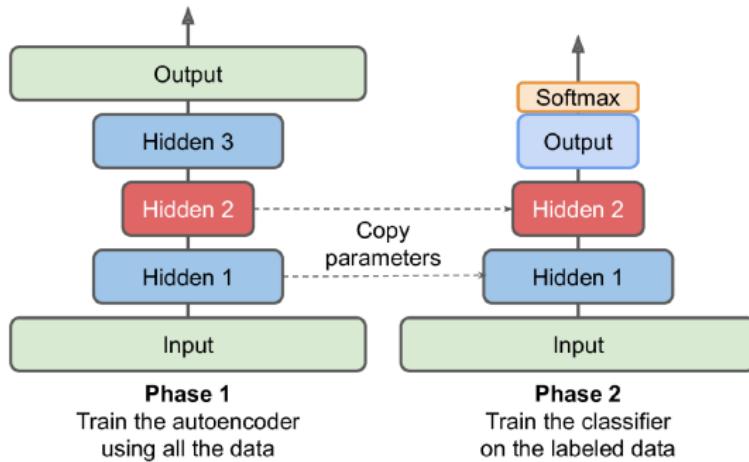


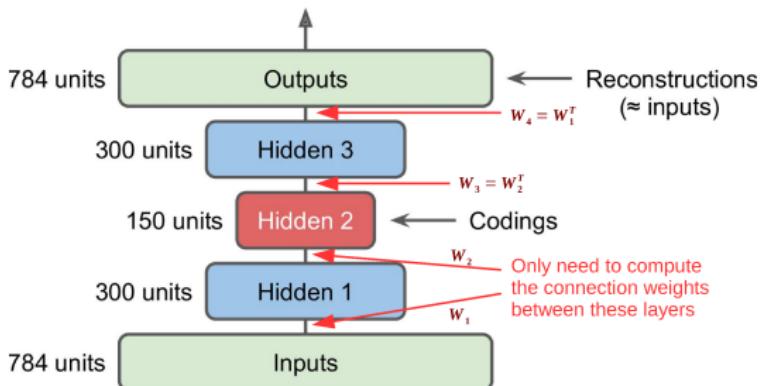
Figure 17-6. Unsupervised pretraining using autoencoders.

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network.

Stacked Autoencoders: Tying the Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie the weights* of the decoder layers to the weights of the encoder layers. This **halves the number of weights** in the model, speeding up training and limiting the risk of overfitting.

Stacked Autoencoders: Tying the Weights (cont.)



Tying the weights in a stacked autoencoder ($N = 4$ in this example).

Specifically, if the autoencoder has a total of N layers (not counting the input layer), and \mathbf{W}_ℓ represents the connection weights of the ℓ^{th} layer (e.g., layer 1 is the first hidden layer, layer $\frac{N}{2}$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined simply as:

$$\mathbf{W}_{N-\ell+1} = \mathbf{W}_\ell^T \quad (\text{with } \ell = 1, 2, \dots, N/2).$$

Stacked Autoencoders: Tying the Weights (cont.)

```
class DenseTranspose(keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)
```

Stacked Autoencoders: Tying the Weights (cont.)

```
dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")
tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])
```

This model achieves a slightly lower reconstruction error than the previous model, with the number of parameters that require training reduced by almost half.

Training One Autoencoder at a Time

It is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown below:

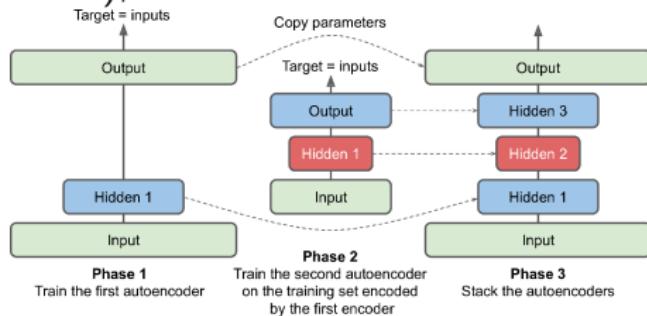


Figure 17-7. Training one autoencoder at a time

(ii) We train a second autoencoder using the new (compressed) training set from phase 2.

• **Phase 3:** Finally, we build a big sandwich using all these autoencoders.

- **Phase 1:** The first autoencoder learns to reconstruct the inputs.
- **Phase 2:** (i) we encode the whole training set using this first autoencoder.

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a convolutional autoencoder.

The *encoder* of a convolutional autoencoder is just a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., number of channels).

The *decoder* must do the reverse operation (upscale the image and reduce its depth back to the original dimensions), and for this you can use *transpose convolutional layers* (alternatively, you could combine upsampling layers with convolutional layers).

Convolutional Autoencoders (cont.)

Here is a simple convolutional autoencoder for Fashion MNIST:

```
conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])

conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3, strides=2, padding="valid",
                               activation="selu", input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3, strides=2, padding="same",
                               activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3, strides=2, padding="same",
                               activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])
```

Other Types of Autoencoders

So far we have seen various kinds of autoencoders:

- basic, stacked, and convolutional.

We have looked at how to train them :

- either in one shot or in several phases.

We have also looked at a couple of applications:

- data visualization and **unsupervised pretraining**

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it ***undercomplete***.

There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an ***overcomplete autoencoder***.

Denoising Autoencoders

Another way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. In a 2010 paper of Vincent et al.¹, the authors introduced the *stacked denoising autoencoders*. The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in *dropout*:

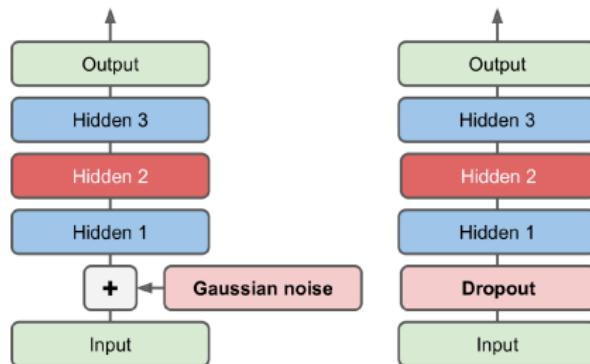


Figure 17-8. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

¹ Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion," Journal of Machine Learning Research 11 (2010): 3371-3408.

Denoising Autoencoders (cont.)

An Implementation Example

```
dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder, dropout_decoder])
```



Figure 17-9. Noisy images (top) and their reconstructions (bottom)

Summary

- Understand that autoencoders can be trained and used as a way to efficiently represent data (analogous to PCA).
- Understand the two main parts in an autoencoder: the *encoder* and the *decoder*.
- Know how to implement an *undercomplete* linear autoencoder in TensorFlow to perform PCA on the input data.
- Understand stacked autoencoders and know how to implement one using TensorFlow.
- Understand the concept of tying the weights in a stacked autoencoder and know how to visualize the reconstruction.
- Understand how to use stacked autoencoders in unsupervised pre-training tasks.
- Understand convolutional autoencoders and denoising autoencoders.

And that's all for the final week's lecture.
Good luck for the exam!

Have a good week.