Machine Learning with Big-Data

Jason McEwen, Tom Kitching, Matt Graham

Feb 04, 2023

CONTENTS

CHAPTER

ONE

LECTURE 1: INTRODUCTION TO MACHINE LEARNING


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:12:03
```

1.1 Course overview

1.1.1 Description and objectives

This module covers how to apply machine learning techniques to large data-sets, so-called *big-data*.

An introduction to machine learning (ML) is presented to provide a general understanding of the concepts of machine learning, common machine learning techniques, and how to apply these methods to data-sets of moderate sizes.

Deep learning and computing frameworks to scale machine learning techniques to big-data are then presented.

Scientific data formats and data curation methods are also discussed.

1.1.2 Syllabus

Foundations of ML (e.g. overview of ML, training, data wrangling, scikit-learn, performance analysis, gradient descent), data formats and curation (e.g. data pipelines, data version control, databases, big-data), ML methods (e.g. logistic regression, SVMs, ANNs, decision trees, ensemble learning and random forests, dimensionality reduction), deep learning and scaling to big-data (e.g. TensorFlow, Deep ANNs, CNNs, RNNs, Autoencoders) and applications of ML in astrophysics, high-energy physics and industry.

1.1.3 Prerequisites

Students should have a reasonable working knowledge of Python, some familiarity with working in the command line environment in Linux/Unix based operating systems, and a general understanding of elementary mathematics, including linear algebra and calculus.

No previous familiarity with machine learning is required.

1.1.4 Resources

Textbooks

- VanderPlas, "*Python data science handbook*", O'Reilly, 2017, ISBN 9781491912058 (Example code)
- Geron (1st Edition), "*Hands-on machine learning with Scikit-Learn and TensorFlow*", O'Reilly, 2017, ISBN 9781491962299 (Example code)
- Geron (2nd Edition), "*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*", O'Reilly, 2019, ISBN 9781492032649 (Example code)
- Goodfellow, Bengio, Courville (GBC), "*Deep learning*", MIT Press, 2016, ISBN 9780262035613

Tutorials

- Scikit-Learn tutorial, VanderPlas

Main code frameworks and libraries

- Scikit-Learn
- TensorFlow

1.1.5 Schedule

Lectures will run on Friday's from 10am-1pm.

1.1.6 Jupyter notebooks

Each lecture has an accompaning Jupyter notebook, with executable code.

These slides are a Jupyter notebook.

Notebooks can be viewed in slide mode using RISE.

The supporting Jupyter notebooks thus serve as the course *slides*, *lecture notes*, and *examples*.

A book version is also made available.

1.1.7 Course philosophy

This is a practical, hands-on course. While we will cover basic concepts and background theory (but not in great mathematical depth or rigor), a large component of the course will focus on implementing and running machine learning algorithms. Many code examples and exercises will be considered.

The course Jupyter notebooks will be made available weekly, in advance of lectures. Students can then follow examples in the lectures by running code live (and inspecting variables and making modifications).

1.1.8 Exercises

A number of lectures are accompanies by an additional Jupyter notebook with related examples for you to complete. The solutions to these exercises will be made available as the module progresses. These exercises will not be graded but are intended to help improve your understanding of the lecture material.

1.1.9 Assessment

- Courseworks: 2 x 20% = 40%
- Exam: 60%

Coursework

Courseworks will involve downloading a Jupyter notebook, which you will need to complete.

Throughout the notebook you will need to complete code, analytic exercises and descriptive answers. Much of the grading of the coursework will be performed automatically.

There will be two courseworks. The first coursework will be issued after the first 9 lectures, when all the material required to complete the first coursework will be covered. The second coursework will be issued after the first 15 lectures, when all the material required to complete the second coursework will be covered.

Exam

Answer THREE questions of the FOUR questions provided.

Each question has equal mark (15 marks per question).

Markers place importance on clarity and a portion of the marks are awarded for clear descriptions, answers, drawings, and diagrams, and attention to precision in quantitative answers.

1.1.10 Computing setup

Students are expected to bring their own laptops to class in order to run notebooks and complete examples.

All examples are implemented in Python 3.

The main Python libraries that are required include the following:

```
- numpy
- scipy
- matplotlib
- scikit-learn
- ipython/jupyter
- seaborn
- tensorflow
- astroML
```

An environment to run the notebooks can be set up with the versions of the libraries in `requirements.txt` (details below), following the steps below in terminal (MacOS, Linux) or anaconda prompt (Windows):

1. Create an environment named mlbd with Python 3.8.

```
conda create --name mlbd python=3.8
```

2. Activate the MLBDenv environment and then install the libraries in the requirements.txt file.

```
conda activate mlbd
pip install -r requirements.txt
```

3. Finally, start Jupyter, which will open the explorer and let you run the notebooks.

```
jupyter notebook
```

Content of requirements.txt:

```
numpy==1.23.2
matplotlib==3.6.2
pandas==1.5.2
scikit-learn==1.2.0
seaborn==0.12.1
tensorflow==2.10.0
boto3==1.20.24
pyarrow==10.0.1
pyspark==3.3.1
pyppeteer==1.0.2
rise==5.7.1
jupyterlab==3.5.1
jupyter-book==0.13.1
astroML==1.0.2.post1
tensorflow_datasets==4.7.0
fugue[sql]==0.7.3
```

1.2 What is machine learning?

1.2.1 Artifical intelligence (AI)

Ironically…

- Solving "computational problems" that are difficult for humans is straightforward for machines (i.e. problems described by list of formal mathematical rules).

- Solving "intuitive problems" that are easy for humans is difficult for machines (i.e. problems difficult to describe formally).

This is often known as Moravec's paradox (although formal definition is a little more specific).

Solution is to allow computers to learn from experience and to build an understanding of the world through a hierarchy of concepts.

1.2.2 Knowledge base approach

Hard-code knowledge about world in formal set of rules and use logical inference.

Very difficult to capture complexity of intuitive problems in this manner.

1.2.3 Machine learning (ML)

Arthur Samuel (1959):

> "[Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed."

Tom Mitchell (1997):

> "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."

1.2.4 Uses of machine learning

1. **Prediction:** Predict outcome given data.
2. **Inference:** Better understand data (and their distribution).

1.2.5 Data representations

Performance of machine learning depends on representation of data given.

Data presented to learning algorithm as *features*.

Traditional approach to machine learning involved *"feature engineering"*, where a practitioner with domain expertise would develop techniques to extract informative features from raw data.

Examples of features

- Computer visions: edges and corners
- Spam: frequency of words
- Character recognition: histograms of black pixels along rows/columns, number of holes, number of strokes

1.2.6 Learning representations

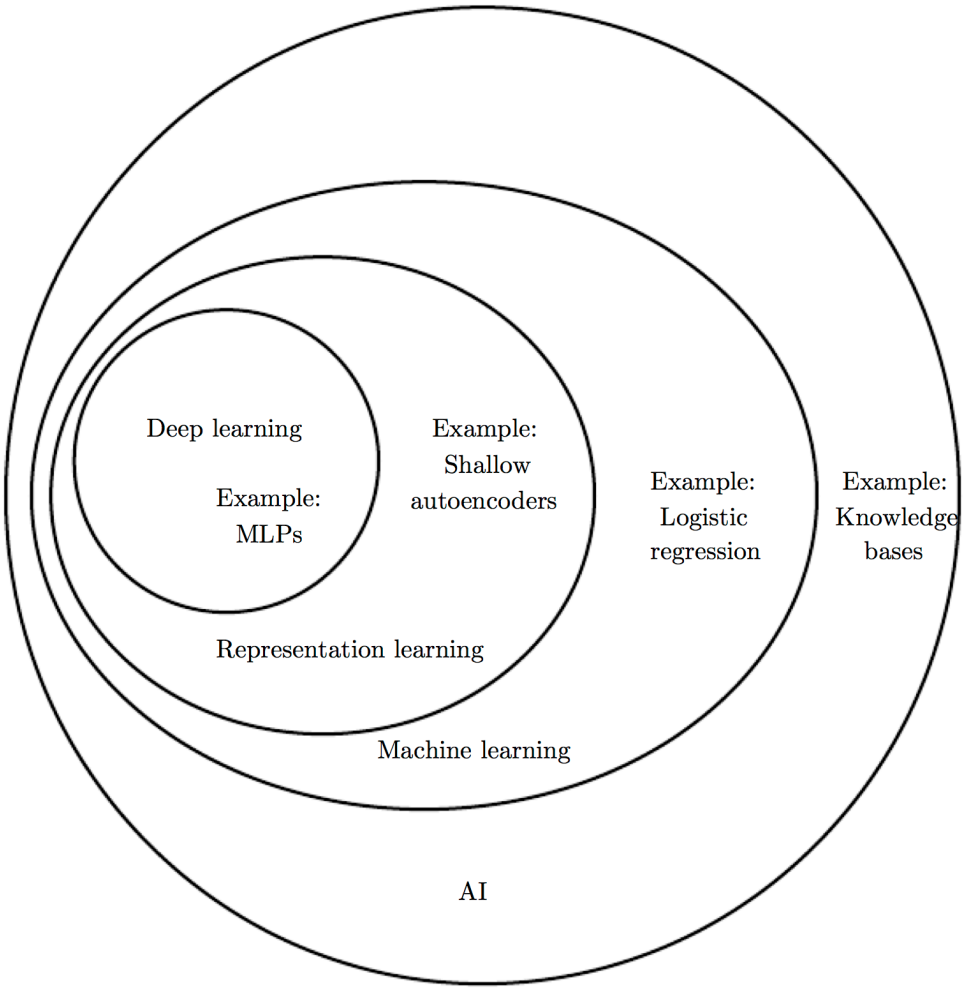
Alternative is to learn features.

- Can discover informative features from data.
- Minimal human intervention.

1.2.7 Approaches to representation learning

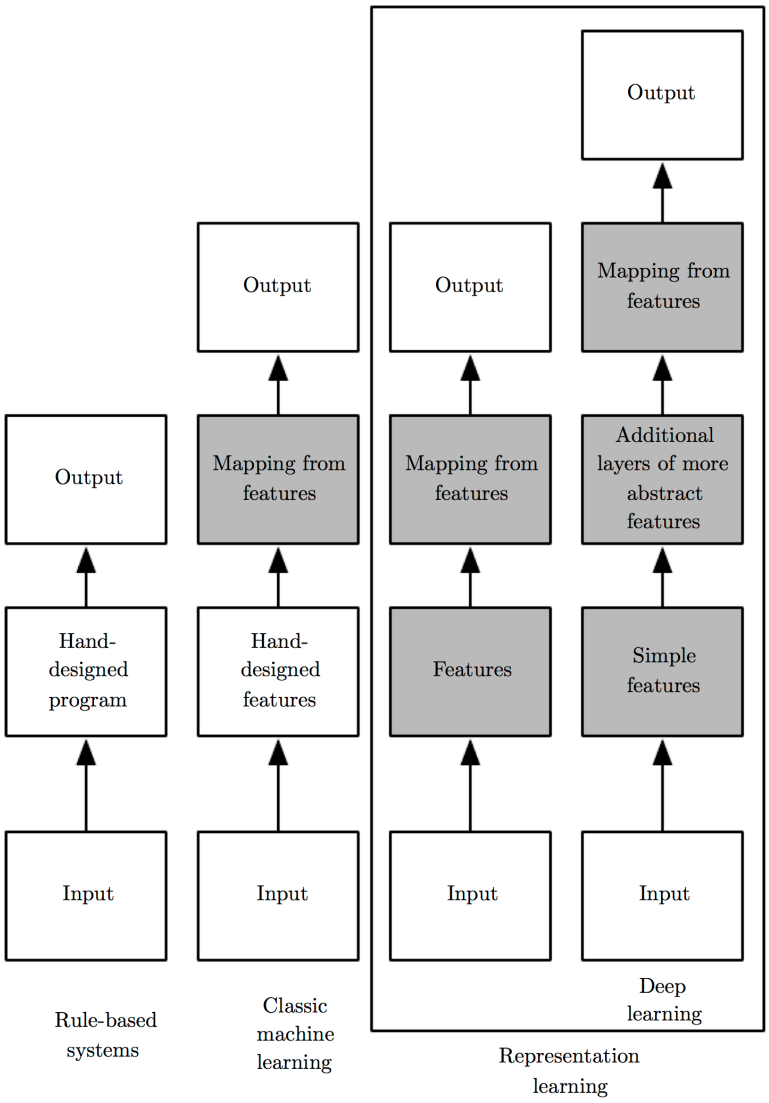
- Dedicated feature learning, e.g. autoencoder combining encoder and decoder.
- Representation learning integral to overall machine learning technique, e.g. deep learning.

1.2.8 Approaches to artifical intelligence



[Image credit: GBC]

1.2.9 AI pipelines



[Image credit: GBC]

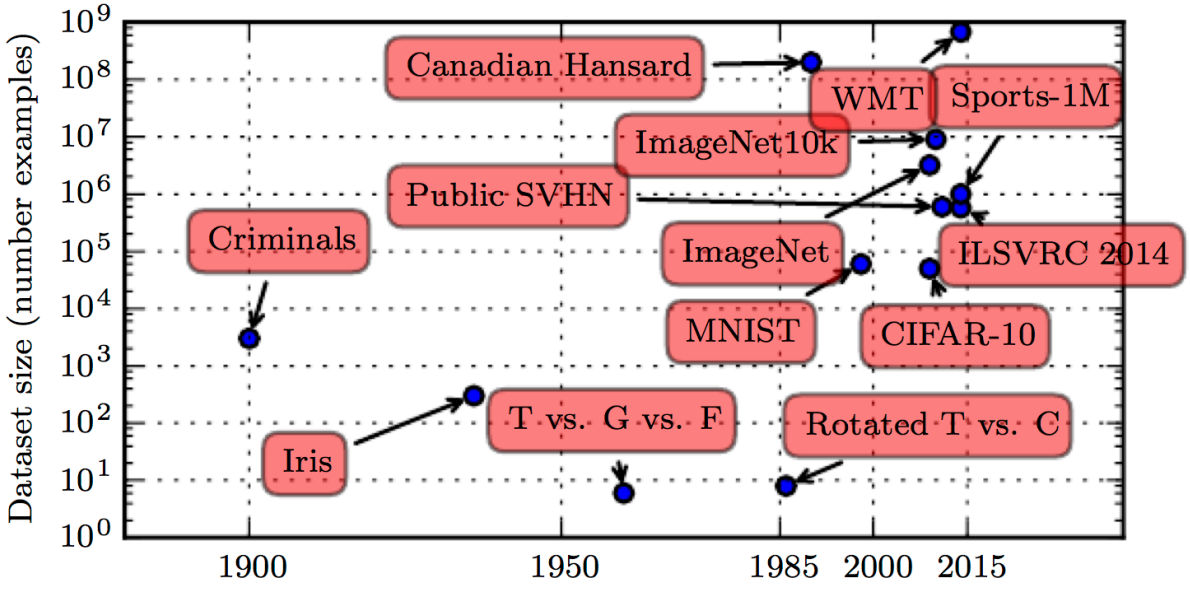
1.3 The unreasonable effectiveness of data

As society becomes increasing digitised, the volume of available data is exploding.

A significant increase in the volume of data can lead to dramatic increases in the performance of machine learning techniques.

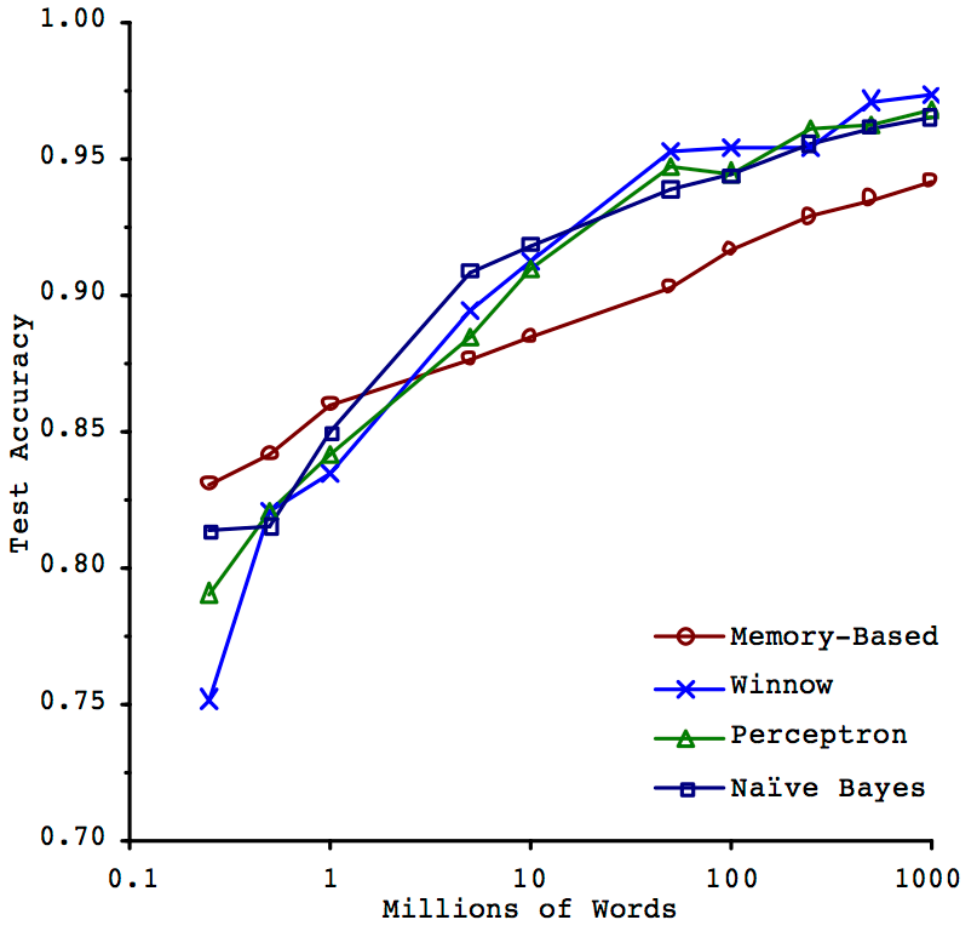
(Term coined in Halevy, Norbig & Pereira, 2009, *The unreasonable effectiveness of data*.)

1.3.1 Size of benchmark data-sets



[Image credit: GBC]

1.3.2 Size of data can have a larger impact than algorihm



Source: Banko & Brill, 2001, *Scaling to very very large corpora for natural language disambiguation*

> As a rule of thumb, a supervised deep learning algorithm will perform reasonably well with around 5,000 labelled samples.
>
> With 10 million samples, it will match or exceed human performance.

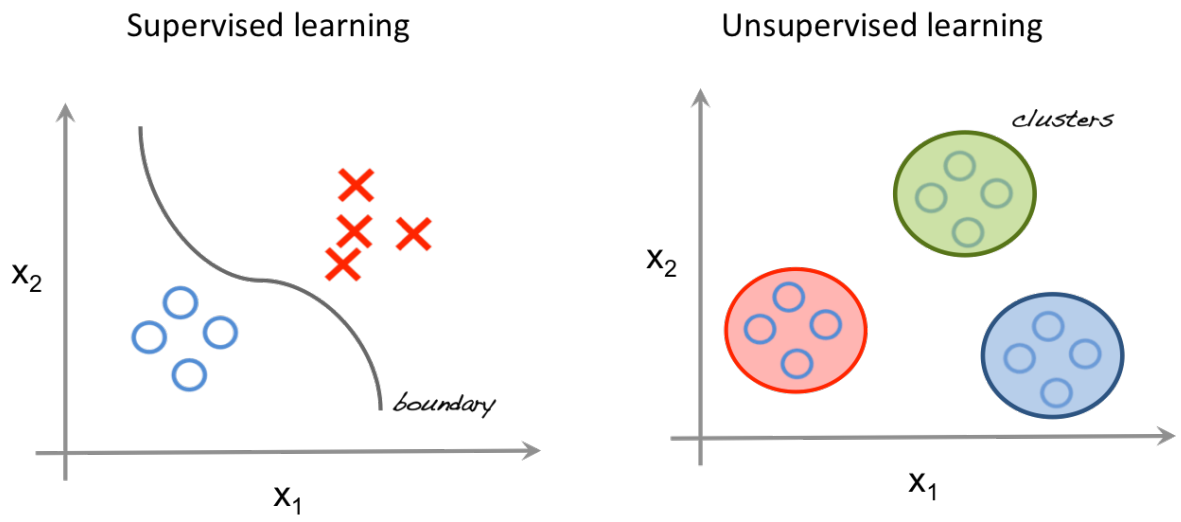
[Source: GBC]

However, in many cases very large datasets are not available and in some cases not possible.

Hence, developing effective algorithms remains critical.

1.4 Classes of machine learning

1. **Supervised:** Learn to predict output given input (given labelled training data).
2. **Unsupervised:** Discover internal representation of input.
3. **Reinforcement:** Learn action to maximise payoff.



[Image source]

1.4.1 Supervised learning

Learn to predict output given input (given labelled training data).

1. **Regression:** Target output is a (real) number, e.g. estimate flux intensity.
2. **Classification:** Target output is a class label, e.g. classify galaxy morphology.

How supervised learning works

- Select model defined by function f, and model target y from inputs x by $y = f(x, \theta)$, where θ are the parameters of the model that are learnt during training.
- Learning typically involves minimising the difference between the inputs and outputs for the model, given a training data-set (more on training, validation and test data-sets later).

1.4.2 Unsupervised learning

Discover internal representation of input.

1. **Cluster finding:** Learn cluster of similar structure in data.
2. **Density estimation:** Learn representations of data (probability distributions).
3. **Dimensionality reduction:** Provides compact, low-dimensional representation of data.

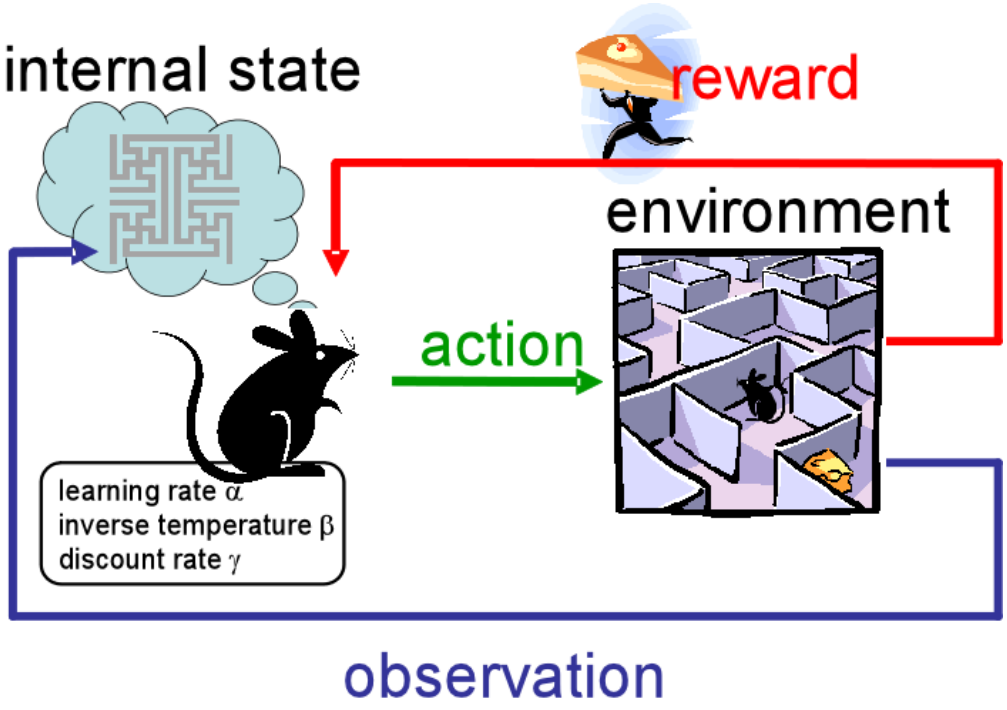
Unsupervised learning examples

Anomaly detection, clustering groups of similar objects, visualising high-dimensional data in 2D or 3D plots are examples of unsupervised learning.

1.4.3 Reinforcement learning

Learn action to maximise payoff.

- Output is an action or sequence of actions and the only supervisory signal is an occasional numerical (scalar) reward.
- Difficult since rewards are delayed.
- Not covered in this course.



[Image credit]

1.4.4 Reinforcement learning examples

Go, playing computer games, driverless cars, self navigating vaccum cleaners, scheduling of elevators are all applications of reinforcement learning.

E.g. Google [DeepMind] machine learns to master video games

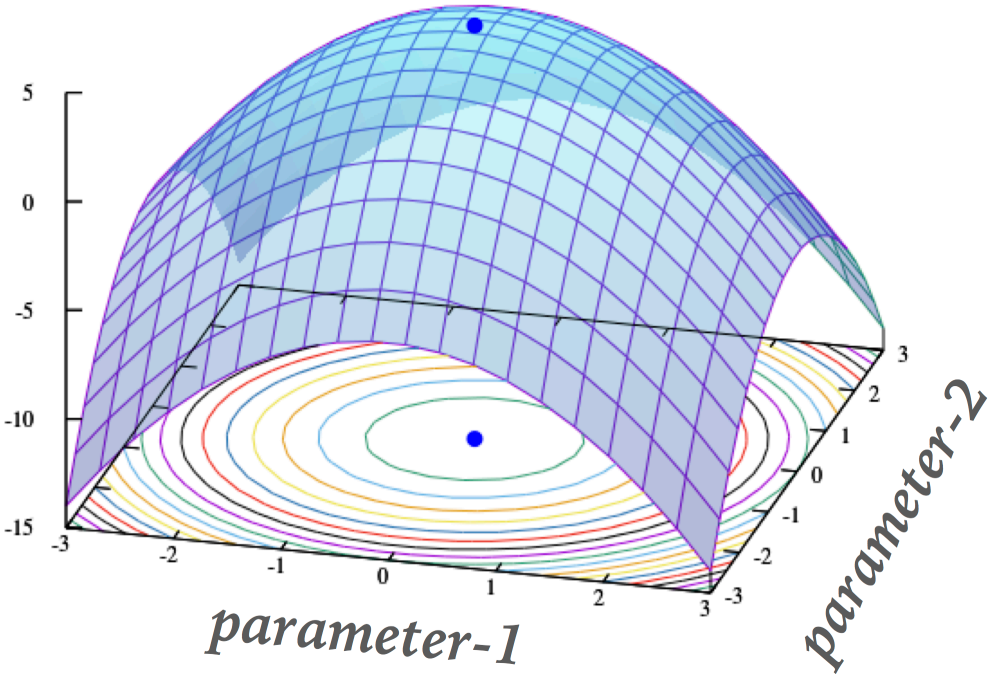
1.5 Training

Machine *learning* often involves solving an *optimization* problem, i.e. finding the parameters θ of the model f to best represent the training data (for supervised learning).

1.5.1 Objective function

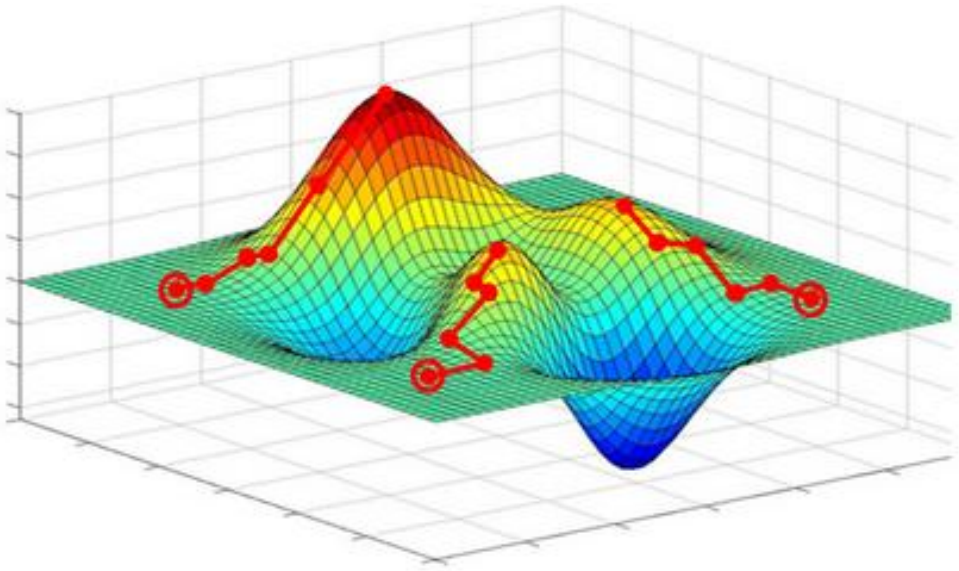
Typically maximise/minimise some goodness-of-fit/cost function.

Example of convex objective function



[Image credit: Kirkby, UC Irvine, LSST Dark Energy Summer School 2017]

Example of non-convex objective function

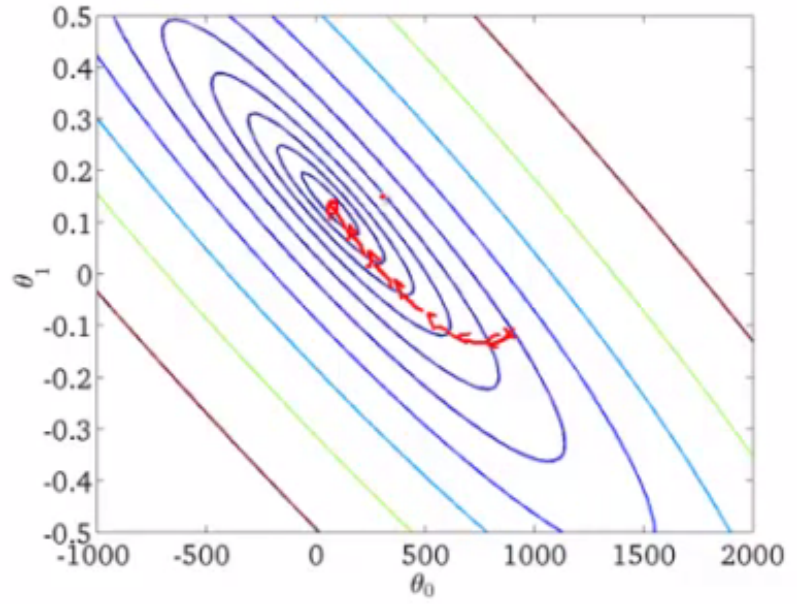


[Image source]

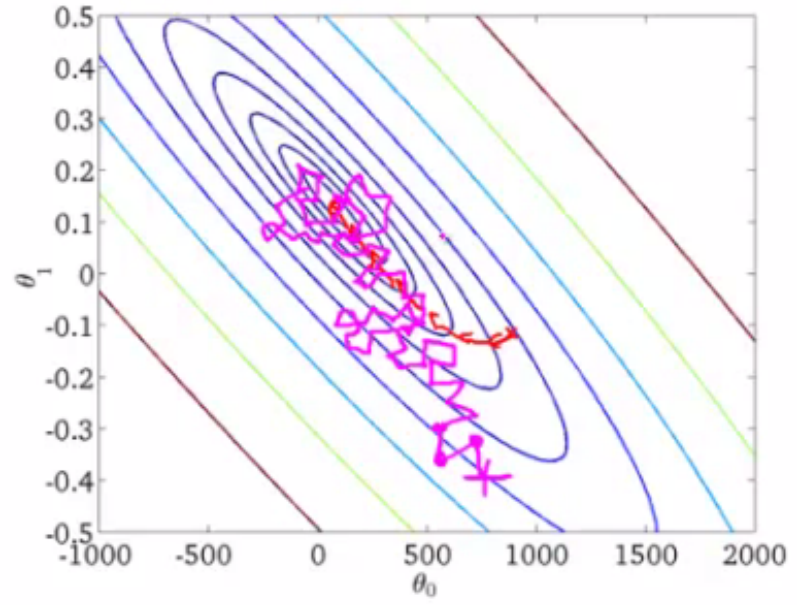
1.5.2 Using gradients to optimize objective function (i.e. perform training)

- **(Batch) Gradient descent:** Use all data at each iteration (full dimension).
- **Stochastic gradient descent:** Use a random data-point at each iteration (1 dimension).
- **Backpropagation:** propagate errors backwards through networks.

Batch gradient descent



Stochastic gradient descent



[Image source]

1.5.3 Batch and online learning

Batch learning

Algorithm is trained using all available training data at once.

Also called *offline learning*.

- Requires substantial resources (CPU, memory space, disk space).
- If want to add new training data, must re-train from scratch on new full set of data (i.e. not just the new data but also the old data).

Online learning

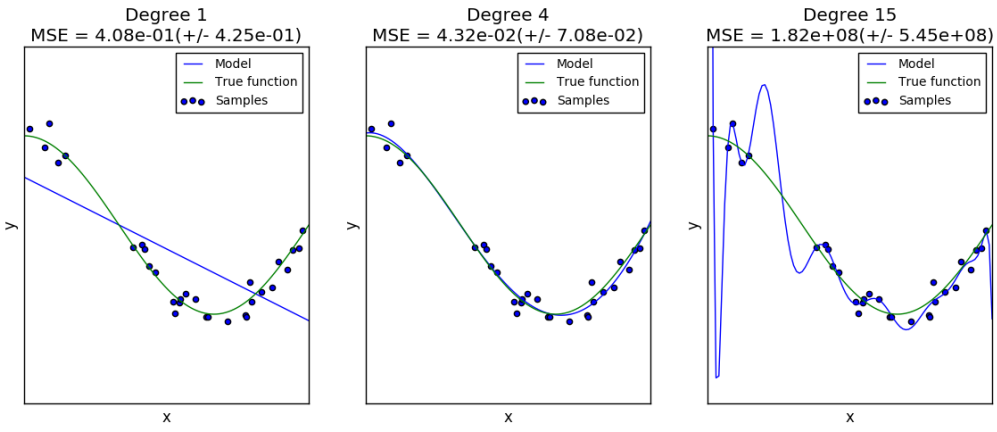
Algorithm is trained using a sub-set of the training data.

- Each learning step does *not* require substantial resources.
- Can integate new training data on the fly.
- May be able to throw away data once used it (although might not want to).
- If fed bad data, performance will decline.
- Noisy training.

1.6 Overfitting and underfitting

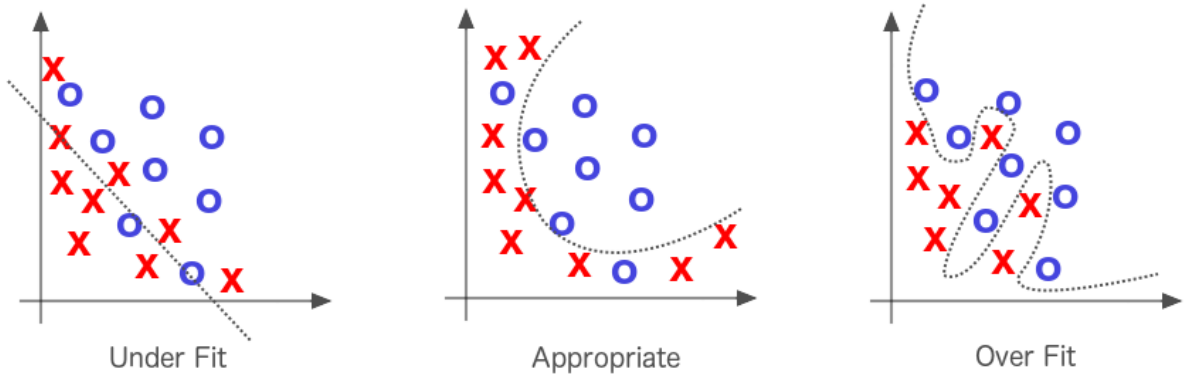
- **Problem:** The learned model may fit the training set extremely well but fail to generalise to new examples.

1.6.1 1D example



[Image source]

1.6.2 2D example



[Image source]

1.6.3 Techniques to avoid overfitting

- Reduce complexity of model.
- Regularization:
  - Place additional constraints (priors) on features/parameters.
  - E.g. smoothness of parameters, sparsity of model (i.e. limit complexity).
- Split data into training, validation and test sets (e.g. cross-validation).

1.7 Testing and validation

1.7.1 No free lunch theorem

Essentially, all algorithms are equivalent when performance is averaged over all possible problems.

Consequently, there is no a priori model that is guaranteed to work best on all problems.

(Wolpert, 1996, *The lack of a priori distinctions between learning algorithms*)

It is therefore a matter of validating models empirically.

1.7.2 Training and test datasets

Split data into training and test sets (e.g. 80% for training and 20% for testing).

The model is trained on the *training set* and then tested on the *test set*.

No data used in training the method is then used to evaluate it.

Error rate on the test set is called the *generalization error* or *out of sample error*.

If the training error is low but the generalization error is high, it suggests the model is overfitted.

1.7.3 Hyperparameters

Many machine learning algorithms contain hyperparameters to control the model.

One (**bad**) approach is to evaluate alternative models defined by different hyperparameters on test set and select the model that performs best.

However, this optimizes the model for the test set and may not generalise to other data well.

1.7.4 Validation

A better approach is to split the data into three sets:

1. Training set
2. Validation set
3. Test set

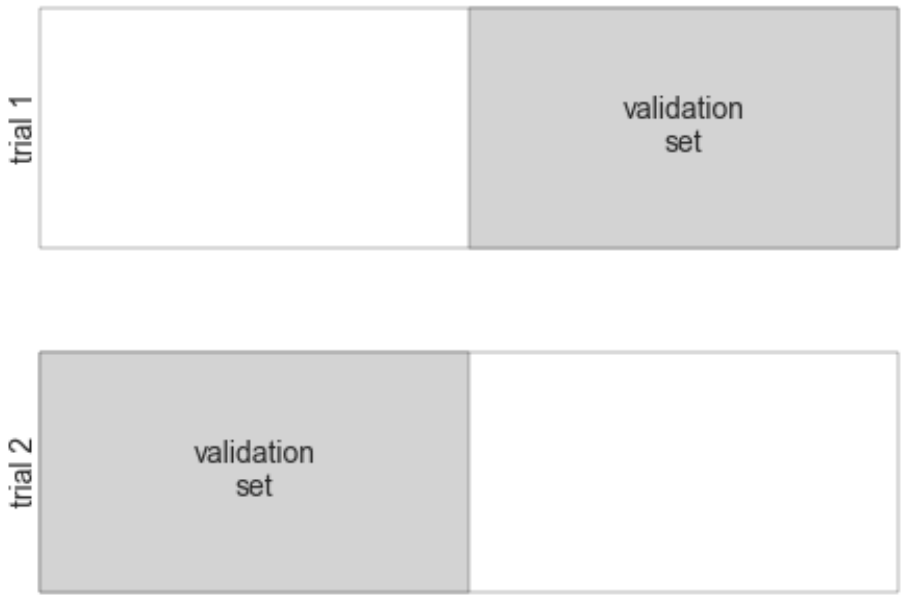
Train models on the training set and evaluate different models (with different hyperparameters) on the validation set.

Only once the final model to be used is fully specified should it be applied to the test set to estimate its generalization performance.

1.7.5 Cross-validation

A disadvantage of the previous approach is that less data are available for training.

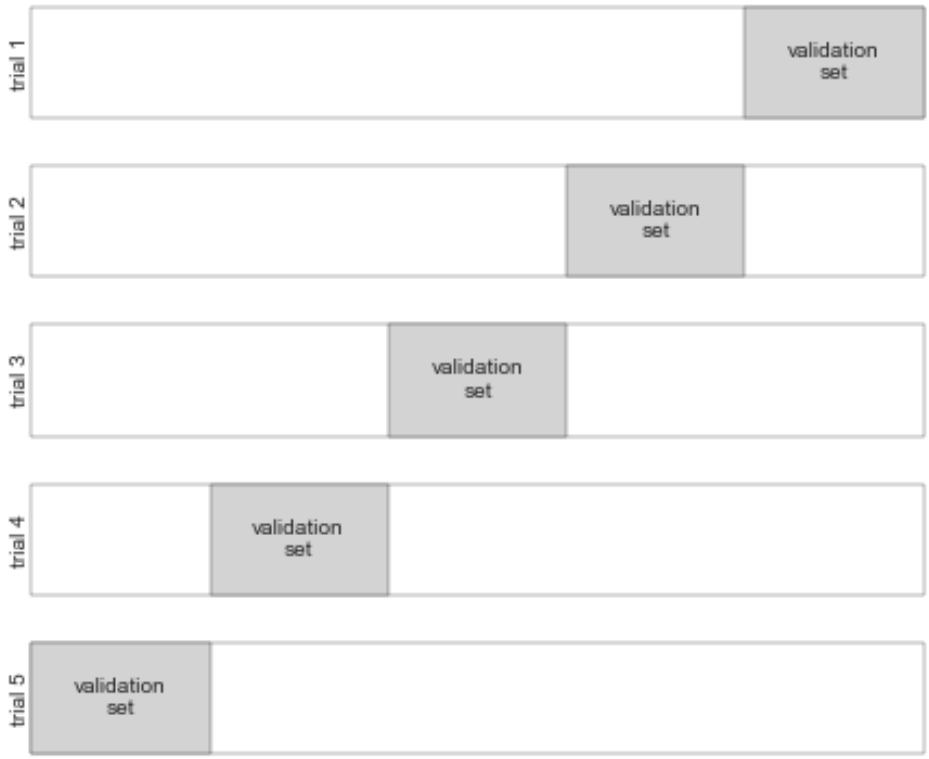
Cross-validation addresses this issue by performing a sequence of fits where each subset of the data is used both as a training set and a validation set.



[Image credit: VanderPlas]

Get validation accuracy scores for each trial, which could be combined.

Extension to n-fold cross-validation



[Image credit: VanderPlas]

CHAPTER

TWO

LECTURE 2: DATA WRANGLING WITH PANDAS


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:12:07
```

2.1 Why Pandas?

Pandas is a very useful package for data wrangling.

Particularly useful when working with real data, which can be messy.

Combines advantages of a number of different data structures (NumPy arrays, dictionaries, relational databases).

Can also be more efficient than native Python data structures for certain operators (as we will see).

Particularly useful for dealing with:

- Labelled data
- Missing data
- Heteterogenous types
- Groupings

We will focus mostly on Pandas `Series` and `DataFrame` objects.

2.1.1 Import Pandas

```
import pandas as pd
import numpy as np
```

2.1.2 Documentation

Recall can check documentation with `pd?`, `pd.<TAB>`, and/or print documentation for specific function with `print(pd.<function_name>.__doc__)`.

```
#pd?
```

```
#pd.
```

```
#pd.concat?
```

```
#print(pd.concat.__doc__)
```

2.2 Pandas `Series`

A Pandas `Series` is a *1D* array of *indexed* data.

Can be created from a list or array:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

The `Series` wraps both a sequence of *values* and a sequence of *indices*, which we can access with the `values` and `index` attributes.

```
data
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

```
data.values
```

```
array([0.25, 0.5 , 0.75, 1.  ])
```

```
data.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

2.2.1 `Series` as generalized NumPy array

Values are simply NumPy array.

Index need not be an integer, but can consist of values of any desired type.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

```
data['b']
```

```
0.5
```

2.2.2 `Series` as specialized dictionary

Can also think of a Pandas `Series` like a specialization of a Python dictionary.

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict) # Instantiate from dictionary
```

```
type(population_dict), type(population)
```

```
(dict, pandas.core.series.Series)
```

```
population['California']
```

```
38332521
```

- Python dictionary: maps *arbitrary* keys to *arbitrary* values.
- Pandas `Series`: maps *typed* indices to *typed* values.

Type information of Pandas `Series` makes it much more efficient than Python dictionaries for certain operations.

2.3 Pandas `DataFrame`

`DataFrame` can be thought of as a sequence of aligned `Series` objects, with *indices* and *columns*.

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

```
        foo       bar
a  0.017944  0.510359
b  0.146040  0.251697
c  0.040243  0.642187
```

2.3.1 `DataFrame` as generalized NumPy array

`DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names.

Contstruct another `Series` with same indices.

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
```

Combine two `Series` into a `DataFrame`.

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

```
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
Florida       19552860  170312
Illinois      12882135  149995
```

`DataFrame` has both `index` and `column` attributes.

```
states.index
```

```
Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')
```

```
states.columns
```

```
Index(['population', 'area'], dtype='object')
```

2.3.2 `DataFrame` as specialized dictionary

Can also think of a Pandas `DataFrame` like a specialization of a Python dictionary.

`DataFrame` maps a column name to a `Series`.

```
states['area']
```

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

```
type(states['area'])
```

```
pandas.core.series.Series
```

2.4 Pandas `Index`

Both Pandas `Series` and `DataFrame` contain `Index` object(s).

Can be thought of as *immutable array* (i.e. cannot be changed) or *ordered multi-set* (may contain repeated values).

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

2.4.1 `Index` as immutable array

Immutability makes it safer to share indices between multiple `DataFrames`.

```
ind[1]
```

```
3
```

```
#ind[1] = 0
```

2.4.2 `Index` as ordered multi-set

`Index` objects support many set operations, e.g. joins, unions, intersections, differences.

2.4.3 Example: Compute the intersection and union of the following two `Index` objects.

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

```
indA.intersection(indB)  # intersection
```

```
Int64Index([3, 5, 7], dtype='int64')
```

```
indA.union(indB)  # union
```

```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

2.5 Data indexing and selection

2.5.1 Data selection in a `Series`

In additional to acting like a dictionary, a `Series` also provies array-style selection like NumPy arrays.

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

```
# slicing by explicit index
data['a':'c']
```

```
a    0.25
b    0.50
c    0.75
dtype: float64
```

```
# slicing by implicit integer index
data[0:2]
```

```
a    0.25
b    0.50
dtype: float64
```

When slicing by an explicit index (e.g. `data['a':'c']`), the final index *is* included.

When slicing by an implicit index (e.g. `data[0:2]`), the final index *is not* included.

This can be a source of much confusion.

Consider a Series with integer indices.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

```
1    a
3    b
5    c
dtype: object
```

```
# explicit index when indexing
data[1]
```

```
'a'
```

```
# implicit index when slicing
data[1:3]
```

```
/tmp/ipykernel_2516/1915449024.py:2: FutureWarning: The behavior of `series[i:j]`␣
↪with an integer-dtype index is deprecated. In a future version, this will be␣
↪treated as *label-based* indexing, consistent with e.g. `series[i]` lookups. To␣
↪retain the old behavior, use `series.iloc[i:j]`. To get the future behavior, use␣
↪`series.loc[i:j]`.
  data[1:3]
```

```
3    b
5    c
dtype: object
```

Indexers

Indexers `loc` (explicit) and `iloc` (implicit) are introduced to avoid confusion.

```
data
```

```
1    a
3    b
5    c
dtype: object
```

```
data.loc[1]
```

```
'a'
```

```
data.loc[1:3]
```

```
1    a
3    b
dtype: object
```

```
data.iloc[1]
```

```
'b'
```

```
data.iloc[1:3]
```

```
3    b
5    c
dtype: object
```

2.5.2 Data selection in a `DataFrame`

In additional to acting like a dictionary of `Series` objects with the same index, a `DataFrame` also provies array-style selection like NumPy arrays.

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'population':pop})
data
```

```
             area  population
California  423967    38332521
Texas       695662    26448193
New York    141297    19651127
Florida     170312    19552860
Illinois    149995    12882135
```

Indexers

Indexers `loc` (explicit) and `iloc` (implicit) are also available to avoid confusion when selecting data.

Note that index and column labels are preserved in the result.

```
data
```

```
             area  population
California  423967    38332521
Texas       695662    26448193
New York    141297    19651127
Florida     170312    19552860
Illinois    149995    12882135
```

```
data.iloc[:3, :2]
```

```
             area  population
California  423967    38332521
Texas       695662    26448193
New York    141297    19651127
```

```
data.loc[:'Illinois', :'population']
```

```
             area  population
California  423967    38332521
Texas       695662    26448193
New York    141297    19651127
Florida     170312    19552860
Illinois    149995    12882135
```

2.5.3 Additional array-style selection

Other NumPy selection approaches can also be applied (e.g. masking).

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

2.6 Operating on data in Pandas

Elementwise operations in Pandas automatically aligns indices and preserves index/column labels.

Can avoid many errors and bugs in data wrangling.

2.6.1 Index preservation

```
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
```

```
0    6
1    3
2    7
3    4
dtype: int64
```

```
np.exp(ser)
```

```
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

```
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

```
   A  B  C  D
0  6  9  2  6
1  7  4  3  7
2  7  2  5  4
```

```
np.exp(df)
```

```
             A            B           C            D
0   403.428793  8103.083928    7.389056   403.428793
1  1096.633158    54.598150   20.085537  1096.633158
2  1096.633158     7.389056  148.413159    54.598150
```

Exercises: *You can now complete Exercise 2 in the exercises associated with this lecture.*

2.6.2 Index alignment

Index alignment works similarly for `DataFrames`.

```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=list('AB'))
A
```

```
   A   B
0  1  11
1  5   1
```

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
B
```

```
   B  A  C
0  4  0  9
1  5  8  0
2  9  2  6
```

```
A + B
```

```
      A     B   C
0   1.0  15.0 NaN
1  13.0   6.0 NaN
2   NaN   NaN NaN
```

2.6.3 Operations between `DataFrame` and `Series` objects

```
A = rng.randint(10, size=(3, 4))
df = pd.DataFrame(A, columns=list('QRST'))
df
```

```
   Q  R  S  T
0  3  8  2  4
1  2  6  4  8
2  6  1  3  8
```

```
s = df.iloc[0]
s
```

```
Q    3
R    8
S    2
T    4
Name: 0, dtype: int64
```

Difference between the `DataFrame` and `Series`:

```
df - s
```

```
   Q  R  S  T
0  0  0  0  0
1 -1 -2  2  4
2  3 -7  1  4
```

Convention is to operate row-wise.

Can also operate column-wise using object methods.

```
df.subtract(df['R'], axis=0)
```

```
   Q  R  S  T
0 -5  0 -6 -4
1 -4  0 -2  2
2  5  0  2  7
```

2.7 Handling missing data

Real data is messy. Often some data are missing.

Various conventions can be considered to handle missing data.

We will focus on the use of the floating point IEEE value NaN (not a number) to represent missing data.

Pandas interprets NaN as Null values.

(Pandas also supports `None` but we will focus on NaN here.)

Arithematic operations with NaN values result in NaN.

```
1 + np.nan
```

```
nan
```

2.7.1 Operating on Null values

Several useful methods exist to work with NaNs, for example to detect, drop or replace:

- `isnull()`: Generate a boolean mask indicating missing values.
- `notnull()`: Opposite of `isnull()`.
- `dropna()`: Return a filtered version of the data.
- `fillna()`: Return a copy of the data with missing values filled.

2.7.2 Detecting null values

Pandas `isnull` and `notnull` are useful for detecting null values.

Exercises: *You can now complete Exercise 3 in the exercises associated with this lecture.*

2.8 Dropping null values

Direct routines may be used to drop null values (i.e. `dropna`), rather than constructing masks as performed above.

Exercises: *You can now complete Exercise 4 in the exercises associated with this lecture.*

2.8.1 Dropping null values from `DataFrames`

For `DataFrames`, there are multiple ways null values can be dropped.

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

By default `dropna` operates row-wise and drops all rows that contain any NaNs.

```
df.dropna()
```

```
     0    1  2
1  2.0  3.0  5
```

Can also operate column-wise.

```
df
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

```
df.dropna(axis='columns')
```

```
   2
0  2
1  5
2  6
```

More sophisticated approaches can also be considered (e.g. only dropping rows/columns if all entries or a certain number of NaNs appear).

2.8.2 Replacing null values

Null values can be easily replaced using `fillna`.

```
df
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

```
df.fillna(0.0)
```

```
     0    1  2
0  1.0  0.0  2
1  2.0  3.0  5
2  0.0  4.0  6
```

Can also fill using adjacent values.

```
df
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6
```

```
df.fillna(method='ffill', axis='columns')
```

```
     0    1    2
0  1.0  1.0  2.0
1  2.0  3.0  5.0
2  NaN  4.0  6.0
```

```
df.fillna(method='ffill', axis='rows')
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  2.0  4.0  6
```

```
df.fillna(method='bfill', axis='columns')
```

```
     0    1    2
0  1.0  2.0  2.0
1  2.0  3.0  5.0
2  4.0  4.0  6.0
```

2.9 Combining data-sets

2.9.1 Define helper functions

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

```
    A   B   C
0  A0  B0  C0
1  A1  B1  C1
2  A2  B2  C2
```

```
from IPython.display import display
```

2.9.2 Concatenation

Can concatenate `Series` and `DataFrame` objects with `pd.concat()`.

Default is to concatenate over rows.

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display(df1, df2, pd.concat([df1, df2]))
```

```
    A   B
1  A1  B1
2  A2  B2
```

```
    A   B
3  A3  B3
4  A4  B4
```

```
    A   B
1  A1  B1
2  A2  B2
3  A3  B3
4  A4  B4
```

Can also concatenate over columns.

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display(df3, df4, pd.concat([df3, df4], axis='columns'))
```

```
    A   B
0  A0  B0
1  A1  B1
```

```
    C   D
0  C0  D0
1  C1  D1
```

```
    A   B   C   D
0  A0  B0  C0  D0
1  A1  B1  C1  D1
```

Duplicated indices

Can have duplicated indices.

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make duplicate indices!
display(x, y, pd.concat([x, y]))
```

```
    A   B
0  A0  B0
1  A1  B1
```

```
    A   B
0  A2  B2
1  A3  B3
```

```
    A   B
0  A0  B0
1  A1  B1
0  A2  B2
1  A3  B3
```

Ignoring index

Can ignore index.

```
display(x, y, pd.concat([x, y], ignore_index=True))
```

```
    A   B
0  A0  B0
1  A1  B1
```

```
    A   B
0  A2  B2
1  A3  B3
```

```
    A   B
0  A0  B0
1  A1  B1
2  A2  B2
3  A3  B3
```

Concantenation with joins

Can join `DataFrames` with different column names.

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display(df5, df6, pd.concat([df5, df6]))
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
```

```
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

```
     A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

Entries with no data are filled with NaN.

Default join is the *union* of the columns of the two `DataFrames`.

Can also perform different types of joins.

For example, the *intersection* of the columns of the two DataFrames.

```
display(df5, df6, pd.concat([df5, df6], join='inner'))
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
```

```
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

```
    B   C
1  B1  C1
2  B2  C2
3  B3  C3
4  B4  C4
```

2.9.3 Relational combinations

Pandas also provides functionality to perform relational algebra (cf. relational databases).

Hence, Pandas data structures provide analog not only of NumPy array and dictionary, but also relational database.

Functionality provied by `pd.merge()` function.

One-to-one join

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.merge(df1, df2)
display(df1,df2,df3)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
```

```
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

Recognises that “employee” column common and automatically selects as key for the relational join.

Many-to-one joins

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display(df3, df4, pd.merge(df3, df4))
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

```
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
```

```
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

The `on` keyword

```
display(df1, df2, pd.merge(df1, df2, on='employee'))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
```

```
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

The `left_on` and `right_on` keywords

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display(df1, df3, pd.merge(df1, df3, left_on="employee", right_on="name"))
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
```

```
   name  salary
0   Bob   70000
1  Jake   80000
```

(continues on next page)

(continued from previous page)

```
2  Lisa  120000
3   Sue   90000
```

```
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

Employee and name both included now, so may want to drop one.

```
pd.merge(df1, df3, left_on="employee", right_on="name")
```

```
  employee        group  name  salary
0      Bob   Accounting   Bob   70000
1     Jake  Engineering  Jake   80000
2     Lisa  Engineering  Lisa  120000
3      Sue           HR   Sue   90000
```

```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis='columns')
```

```
  employee        group  salary
0      Bob   Accounting   70000
1     Jake  Engineering   80000
2     Lisa  Engineering  120000
3      Sue           HR   90000
```

The `left_index` and `right_index` keywords

Often one wants to join on index.

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display(df1a, df2a)
```

```
                group
employee
Bob        Accounting
Jake      Engineering
Lisa      Engineering
Sue                HR
```

```
          hire_date
employee
Lisa           2004
Bob            2008
Jake           2012
Sue            2014
```

```
display(pd.merge(df1a, df2a, left_index=True, right_index=True))
```

```
                group  hire_date
employee
Bob        Accounting       2008
Jake      Engineering       2012
Lisa      Engineering       2004
Sue                HR       2014
```

Set arithmetic for joins

Have so far been considering relational joins based on *intersection* (also called *inner* join).

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display(df6, df7, pd.merge(df6, df7, how='inner'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
```

```
     name drink
0    Mary  wine
1  Joseph  beer
```

```
   name   food drink
0  Mary  bread  wine
```

Outer join

Can also join based on *union* (missing entries filled with NaNs).

```
display(df6, df7, pd.merge(df6, df7, how='outer'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
```

```
     name drink
0    Mary  wine
1  Joseph  beer
```

```
     name   food drink
0   Peter   fish   NaN
1    Paul  beans   NaN
2    Mary  bread  wine
3  Joseph    NaN  beer
```

Left and right join

Can also join based on *left* or *right* entries.

```
display(df6, df7, pd.merge(df6, df7, how='left'))
```

```
    name   food
0  Peter   fish
1   Paul  beans
2   Mary  bread
```

```
     name drink
0    Mary  wine
1  Joseph  beer
```

```
    name   food drink
0  Peter   fish   NaN
1   Paul  beans   NaN
2   Mary  bread  wine
```

Overlapping column names

Possible for DataFrames to have conflicting columns.

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display(df8, df9)
```

```
   name  rank
0   Bob     1
1  Jake     2
2  Lisa     3
3   Sue     4
```

```
   name  rank
0   Bob     3
1  Jake     1
2  Lisa     4
3   Sue     2
```

```
display(pd.merge(df8, df9, on="name", suffixes=["_L", "_R"]))
```

```
   name  rank_L  rank_R
0   Bob       1       3
1  Jake       2       1
2  Lisa       3       4
3   Sue       4       2
```

CHAPTER

THREE

LECTURE 3: INTRODUCTION TO SCIKIT-LEARN


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:12:16
```

3.1 Scikit-Learn overview

Scikit-Learn is an extremely popular python machine learning package.

Provides implementations of a number of different machine learning algorithms.

- Clean, uniform and streamlined API.
- Useful and complete online documentation.
- Straightforward to switch models or algorithms.

Two main general concepts:

- Data representation
- Estimator API

3.2 Data representations

3.2.1 Scikit-Learn includes a number of example data-sets

```
from sklearn import datasets
```

```
# Type datasets.<TAB> to see more
#datasets.
```

3.2.2 Data as a table

Best way to think about data in Scikit-Learn is in terms of tables of data.

Using the `seaborn` library we can read example data-sets as a Pandas `DataFrame`.

```
import seaborn as sns
iris = sns.load_dataset('iris')
type(iris)
```

```
pandas.core.frame.DataFrame
```

```
iris.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

3.2.3 Iris data

Here we consider the Iris flower data.

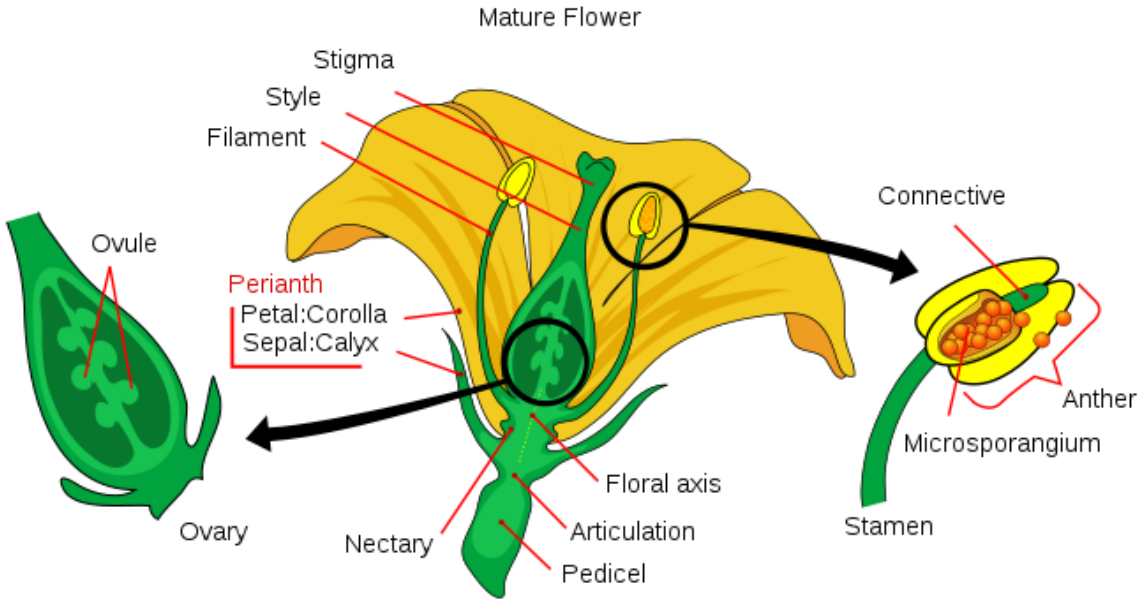
- Introduced by statistician and biologist Ronald Fisher in 1936 paper.
- Consists of 50 samples of three different species of Iris (Iris Setosa, Iris Virginica and Iris Versicolor).
- Four features were measured from each sample: the length and the width of the sepals and petals, in centimetres.

```
iris.tail()
```

```
     sepal_length  sepal_width  petal_length  petal_width    species
145           6.7          3.0           5.2          2.3  virginica
146           6.3          2.5           5.0          1.9  virginica
147           6.5          3.0           5.2          2.0  virginica
148           6.2          3.4           5.4          2.3  virginica
149           5.9          3.0           5.1          1.8  virginica
```

Parts of a flower

Measured flower petals and sepals.



[Image credit: Mariana Ruiz]

Images of different species

Iris Setosa

Iris Versicolor

Iris Virginica

[Image source]

3.2.4 Features matrix

Recall data represented to learning algorithm as "*features*".

Each row corresponds to an observed (*sampled*) flower, with a number of *features*.

```
iris.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width species
0           5.1          3.5           1.4          0.2  setosa
1           4.9          3.0           1.4          0.2  setosa
2           4.7          3.2           1.3          0.2  setosa
3           4.6          3.1           1.5          0.2  setosa
4           5.0          3.6           1.4          0.2  setosa
```

In this example we extract a feature matrix, removing species (which we want to predict).

```
X_iris = iris.drop('species', axis='columns')
X_iris.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width
0           5.1          3.5           1.4          0.2
1           4.9          3.0           1.4          0.2
2           4.7          3.2           1.3          0.2
3           4.6          3.1           1.5          0.2
4           5.0          3.6           1.4          0.2
```

```
type(X_iris)
```

```
pandas.core.frame.DataFrame
```

3.2.5 Target array

Consider 1D *target array* containing labels or targets that we want to predict.

May be numerical values or discrete classes/labels.

In this example we want to predict the flower species from other measurements.

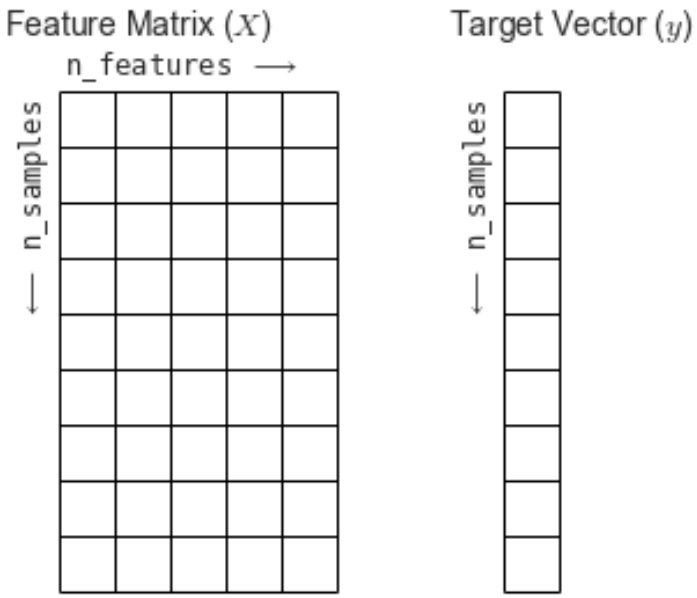
```
y_iris = iris['species']
y_iris.head()
```

```
0    setosa
1    setosa
2    setosa
3    setosa
4    setosa
Name: species, dtype: object
```

```
type(y_iris)
```

```
pandas.core.series.Series
```

3.2.6 Features matrix and target vector



[Image source]

```
X_iris.shape
```

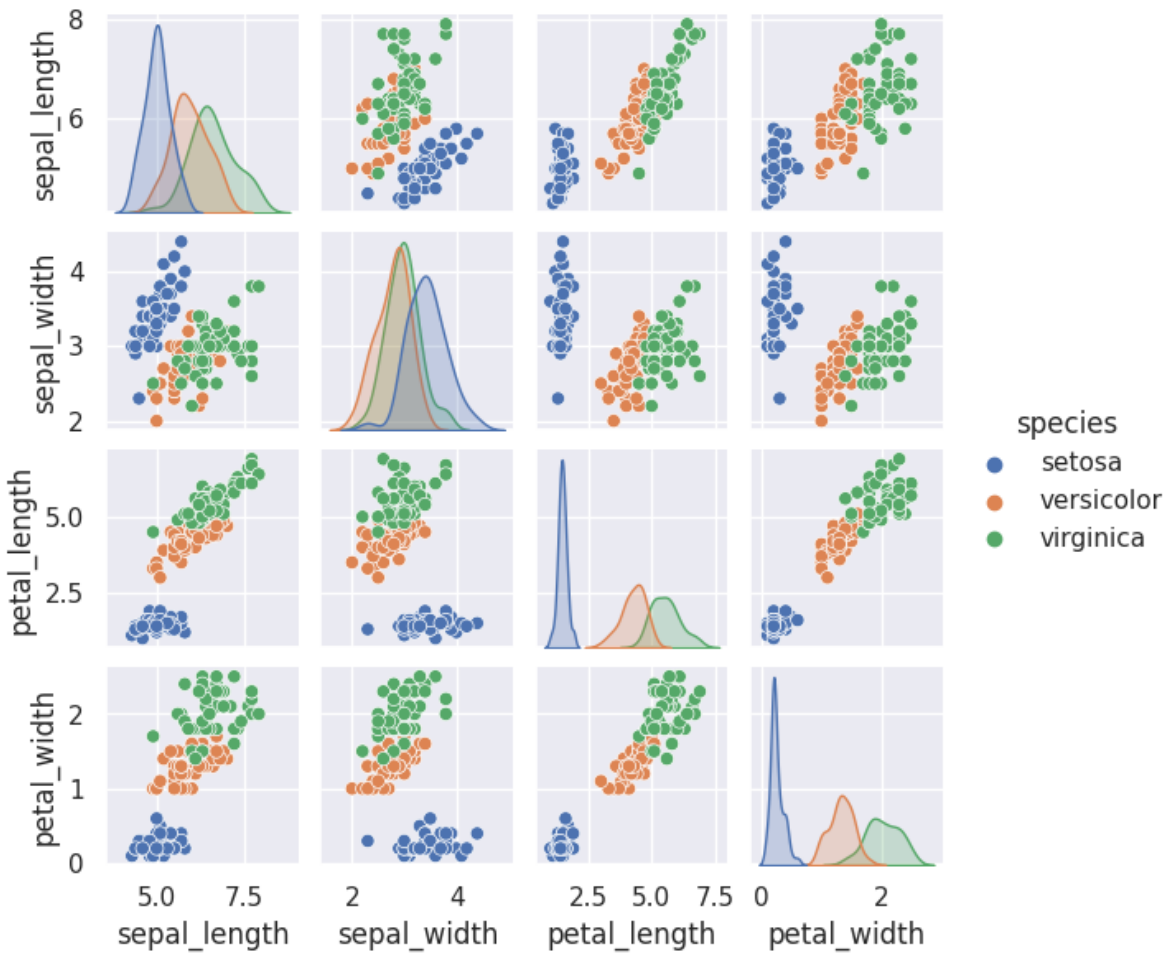
```
(150, 4)
```

```
y_iris.shape
```

```
(150,)
```

3.2.7 Visualizing the data

```
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', height=1.5);
```



How well do you expect classification to perform with these features and why?

Fairly well since the different classes are reasonably well separated in feature space.

3.3 Scikit-Learn's Estimator API

3.3.1 Scikit-Learn API design principles

- Consistency: All objects share a common interface.
- Inspection: All specified parameter values exposed as public attributes.
- Limited object hierarchy: Only algorithms are represented by Python classes; data-sets/parameters represented in standard formats.
- Composition: Many machine learning tasks can be expressed as sequences of more fundamental algorithms.
- Sensible defaults: Library defines appropriate default value.

3.3.2 Impact of design principles

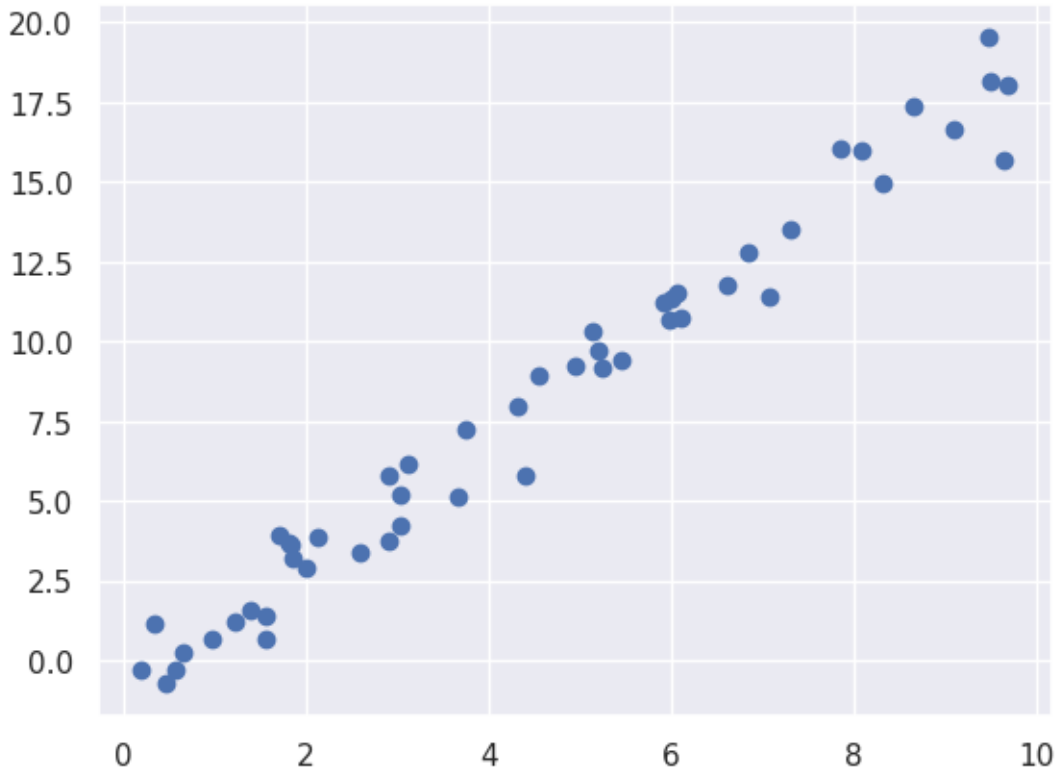
- Makes Scikit-Learn easy to use, once the basic principles are understood.
- Every machine learning algorithm in Scikit-Learn implemented via the Estimator API.
- Provides a consistent interface for a wide range of machine learning applications.

3.3.3 Typical Scikit-Learn Estimator API steps

1. Choose a class of model (import appropriate estimator class).
2. Choose model hyperparameters (instantiate class with desired values).
3. Arrange data into a features matrix and target vector.
4. Fit the model to data (calling `fit` method of model instance).
5. Apply model to new data:
    - Supervised learning: often predict targets for unknown data using the `predict` method.
    - For unsupervised learning: often transform or infer properties of the data using the `transform` or `predict` method.

3.4 Linear regression as machine learning

```
import matplotlib.pyplot as plt
import numpy as np

n_samples = 50
rng = np.random.RandomState(42)
x = 10 * rng.rand(n_samples)
y = 2 * x - 1 + rng.randn(n_samples)
plt.scatter(x, y);
```



3.4.1 1. Choose a class of model

Every class of model is represented by a Python class.

```
from sklearn.linear_model import LinearRegression
```

3.4.2 2. Choose model hyperparameters

Make instance of model with defined hyperparameters (e.g. y-intersect, regularization).

```
model = LinearRegression(fit_intercept=True)
model
```

```
LinearRegression()
```

3.4.3 3. Arrange data into a features matrix and target vector

```
X = x.reshape(n_samples,1)
X.shape
```

```
(50, 1)
```

```
y.shape
```

```
(50,)
```

3.4.4 4. Fit the model to data

```
model.fit(X, y)
```

```
LinearRegression()
```

All model parameters that were learned during the `fit()` process have *trailing underscores*.

```
model.intercept_
```

```
-0.9033107255311146
```

```
model.coef_
```

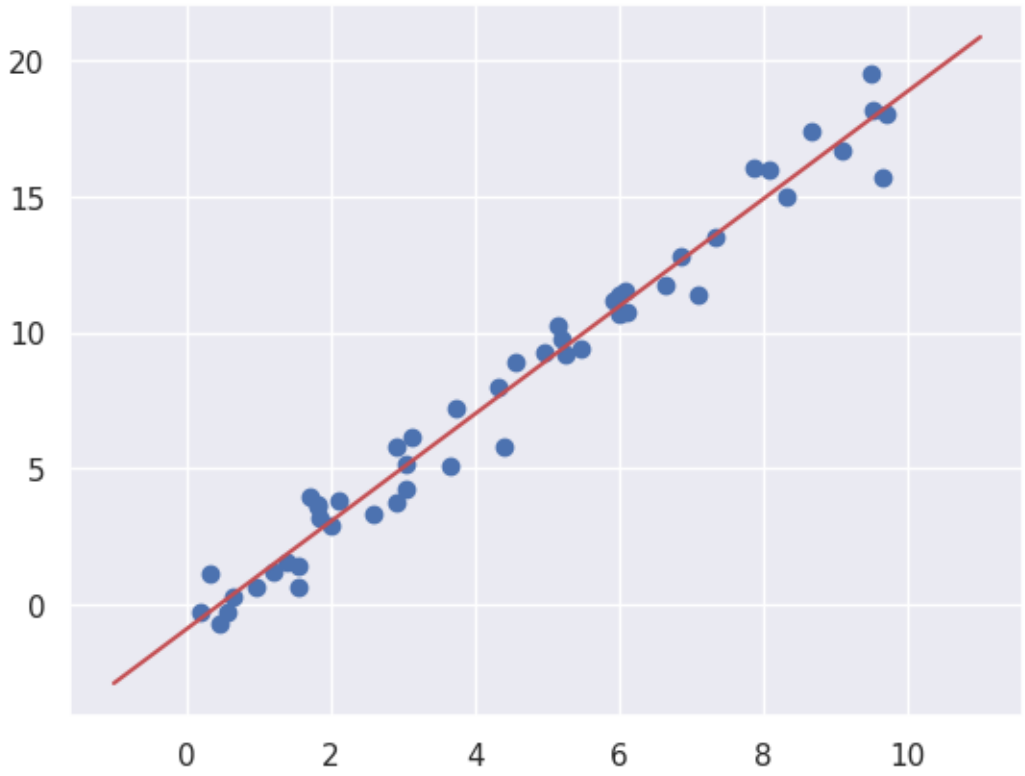
```
array([1.9776566])
```

Intercept and slope are close to the model used to generate the data (-1 and 2 respectively).

3.4.5 5. Predict targets for unknown data

```
n_fit = 50
xfit = np.linspace(-1, 11, n_fit)
Xfit = xfit.reshape(n_fit,1)
yfit = model.predict(Xfit)
```

```
plt.scatter(x, y)
plt.plot(xfit, yfit, 'r');
```



3.5 Supervised learning: classification

Consider Iris data-set and predict species.

Split data into training and test sets (hint: `train_test_split` is a convenient scikit-learn function for this task).

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=0.5,␣
↪random_state=1)
```

```
X_train.head()
```

```
     sepal_length  sepal_width  petal_length  petal_width
74            6.4          2.9           4.3          1.3
116           6.5          3.0           5.5          1.8
93            5.0          2.3           3.3          1.0
100           6.3          3.3           6.0          2.5
89            5.5          2.5           4.0          1.3
```

Use a Gaussian Naive Bayes (`GaussianNB`) model to predict Iris species. Then evaluate performance on test data.

(Hint: choose, instantiate, fit and predict.)

See Scikit-Learn documentation on `GaussianNB`.

Evaluate performance using simple `accuracy_score`.

(Do not set any priors.)

```
from sklearn.naive_bayes import GaussianNB  # 1. choose model class
model = GaussianNB()                        # 2. instantiate model
model.fit(X_train, y_train)                 # 3. fit model to data
y_model = model.predict(X_test)             # 4. predict on new data
```

Evaluate performance on test data.

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_model)
```

```
0.96
```

3.6 Unsupervised learning: dimensionality reduction

Reduce dimensionality of Iris data for visualisation or to discover structure.

Recall the original Iris data has four features.

```
X_iris.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

```
X_iris.shape
```

```
(150, 4)
```

Compute principle component analysis (`PCA`), with 2 components, and apply transform. Plot data in PCA space.

(Hint: choose, instantiate, fit and transform.)

See Scikit-Learn documentation on `PCA`.

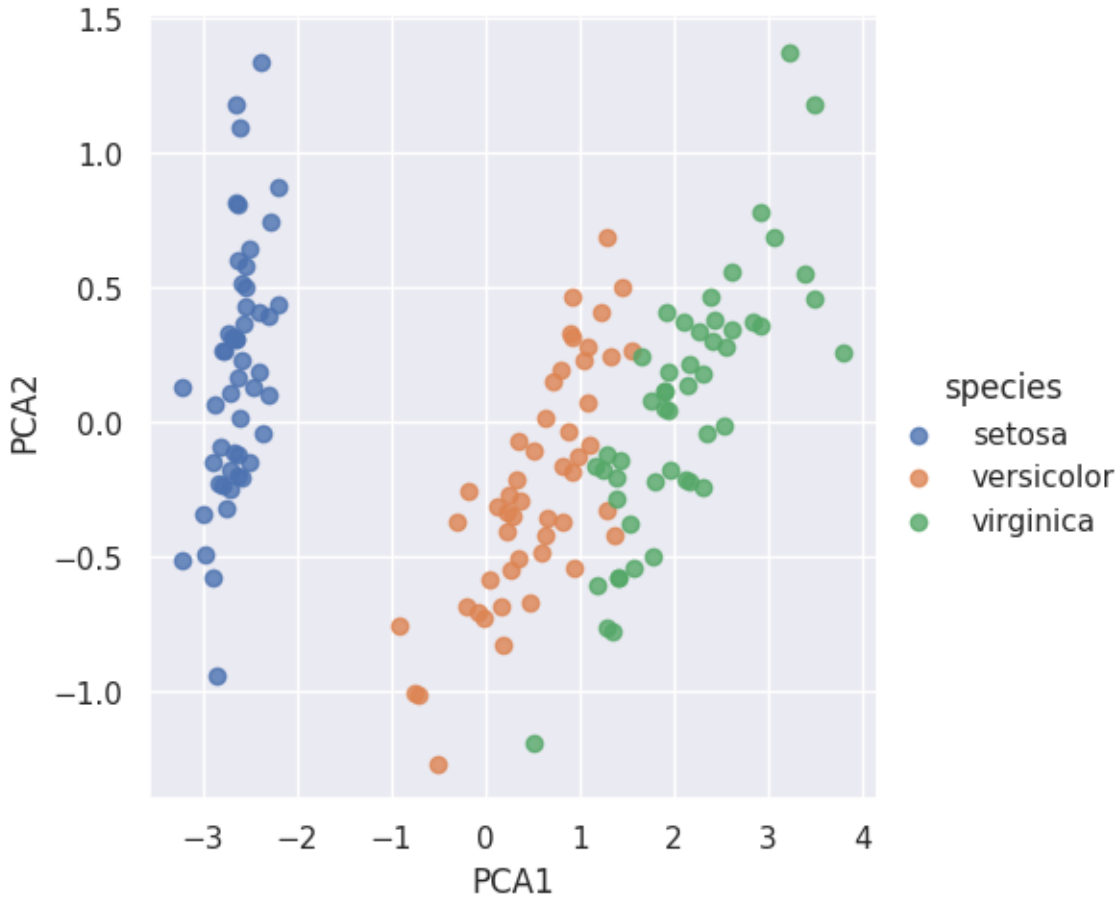
See Seaborn documentation on `lmplot`.

```
from sklearn.decomposition import PCA  # 1. Choose the model class
model = PCA(n_components=2)            # 2. Instantiate the model with hyperparameters
model.fit(X_iris)                      # 3. Fit to data. Notice y is not specified!
X_2D = model.transform(X_iris)         # 4. Transform the data to two dimensions
```

```
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
iris.head()
```

```
   sepal_length  sepal_width  petal_length  petal_width species      PCA1  \
0           5.1          3.5           1.4          0.2  setosa -2.684126
1           4.9          3.0           1.4          0.2  setosa -2.714142
2           4.7          3.2           1.3          0.2  setosa -2.888991
3           4.6          3.1           1.5          0.2  setosa -2.745343
4           5.0          3.6           1.4          0.2  setosa -2.728717

       PCA2
0  0.319397
1 -0.177001
2 -0.144949
3 -0.318299
4  0.326755
```

```
sns.lmplot(data=iris, x="PCA1", y="PCA2", hue='species', fit_reg=False);
```



How well do you expect classification to perform using PCA components as features and why?

Very well since the different classes are well separated in PCA feature space.

3.7 Unsupervised learning: clustering

Attempt to find "groups" in Iris data without given labels or training data.

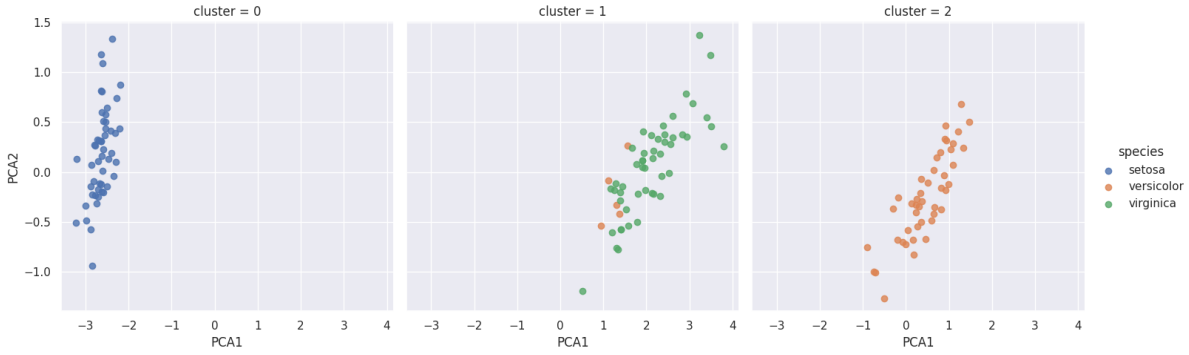
Cluster Iris data into 3 components using Gaussian Mixture Model (GMM). Plot the 3 components separately in PCA space.

(Hint: choose, instantiate, fit and predict.)

See Scikit-Learn documentation on `GaussianMixture`.

```
from sklearn.mixture import GaussianMixture      # 1. Choose the model class
model = GaussianMixture(n_components=3)          # 2. Instantiate the model with␣
 ↪hyperparameters
model.fit(X_iris)                                # 3. Fit to data. Notice y is not␣
 ↪specified!
y_gmm = model.predict(X_iris)                    # 4. Determine cluster labels
```

```
iris['cluster'] = y_gmm
sns.lmplot(data=iris, x="PCA1", y="PCA2", hue='species',
           col='cluster', fit_reg=False);
```



The GMM has done a reasonably good job of separating the different classes. Setosa is perfectly separated in one cluster, while there remains some mixing between versicolor and viginica.

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

CHAPTER

FOUR

LECTURE 4: PERFORMANCE ANALYSIS


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:12:42
```

4.1 Examining datasets

Use the MNIST digit dataset as a worked example in this lecture.

4.1.1 Fetch MNIST data

```
# Common imports
import os
import numpy as np
np.random.seed(42) # To make this notebook's output stable across runs
```

```
# Fetch MNIST dataset
from sklearn.datasets import fetch_openml
#mnist = fetch_openml('mnist_784')
mnist = fetch_openml('mnist_784', parser="pandas")
```

4.1.2 Extract features and targets

MNIST dataset is already split into standard training set (first 60,000 images) and test set (last 10,000 images).

```
y_train = mnist.target[:60000].to_numpy(dtype=int)
y_test = mnist.target[-10000:].to_numpy(dtype=int)
y_train.shape, y_test.shape
```

```
((60000,), (10000,))
```

```
X_train = mnist.data[:60000].to_numpy()
X_test = mnist.data[-10000:].to_numpy()
X_train.shape, X_test.shape
```

```
((60000, 784), (10000, 784))
```

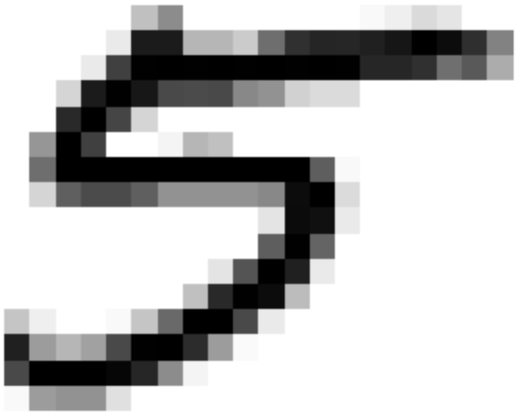
Each datum corresponds to a 28 x 28 image.

```
import math
n_float = np.sqrt(X_train.shape[1])
n = math.floor(n_float)
print(n_float, n)
```

```
28.0 28
```

4.1.3 Plot image of digit

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

some_digit = X_train[41000]
some_digit_image = some_digit.reshape(n, n)
plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
           interpolation="nearest")
plt.axis("off");
```



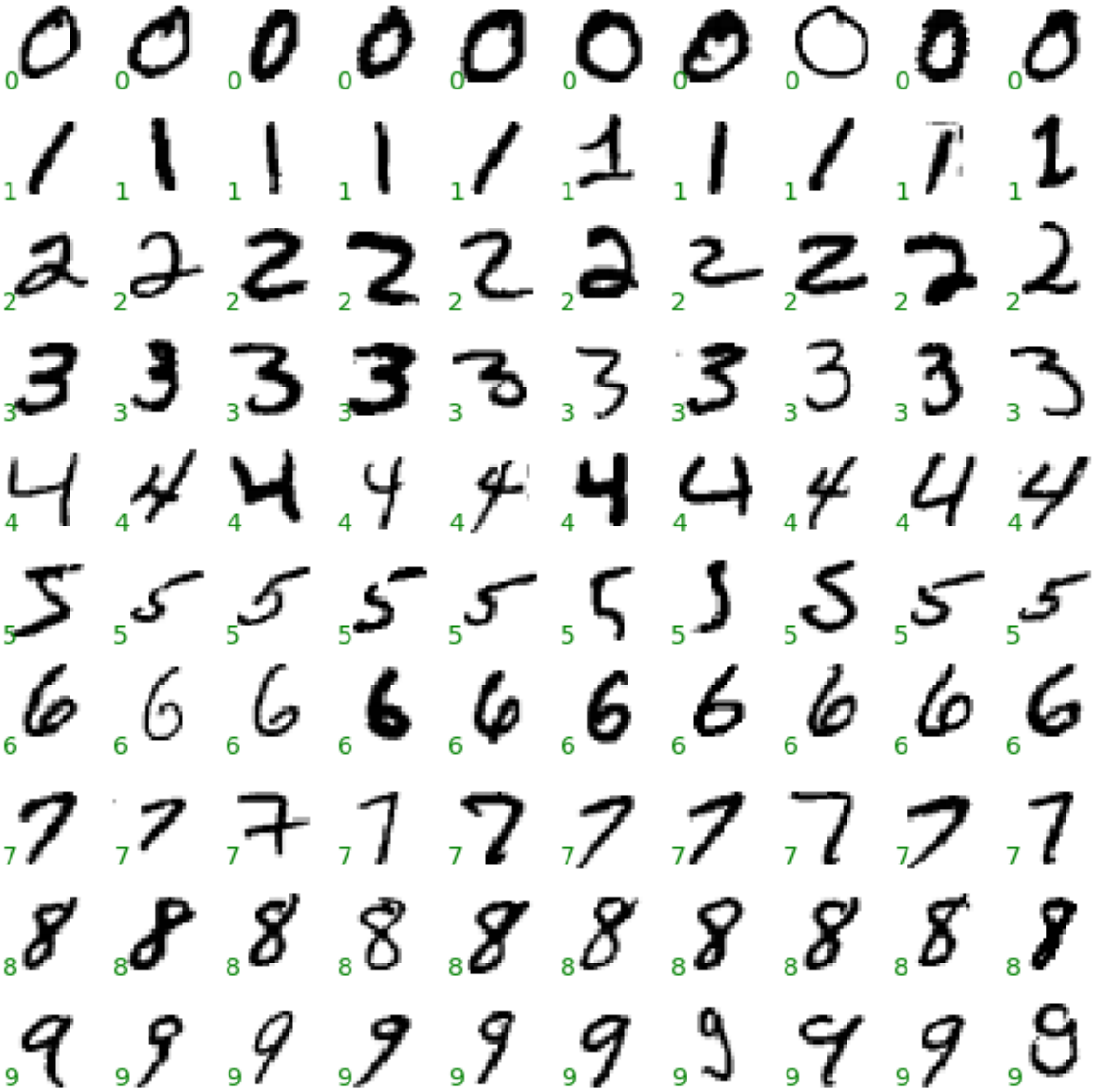
Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

4.1.4 Plot selection of digits

```
# Extract digits
n_digits = 10
n_images = 10
example_images = np.zeros([n_images * n_digits, n*n])
for i in range(n_digits):
    example_images[i*n_images:(i+1)*n_images,:] =  X_train[np.where(y_train ==␣
 ↪i)][0:n_images,:]
```

```
# Plot digits
plt.figure(figsize=(10,10))
fig, axes = plt.subplots(n_digits, n_images, figsize=(8, 8),
                         subplot_kw={'xticks':[], 'yticks':[]},
                         gridspec_kw=dict(hspace=0.1, wspace=0.1))

for i, ax in enumerate(axes.flat):
    ax.imshow(example_images[i].reshape(n,n), cmap='binary', interpolation='nearest')
    ax.axis("off")
    ax.text(0.05, 0.05, str(i // n_images),
            transform=ax.transAxes, color='green')
```

```
<Figure size 1000x1000 with 0 Axes>
```



```
def plot_digit(data):
    image = data.reshape(n, n)
    plt.imshow(image, cmap = matplotlib.cm.binary,
               interpolation="nearest")
    plt.axis("off");
```

Shuffle training data so not ordered by type.

```
# Shuffle training data
import numpy as np
shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

4.2 Binary classifier

Construct a classify to distinguish between 5s and all other digits.

```
y_train_5 = (y_train == 5)
y_test_5 = (y_test == 5)
```

```
y_test_5
```

```
array([False, False, False, ..., False,  True, False])
```

4.2.1 Train

Train a linear model using Stochastic Gradient Descent (good for large data-sets, as we will see later…).

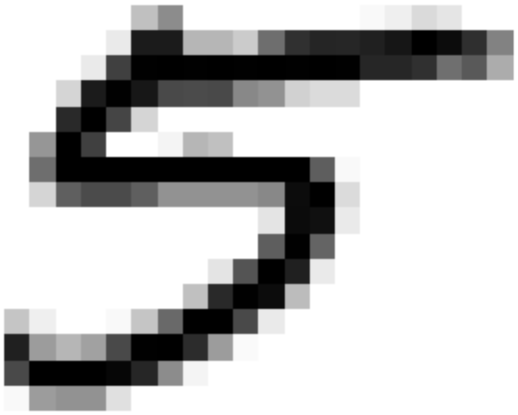
```
# disable convergence warning from early stopping
from warnings import simplefilter
from sklearn.exceptions import ConvergenceWarning
simplefilter("ignore", category=ConvergenceWarning)
```

```
from sklearn.linear_model import SGDClassifier
sgd_clf = SGDClassifier(random_state=42, max_iter=10);
sgd_clf.fit(X_train, y_train_5)
```

```
SGDClassifier(max_iter=10, random_state=42)
```

Recall extracted `some_digit` previously, which was a 5.

```
plot_digit(some_digit)
```



Predict class:

```
some_digit.shape

sgd_clf.predict([some_digit])
```

```
array([ True])
```

4.2.2 Test accuracy

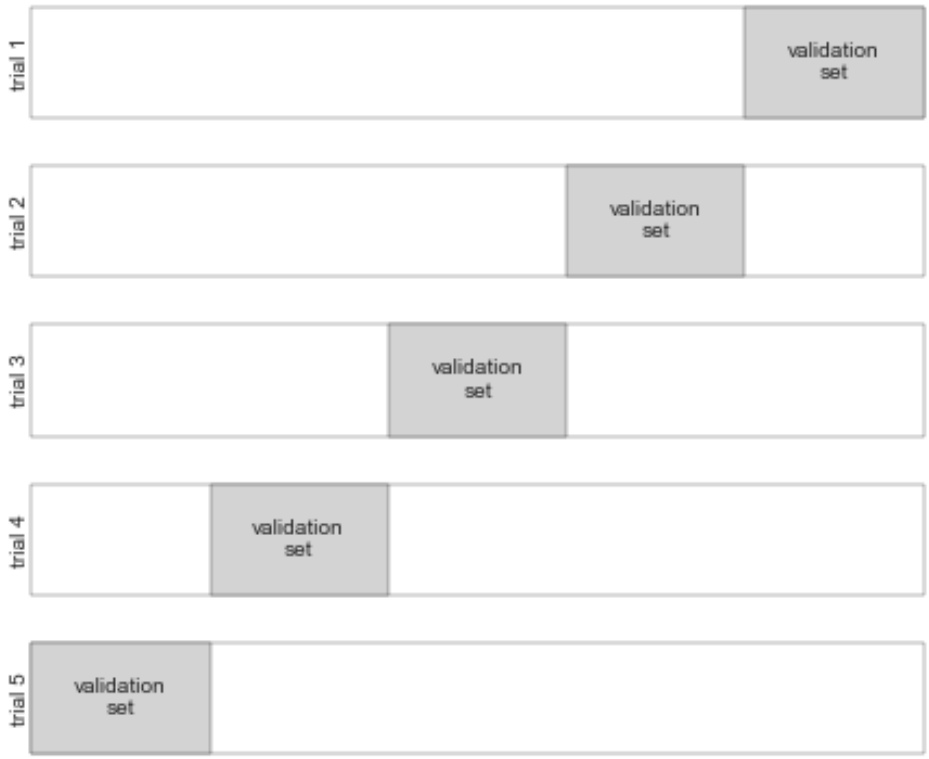
```
y_test = sgd_clf.predict(X_test)
```

```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_test_5)
```

```
0.9601
```

4.2.3 Cross-validation

n-fold cross-validation



[Image credit: VanderPlas]

Exercises: *You can now complete Exercises 2-3 in the exercises associated with this lecture.*

4.2.4 Consider naive classifier

Classify everying as not 5.

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

What accuracy expect?

```
from sklearn.model_selection import cross_val_score

never_5_clf = Never5Classifier()
cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

```
array([0.909  , 0.90745, 0.9125 ])
```

Need to go beyond classification accuracy, especially for skewed datasets.

4.3 Confusion matrix

Can gain further insight into performance by examining confusion matrix.

4.3.1 Confusion matrix shows true/false-positive/negative classifications

- True-positive TP: number of true positives (i.e. *correctly classified* as *positive*)
- False-positive FP: number of false positives (i.e. *incorrectly classified* as *positive*)
- True-negative TN: number of true negatives (i.e. *correctly classified* as *negative*)
- False-negative FN: number of false negatives (i.e. *incorrectly classified* as *negative*)

4.3.2 Cross-validation prediction

`cross_val_predict` performs K-fold cross-validation, returing predictions made on each test fold. Get clean prediction on each test fold, i.e. clean prediction for each instance in the training set.

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

4.3.3 Compute confusion matrix

```
from sklearn.metrics import confusion_matrix
conf_matrix = confusion_matrix(y_train_5, y_train_pred)
conf_matrix
```

```
array([[52953,  1626],
       [  807,  4614]])
```

Each row represents actual class, while each colum represents predicted class.

4.3.4 Perfect confusion matrix

```
y_train_perfect_predictions = y_train_5
confusion_matrix(y_train_5, y_train_perfect_predictions)
```

```
array([[54579,     0],
       [    0,  5421]])
```

Exercises: *You can now complete Exercise 4 in the exercises associated with this lecture.*

4.4 Precision and recall

- **Precision**: of predicted positives, proportion that are correctly classified (also called *positive predictive value*).

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall**: of actual positives, proportion that are correctly classified (also called *true positive rate* or *sensitivity*).

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Remember:

Exercises: *You can now complete Exercise 5 in the exercises associated with this lecture.*

4.4.1 F_1 score

F_1 score is the *harmonic mean* of the precision and recall.

$$F_1 = \frac{2}{1/\text{precision} + 1/\text{recall}} = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{\text{TP}}{\text{TP} + \frac{\text{FN}+\text{FP}}{2}}$$

F_1 favours classifiers that have similar (and high) precision and recall.

Sometimes may wish to favour precision or recall.

Exercises: *You can now complete Exercise 6 in the exercises associated with this lecture.*

4.4.2 Precision-recall tradeoff

Under the hood the classifier computes a *score*. Binary decision is then made depending on whether score exceeds some *threshold*.

By changing the threshold, one can change the tradeoff between precision and recall.

Scikit-Learn does not let you set the threshold directly but can access scores (confidence score for a sample is, e.g., the signed distance of that sample to classifying hyperplane).

```
y_scores = sgd_clf.decision_function([some_digit])
y_scores
```

```
array([24299.40524152])
```

Can then make prediction for given threshold.

```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

```
array([ True])
```

```
threshold = 200000
y_some_digit_pred = (y_scores > threshold)
y_some_digit_pred
```

```
array([False])
```

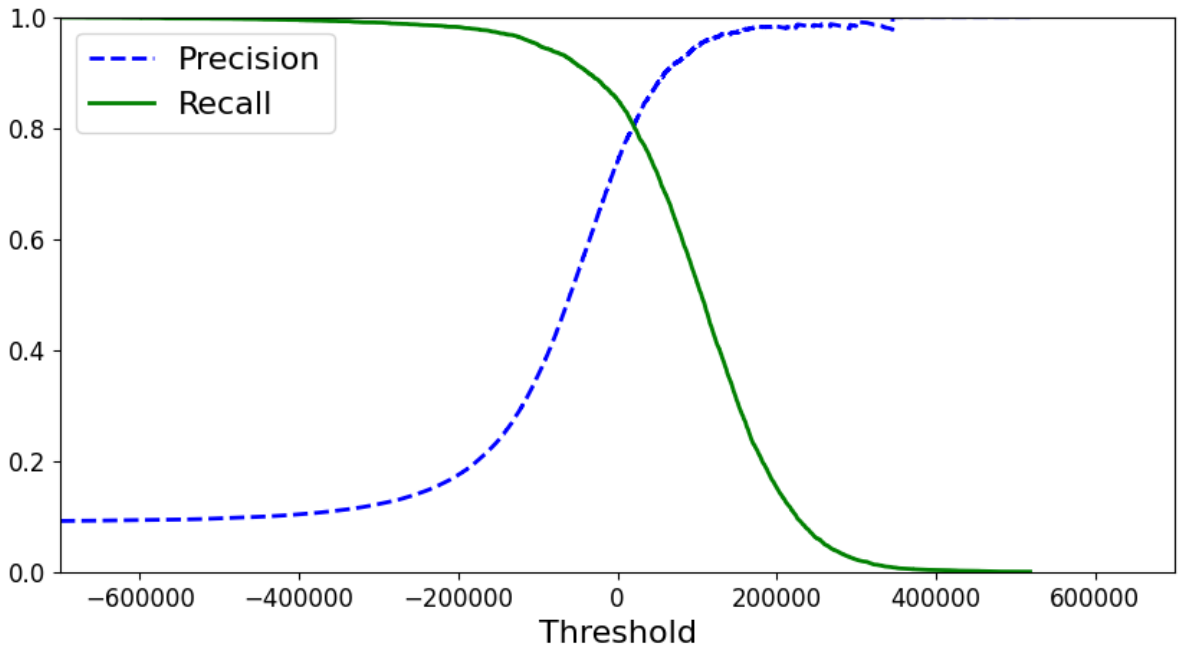
Compute precision and recall for range of thresholds

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

```
def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
    plt.xlabel("Threshold", fontsize=16)
    plt.legend(loc="upper left", fontsize=16)
    plt.ylim([0, 1])

plt.figure(figsize=(10, 5))
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.xlim([-700000, 700000])
```

```
(-700000.0, 700000.0)
```



Raising the threshold *increases precision* and *reduces recall*.

Can select threshold of appropriate trade-off for problem at hand.

Note recall curve smoother than precision since recall related to actual positives and precision related to predicted positives.

4.5 ROC curve

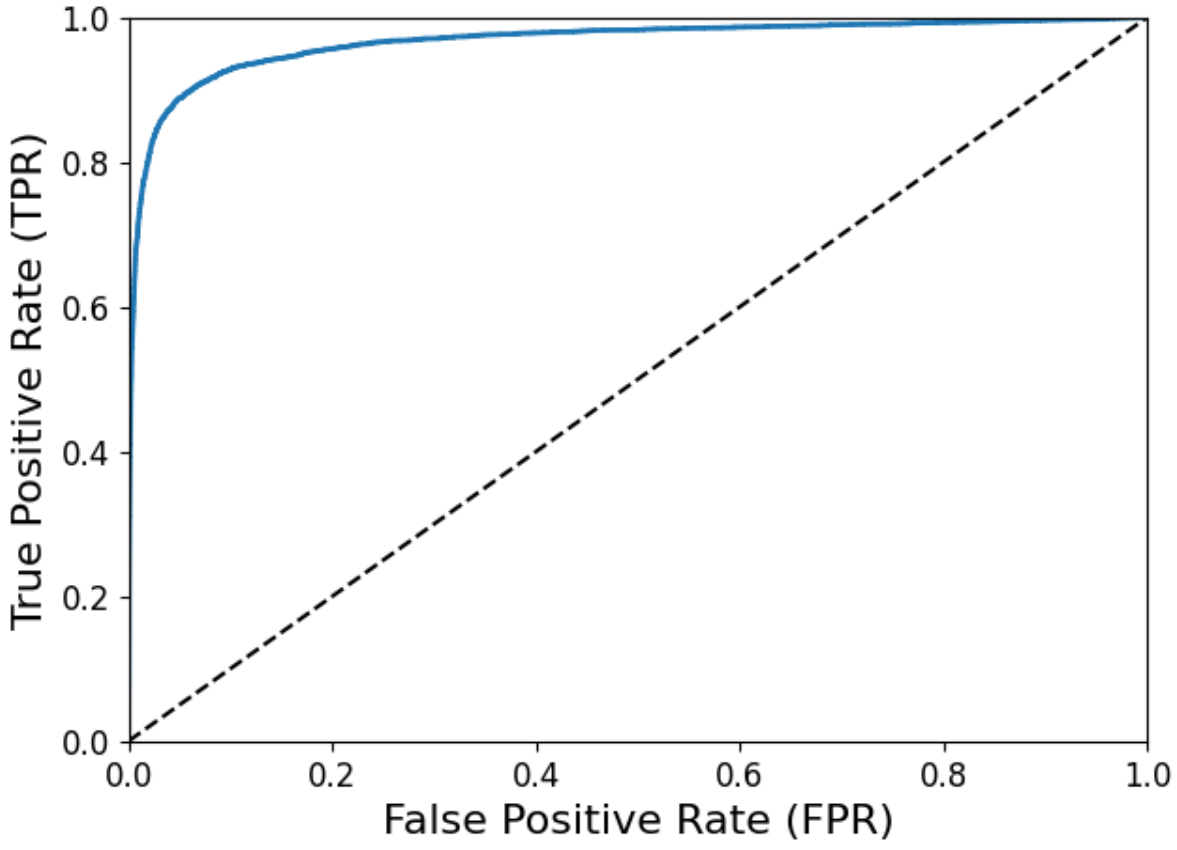
Receiver operating characteristic (ROC) curve plots *true positive rate* (i.e. recall) against the *false positive rate* for different *thresholds*.

Exercises: *You can now complete Exercise 7 in the exercises associated with this lecture.*

4.5.1 Plot ROC curve

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (FPR)', fontsize=16)
    plt.ylabel('True Positive Rate (TPR)', fontsize=16)

plt.figure(figsize=(7, 5))
plot_roc_curve(fpr, tpr)
```



Dashed line corresponds to random classifier.

Again, there is a trade-off. As the threshold is reduced to increase the true positive rate, we get a larger false positive rate.

Exercises: *You can now complete Exercise 8 in the exercises associated with this lecture.*

4.5.2 Area under the ROC curve

Area under the ROC curve (AUC) is a common performance metric.

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_train_5, y_scores)
```

```
0.9668396102663849
```

Exercises: *You can now complete Exercise 9 in the exercises associated with this lecture.*

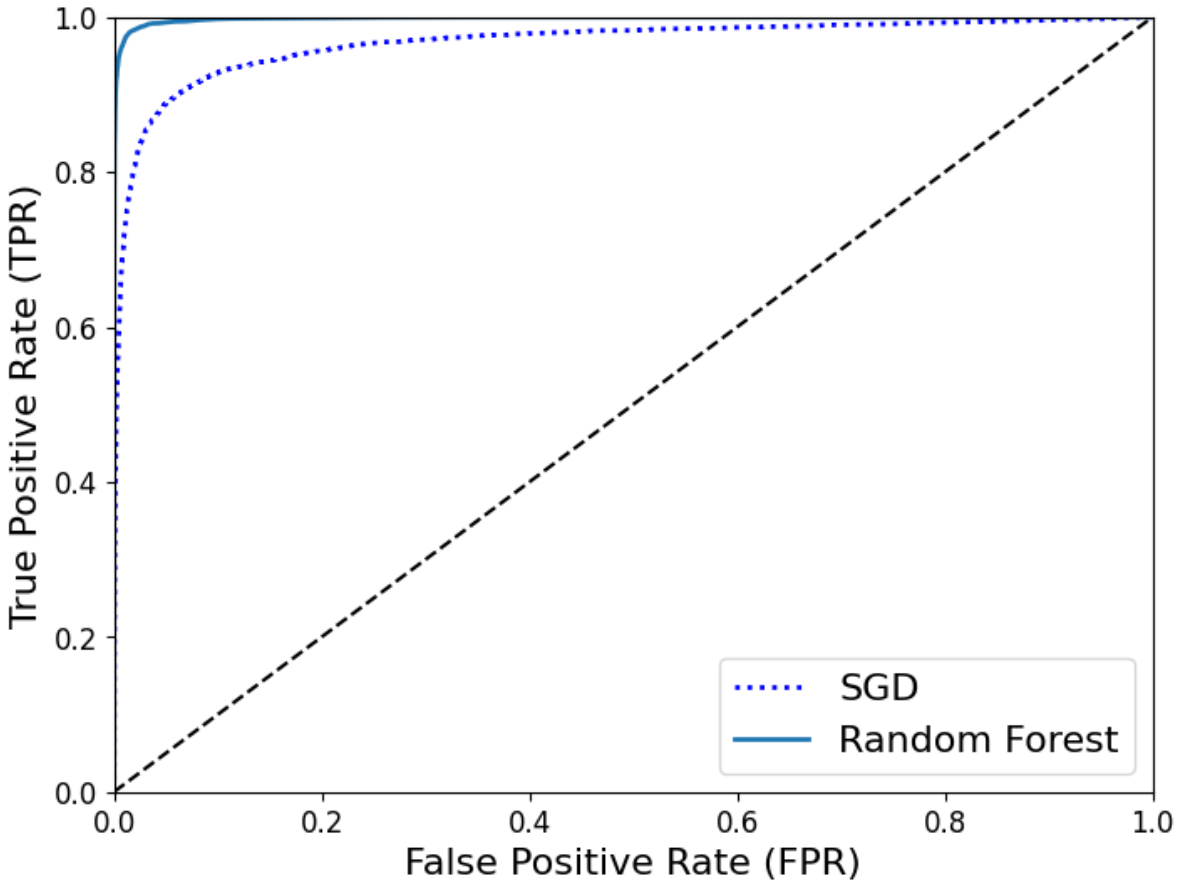
4.5.3 Comparing classifier ROC curves

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                    method="predict_proba")
```

```
y_scores_forest = y_probas_forest[:, 1] # score = proba of positive class
fpr_forest, tpr_forest, thresholds_forest = roc_curve(y_train_5,y_scores_forest)
```

```
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, "b:", linewidth=2, label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Random Forest")
plt.legend(loc="lower right", fontsize=16)
```

```
<matplotlib.legend.Legend at 0x7f935f83f1f0>
```



4.5.4 Exercise: from the ROC curve, which method appears to work better?

Random Forests since get closer to the ideal point (i.e. top left of plot).

4.5.5 Comparing metrics

```
# AUC
roc_auc_score(y_train_5, y_scores_forest), roc_auc_score(y_train_5, y_scores)
```

```
(0.9983631764491033, 0.9668396102663849)
```

```
# Precision
from sklearn.metrics import precision_score, recall_score, f1_score

y_train_pred_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3)
precision_score(y_train_5, y_train_pred_forest), precision_score(y_train_5, y_train_
↪pred)
```

```
(0.9890893831305078, 0.739423076923077)
```

```
# Recall
recall_score(y_train_5, y_train_pred_forest), recall_score(y_train_5, y_train_pred)
```

```
(0.8695812580704667, 0.8511344770337577)
```

```
# F_1
f1_score(y_train_5, y_train_pred_forest), f1_score(y_train_5, y_train_pred)
```

```
(0.9254932757435947, 0.7913558013892462)
```

4.5.6 Progress so far

So far we have considered binary classification only (e.g. five and not five).

Clearly in many scenarios we want to classify multiple classes.

4.6 Multiclass classification

Binary classifiers distinguish between two classes. Multiclass classifiers can distinguish between more than two classes.

Some algorithms can handle multiple classes directly (e.g. Random Forests, naive Bayes). Others are strictly binary classifiers (e.g. Support Vector Machines, Linear classifiers).

4.6.1 Multiclass classification strategies

However, there are various strategies that can be used to perform multiclass classification with binary classifiers.

- **One-versus-rest (OvR) / one-versus-all (OvA)**: train a binary classifier for each class, then select classification with greatest score across classifiers (e.g. train a binary classifier for each digit).
- **One-versus-one (OvO)**: train a binary classifier for each pair of classes, then select classification that wins most duels (e.g. train a binary classifier for each pairs of digits: 0 vs 1, 0 vs 2, …, 1 vs 2, 1 vs 3, …).

4.6.2 Comparison of multiclass classification strategies

One-versus-rest (OvR):

- N classifiers for N classes
- each classifier uses all of the training data

$\Rightarrow$ requires training relatively *few classifiers* but training each classifier can be *slow*.

One-versus-one (OvO):

- $N(N-1)/2$ classifiers for N classes
- each classifier uses a subset of the training data (typically much smaller than overall training dataset)

$\Rightarrow$ requires training *many* classifiers but training each classifier can be *fast*.

4.6.3 Preferred approach

OvR usually preferred, unless training binary classifier is very slow with large data-sets.

In Scikit-Learn, if try to use binary classifier for a multiclass classification problem, OvR is automatically run.

```
sgd_clf.fit(X_train, y_train)
sgd_clf.predict([some_digit])
```

```
array([5])
```

Can see OvR performed by inspecting scores, where we have a score per classifier.

The 5th score (starting from 0) is clearly largest.

```
some_digit_scores = sgd_clf.decision_function([some_digit])
some_digit_scores
```

```
array([[ -81742.80600673, -226403.87485225, -310042.19433877,
         -173577.43026798,  -82855.74343468,   39922.35938292,
         -183200.20815396,   10437.2327332 , -240036.68142135,
         -160691.66786235]])
```

```
sgd_clf.classes_
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
sgd_clf.classes_[np.argmax(some_digit_scores)]
```

```
5
```

4.6.4 OvO with Scikit-Learn

Can also perform OvO multiclass classification.

```
from sklearn.multiclass import OneVsOneClassifier
ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42, max_iter=10))
ovo_clf.fit(X_train, y_train)
ovo_clf.predict([some_digit]), len(ovo_clf.estimators_)
```

```
(array([5]), 45)
```

4.6.5 Many classifiers can inherently classify multiple classes

Random Forest can directly classify multiple classes so OvR or OvO classification not required.

```
forest_clf.fit(X_train, y_train)
forest_clf.predict([some_digit])
```

```
array([5])
```

```
forest_clf.predict_proba([some_digit])
```

```
array([[0.03, 0.01, 0.  , 0.02, 0.02, 0.85, 0.02, 0.03, 0.01, 0.01]])
```

```
cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
```

```
array([0.8687 , 0.86975, 0.8449 ])
```

4.6.6 Error analysis

Compute confusion matrix for multiclass classification.

```
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train, cv=3)
conf_mx = confusion_matrix(y_train, y_train_pred)
conf_mx
```

```
array([[5765,    2,   16,   16,    7,   21,   33,    9,   48,    6],
       [   1, 6470,   41,   17,    9,   35,   13,   16,  138,    2],
       [  79,   52, 5155,  153,   39,   31,  171,   85,  181,   12],
       [  80,   40,  224, 5087,   19,  286,   62,   78,  217,   38],
       [  37,   30,   46,   15, 5101,   22,   64,   75,  315,  137],
       [ 127,   27,   35,  208,   48, 4451,  175,   35,  259,   56],
       [  45,   10,   57,    3,   19,  112, 5611,   11,   45,    5],
       [  25,   32,   91,   49,   50,   15,    8, 5824,   81,   90],
       [  70,  175,  127,  278,   43,  394,   86,   51, 4571,   56],
       [  48,   33,   54,  117,  326,   99,    5,  801,  834, 3632]])
```

Exercises: *You can now complete Exercise 10 in the exercises associated with this lecture.*

Performance analysis can provide insight into how to make improvements.

For example, for the previous dataset, one might want to consider trying to improve the performane of classifying 9 by collecting more training data for 7s and 9s.

CHAPTER

FIVE

LECTURE 5: TRAINING I


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:19:20
```

5.1 Linear regression

5.1.1 Linear regression mode (scalar form)

$$\hat{y} = \theta_0 + \theta_1 x_1 + ... + \theta_n x_n$$

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_j is the jth feature, for $j = 0, 1, ..., n$.
- θ_j is the jth model parameter (with bias θ_0 and feature weights $\theta_1, ..., \theta_n$).

5.1.2 Linear regression mode (vector form)

$$\hat{y} = \theta^{\mathrm{T}} x = h_\theta(x)$$

- x is the instance feature vector $x = (x_0, x_1, ...x_n)^{\mathrm{T}}$, where $x_0 = 1$.
- θ is the vector of model parameters $\theta = (\theta_0, \theta_1, ...\theta_n)^{\mathrm{T}}$.
- $.^{\mathrm{T}}$ denotes transpose.
- $h_\theta(x)$ is the hypothesis function, with parameters θ.

5.1.3 Train linear regression

Give training data $\{x^{(i)}, y^{(i)}\}$, for m instances $i = 1, 2, ..., m$.

Minimise mean square error (MSE):

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} \left(\theta^{\text{T}} x^{(i)} - y^{(i)}\right)^2.$$

5.1.4 Concise matrix-vector notation

Recall:

- $x_j^{(i)}$ is (scalar) value of feature j in ith training example.
- $x^{(i)}$ is the (column) vector of features of the ith training example.
- $y^{(i)}$ is the (scalar) value of target of the ith training example.
- n features and m training instances

Define:

- Feature matrix: $X_{m \times n} = [x^{(1)},\ x^{(2)},\ ...,\ x^{(m)}]^{\text{T}}$.
- Target vector: $y_{m \times 1} = [y^{(1)},\ y^{(2)},\ ...,\ y^{(m)}]^{\text{T}}$.

5.1.5 Recall features matrix and target vector

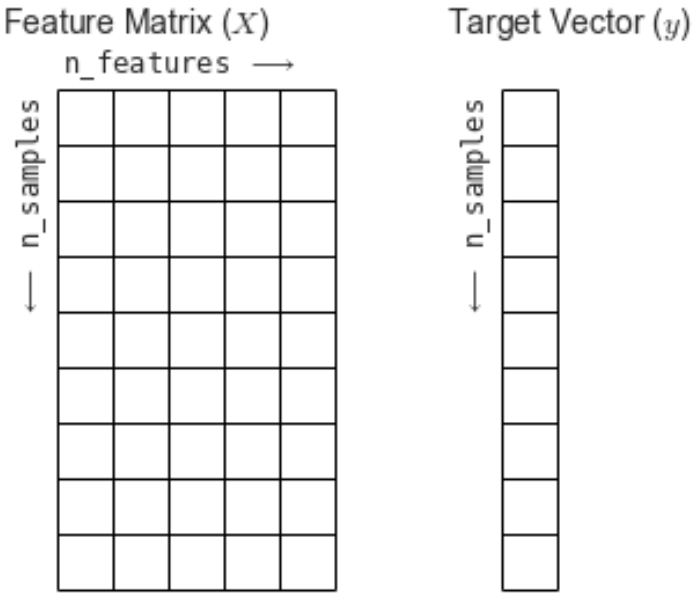


Image source

Minimising the MSE is equivalent to minimising the cost function

$$C(\theta) = \frac{1}{m}(X\theta - y)^{\text{T}}(X\theta - y),$$

where for notational convenience we denote the dependence on θ only and consider X and y fixed.

5.2 Normal equations

Minimise the cost function analytically

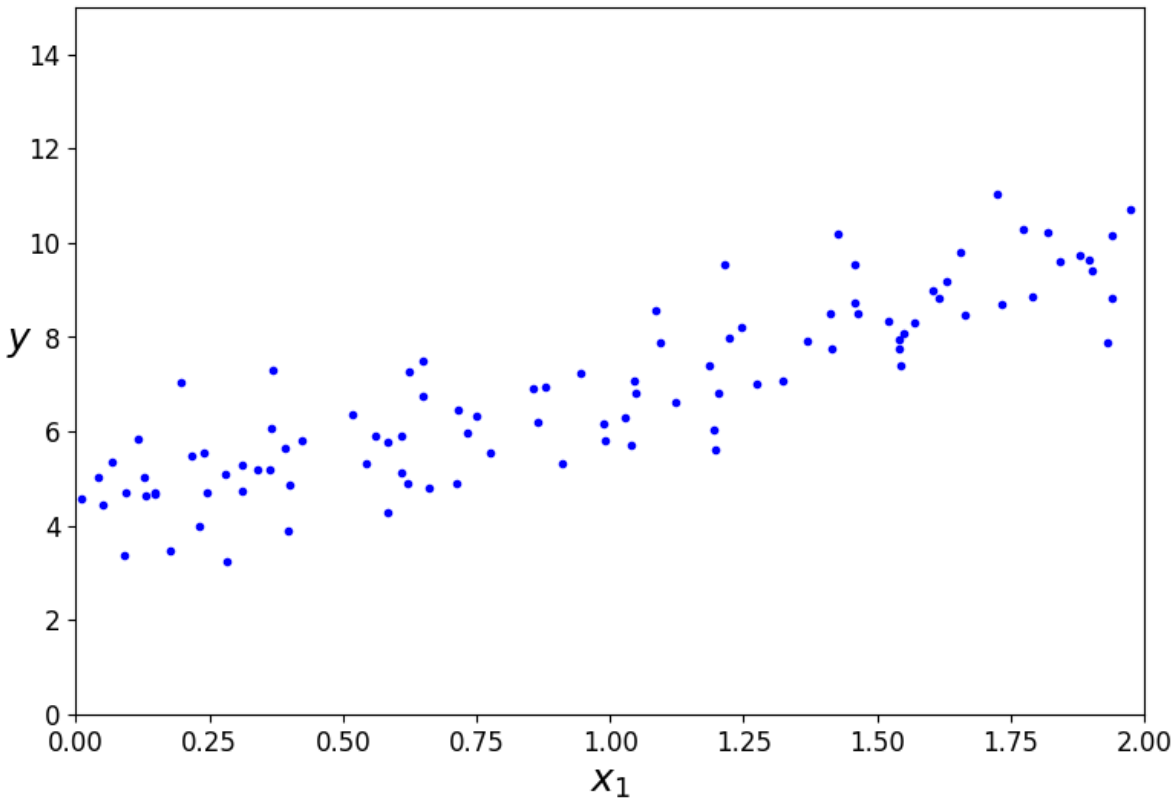
$$\min_{\theta} C(\theta) = \min_{\theta} (X\theta - y)^{\mathrm{T}}(X\theta - y).$$

Solution given by

$$\hat{\theta} = \left(X^{\mathrm{T}}X\right)^{-1} X^{\mathrm{T}}y.$$

```
# Common imports
import os
import numpy as np
np.random.seed(42) # To make this notebook's output stable across runs

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

```
import numpy as np
m = 100
X = 2 * np.random.rand(m, 1)
y = 4 + 3 * X + np.random.randn(m, 1)
plt.figure(figsize=(9,6))
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15]);
```



Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

5.2.1 Solve using Scikit-Learn

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([4.21509616]), array([[2.77011339]]))
```

```
X_new = np.array([[0], [2]])
lin_reg.predict(X_new)
```

```
array([[4.21509616],
       [9.75532293]])
```

5.2.2 Computational complexity

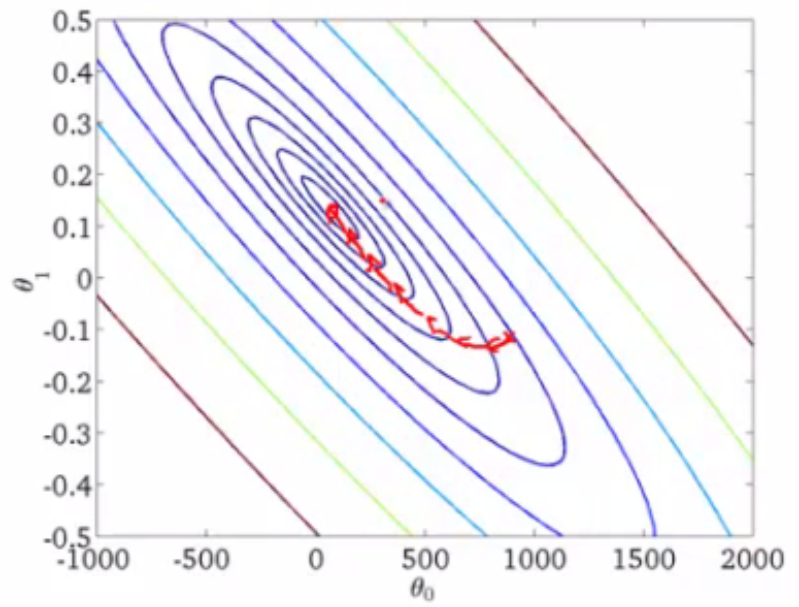
Solving the normal equation requires inverting $X^{\mathrm{T}}X$, which is an $n \times n$ matrix (where n is the number of features).

Inverting an $n \times n$ matrix is naively of complexity $\mathcal{O}(n^3)$ ($\mathcal{O}(n^{2.4})$ if certain fast algorithms are used).

Also, typically requires all training instances to be held in memory at once.

Other methods are required to handle large data-sets, i.e. when considering large numbers of features and large numbers of training instances.

5.3 Batch gradient descent



[Image source]

Gradient of the cost function is given by

$$\frac{\partial}{\partial\theta}C(\theta) = \frac{2}{m}X^{\mathrm{T}}\left(X\theta - y\right) = \left[\frac{\partial C}{\partial\theta_0},\ \frac{\partial C}{\partial\theta_1},\ ...,\ \frac{\partial C}{\partial\theta_n}\right]^{\mathrm{T}}.$$

Batch gradient descent algorithm defined by taking a step α in the direction of the gradient:

$$\theta^{(t)} = \theta^{(t-1)} - \alpha\frac{\partial C}{\partial\theta},$$

where t denotes the iteration number.

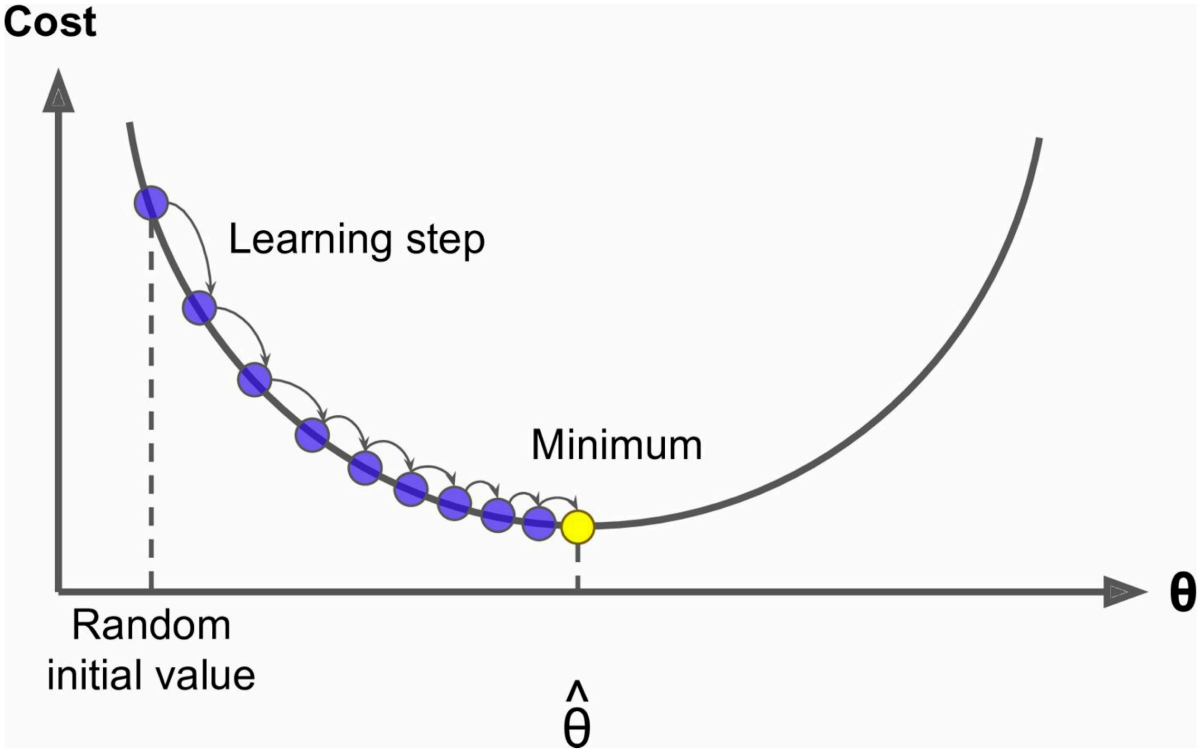
This algorithm is called *batch* gradient descent since it uses the full *batch* of training data (X) at each iteration. We will see other forms of gradient descent soon…

Exercises: *You can now complete Exercise 2 in the exercises associated with this lecture.*

5.4 Learning rate

The step size α is also called the *learning rate*.

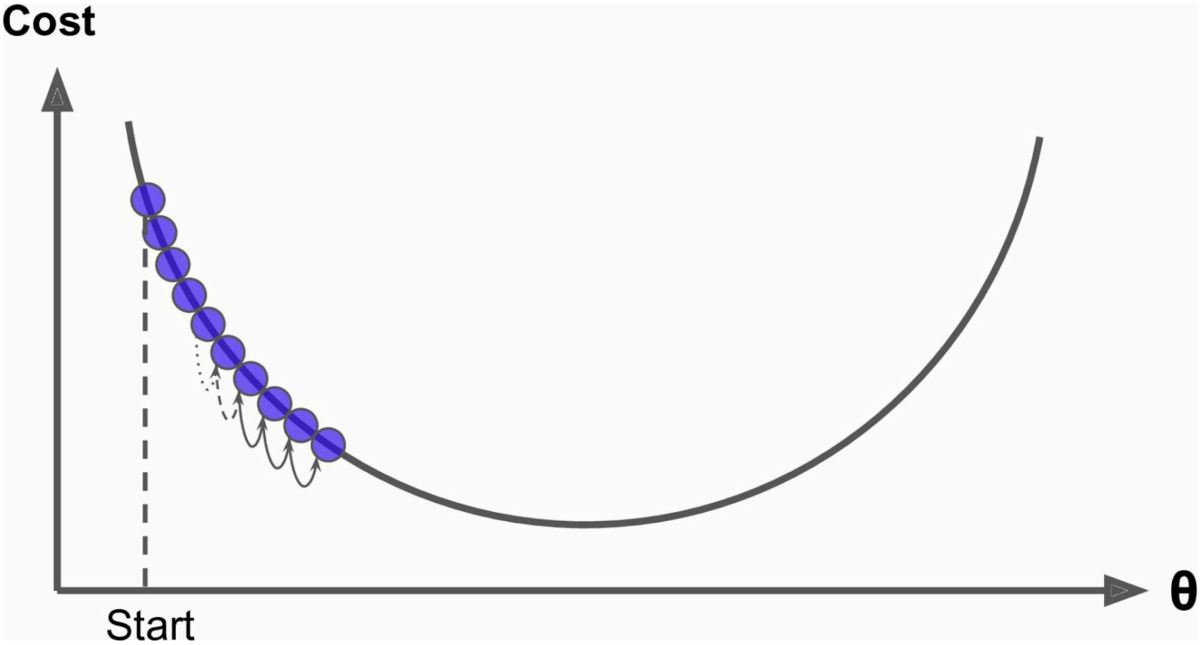
5.4.1 Good learning rate



[Source: Geron]

5.4.2 Learning rate too small

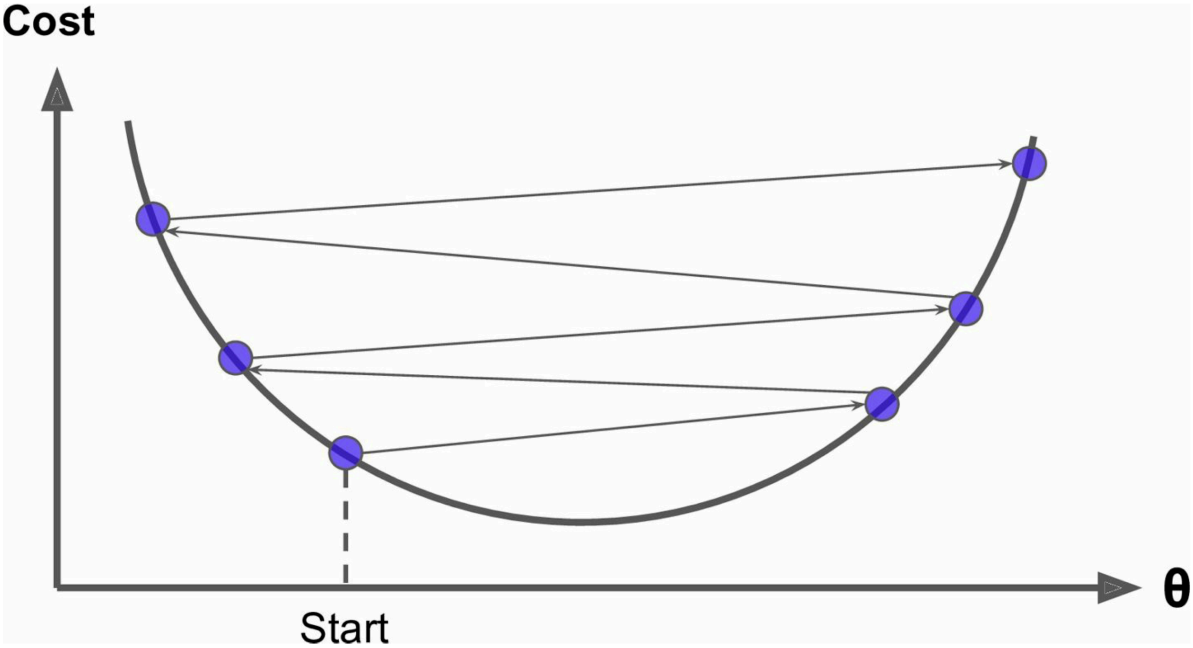
Convergence will be slow.



[Source: Geron]

5.4.3 Learning rate too large

May not converge.



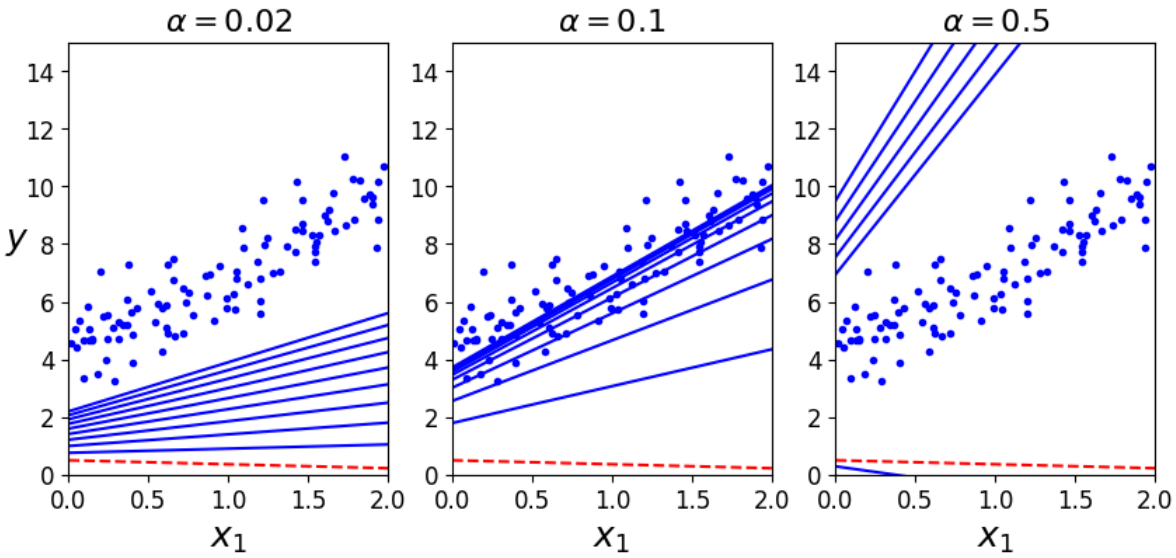
[Source: Geron]

5.4.4 Evolution of fitted curve

Consider the evolution of the curve corresponding to the best fit parameters for the first 10 iterations for different learning rates α.

```
theta_path_bgd = []

X_b = np.c_[np.ones((m, 1)), X]  # add x0 = 1 to each instance
X_new_b = np.c_[np.ones((2, 1)), X_new]

def plot_gradient_descent(theta, alpha, theta_path=None):
    m = len(X_b)
    plt.plot(X, y, "b.")
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta = theta - alpha * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\alpha = {}$".format(alpha), fontsize=16)
```

```
np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, alpha=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, alpha=0.1)
plt.subplot(133); plot_gradient_descent(theta, alpha=0.5)
```

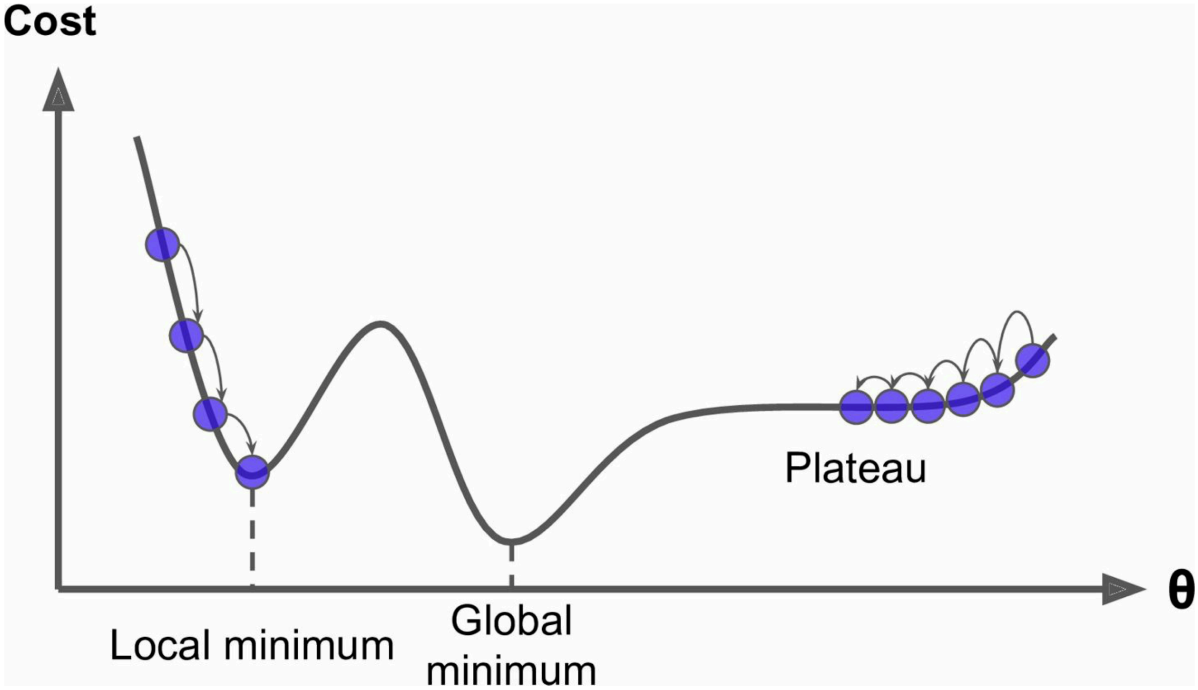


- $\alpha = 0.02$: learning rate is too small and will take a long time to reach the global optimum.

- $\alpha = 0.1$: learning rate is good and global optimum will be found quickly.
- $\alpha = 0.5$: learning rate is too large and parameters jump about (may not converge).

5.5 Convergence

Cost function may not be nice with a single local minimum that coincides with the global minimum.



[Source: Geron]

5.5.1 Convex objective function

MSE cost function for linear regression is convex (for any two points on the curve, the line segment connecting those points lies above the curve).

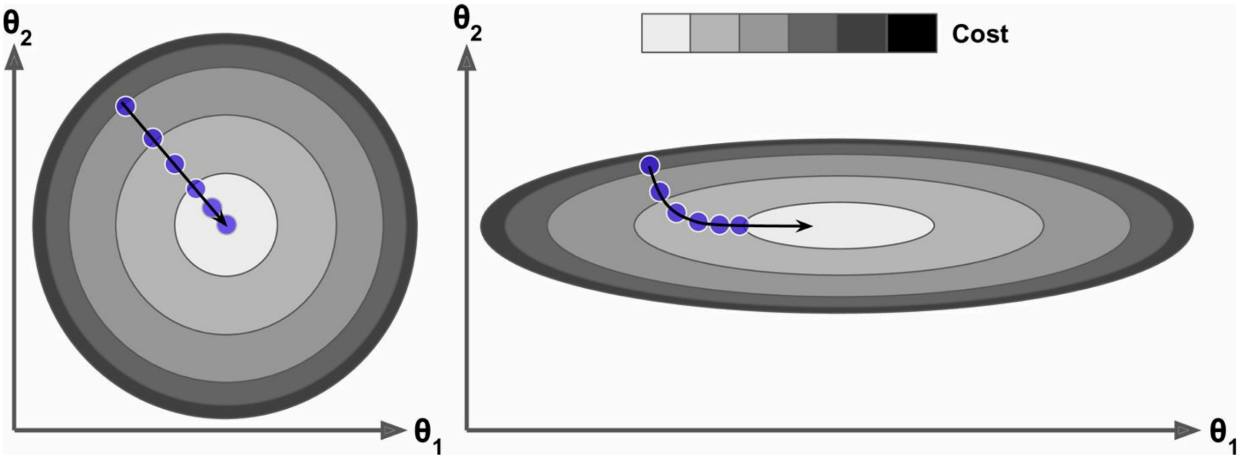
Consequently, the cost function has a single local minimum (that therefore coincides with the global minimum).

Furthermore, the cost function is smooth with a slope that never changes abruptly (i.e. it is Lipschitz continuous).

Gradient descent is therefore guaranteed to converge to the global minimum.

5.6 Feature scaling

For gradient descent to work well it is critical for all features to have a similar scale.



[Source: Geron]

In the case on the left, the features have the same scale and even intially the solution moves towards the minimum, thus reaching it quickly.

In the case on the right, the features have different scales and initially the solution does not move directly towards the minimum. Convergence is therefore slow.

5.6.1 Feature scaling methods

In general should ensure features have a similar scale since many machine learning methods do not work well when numerical attributes have very different scales.

Min-max scaling

Min-max scaling given by

$$x_j \to \frac{x_j - \min_i x_i}{\max_i x_i - \min_i x_i}.$$

See Scikit-Learn `MinMaxScalar`.

Standardization

Standardization scaling given by

$$x_j \to \frac{x_j - \mu_j}{\sigma_j},$$

where μ_j and σ_j are the mean and standard deviation of feature j computed from the training instances. See Scikit-Learn `StandardScaler`.

CHAPTER

SIX

LECTURE 6: TRAINING II


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:19:31
```

6.1 Stochastic gradient descent

6.1.1 Problems with batch gradient descent

- Uses the entire training set to compute gradients at every step (slow when the training set is large).
- Full training set needs to be held in memory.

6.1.2 Properties of stochastic gradient descent

- Uses a (random) single instance from the training set to compute gradients at each iteration (fast since very little data considered for each iteration).
- Only one instance of training data then needs to be held in memory.
- Less regular than batch gradient descent.
  - Helps to escape local minima.
  - Ends up close to a minimum but continues to explore vacinity around minimum (“bounces” around).

6.1.3 Simulated annealing

To mitigate issue of bouncing around minimum, can reduce learning rate as algorithm proceeds.

Called *simulated annealing* by analogy with annealing in metallurgy.

Learning schedule defines how learning rate changes over time.

- If learning rate reduces too quickly, may get stuck on local minimum or end up frozen half-way to minimum.
- If learning rate reduces too slowly, may jump around minimum for long time.

Example learning schedule

Set learning rate α at iteration t by

$$\alpha(t) = \frac{t_0}{t + t_1},$$

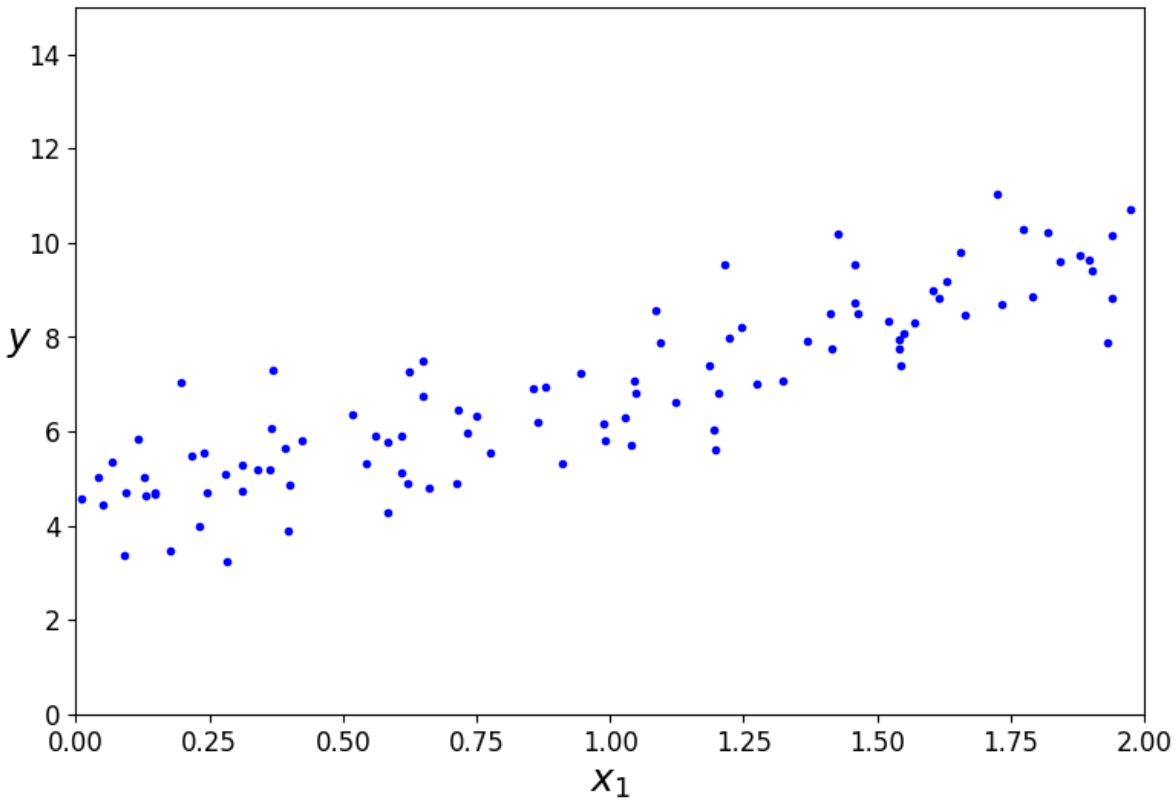
where t_0 and t_1 are parameters.

6.1.4 Stochastic gradient descent example

```
# Common imports
import os
import numpy as np
np.random.seed(42) # To make this notebook's output stable across runs

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

Set up training data (repeating example from previous lecture)

```
m = 100
X = 2 * np.random.rand(m, 1)
y = 4 + 3 * X + np.random.randn(m, 1)
plt.figure(figsize=(9,6))
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15]);
```



Add bias terms

```
X_b = np.c_[np.ones((m, 1)), X]          # add x0 = 1 to each instance
X_new = np.array([[0], [2]])
X_new_b = np.c_[np.ones((2, 1)), X_new] # add x0 = 1 to each instance
```

Solve by SGD with learning schedule

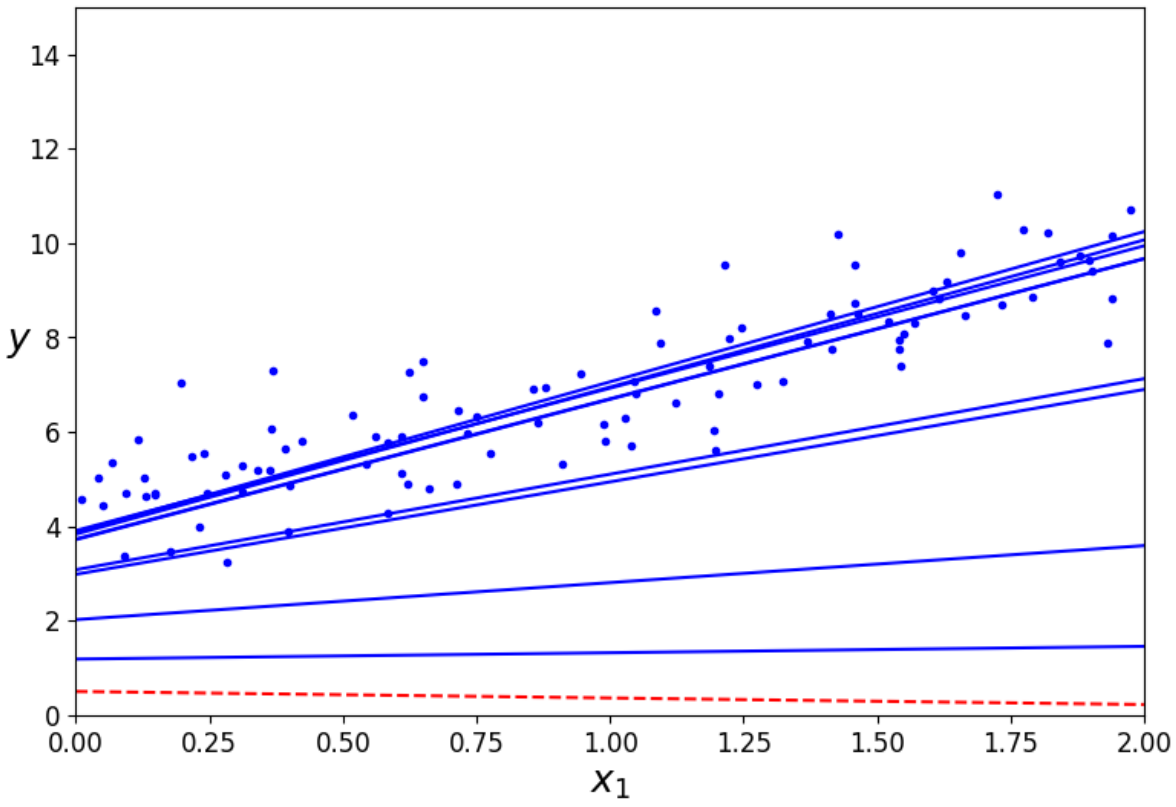
```
theta_path_sgd = []
m = len(X_b)
np.random.seed(42)

n_epochs = 50
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)  # random initialization
```

```
plt.figure(figsize=(9,6))
for epoch in range(n_epochs):
    for i in range(m):
```

(continues on next page)

(continued from previous page)

```
        # Plot current model
        if epoch == 0 and i < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if i > 0 else "r--"
            plt.plot(X_new, y_predict, style)

        # SGD update
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        alpha = learning_schedule(epoch * m + i)
        theta = theta - alpha * gradients
        theta_path_sgd.append(theta)

plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([0, 2, 0, 15]);
```




```
theta
```

```
array([[4.21076011],
       [2.74856079]])
```

Use only 50 passes over the data, compared to 1000 for batch gradient descent.

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

6.2 Mini-batch gradient descent

Use *mini-batches* of small random sets of instances of training data.

Trades off properties of batch GD and stochastic GD.

Can get a performance boost over SGD by exploiting hardware optimisation for matrix operations, particuarly for GPUs.

6.2.1 Shuffling training data

First step is to randomly shuffle or reorder data-set since do not want to be sensitive to ordering of data (want mini-batch considered to be representative).

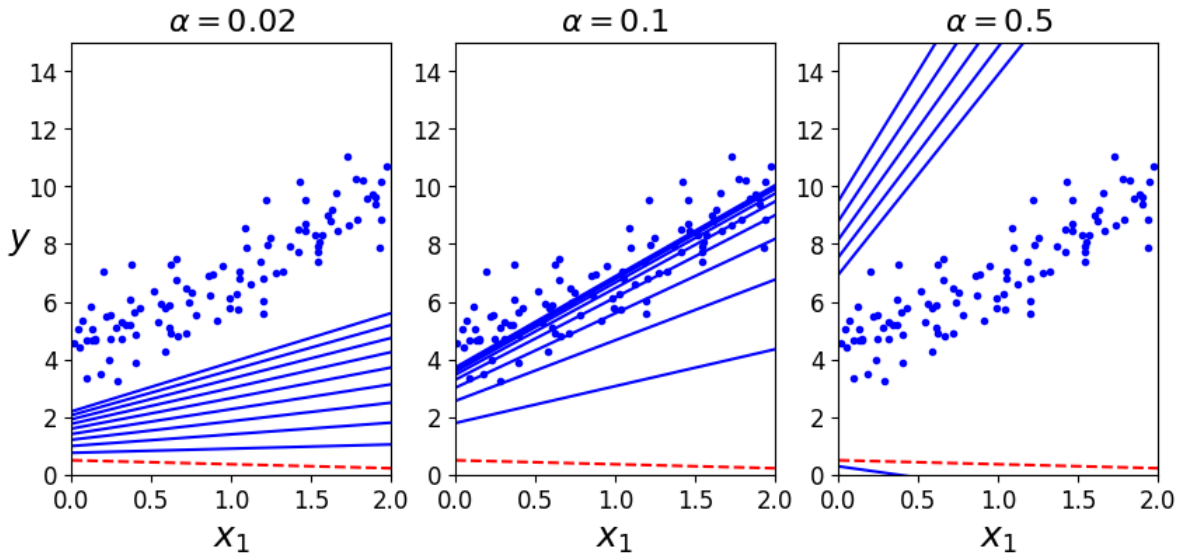
Exercises: *You can now complete Exercise 2 in the exercises associated with this lecture.*

6.3 Comparing gradient descent algorithms

6.3.1 Batch gradient descent (from previous lecture)

```
theta_path_bgd = []

def plot_gradient_descent(theta, alpha, theta_path=None):
    m = len(X_b)
    plt.plot(X, y, "b.")
    n_iterations = 1000
    for iteration in range(n_iterations):
        if iteration < 10:
            y_predict = X_new_b.dot(theta)
            style = "b-" if iteration > 0 else "r--"
            plt.plot(X_new, y_predict, style)
        gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
        theta = theta - alpha * gradients
        if theta_path is not None:
            theta_path.append(theta)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 2, 0, 15])
    plt.title(r"$\alpha = {}$".format(alpha), fontsize=16)
```

```
np.random.seed(42)
theta = np.random.randn(2,1)  # random initialization

plt.figure(figsize=(10,4))
plt.subplot(131); plot_gradient_descent(theta, alpha=0.02)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(132); plot_gradient_descent(theta, alpha=0.1, theta_path=theta_path_bgd)
plt.subplot(133); plot_gradient_descent(theta, alpha=0.5)
```



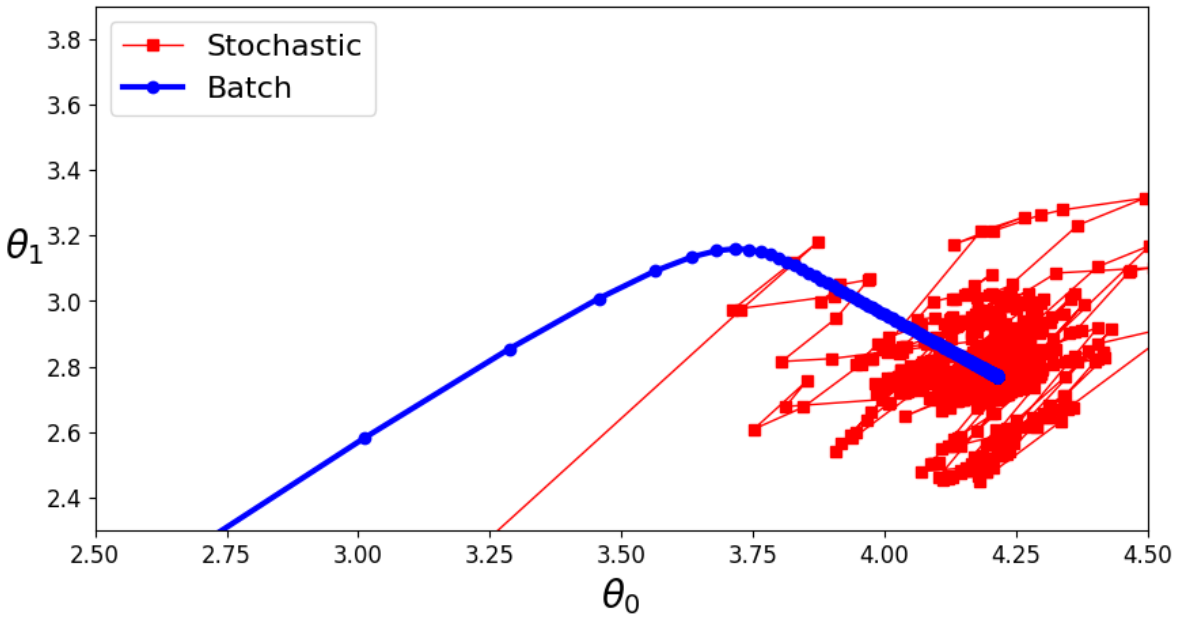
6.3.2 Convert lists to numpy arrays

```
theta_path_bgd = np.array(theta_path_bgd)
theta_path_sgd = np.array(theta_path_sgd)
```

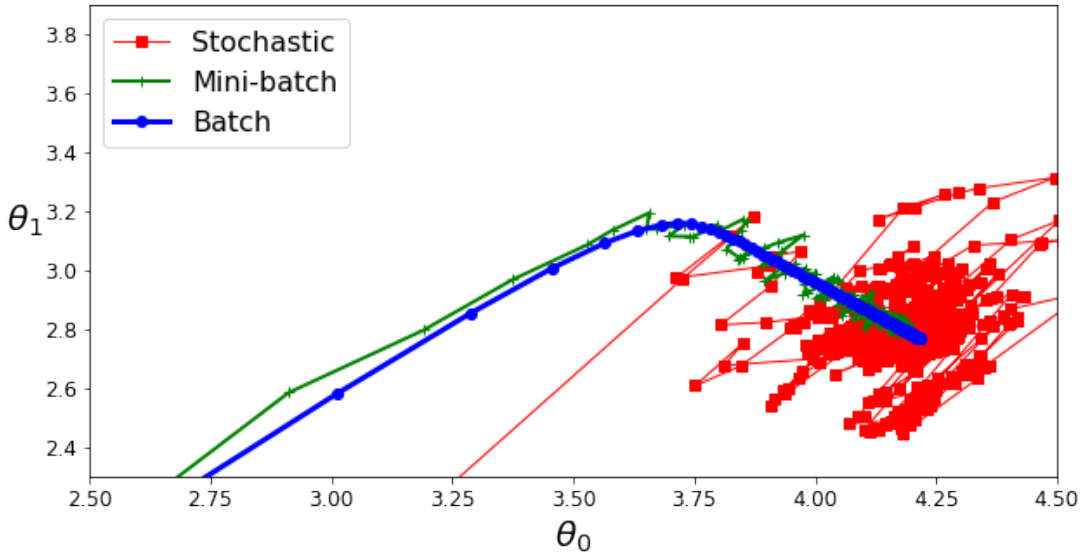
6.3.3 Algorithm trajectories

```
plt.figure(figsize=(10,5))
plt.plot(theta_path_sgd[:, 0], theta_path_sgd[:, 1], "r-s", linewidth=1, label=
 ↪"Stochastic")
plt.plot(theta_path_bgd[:, 0], theta_path_bgd[:, 1], "b-o", linewidth=3, label="Batch
 ↪")
plt.legend(loc="upper left", fontsize=16)
plt.xlabel(r"$\theta_0$", fontsize=20)
plt.ylabel(r"$\theta_1$    ", fontsize=20, rotation=0)
plt.axis([2.5, 4.5, 2.3, 3.9])
```

```
(2.5, 4.5, 2.3, 3.9)
```



We finally show the full trajectory for all optimisation methods, including mini-batch gradient descent (computed in exercises).



CHAPTER

SEVEN

LECTURE 7: TRAINING III


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:19:41
```

7.1 Polynomial regression

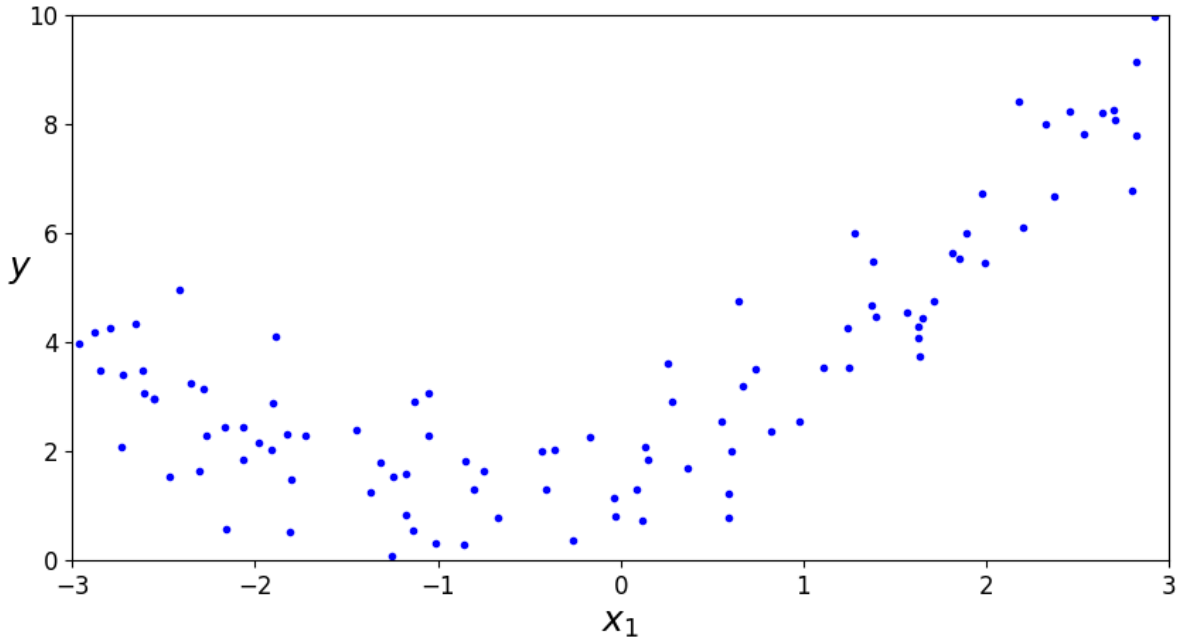
So far we have considered only linear regression. Polynomial regression can also be performed with a model that is linear (in the parameters).

```
# Common imports
import os
import numpy as np
np.random.seed(42) # To make this notebook's output stable across runs

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

7.1.1 Example data

```
import numpy.random as rnd
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

```
plt.figure(figsize=(10,5))
plt.plot(X, y, "b.")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10]);
```



Clearly a straight line will not fit the data well.

7.1.2 Construct new features

Can use a linear model by constructing additional features that are powers of existing features:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_2 + \theta_4 x_2^2 + \theta_5 x_1 x_2 + ...$$

Model remains linear in the parameters θ_j.

7.1.3 Generate polynomial features

```
from sklearn.preprocessing import PolynomialFeatures
poly_features = PolynomialFeatures(degree=2, include_bias=False)
X_poly = poly_features.fit_transform(X)
X[0], X_poly[0]
```

```
(array([-0.75275929]), array([-0.75275929,  0.56664654]))
```

```
X.shape, X_poly.shape
```

```
((100, 1), (100, 2))
```

7.1.4 Fit model

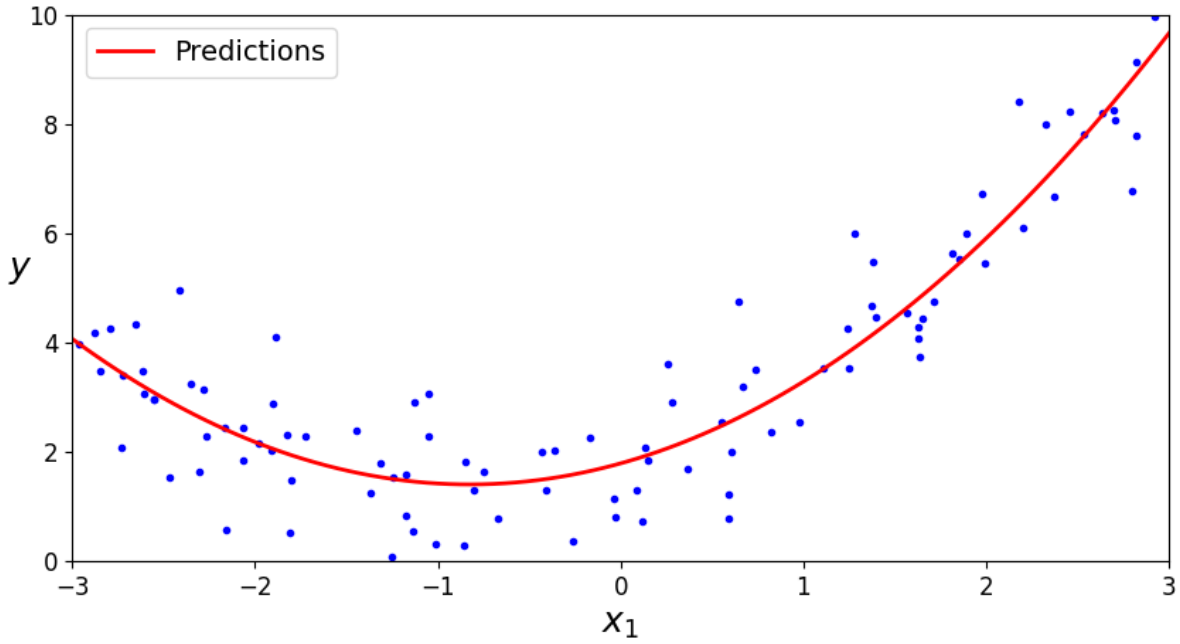
```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
lin_reg.intercept_, lin_reg.coef_
```

```
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

Parameters are close to the model used to generate the data (2, 1 and 0.5 respectively).

7.1.5 Predictions

```
X_new = np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
plt.figure(figsize=(10,5))
plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10]);
```



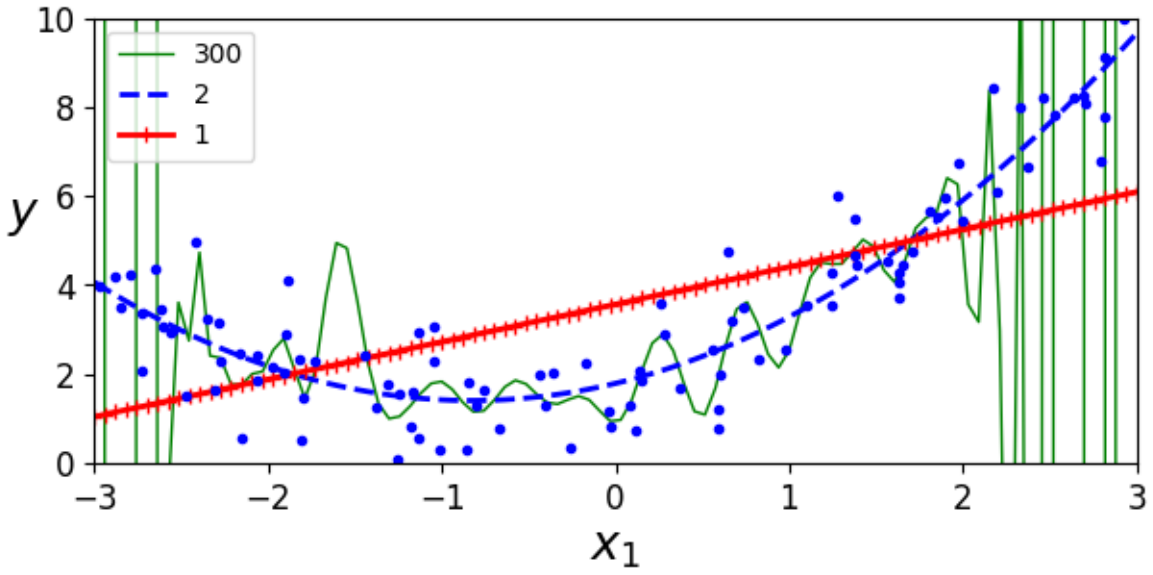
Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

7.2 Learning curves

How determine whether overfitting or underfitting?

7.2.1 Overfitting with high degree polynomials

```
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

plt.figure(figsize=(7,3))
for style, width, degree in (("g-", 1, 300), ("b--", 2, 2), ("r-+", 2, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline(
        (("poly_features", polybig_features), ("std_scaler", std_scaler), ("lin_reg",␣
 ↪lin_reg)) )
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)

plt.plot(X, y, "b.", linewidth=3)
plt.legend(loc="upper left")
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10]);
```



Training data is underfitted for degree 1, fitted well for degree 2, and overfitted for degree 300.

Learning curves provide another way to determine whether model underfitted or overfitted.

Consider performance on training and validation set *as size of the training set increases*.

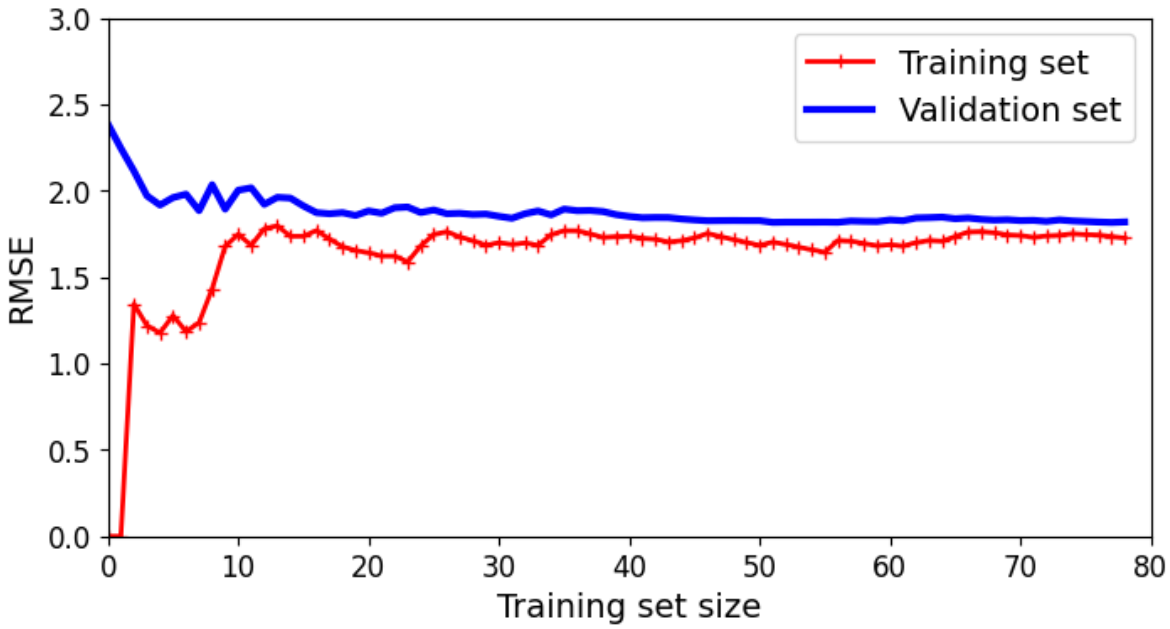
7.2.2 Plotting learning curves

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_
 ↪state=10)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict, y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))

    plt.figure(figsize=(8,4))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2, label="Training set")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
    plt.legend(loc="upper right", fontsize=14)
    plt.xlabel("Training set size", fontsize=14)
    plt.ylabel("RMSE", fontsize=14)
```

7.2.3 Underfitted learning curves

Learning curve for linear model

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
plt.axis([0, 80, 0, 3]);
```

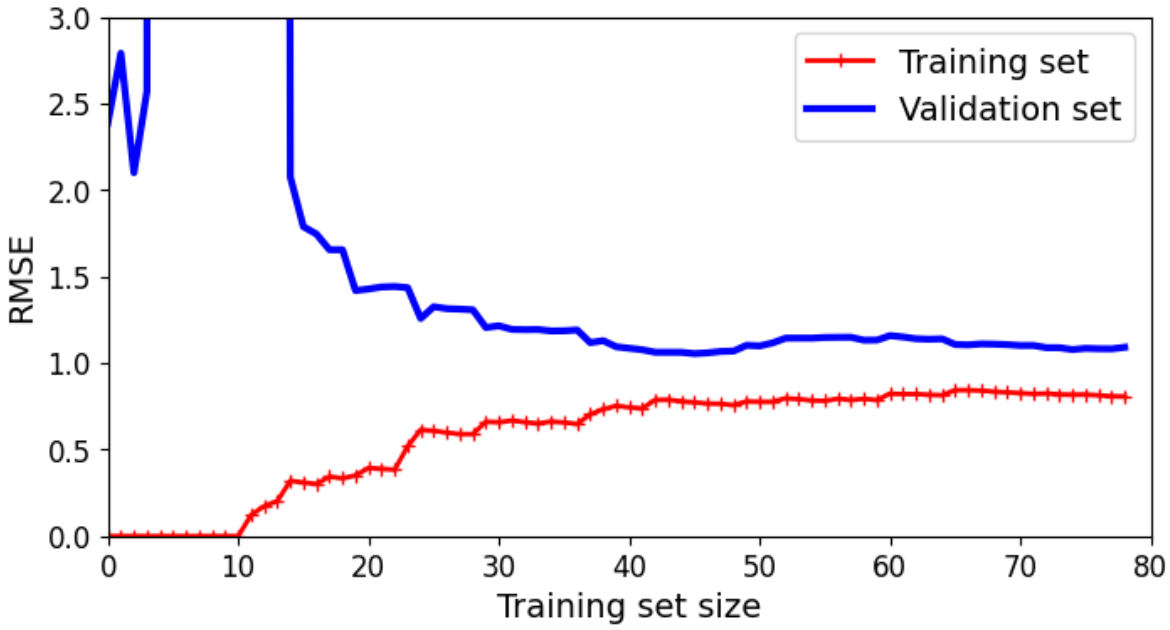


- RMSE on training set small for small training set size since model can fit data well for very few data points (perfect for one or two points).
- RMSE performance on training and validation eventually similar but high since linear model cannot fit the data well (recall data generated by quadratic).

7.2.4 Overfitted learning curves

Learning curve for poynomial model of degree 10

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline((
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ))
plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3]);
```



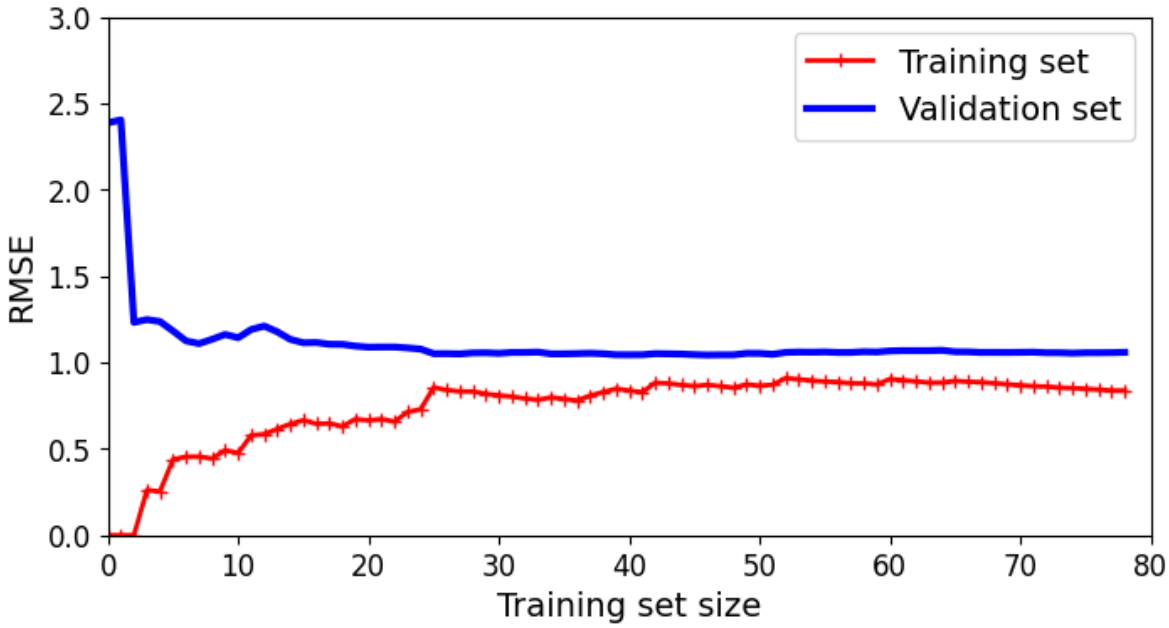
- RMSE now much smaller.
- But training and validation set errors remain quite different.

Model performs much better on training set than validation set, suggesting overfitting.

7.2.5 Well-fitted learning curves

Learning curve for poynomial model of degree 2

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline((
        ("poly_features", PolynomialFeatures(degree=2, include_bias=False)),
        ("lin_reg", LinearRegression()),
    ))
plot_learning_curves(polynomial_regression, X, y)
plt.axis([0, 80, 0, 3]);
```



7.3 Bias-variance tradeoff

Bias-variance tradeoff refers to the problem of simultaneously reducing the two types of errors that prevents supervised learning algorithms from generalising to other data.

- **Bias**: Expected difference between data and prediction.
- **Variance**: Expected ability of the model to fluctuate.

One seeks a model that accurately fits the training data, while also generalising to unseen data.

Typically impossible to do both simultaneously.

- On one hand, high-variance models may fit training data well but typically overfit to noise or unrepresentative training data.
- On the other hand, low-complexity models with a high bias typically underfit training data.

7.3.1 Contributions to the mean square error

Consider underlying (true) model: $y = f(x) + \epsilon$,

where

- y is the target and x features.
- f is the true model, that we will approximate by h.
- ϵ is the noise, with zero mean and variance σ^2.

Approximate f by h, which is fitted by a learning algorithm and training data.

Expected value of the mean square error is given by

$$\mathrm{E}\left[(y - h(x))^2\right] = \mathrm{Bias}^2\left[h(x)\right] + \mathrm{Var}\left[h(x)\right] + \sigma^2$$

Three contributions to the error:

1. Bias: $\mathrm{Bias}\left[h(x)\right] = \mathrm{E}\left[h(x) - f(x)\right]$.
2. Variance: $\mathrm{Var}\left[h(x)\right] = \mathrm{E}\left[h^2(x)\right] - \mathrm{E}\left[h(x)\right]^2$.
3. Irreducible error σ^2 due to noise in observations.

7.3.2 Tradeoff

By choosing a complex model, the bias can be made small but the variance will be large.

By choosing a simple model, the variance can be made small but the bias will be large.

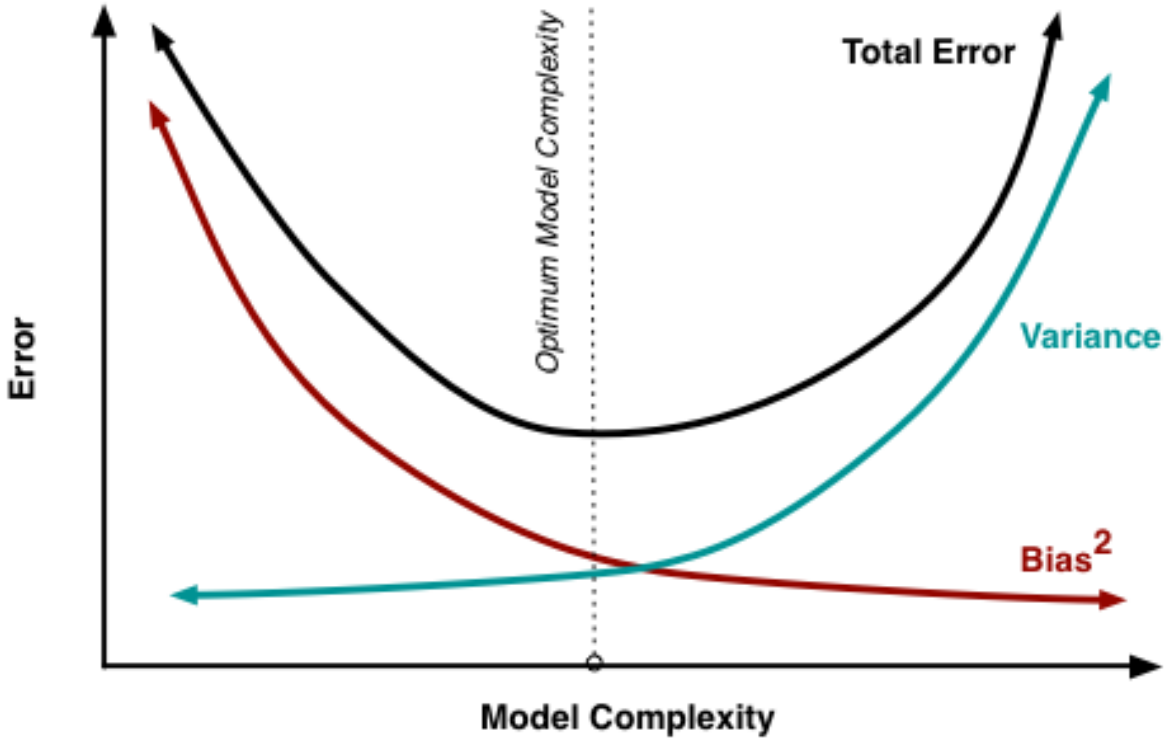


Image source

7.4 Regularization

One approach to mitigate the bias-variance tradeoff is by *regularization*.

Consider a complex model but place additional constraints to reduce its variance.

As a consequence the bias is increased but can introduce a regularisation parameter to control the tradeoff.

Add regularization term $R(\theta)$ to the cost function:

$$C_\lambda(\theta) = C(\theta) + \lambda\, R(\theta).$$

The regularization parameter λ controls the amount of regularization.

Regularization should only be added when training. When using fitted model to make predictions, should evaluate cost without regularization term.

7.4.1 Tikhonov regularization

Tikhonov regularization adopts ℓ_2 regularising term (also called *Ridge regression*):

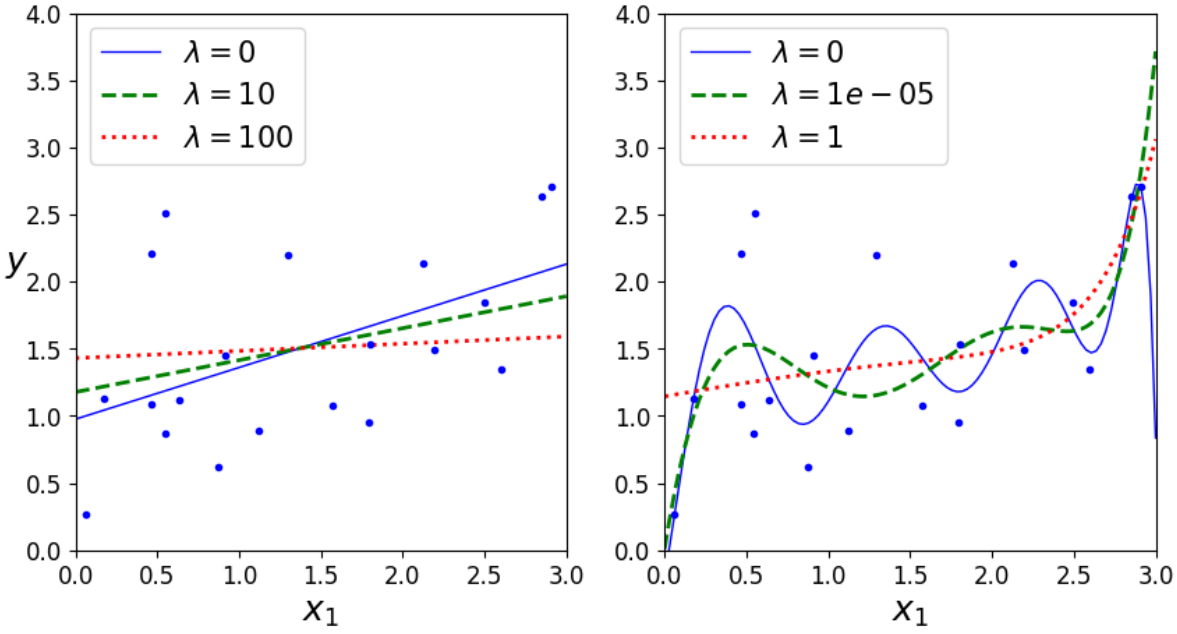
$$R(\theta) = \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\theta^{\mathrm{T}}\theta.$$

Acts to keep parameters small.

Note that the bias term θ_0 is not regularized (i.e. sum starts from 1 not 0).

```
from sklearn.linear_model import Ridge

np.random.seed(42)
m = 20
X = 3 * np.random.rand(m, 1)
y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)

def plot_model(model_class, polynomial, alphas, **model_kargs):
    # Use alpha for regularization parameter (lambda used already)
    for alpha, style in zip(alphas, ("b-", "g--", "r:")):
        model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegression()
        if polynomial:
            model = Pipeline((
                    ("poly_features", PolynomialFeatures(degree=10, include_
↪bias=False)),
                    ("std_scaler", StandardScaler()),
                    ("regul_reg", model),
                ))
        model.fit(X, y)
        y_new_regul = model.predict(X_new)
        lw = 2 if alpha > 0 else 1
        plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\lambda = {}$".
↪format(alpha))
    plt.plot(X, y, "b.", linewidth=3)
    plt.legend(loc="upper left", fontsize=15)
    plt.xlabel("$x_1$", fontsize=18)
    plt.axis([0, 3, 0, 4])
```

```
plt.figure(figsize=(10,5))
plt.subplot(121)
plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)
```



Exercises: *You can now complete Exercises 2 and 3 in the exercises associated with this lecture.*

7.4.2 Lasso regularization

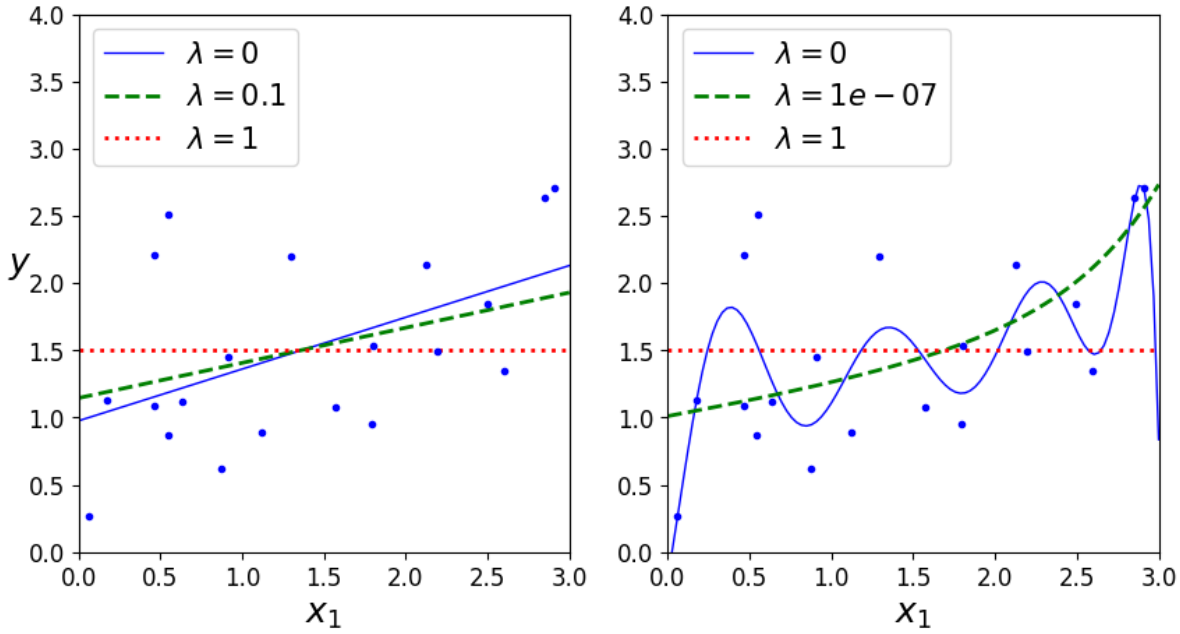
Lasso regularization adopts ℓ_1 regularising term:

$$R(\theta) = \sum_{j=1}^{n} |\theta_j| .$$

Acts to promote sparsity.

Again, note that the bias term θ_0 is not regularized (i.e. sum starts from 1 not 0).

```
from sklearn.linear_model import Lasso

plt.figure(figsize=(10,5))
plt.subplot(121)
plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.subplot(122)
plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), tol=1, random_state=42)
```



Differentiability

Note that the Lasso penality is non-differentiable at zero.

Gradient descent can still be used but with gradients replaced by sub-gradients when any $\theta_j = 0$.

7.4.3 Elastic Net regularization

Provides a mix of Tikhonov and Lasso regularization, controlled by mix ratio r:

$$R(\theta) = r \sum_{j=1}^{n} |\theta_j| + \frac{1-r}{2} \sum_{j=1}^{n} \theta_j^2.$$

- For $r = 0$, corresponds to Tikhonov regularization.
- For $r = 1$, corresponds to Lasso regularization.

7.4.4 Stopping early

Compute RMSE on validation set as train and stop when starts to increase.

```
from sklearn.linear_model import SGDRegressor
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 2 + X + 0.5 * X**2 + np.random.randn(m, 1)

X_train, X_val, y_train, y_val = train_test_split(X[:50], y[:50].ravel(), test_size=0.
↪5, random_state=10)

poly_scaler = Pipeline((
```

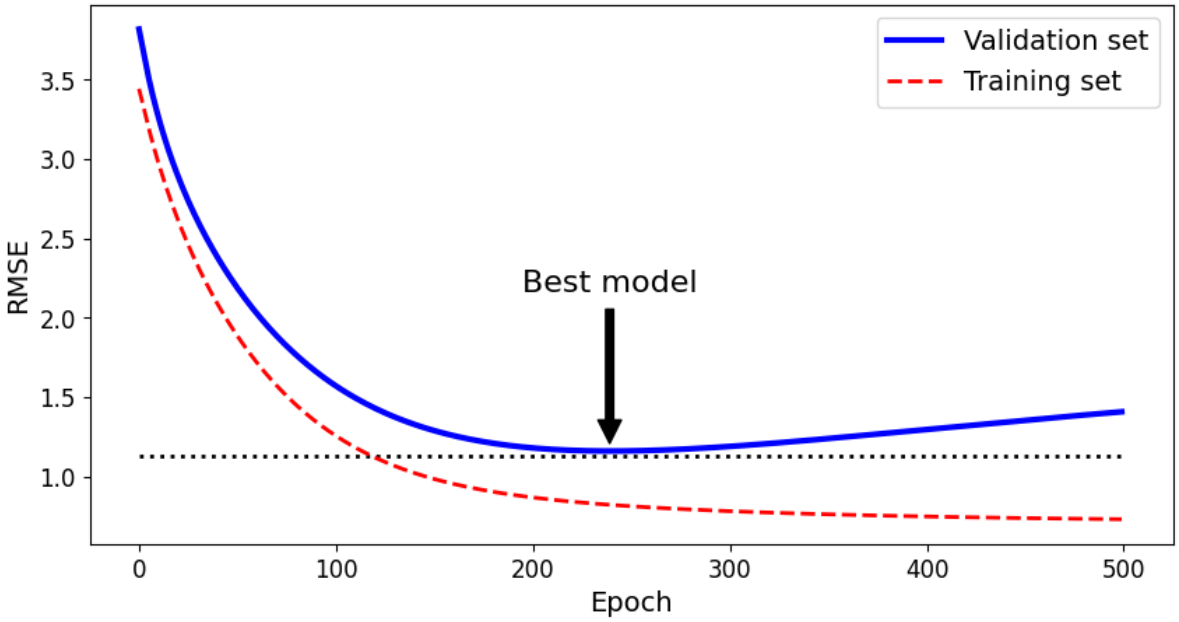
(continues on next page)

(continued from previous page)

```
        ("poly_features", PolynomialFeatures(degree=90, include_bias=False)),
        ("std_scaler", StandardScaler()),
    ))

X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(max_iter=1,
                       penalty=None,
                       eta0=0.0005,
                       warm_start=True,
                       learning_rate="constant",
                       random_state=42)
```

```
import warnings
warnings.filterwarnings(action='ignore')
n_epochs = 500
train_errors, val_errors = [], []
for epoch in range(n_epochs):
    sgd_reg.fit(X_train_poly_scaled, y_train)
    y_train_predict = sgd_reg.predict(X_train_poly_scaled)
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    train_errors.append(mean_squared_error(y_train_predict, y_train))
    val_errors.append(mean_squared_error(y_val_predict, y_val))

best_epoch = np.argmin(val_errors)
best_val_rmse = np.sqrt(val_errors[best_epoch])
```

```
plt.figure(figsize=(10,5))
plt.annotate('Best model',
             xy=(best_epoch, best_val_rmse),
             xytext=(best_epoch, best_val_rmse + 1),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.05),
             fontsize=16,
            )

best_val_rmse -= 0.03  # just to make the graph look better
plt.plot([0, n_epochs], [best_val_rmse, best_val_rmse], "k:", linewidth=2)
plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="Validation set")
plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="Training set")
plt.legend(loc="upper right", fontsize=14)
plt.xlabel("Epoch", fontsize=14)
plt.ylabel("RMSE", fontsize=14);
```



Note that for stochastic or mini-batch gradient descent the RMSE is noisy and may be difficult to know when reach minimum (can employ simple strategies to deal with that).

CHAPTER

EIGHT

LECTURE 8: LOGISTIC REGRESSION


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:19:54
```

8.1 Estimating probabilities

Estimate the probability of an instance belonging to a particular class.

Can adapt linear regression algorithm for this purpose to perform *logistic regression*.

8.1.1 Sigmoid function

Consider linear weighted sum of inputs $\theta^{\mathrm{T}}x$ again but then apply sigmoid function σ:

$$\hat{p} = h_\theta(x) = \sigma(\theta^{\mathrm{T}}x),$$

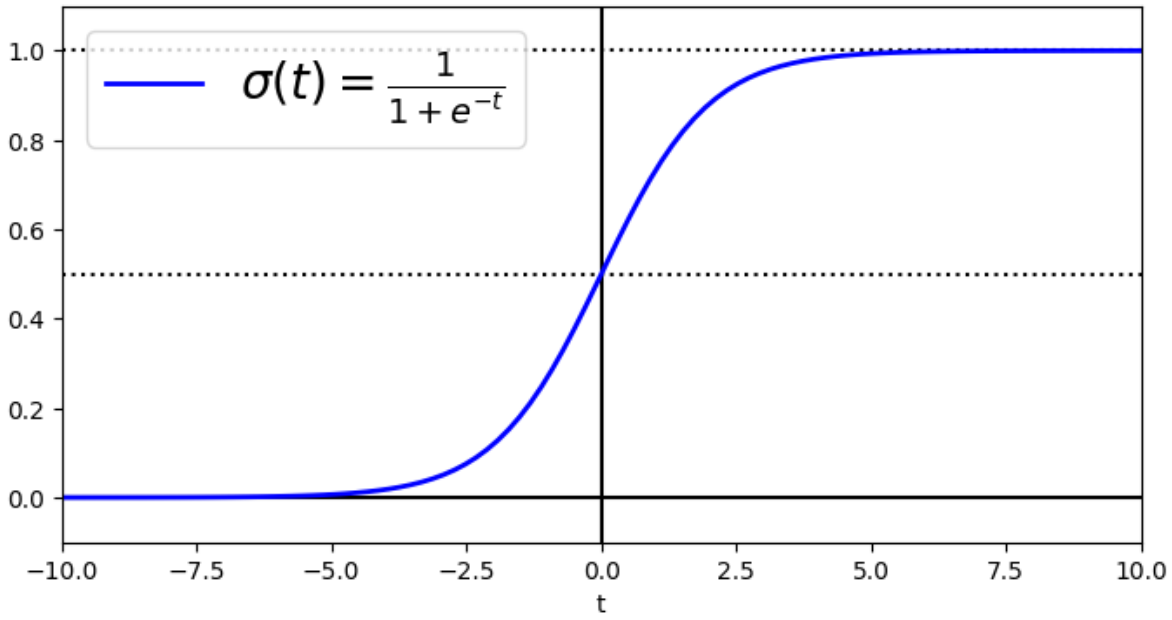
where

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

Plot the sigmoid function

```
import numpy as np
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

```
t = np.linspace(-10, 10, 100)
sig = 1 / (1 + np.exp(-t))
plt.figure(figsize=(8, 4))
plt.plot([-10, 10], [0, 0], "k-")
plt.plot([-10, 10], [0.5, 0.5], "k:")
plt.plot([-10, 10], [1, 1], "k:")
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlabel("t")
plt.legend(loc="upper left", fontsize=20)
plt.axis([-10, 10, -0.1, 1.1]);
```



8.1.2 Predictions

Can then make class predictions depending on whether the predicted probability $\hat{p}$ is greater than 0.5, i.e.

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

where we recall $\hat{p} = h_\theta(x) = \sigma(\theta^{\mathrm{T}} x)$ and $\sigma(t) = \frac{1}{1+\exp(-t)}$.

Note that $\sigma(t) < 0.5$ when $t < 0$ and $\sigma(t) \geq 0.5$ when $t \geq 0$.

That is, logistic regression predicts model 1 when $\theta^{\mathrm{T}} x$ is positive, and model 0 when it is negative.

The decision boundary is defined by $\theta^{\mathrm{T}} x = 0$.

8.2 Cost functions

Consider the cost function:

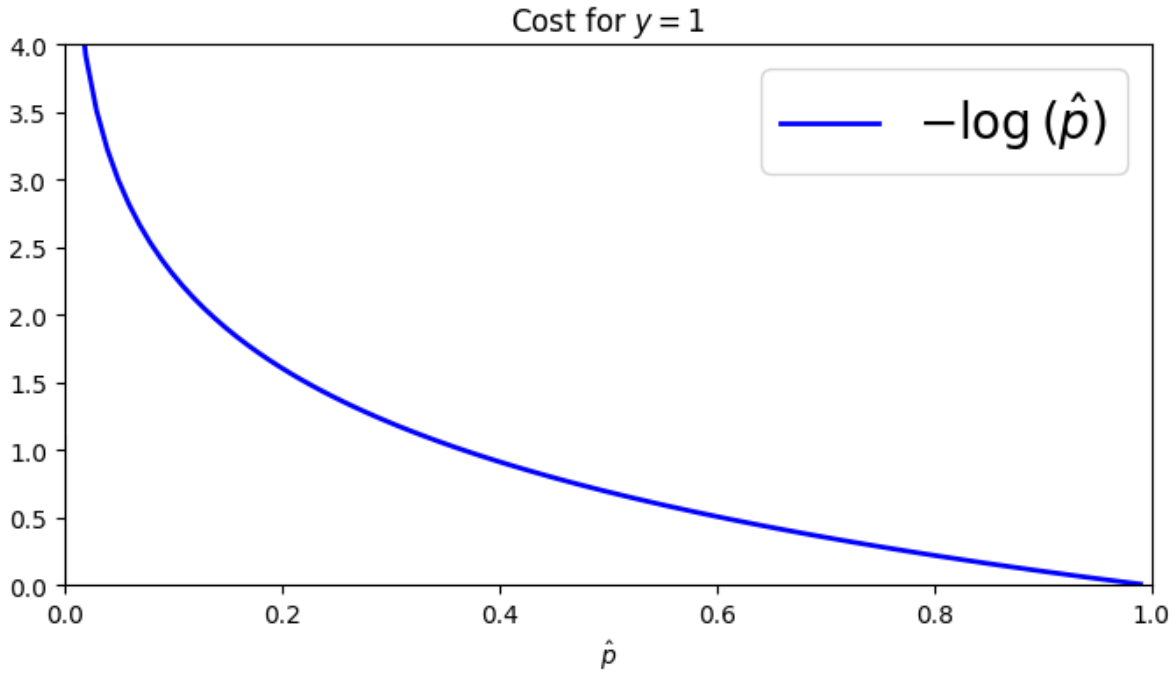
$$C(\theta) = \begin{cases} -\log(\hat{p}), \text{ if } y = 1 \\ -\log(1 - \hat{p}), \text{ if } y = 0 \end{cases}.$$

8.2.1 Exercise: Plot the cost function for $y = 1$ as a function of $\hat{p}$.

Consider the cost function:

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

```
ph = np.linspace(0.01, 0.99, 100)
cost_one = -np.log(ph)
plt.figure(figsize=(8, 4))
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(ph, cost_one, "b-", linewidth=2, label=r"$-\log{(\hat{p})}$")
plt.xlabel("$\hat{p}$")
plt.legend(loc="upper right", fontsize=20)
plt.axis([0, 1, 0, 4]);
plt.title('Cost for $y=1$');
```

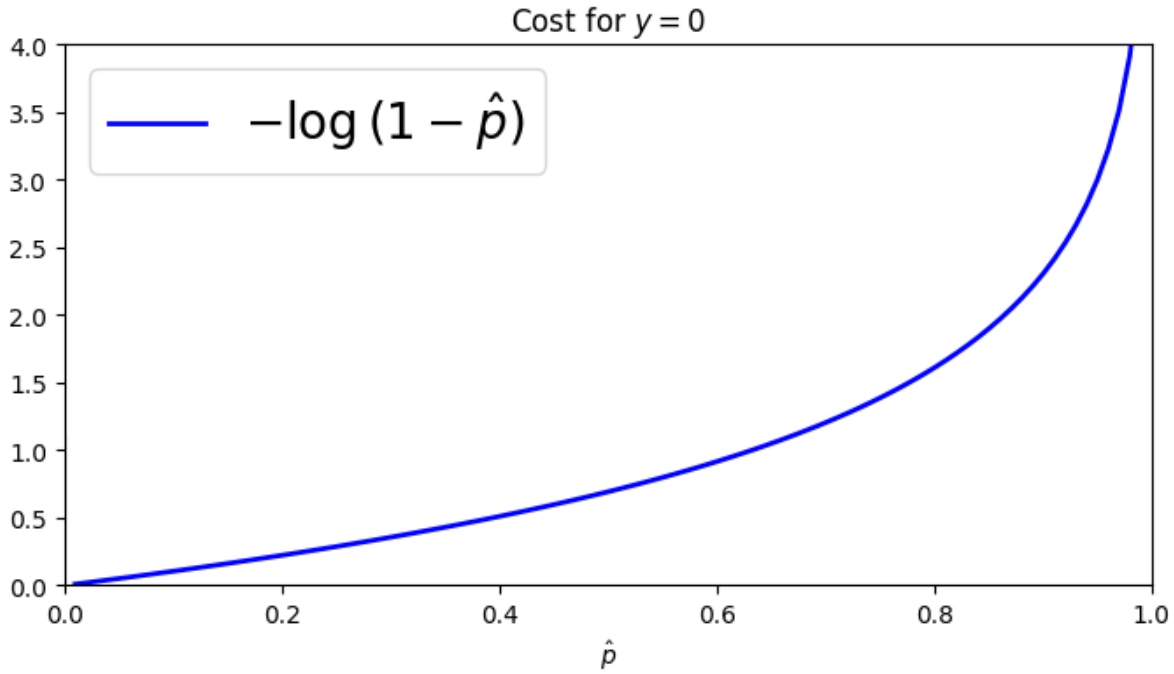


What can you say intuitively about the cost function at the edges of the domain?

- For $\hat{p} = 1$, $C(\hat{p}) = 0$.
- For $\hat{p} = 0$, $C(\hat{p}) \to \infty$.

8.2.2 Exercise: Plot the cost function for $y = 0$ as a function of $\hat{p}$.

```
cost_zero = -np.log(1-ph)
plt.figure(figsize=(8, 4))
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(ph, cost_zero, "b-", linewidth=2, label=r"$-\log{(1-\hat{p})}$")
plt.xlabel("$\hat{p}$")
plt.legend(loc="upper left", fontsize=20)
plt.axis([0, 1, 0, 4]);
plt.title('Cost for $y=0$');
```



What can you say intuitively about the cost function at the edges of the domain?

- For $\hat{p} = 0$, $C(\hat{p}) = 0$.
- For $\hat{p} = 1$, $C(\hat{p}) \to \infty$.

8.2.3 Log-loss function for logistic regression

Cost function can be written by the single expression

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

since $y^{(i)}$ is always 0 or 1 and we thus recover the separate cases considered above.

Aside: statistical interpretation

Interpret p as probability of target y:

$$\mathrm{P}(y \mid p) = p^y(1-p)^{1-y}$$

$$\log\mathrm{P}(y \mid p, x) = y\log(p) + (1-y)\log(1-p)$$

See MacKay [Chapter 41] for further details.

8.3 Minimising the cost function

No closed form solution like linear regression.

But since the cost function is convex guaranteed to find global minimum by gradient descent.

8.3.1 Derivative of the cost function

$$\begin{aligned}\frac{\partial C}{\partial \theta} &= \frac{1}{m}\sum_{i=1}^{m}\left[\sigma\left(\theta^{\mathrm{T}}x^{(i)}\right) - y^{(i)}\right]x^{(i)} \\ &= \frac{1}{m}X^{\mathrm{T}}\left[\sigma\left(X\theta\right) - y\right] \\ &= \frac{1}{m}X^{\mathrm{T}}\left[h_\theta\left(X\right) - y\right]\end{aligned}$$

8.3.2 Similarity with linear regression

Identical to linear regression (up to factor of 2 depending on conventions adopted) but with a different prediction function:

$$h_\theta(x) = \sigma(\theta^{\mathrm{T}}x),$$

instead of

$$h_\theta(x) = \theta^{\mathrm{T}}x.$$

8.4 Example of logistic regression

Consider Iris flower data again.

```
from sklearn import datasets
iris = datasets.load_iris()
list(iris.keys())
```

```
['data',
 'target',
 'frame',
 'target_names',
```

(continues on next page)

(continued from previous page)

```
 'DESCR',
 'feature_names',
 'filename',
 'data_module']
```

```
import numpy as np
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

8.4.1 Train model

Use petal width to classify whether Virginica or not.

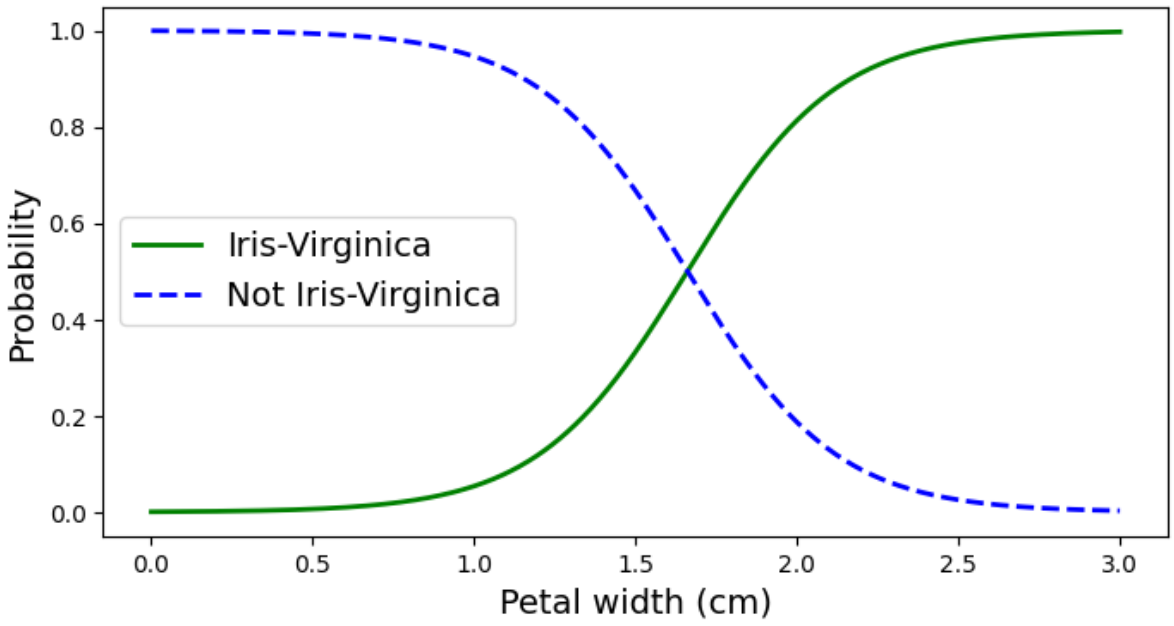
```
# Set up training data
X_1d = iris["data"][:, 3:]  # petal width
y = (iris["target"] == 2).astype(int)  # 1 if Iris-Virginica, else 0
```

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_1d, y);
```

Note that Scikit-Learn automatically adds ℓ_2 regularizer to cost function.

8.4.2 Prediction

```
X_1d_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_1d_proba = log_reg.predict_proba(X_1d_new)

plt.figure(figsize=(8,4))
plt.plot(X_1d_new, y_1d_proba[:, 1], "g-", linewidth=2, label="Iris-Virginica")
plt.plot(X_1d_new, y_1d_proba[:, 0], "b--", linewidth=2, label="Not Iris-Virginica")
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14);
```



8.4.3 Decision boundary

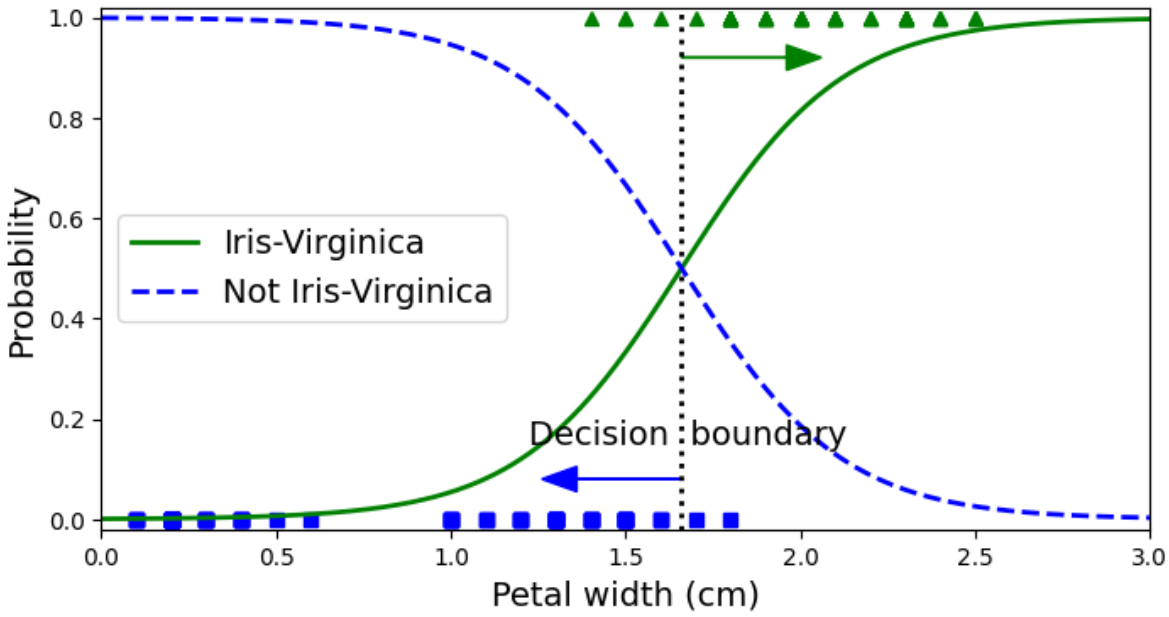
Recall the decision boundary is given by $\hat{p} = 0.5$ or, equivalently, $\theta^{\mathrm{T}} x = 0$.

```
decision_boundary = X_1d_new[y_1d_proba[:, 1] >= 0.5][0]
decision_boundary
```

```
array([1.66066066])
```

Updating plot with decision boundary and training data

```
plt.figure(figsize=(8, 4))
plt.plot(X_1d[y==0], y[y==0], "bs")
plt.plot(X_1d[y==1], y[y==1], "g^")
plt.plot([decision_boundary, decision_boundary], [-1, 2], "k:", linewidth=2)
plt.plot(X_1d_new, y_1d_proba[:, 1], "g-", linewidth=2, label="Iris-Virginica")
plt.plot(X_1d_new, y_1d_proba[:, 0], "b--", linewidth=2, label="Not Iris-Virginica")
plt.text(decision_boundary+0.02, 0.15, "Decision  boundary", fontsize=14, color="k",␣
 ↪ha="center")
plt.arrow(decision_boundary[0], 0.08, -0.3, 0, head_width=0.05, head_length=0.1, fc='b
 ↪', ec='b')
plt.arrow(decision_boundary[0], 0.92, 0.3, 0, head_width=0.05, head_length=0.1, fc='g
 ↪', ec='g')
plt.xlabel("Petal width (cm)", fontsize=14)
plt.ylabel("Probability", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 3, -0.02, 1.02]);
```



Predictions depend on what side of decision boundary fall.

```
log_reg.predict([[1.7], [1.5]])
```

```
array([1, 0])
```

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

8.4.4 Extending to two features

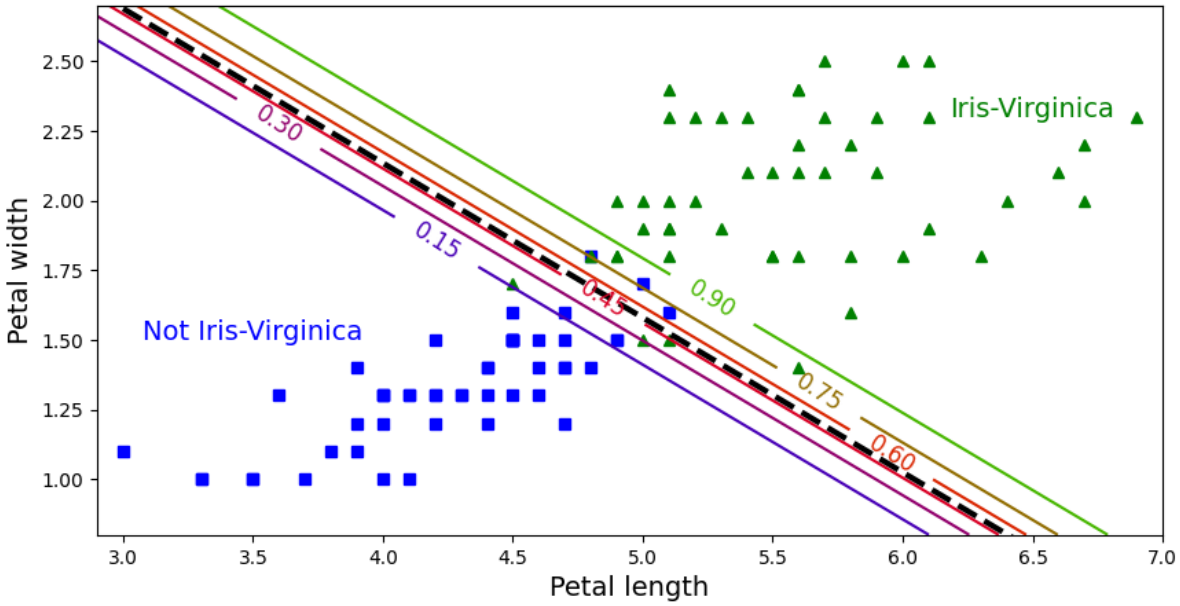
```
from sklearn.linear_model import LogisticRegression

X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(int)  # 1 if Iris-Virginica, else 0

C = 1000 # inverse regularization (smaller values correspond to stronger␣
 ↪regularization)
log_reg = LogisticRegression(C=C, random_state=42)
log_reg.fit(X, y)

x0, x1 = np.meshgrid(
        np.linspace(2.9, 7, 500).reshape(-1, 1),
        np.linspace(0.8, 2.7, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

y_proba = log_reg.predict_proba(X_new)
```

```
plt.figure(figsize=(10, 5))
plt.plot(X[y==0, 0], X[y==0, 1], "bs")
plt.plot(X[y==1, 0], X[y==1, 1], "g^")
```

(continues on next page)

(continued from previous page)

```
zz = y_proba[:, 1].reshape(x0.shape)
contour = plt.contour(x0, x1, zz, cmap=plt.cm.brg)

# Solve theta^T x = 0 to determine boundary
left_right = np.array([2.9, 7])
boundary = -(log_reg.coef_[0][0] * left_right + log_reg.intercept_[0]) / log_reg.coef_
 ↪[0][1]

plt.clabel(contour, inline=1, fontsize=12)
plt.plot(left_right, boundary, "k--", linewidth=3)
plt.text(3.5, 1.5, "Not Iris-Virginica", fontsize=14, color="b", ha="center")
plt.text(6.5, 2.3, "Iris-Virginica", fontsize=14, color="g", ha="center")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.axis([2.9, 7, 0.8, 2.7]);
```



Exercises: *You can now complete Exercise 2 in the exercises associated with this lecture.*

8.5 Softmax regression

Can generalise logistic regression to classify multiple classes.

8.5.1 Softmax score

Consider the softmax score function for class k:

$$s_k(x) = \left(\theta^{(k)}\right)^{\mathrm{T}} x.$$

Important note: each class k has its own score and set of parameters $\theta^{(k)}$, for K classes (i.e. $k = 1, \ldots, K$).

Define:

- Parameter matrix: $\Theta_{K \times n} = [\theta^{(1)},\ \theta^{(2)},\ \ldots,\ \theta^{(K)}]^{\mathrm{T}}$.

8.5.2 Softmax function

Predictions are then given by the softmax function $\sigma_k(s(x))$ for each k:

$$\hat{p}_k = \sigma_k(s(x)) = \frac{\exp\left(s_k(x)\right)}{\sum_{k'=1}^{K} \exp\left(s_{k'}(x)\right)}.$$

Normalised such that

- $\sum_k \hat{p}_k = 1$
- $0 \leq \hat{p}_k \leq 1$

8.5.3 Predictions

Can then make class predictions based on which class has the highest predicted probability, i.e.

$$\hat{y} = \arg\max_k \hat{p}_k = \arg\max_k s_k(x) = \arg\max_k \left(\theta^{(k)}\right)^{\mathrm{T}} x,$$

where we recall $\hat{p}_k = \sigma_k(s(x)) = \frac{\exp(s_k(x))}{\sum_{k'=1}^{K} \exp(s_{k'}(x))}$ and $s_k(x) = \left(\theta^{(k)}\right)^{\mathrm{T}} x$.

8.5.4 Cost function

Generalization of the logistic regression cost function is given by the *cross-entropy* (measure of similarity of probability distributions):

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

For the case $K = 2$, the cost functions reduces to the standard cost function for logistic regression.

8.5.5 Minimising the cost function

Can solve by gradient descent.

Derivative of cost function given by

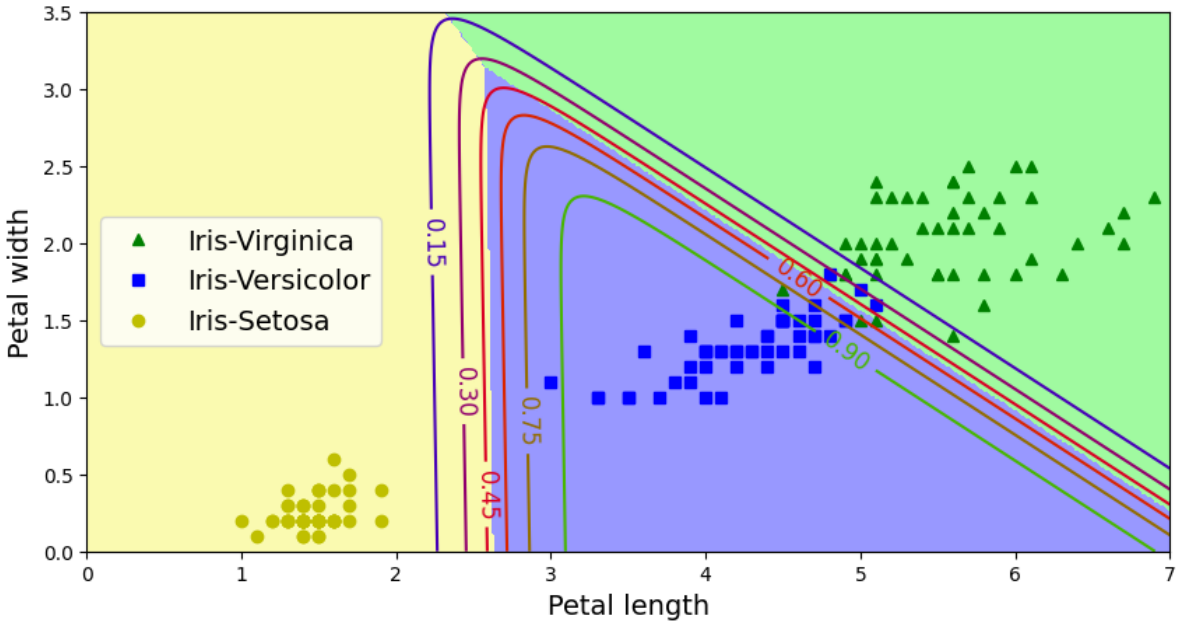
$$\frac{\partial C}{\partial \theta^{(k)}} = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)}\right) x^{(i)}.$$

8.5.6 Example of softmax regression

```
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"] # consider all three target classes

C = 10
softmax_reg = LogisticRegression(multi_class="multinomial", solver="lbfgs", C=C,␣
↪random_state=42)
softmax_reg.fit(X, y);
```

```
x0, x1 = np.meshgrid(
        np.linspace(0, 8, 500).reshape(-1, 1),
        np.linspace(0, 3.5, 200).reshape(-1, 1),
    )
X_new = np.c_[x0.ravel(), x1.ravel()]

y_proba = softmax_reg.predict_proba(X_new)
y_predict = softmax_reg.predict(X_new)

# Select contours to plot
# zz1 = y_proba[:, 0].reshape(x0.shape)
zz1 = y_proba[:, 1].reshape(x0.shape)
# zz1 = y_proba[:, 2].reshape(x0.shape)
zz = y_predict.reshape(x0.shape)
```

```
plt.figure(figsize=(10, 5))
plt.plot(X[y==2, 0], X[y==2, 1], "g^", label="Iris-Virginica")
plt.plot(X[y==1, 0], X[y==1, 1], "bs", label="Iris-Versicolor")
plt.plot(X[y==0, 0], X[y==0, 1], "yo", label="Iris-Setosa")

from matplotlib.colors import ListedColormap
custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])

plt.contourf(x0, x1, zz, cmap=custom_cmap)
contour = plt.contour(x0, x1, zz1, cmap=plt.cm.brg)
plt.clabel(contour, inline=1, fontsize=12)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="center left", fontsize=14)
plt.axis([0, 7, 0, 3.5]);
```


Iris-Virginica
Iris-Versicolor
Iris-Setosa
0.15
0.30
0.45
0.60
0.75
0.90
Petal width
Petal length
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
2
3
4
5
6
7

CHAPTER

NINE

LECTURE 9: SUPPORT VECTOR MACHINES (SVMS)


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:20:06
```

9.1 Large margin classification

9.1.1 Logistic regression with Hinge loss

Recall cost function for logistic regression, with ℓ_2 regularisation, is given by

$$C(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2,$$

where $\hat{p} = h_\theta(x) = \sigma(\theta^{\mathrm{T}}x)$ and $\sigma(t) = \frac{1}{1+\exp(-t)}$.

Recall, the bias θ_0 is not regularised, i.e. sum over j starts at 1.

Plot sigmoid

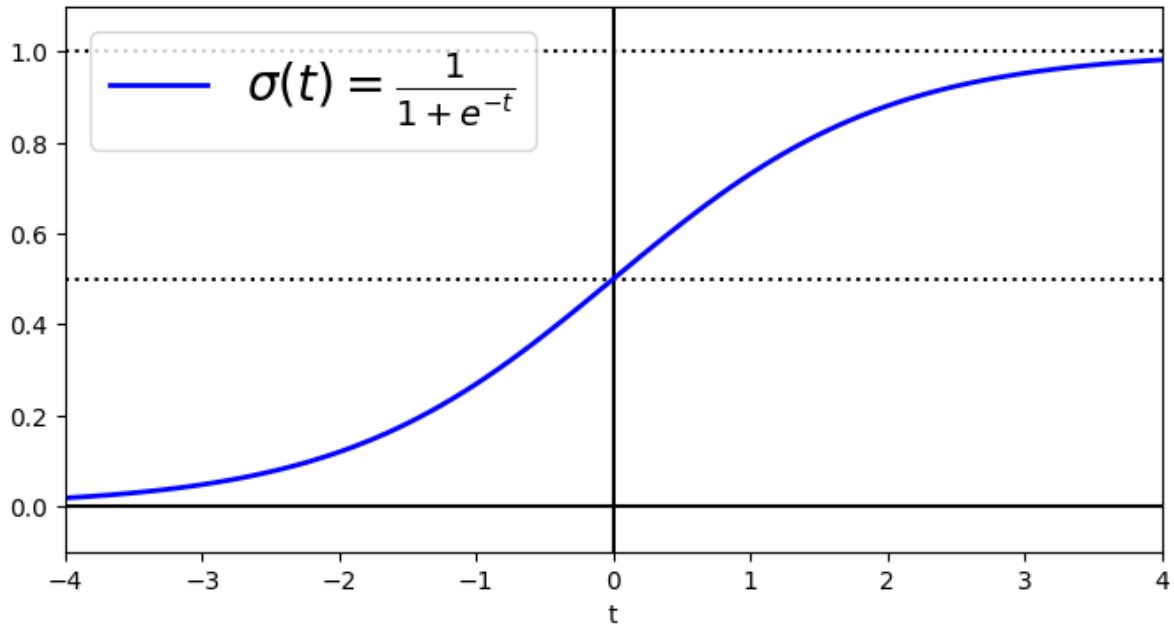
```
import numpy as np
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
```

```
t = np.linspace(-4, 4, 100)
sig = 1 / (1 + np.exp(-t))
plt.figure(figsize=(8, 4))
plt.plot([-4, 4], [0, 0], "k-")
plt.plot([-4, 4], [0.5, 0.5], "k:")
plt.plot([-4, 4], [1, 1], "k:")
```

(continues on next page)

(continued from previous page)

```
plt.plot([0, 0], [-1.1, 1.1], "k-")
plt.plot(t, sig, "b-", linewidth=2, label=r"$\sigma(t) = \frac{1}{1 + e^{-t}}$")
plt.xlabel("t")
plt.legend(loc="upper left", fontsize=20)
plt.axis([-4, 4, -0.1, 1.1]);
```



9.1.2 Hinge loss

Logistic regression cost function for reference:

$$C(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2.$$

Positive training instances

For $y^{(i)} = 1$, replace cost $-\log(\sigma(\theta^{\mathrm{T}}x))$ with *hinge loss* $\max(1-\theta^{\mathrm{T}}x, 0)$.

For training, we want not just $\theta^{\mathrm{T}}x \geq 0$ but $\theta^{\mathrm{T}}x \geq 1$.

Exercise:

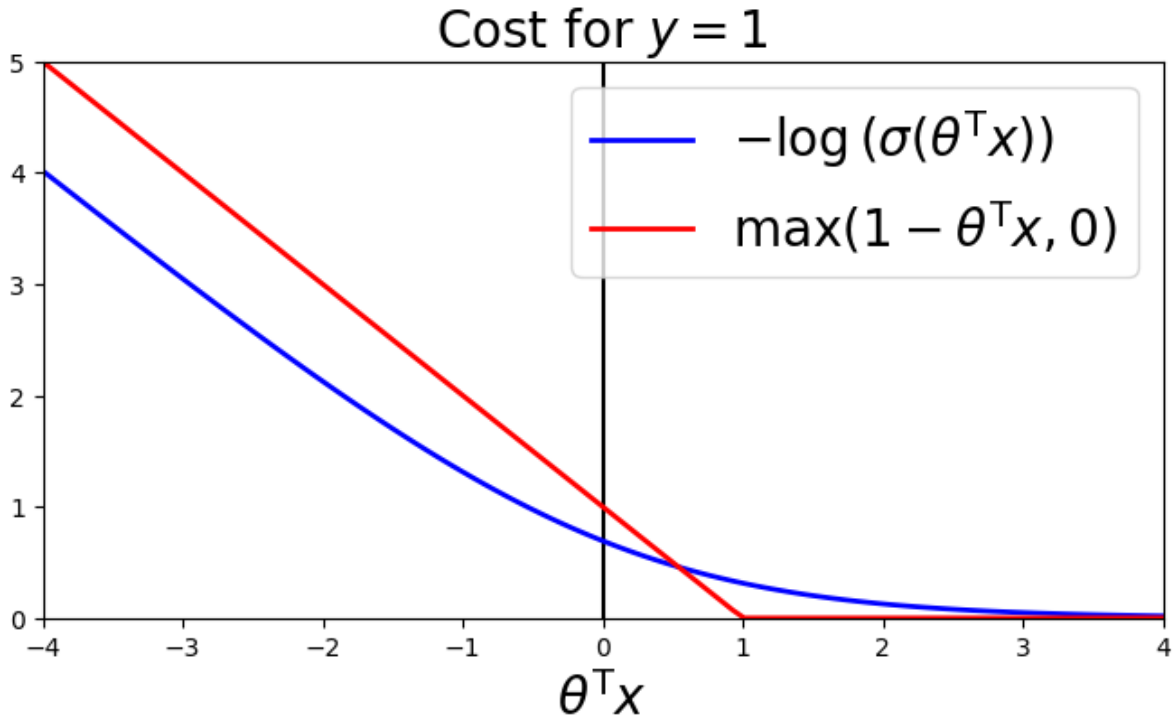
Plot cost $-\log(\sigma(\theta^{\mathrm{T}}x))$ and hinge loss $\max(1-\theta^{\mathrm{T}}x, 0)$.

Logistic regression cost function for reference:

$$C(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2.$$

```
t = np.linspace(-4, 4, 100)
sig = 1 / (1 + np.exp(-t))
cost_one = -np.log(sig)
cost_one_hinge = np.maximum(1 - t, np.zeros(t.size))
```

```
plt.figure(figsize=(8, 4))
plt.plot([0, 0], [0, 5], "k-")
plt.plot(t, cost_one, "b-", linewidth=2, label=r"$-\log{(\sigma(\theta^{\rm T} x))}$")
plt.xlabel(r"$\theta^{\rm T} x$", fontsize=20)
plt.axis([-4, 4, 0, 5]);
plt.title('Cost for $y=1$', fontsize=20)
plt.plot(t, cost_one_hinge, "r-", linewidth=2, label=r"$\max(1-\theta^{\rm T} x, 0)$")
plt.legend(loc="upper right", fontsize=20);
```



Negative training instances

For $y^{(i)} = 0$, replace $-\log(1 - \sigma(\theta^{\mathrm{T}} x))$ with *hinge loss* $\max(1 + \theta^{\mathrm{T}} x, 0)$.

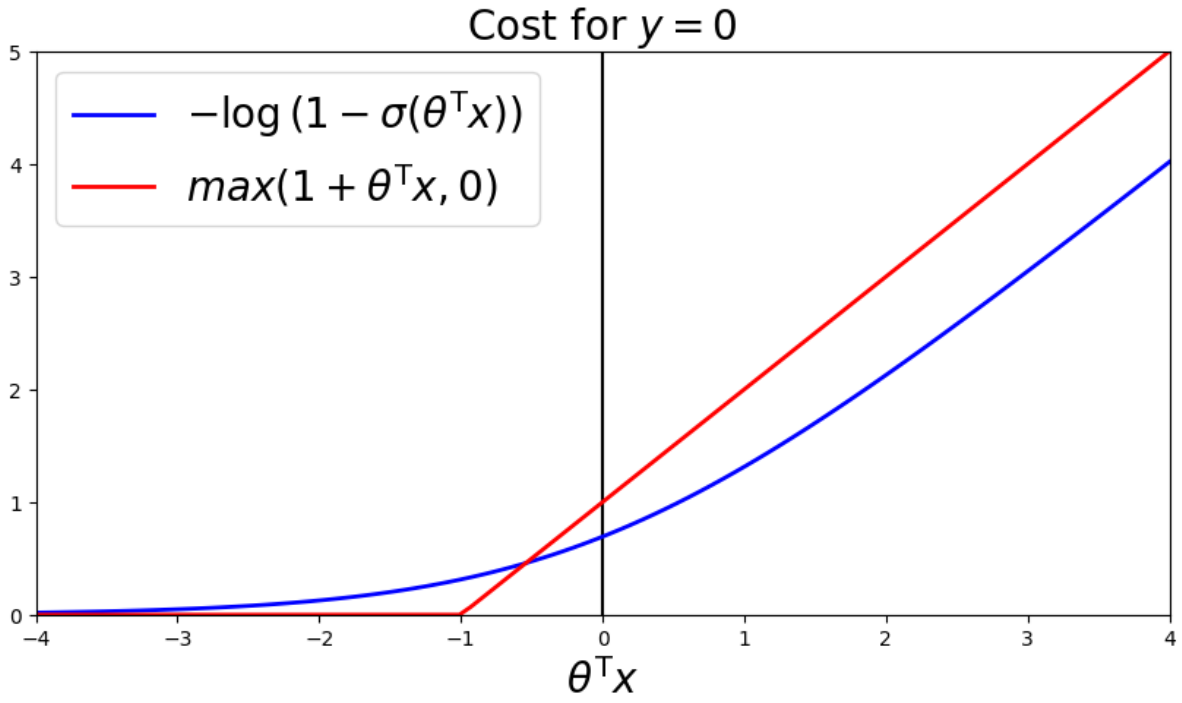
For training, we want not just $\theta^{\mathrm{T}} x < 0$ but $\theta^{\mathrm{T}} x \leq -1$.

Exercise:

Plot cost $-\log(1-\sigma(\theta^{\mathrm{T}}x))$ and hinge loss $\max(1+\theta^{\mathrm{T}}x, 0)$.

```
cost_zero = -np.log(1-sig)
cost_zero_hinge = np.maximum(1 + t, np.zeros(t.size))
```

```
plt.figure(figsize=(10, 5))
plt.plot([0, 0], [0, 5], "k-")
plt.plot(t, cost_zero, "b-", linewidth=2, label=r"$-\log{(1-\sigma(\theta^{\rm T} x))}
↪$")
plt.xlabel(r"$\theta^{\rm T} x$", fontsize=20)
plt.axis([-4, 4, 0, 5]);
plt.title('Cost for $y=0$', fontsize=20);
plt.plot(t, cost_zero_hinge, "r-", linewidth=2, label=r"$max(1+\theta^{\rm T} x, 0)$")
plt.legend(loc="upper left", fontsize=20);
```



9.1.3 Replace costs with hinge loss functions

Recall logistic regression with ℓ_2 regularisation cost function is given by

$$C(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(\hat{p}^{(i)}) + (1-y^{(i)})\log(1-\hat{p}^{(i)})\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2.$$

Replace costs for $y^{(i)}=1$ and $y^{(i)}=0$ with hinge losses given above:

$$\min_{\theta}\sum_{i=1}^{m}\left[y^{(i)}\max(1-\theta^{\mathrm{T}}x^{(i)}, 0) + (1-y^{(i)})\max(1+\theta^{\mathrm{T}}x^{(i)}, 0)\right] + \frac{\lambda}{2}\sum_{j=1}^{n}\theta_j^2.$$

Introduce $k^{(i)} = 1$ for positive instances ($y^{(i)} = 1$) and $k^{(i)} = -1$ for negative instances ($y^{(i)} = 0$):

$$\Rightarrow \min_{\theta} C \sum_{i=1}^{m} \max(1 - k^{(i)} \theta^{\mathrm{T}} x^{(i)}, 0) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

The convention is to weight the fidelity term by C rather than regularisation term by λ (thus C plays the role of $1/\lambda$).

- Large $C \rightarrow$ little regularisation
- Small $C \rightarrow$ greater regularisation

9.1.4 Constrained objective problem

So far we considered the *unconstrained* objective problem by adapting the logistic regression cost function:

$$\min_{\theta} C \sum_{i=1}^{m} \max(1 - k^{(i)} \theta^{\mathrm{T}} x^{(i)}, 0) + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

We can also consider the *constrained* objective problem:

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \quad \text{subject to} \quad k^{(i)} \theta^{\mathrm{T}} x^{(i)} \geq 1 \text{ for } i = 1, 2, \ldots, m$$

(Follows intuitively by considering large C.)

9.1.5 Intuition for large margin classification

Decision boundary defined by $\theta^{\mathrm{T}} x + b = 0$ (where here θ does *not* include bias and $b = \theta_0$; in notation above θ did include bias, i.e. θ_0 term).

Consequently, θ is orthogonal to decision boundary.

Recall constrained objective:

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \quad \text{subject to} \quad k^{(i)} \theta^{\mathrm{T}} x^{(i)} \geq 1 \text{ for } i = 1, 2, \ldots, m$$

Projection

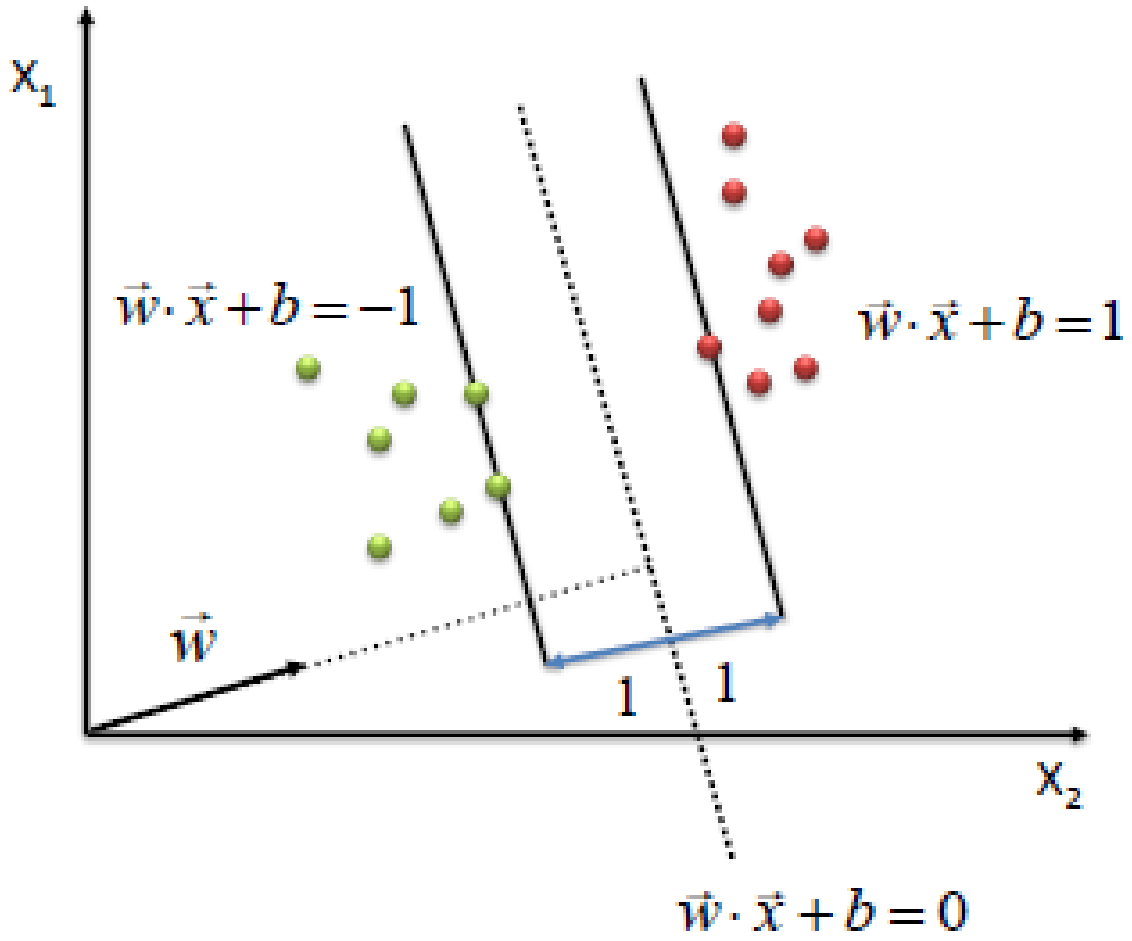
Note that the term $k^{(i)} \theta^{\mathrm{T}} x^{(i)}$ is related to the projection of $x^{(i)}$ onto θ:

$$k^{(i)} \theta^{\mathrm{T}} x^{(i)} = p^{(i)} \|\theta\|,$$

where $p^{(i)}$ is the projection of $x^{(i)}$ onto θ.

Attempting to minimise $\|\theta\|$, hence requires $p^{(i)}$ to be large $\Rightarrow$ large margin classification.

9.1.6 Graphical illustration



(Note difference notation used: $w = \theta$ without bias.)

Image source

9.2 Training SVMs

```
# Common imports
import os
import numpy as np
np.random.seed(42) # To make this notebook's output stable across runs

# To plot pretty figures
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.rcParams['axes.labelsize'] = 14
```

(continues on next page)

(continued from previous page)

```
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
```

9.2.1 Load data and train

```
from sklearn.svm import SVC
from sklearn import datasets

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = iris["target"]

setosa_or_versicolor = (y == 0) | (y == 1)
X = X[setosa_or_versicolor]
y = y[setosa_or_versicolor]

# SVM Classifier model
svm_clf = SVC(kernel="linear", C=np.finfo(float).max)
svm_clf.fit(X, y);
```

9.2.2 Plot decision boundaries

```
# Bad models
x0 = np.linspace(0, 5.5, 200)
#pred_1 = 5*x0 - 20
pred_2 = x0 - 1.8
pred_3 = 0.1 * x0 + 0.5
```

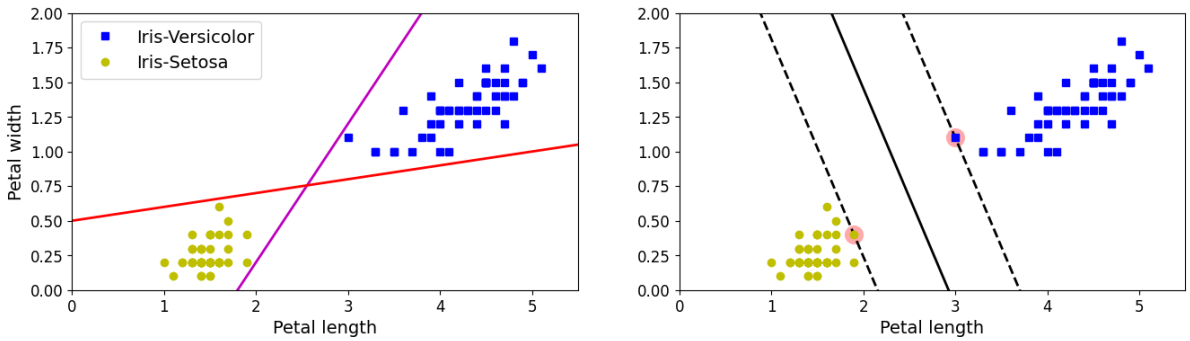
```
def plot_svc_decision_boundary(svm_clf, xmin, xmax):
    w = svm_clf.coef_[0]
    b = svm_clf.intercept_[0]

    # At the decision boundary, w0*x0 + w1*x1 + b = 0
    # => x1 = -w0/w1 * x0 - b/w1
    x0 = np.linspace(xmin, xmax, 200)
    decision_boundary = -w[0]/w[1] * x0 - b/w[1]

    # On the margin, w0*x0 + w1*x1 + b = +/- 1
    margin = 1/w[1]
    gutter_up = decision_boundary + margin
    gutter_down = decision_boundary - margin

    svs = svm_clf.support_vectors_
    plt.scatter(svs[:, 0], svs[:, 1], s=200, facecolors='#FFAAAA')
    plt.plot(x0, decision_boundary, "k-", linewidth=2)
    plt.plot(x0, gutter_up, "k--", linewidth=2)
    plt.plot(x0, gutter_down, "k--", linewidth=2)
```

```
plt.figure(figsize=(16,4))
```

(continues on next page)

(continued from previous page)

```
plt.subplot(121)
#plt.plot(x0, pred_1, "g--", linewidth=2)
plt.plot(x0, pred_2, "m-", linewidth=2)
plt.plot(x0, pred_3, "r-", linewidth=2)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris-Versicolor")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris-Setosa")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.axis([0, 5.5, 0, 2])

plt.subplot(122)
plot_svc_decision_boundary(svm_clf, 0, 5.5)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo")
plt.xlabel("Petal length", fontsize=14)
plt.axis([0, 5.5, 0, 2]);
```



Adding training instances outside the margin will not alter the decision boundary.

Boundary is defined by *support vectors* that are located on the edge of the margin.

Exercise: what value would you guess for the weights θ (excluding the bias)?

From plot appears $w \sim [1, 1]^{\mathrm{T}}$ (or slightly more accurately $w \sim [1.1, 0.9]^{\mathrm{T}}$).

Let's check:

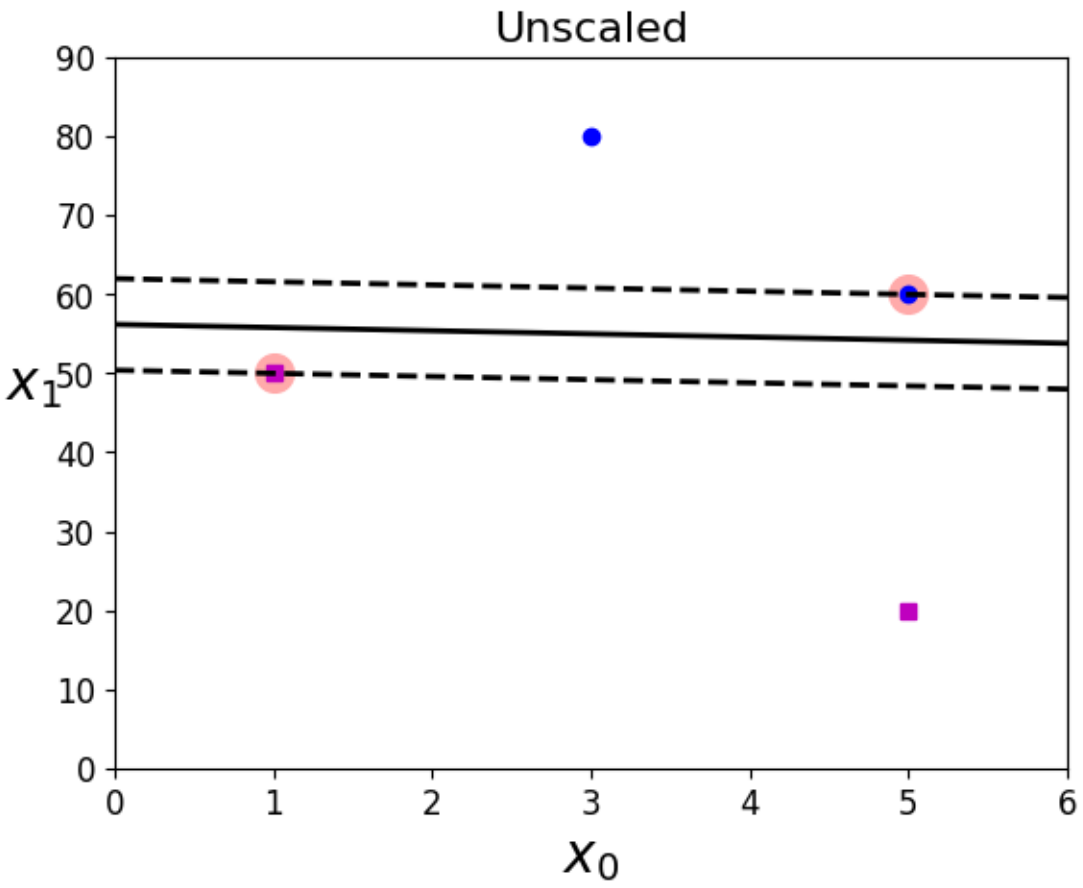
```
svm_clf.coef_[0]
```

```
array([1.29411744, 0.82352928])
```

9.2.3 Feature scaling

SVMs are sensitive to feature scales, hence feature scaling is important if features not already of similar scale.

```
Xs = np.array([[1, 50], [5, 20], [3, 80], [5, 60]]).astype(np.float64)
ys = np.array([0, 0, 1, 1])
svm_clf = SVC(kernel="linear", C=100)
svm_clf.fit(Xs, ys)

#plt.figure(figsize=(16,4))
#plt.subplot(121)
plt.plot(Xs[:, 0][ys==1], Xs[:, 1][ys==1], "bo")
plt.plot(Xs[:, 0][ys==0], Xs[:, 1][ys==0], "ms")
plot_svc_decision_boundary(svm_clf, 0, 6)
plt.xlabel("$x_0$", fontsize=20)
plt.ylabel("$x_1$  ", fontsize=20, rotation=0)
plt.title("Unscaled", fontsize=16)
plt.axis([0, 6, 0, 90]);
```

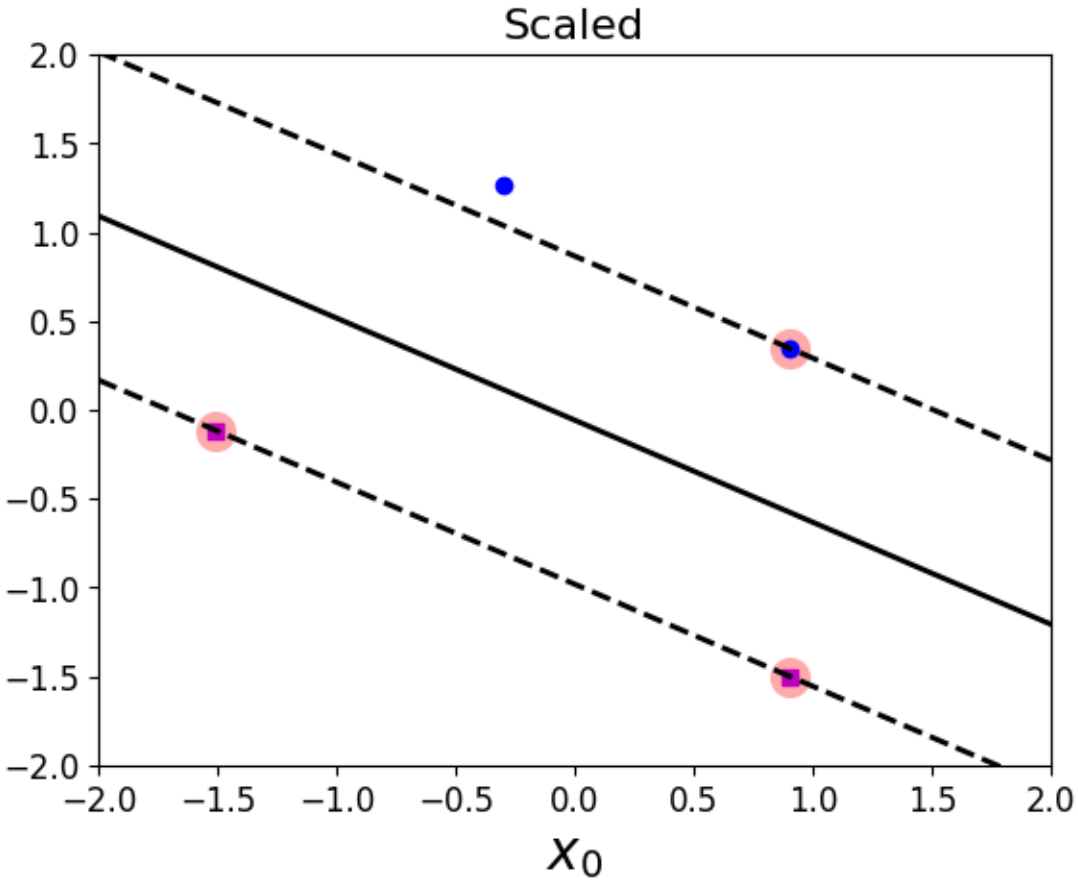



```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(Xs)
svm_clf.fit(X_scaled, ys)

#plt.subplot(122)
```

(continues on next page)

(continued from previous page)

```
plt.plot(X_scaled[:, 0][ys==1], X_scaled[:, 1][ys==1], "bo")
plt.plot(X_scaled[:, 0][ys==0], X_scaled[:, 1][ys==0], "ms")
plot_svc_decision_boundary(svm_clf, -2, 2)
plt.xlabel("$x_0$", fontsize=20)
plt.title("Scaled", fontsize=16)
plt.axis([-2, 2, -2, 2]);
```



9.2.4 Hard margin classification

Hard margin classification corresponds to strictly imposing all training instances correctly classfied.

Problems with hard margin classification

- Can fail if data not linearly separable.
- Sensitive to outliers.

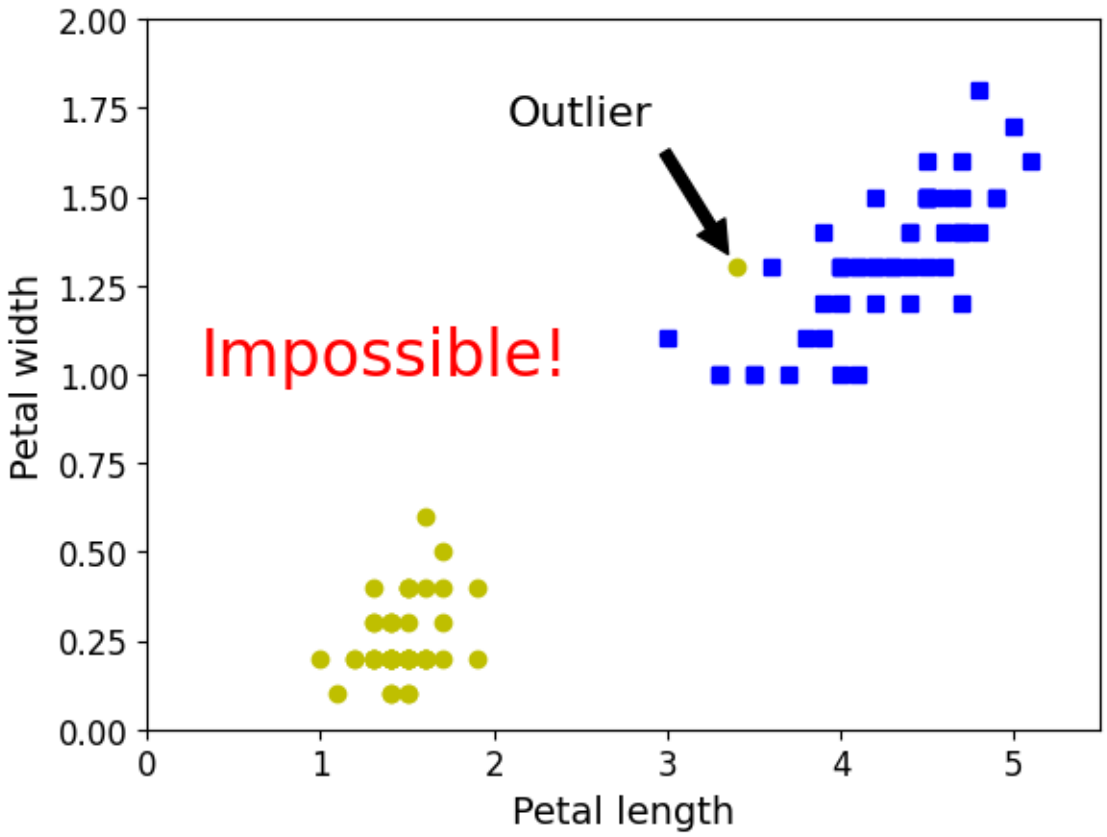
```
X_outliers = np.array([[3.4, 1.3], [3.2, 0.8]])
y_outliers = np.array([0, 0])
Xo1 = np.concatenate([X, X_outliers[:1]], axis=0)
yo1 = np.concatenate([y, y_outliers[:1]], axis=0)
```

(continues on next page)

(continued from previous page)

```
Xo2 = np.concatenate([X, X_outliers[1:]], axis=0)
yo2 = np.concatenate([y, y_outliers[1:]], axis=0)

svm_clf2 = SVC(kernel="linear", C=10**9)
svm_clf2.fit(Xo2, yo2);
```

```
# plt.figure(figsize=(16,5))
# plt.subplot(121)
plt.plot(Xo1[:, 0][yo1==1], Xo1[:, 1][yo1==1], "bs")
plt.plot(Xo1[:, 0][yo1==0], Xo1[:, 1][yo1==0], "yo")
plt.text(0.3, 1.0, "Impossible!", fontsize=24, color="red")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.annotate("Outlier",
             xy=(X_outliers[0][0], X_outliers[0][1]),
             xytext=(2.5, 1.7),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=16,
            )
plt.axis([0, 5.5, 0, 2]);
```

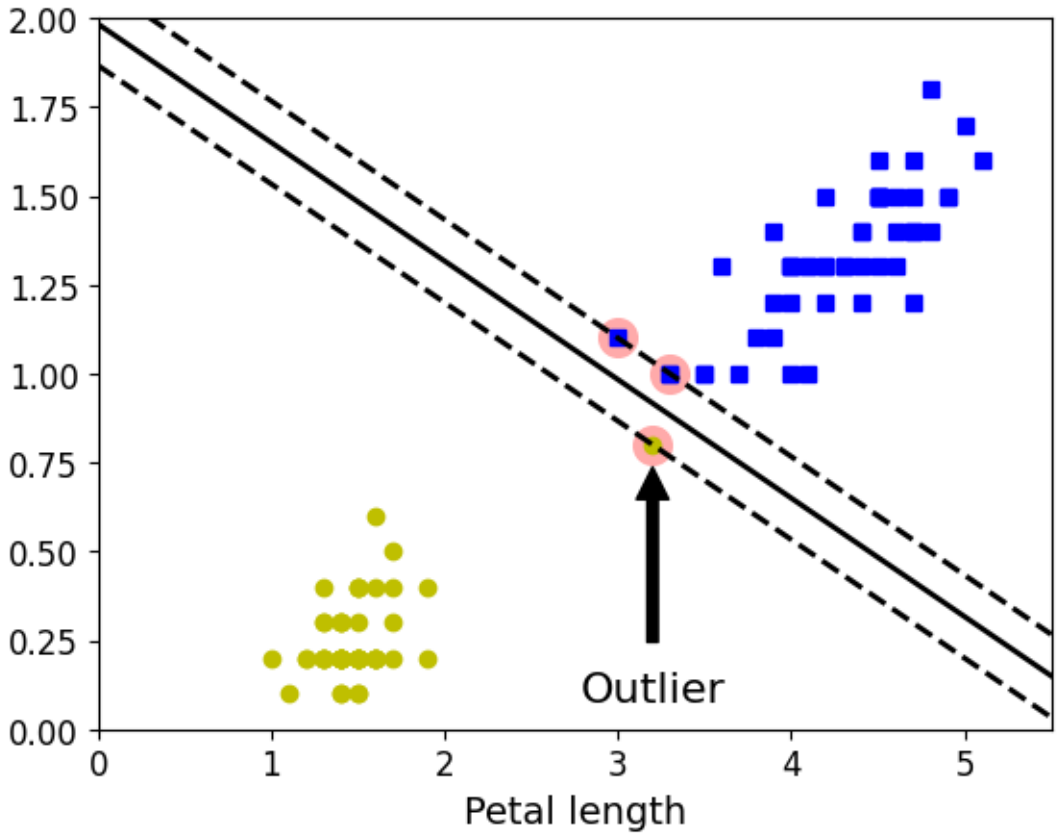



```
# plt.subplot(122)
plt.plot(Xo2[:, 0][yo2==1], Xo2[:, 1][yo2==1], "bs")
plt.plot(Xo2[:, 0][yo2==0], Xo2[:, 1][yo2==0], "yo")
```

(continues on next page)

(continued from previous page)

```
plot_svc_decision_boundary(svm_clf2, 0, 5.5)
plt.xlabel("Petal length", fontsize=14)
plt.annotate("Outlier",
             xy=(X_outliers[1][0], X_outliers[1][1]),
             xytext=(3.2, 0.08),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=16,
            )
plt.axis([0, 5.5, 0, 2]);
```



9.2.5 Soft margin classification

Allow some margin violations by varying hyperparameter C.

Recall, large C corresponds to small regularisation and thus few margin violations. Small C corresponds to greater regularisation and thus more margin violations.

Load data

```
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
```

Train SVMs

```
# disable convergence warning from early stopping
from warnings import simplefilter
from sklearn.exceptions import ConvergenceWarning
simplefilter("ignore", category=ConvergenceWarning)
```

```
scaler = StandardScaler()
svm_clf1 = LinearSVC(C=1, loss="hinge", random_state=42)
svm_clf2 = LinearSVC(C=100, loss="hinge", random_state=42)

scaled_svm_clf1 = Pipeline((
        ("scaler", scaler),
        ("linear_svc", svm_clf1),
    ))
scaled_svm_clf2 = Pipeline((
        ("scaler", scaler),
        ("linear_svc", svm_clf2),
    ))

scaled_svm_clf1.fit(X, y);
scaled_svm_clf2.fit(X, y);
```

Compute support vectors

```
# Convert to unscaled parameters
b1 = svm_clf1.decision_function([-scaler.mean_ / scaler.scale_])
b2 = svm_clf2.decision_function([-scaler.mean_ / scaler.scale_])
w1 = svm_clf1.coef_[0] / scaler.scale_
w2 = svm_clf2.coef_[0] / scaler.scale_
svm_clf1.intercept_ = np.array([b1])
svm_clf2.intercept_ = np.array([b2])
svm_clf1.coef_ = np.array([w1])
svm_clf2.coef_ = np.array([w2])

# Find support vectors (LinearSVC does not do this automatically)
t = y * 2 - 1 # t = +/-1
support_vectors_idx1 = (t * (X.dot(w1) + b1) < 1).ravel()
support_vectors_idx2 = (t * (X.dot(w2) + b2) < 1).ravel()
```

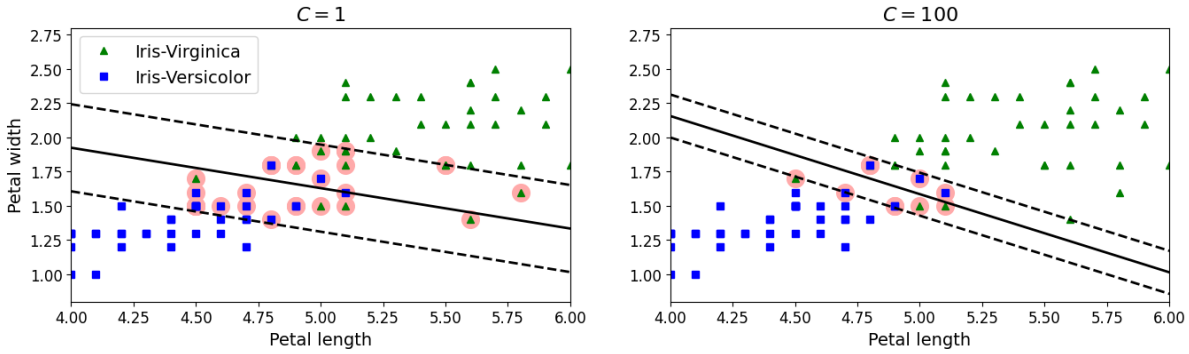
(continues on next page)

(continued from previous page)

```
svm_clf1.support_vectors_ = X[support_vectors_idx1]
svm_clf2.support_vectors_ = X[support_vectors_idx2]
```

Plot

```
plt.figure(figsize=(16,4))
plt.subplot(121)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^", label="Iris-Virginica")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs", label="Iris-Versicolor")
plot_svc_decision_boundary(svm_clf1, 4, 6)
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=14)
plt.title("$C = {}$".format(svm_clf1.C), fontsize=16)
plt.axis([4, 6, 0.8, 2.8])

plt.subplot(122)
plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
plot_svc_decision_boundary(svm_clf2, 4, 6)
plt.xlabel("Petal length", fontsize=14)
plt.title("$C = {}$".format(svm_clf2.C), fontsize=16)
plt.axis([4, 6, 0.8, 2.8]);
```



Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

9.3 Non-linear classification with polynomial features

So far we have considered linear classification only.

Most data-sets are not linearly separable.

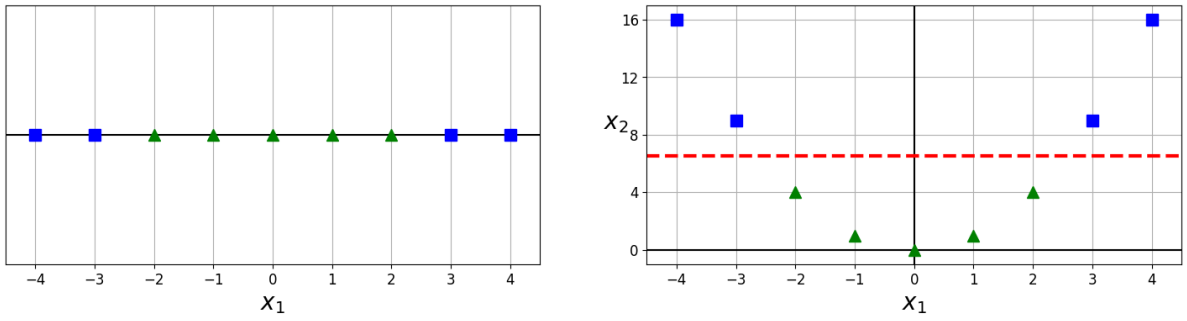
9.3.1 1D example

Consider 1D feature space with x_1, which is clearly not linearly separable.

However, if augment feature space with $x_2 = (x_1)^2$ we see that the resulting 2D feature space is linearly separable.

```
X1D = np.linspace(-4, 4, 9).reshape(-1, 1)
X2D = np.c_[X1D, X1D**2]
y = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])
```

```
plt.figure(figsize=(16, 4))
plt.subplot(121)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.plot(X1D[:, 0][y==0], np.zeros(4), "bs", markersize=10)
plt.plot(X1D[:, 0][y==1], np.zeros(5), "g^", markersize=10)
plt.gca().get_yaxis().set_ticks([])
plt.xlabel(r"$x_1$", fontsize=20)
plt.axis([-4.5, 4.5, -0.2, 0.2])

plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(X2D[:, 0][y==0], X2D[:, 1][y==0], "bs", markersize=10)
plt.plot(X2D[:, 0][y==1], X2D[:, 1][y==1], "g^", markersize=10)
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"$x_2$", fontsize=20, rotation=0)
plt.gca().get_yaxis().set_ticks([0, 4, 8, 12, 16])
plt.plot([-4.5, 4.5], [6.5, 6.5], "r--", linewidth=3)
plt.axis([-4.5, 4.5, -1, 17])

plt.subplots_adjust(right=1)
```



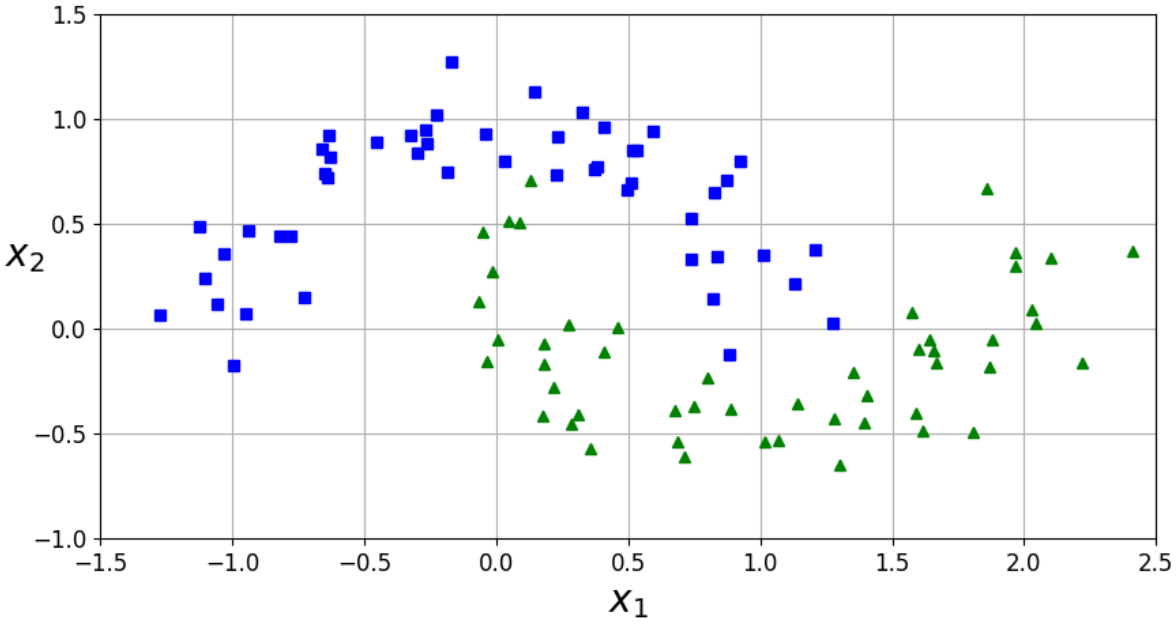
9.3.2 2D example

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=100, noise=0.15, random_state=42)

def plot_dataset(X, y, axes):
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "bs")
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "g^")
    plt.axis(axes)
```

(continues on next page)

(continued from previous page)

```
    plt.grid(True, which='both')
    plt.xlabel(r"$x_1$", fontsize=20)
    plt.ylabel(r"$x_2$", fontsize=20, rotation=0)

plt.figure(figsize=(10, 5))
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
plt.show()
```



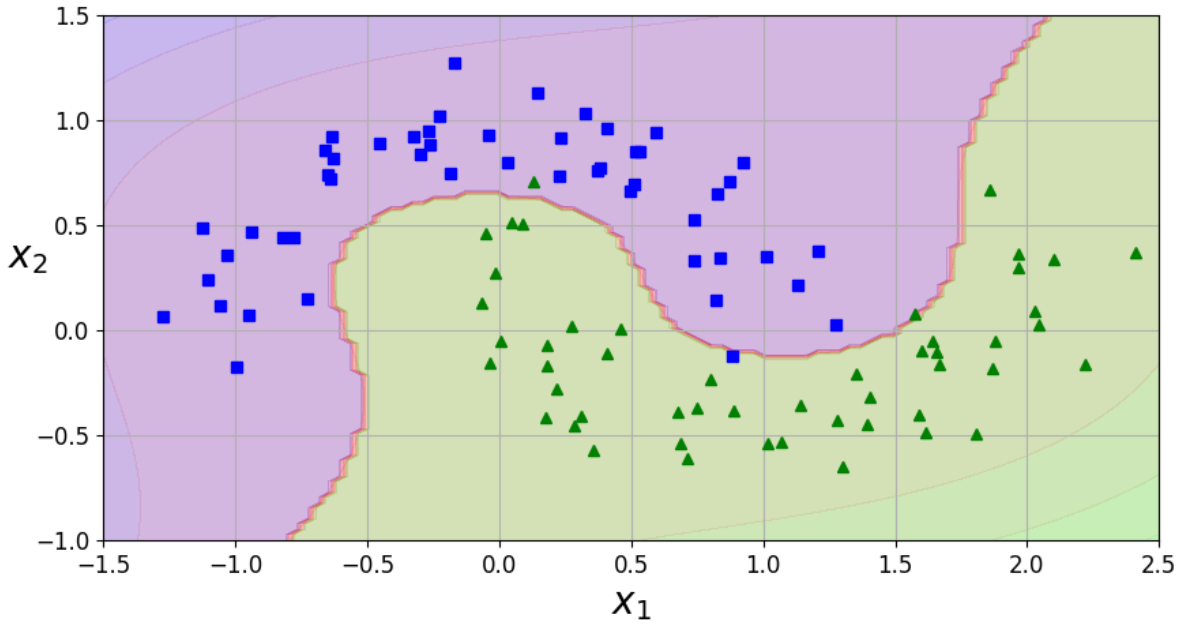

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline((
        ("poly_features", PolynomialFeatures(degree=3)),
        ("scaler", StandardScaler()),
        ("svm_clf", LinearSVC(C=10, loss="hinge", random_state=42))
    ))

polynomial_svm_clf.fit(X, y);
```

```
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plt.figure(figsize=(10,5))
plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
```

(continues on next page)

(continued from previous page)

```
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
```



Adding polynomial features leads to a combinatorial increase in the dimensionality of the feature space and thus is computationally costly.

There are much better ways to perform non-linear classification with SVMs, where computational tricks can be exploited…

9.4 Non-linear classification with kernels

9.4.1 Similarity features and landmarks

Compute new features based on proximity to landmarks.

Consider landmarks $l^{(1)}$, $l^{(2)}$, ...

Then for each training instance x, compute features $f_j = \text{sim}(x, l^{(j)})$, where sim(.,.) defines a *similarity* function.

9.4.2 Similarity functions

The Gaussian radial basis function (RBF) is a common *similarity function*:

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

where γ controls the width of the kernel.

9.4.3 1D example (from above)

Consider landmarks at $x_1 = -2$ and $x_1 = 1$.

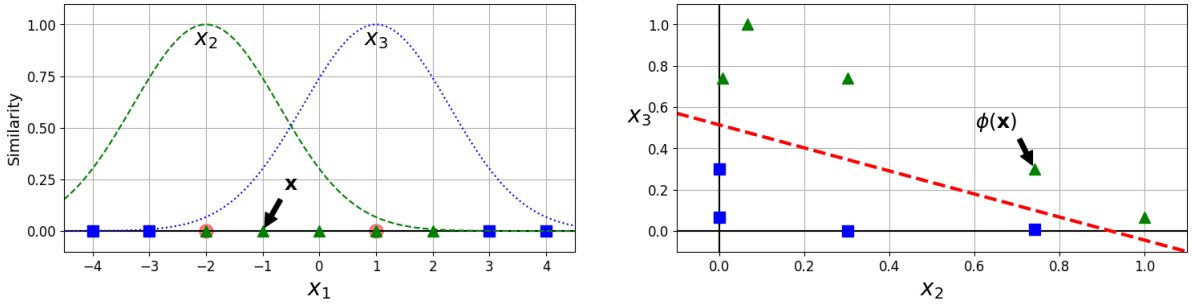
```
def gaussian_rbf(x, landmark, gamma):
    return np.exp(-gamma * np.linalg.norm(x - landmark, axis=1)**2)

gamma = 0.3

x1s = np.linspace(-4.5, 4.5, 200).reshape(-1, 1)
x2s = gaussian_rbf(x1s, -2, gamma)
x3s = gaussian_rbf(x1s, 1, gamma)

XK = np.c_[gaussian_rbf(X1D, -2, gamma), gaussian_rbf(X1D, 1, gamma)]
yk = np.array([0, 0, 1, 1, 1, 1, 1, 0, 0])
```

```
plt.figure(figsize=(16, 4))
plt.subplot(121)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.scatter(x=[-2, 1], y=[0, 0], s=150, alpha=0.5, c="red")
plt.plot(X1D[:, 0][yk==0], np.zeros(4), "bs", markersize=10)
plt.plot(X1D[:, 0][yk==1], np.zeros(5), "g^", markersize=10)
plt.plot(x1s, x2s, "g--")
plt.plot(x1s, x3s, "b:")
plt.gca().get_yaxis().set_ticks([0, 0.25, 0.5, 0.75, 1])
plt.xlabel(r"$x_1$", fontsize=20)
plt.ylabel(r"Similarity", fontsize=14)
plt.annotate(r'$\mathbf{x}$',
             xy=(X1D[3, 0], 0),
             xytext=(-0.5, 0.20),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=18,
            )
plt.text(-2, 0.9, "$x_2$", ha="center", fontsize=20)
plt.text(1, 0.9, "$x_3$", ha="center", fontsize=20)
plt.axis([-4.5, 4.5, -0.1, 1.1])

plt.subplot(122)
plt.grid(True, which='both')
plt.axhline(y=0, color='k')
plt.axvline(x=0, color='k')
plt.plot(XK[:, 0][yk==0], XK[:, 1][yk==0], "bs", markersize=10)
plt.plot(XK[:, 0][yk==1], XK[:, 1][yk==1], "g^", markersize=10)
plt.xlabel(r"$x_2$", fontsize=20)
plt.ylabel(r"$x_3$  ", fontsize=20, rotation=0)
plt.annotate(r'$\phi\left(\mathbf{x}\right)$',
             xy=(XK[3, 0], XK[3, 1]),
             xytext=(0.65, 0.50),
             ha="center",
             arrowprops=dict(facecolor='black', shrink=0.1),
             fontsize=18,
            )
plt.plot([-0.1, 1.1], [0.57, -0.1], "r--", linewidth=3)
plt.axis([-0.1, 1.1, -0.1, 1.1])
```

(continues on next page)

(continued from previous page)

```
plt.subplots_adjust(right=1)
```



Training data *are* linearly separable in new feature space.

9.4.4 Exercise: What are new feature values corresponding to the instance at $x_1 = -1$ ($\gamma = 0.3$)?

$x_2 = \exp(-0.3 \times (1)^2) = 0.74$

$x_3 = \exp(-0.3 \times (2)^2) = 0.30$

Let's check:

```
x1_example = X1D[3, 0]
for landmark in (-2, 1):
    k = gaussian_rbf(np.array([[x1_example]]), np.array([[landmark]]), gamma)
    print("Phi({}, {}) = {}".format(x1_example, landmark, k))
```

```
Phi(-1.0, -2) = [0.74081822]
Phi(-1.0, 1) = [0.30119421]
```

9.4.5 How set landmarks?

Common approach is to set a landmark at the location of each instance in the training data-set.

9.4.6 Kernel trick

For SVMs it is not necessary to actually compute new features for each kernel.

Instead, can be done implicitly, providing *considerable* computational saving.

SVM are well suited for small to medium sized data-sets, with complex structure.

The kernel trick is based on Mercer's theorem.

Mercer's theorem

For (feature) mapping function $\phi(x)$, the inner product of two transformed vectors can be computed implicitly by the evaluation of the kernel function K by $K(x, z) = \langle \phi(x), \phi(z) \rangle$.

There is therefore no need to explicitly compute $\phi(x)$.

Moreoever, it is not necessary to even know the explicit form of the (feature) mapping function $\phi(x)$.

(We will not cover the kernel trick and Mercer's theorem in any further detail.)

9.4.7 Common kernels

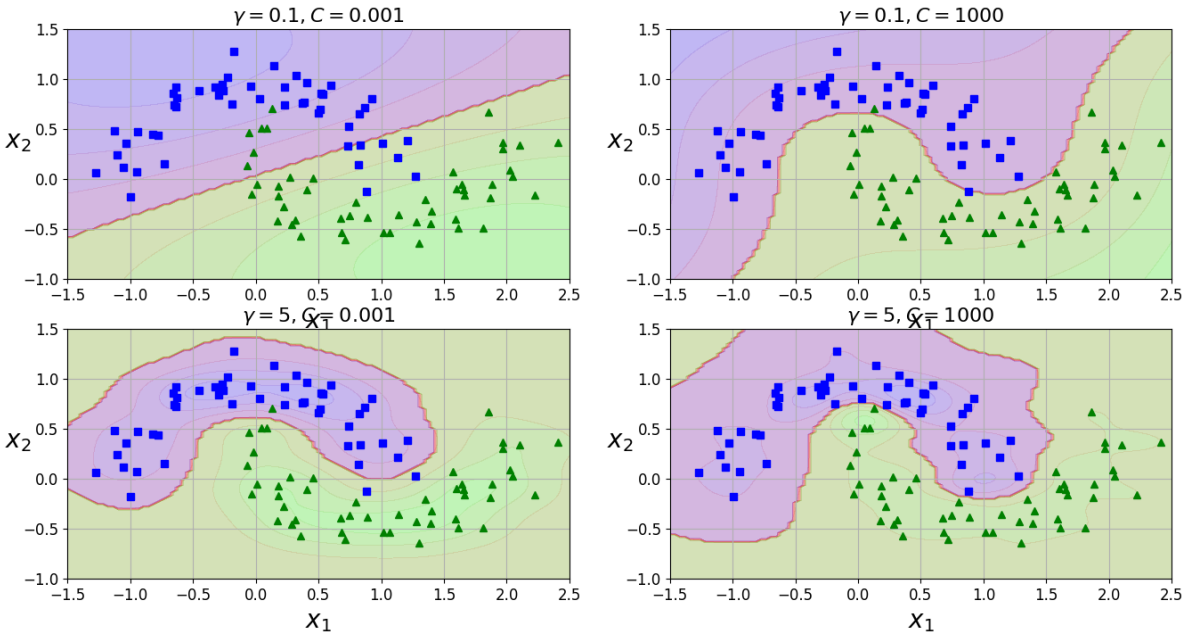
Some common similarity function, or *kernel*, include the following.

1. Gaussian radial basis function (RBF): $K_\gamma(x, l) = exp(-\gamma \|x - l\|^2)$, where γ controls the width of the kernel.
2. Polynomial kernel: $K_{c,d}(x, l) = (x^{\mathrm{T}} l + c)^d$, for constant offset c and degree d.
3. Linear kernel, i.e. linear SVM: $K(x, l) = x^{\mathrm{T}} l$.

9.4.8 2D example (from above)

```
from sklearn.svm import SVC

gamma1, gamma2 = 0.1, 5
C1, C2 = 0.001, 1000
hyperparams = (gamma1, C1), (gamma1, C2), (gamma2, C1), (gamma2, C2)

svm_clfs = []
for gamma, C in hyperparams:
    rbf_kernel_svm_clf = Pipeline((
            ("scaler", StandardScaler()),
            ("svm_clf", SVC(kernel="rbf", gamma=gamma, C=C))
        ))
    rbf_kernel_svm_clf.fit(X, y)
    svm_clfs.append(rbf_kernel_svm_clf)
```

```
plt.figure(figsize=(16, 8))

for i, svm_clf in enumerate(svm_clfs):
    plt.subplot(221 + i)
    plot_predictions(svm_clf, [-1.5, 2.5, -1, 1.5])
    plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])
    gamma, C = hyperparams[i]
    plt.title(r"$\gamma = {}, C = {}$".format(gamma, C), fontsize=16)
```



Exercises: *You can now complete Exercise 2 in the exercises associated with this lecture.*

9.5 Regression

Can also use SVMs to perform regression, in addition to classification.

General idea is to reverse the objective: rather than fitting the largest possible margin between two classes, SVM regression fits as many instances as possible within the margin.

The width of the margin is controlled by the hyperparameter ϵ.

9.5.1 Linear regression

```
np.random.seed(42)
m = 50
X = 2 * np.random.rand(m, 1)
y = (4 + 3 * X + np.random.randn(m, 1)).ravel()
```

```
from sklearn.svm import LinearSVR

svm_reg1 = LinearSVR(epsilon=1.5, random_state=42)
svm_reg2 = LinearSVR(epsilon=0.5, random_state=42)
svm_reg1.fit(X, y)
svm_reg2.fit(X, y)

def find_support_vectors(svm_reg, X, y):
    y_pred = svm_reg.predict(X)
    off_margin = (np.abs(y - y_pred) >= svm_reg.epsilon)
    return np.argwhere(off_margin)

svm_reg1.support_ = find_support_vectors(svm_reg1, X, y)
```

(continues on next page)

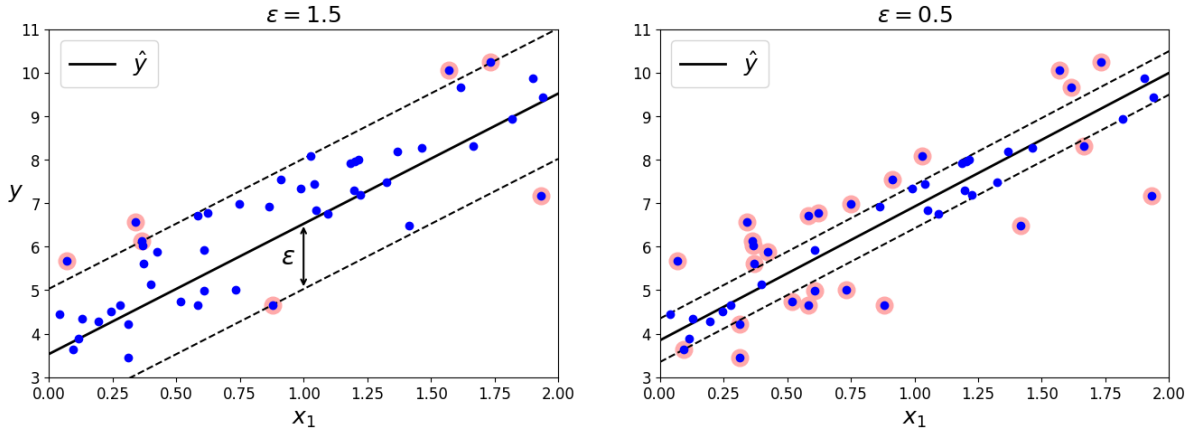
(continued from previous page)

```
svm_reg2.support_ = find_support_vectors(svm_reg2, X, y)

eps_x1 = 1
eps_y_pred = svm_reg1.predict([[eps_x1]])
```

```
eps_y_pred
```

```
array([6.52640746])
```

```
def plot_svm_regression(svm_reg, X, y, axes):
    x1s = np.linspace(axes[0], axes[1], 100).reshape(100, 1)
    y_pred = svm_reg.predict(x1s)
    plt.plot(x1s, y_pred, "k-", linewidth=2, label=r"$\hat{y}$")
    plt.plot(x1s, y_pred + svm_reg.epsilon, "k--")
    plt.plot(x1s, y_pred - svm_reg.epsilon, "k--")
    plt.scatter(X[svm_reg.support_], y[svm_reg.support_], s=180, facecolors='#FFAAAA')
    plt.plot(X, y, "bo")
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.legend(loc="upper left", fontsize=18)
    plt.axis(axes)

plt.figure(figsize=(16, 5))
plt.subplot(121)
plot_svm_regression(svm_reg1, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg1.epsilon), fontsize=18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
#plt.plot([eps_x1, eps_x1], [eps_y_pred, eps_y_pred - svm_reg1.epsilon], "k-",␣
↪linewidth=2)
plt.annotate(
        '', xy=(eps_x1, eps_y_pred), xycoords='data',
        xytext=(eps_x1, eps_y_pred - svm_reg1.epsilon),
        textcoords='data', arrowprops={'arrowstyle': '<->', 'linewidth': 1.5}
    )
plt.text(0.91, 5.6, r"$\epsilon$", fontsize=20)
plt.subplot(122)
plot_svm_regression(svm_reg2, X, y, [0, 2, 3, 11])
plt.title(r"$\epsilon = {}$".format(svm_reg2.epsilon), fontsize=18);
```

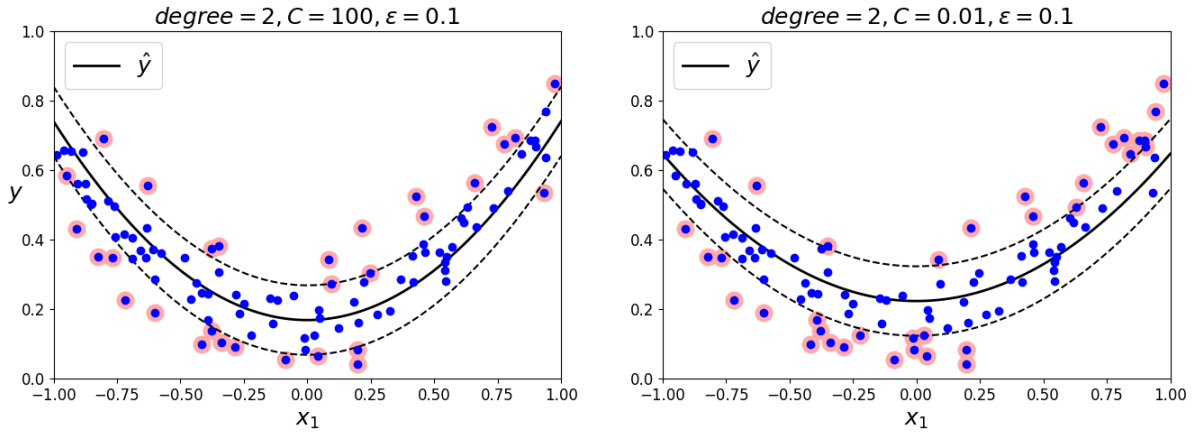


9.5.2 Non-linear regression

```
np.random.seed(42)
m = 100
X = 2 * np.random.rand(m, 1) - 1
y = (0.2 + 0.1 * X + 0.5 * X**2 + np.random.randn(m, 1)/10).ravel()
```

```
from sklearn.svm import SVR

svm_poly_reg1 = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg2 = SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1)
svm_poly_reg1.fit(X, y);
svm_poly_reg2.fit(X, y);
```

```
plt.figure(figsize=(16, 5))
plt.subplot(121)
plot_svm_regression(svm_poly_reg1, X, y, [-1, 1, 0, 1])
plt.title(r"$degree={}, C={}, \epsilon = {}$".format(svm_poly_reg1.degree, svm_poly_
↪reg1.C, svm_poly_reg1.epsilon), fontsize=18)
plt.ylabel(r"$y$", fontsize=18, rotation=0)
plt.subplot(122)
plot_svm_regression(svm_poly_reg2 , X, y, [-1, 1, 0, 1])
plt.title(r"$degree={}, C={}, \epsilon = {}$".format(svm_poly_reg2.degree, svm_poly_
↪reg2.C, svm_poly_reg2.epsilon), fontsize=18);
```



CHAPTER

TEN

LECTURE 10: ARTIFICIAL NEURAL NETWORKS (ANNS)

Run in colab

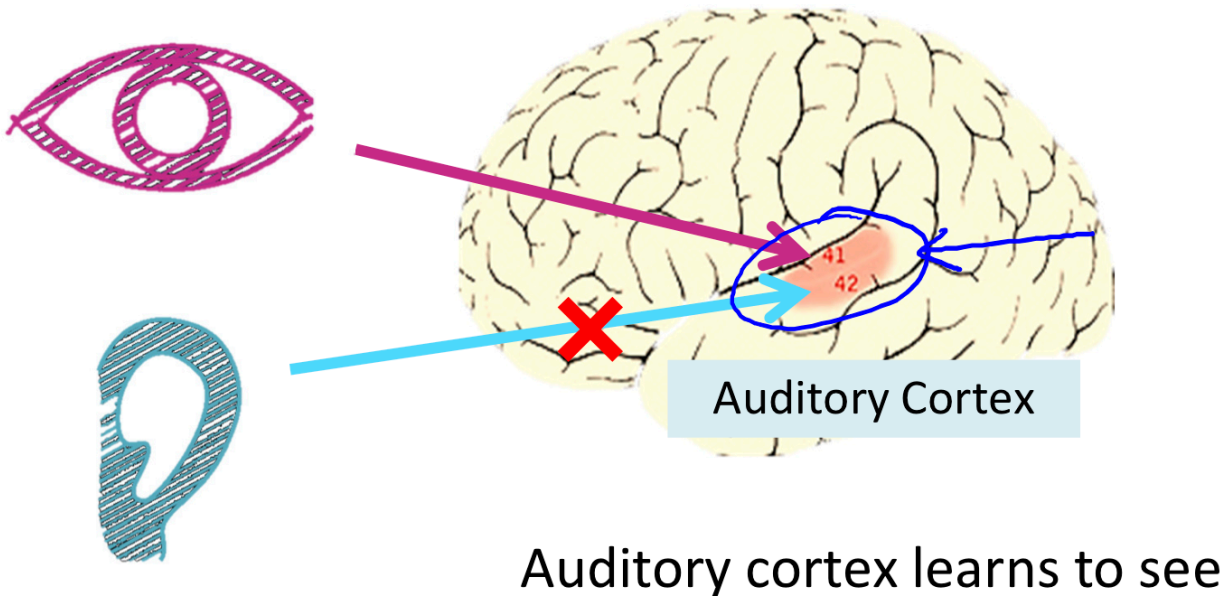
```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:20:23
```

10.1 Biological inspiration

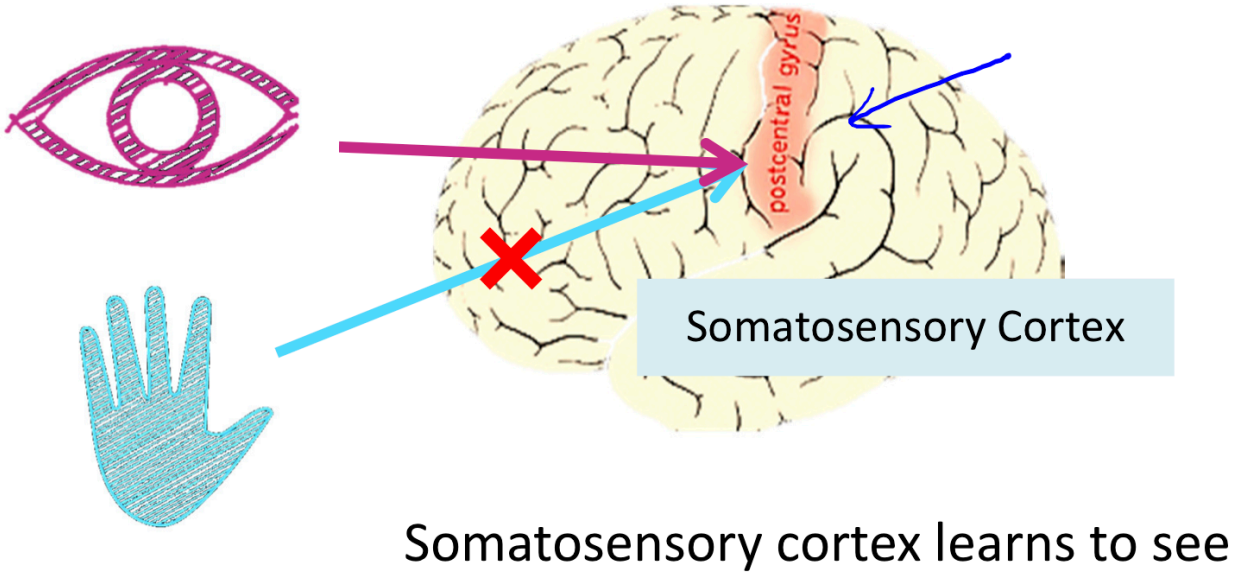
Architecture of neural networks originally inspired by the brain.

10.1.1 Rewriting the brain



Study performed with ferrets by Roe et al. (1992).

[Image credit]

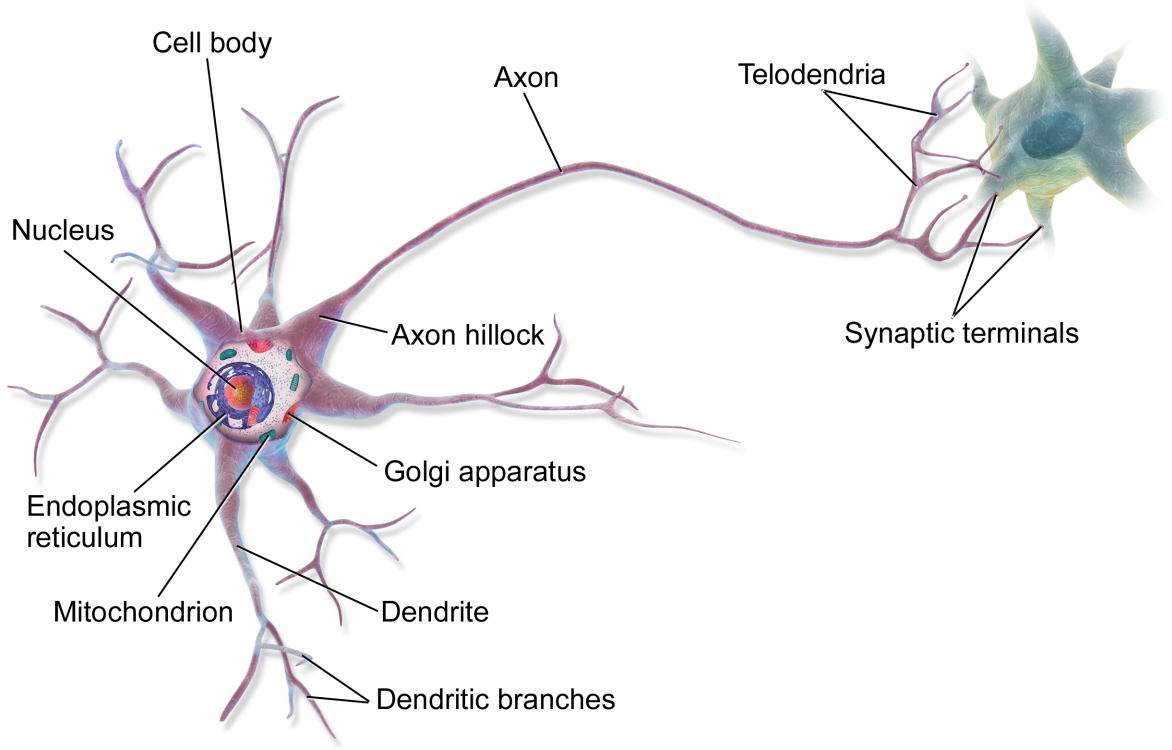


Study performed with hamsters by Metin & Frost (1989).

[Image credit]

Led to "*one learning algorithm*" hypothesis.

10.1.2 Biological neurons



[Image credit: Bruce Blaus, Wikipedia]

Biological neurons consist of cell body containing nucleus, dentrite branches (inputs) and axon (output).

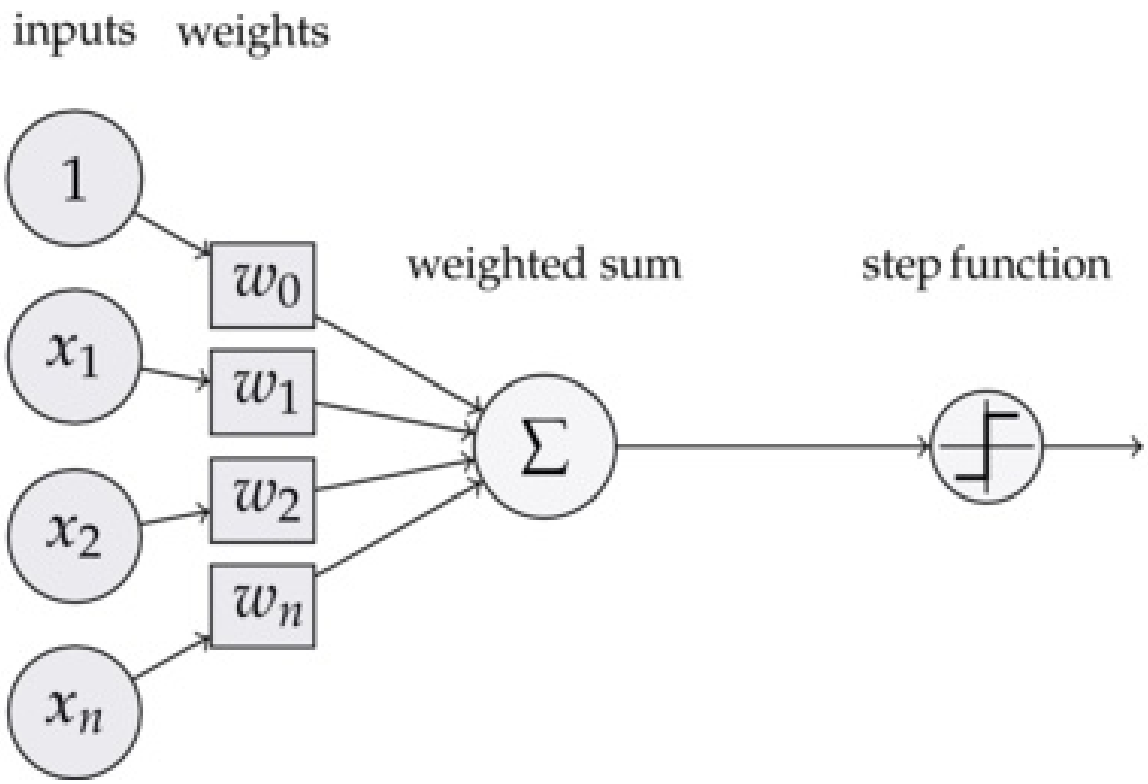
Axon connects neurons and the length of the axon can be a few to 10,000 times the size of the cell body.

Axon splits into telodendria branch, with synaptic terminals at ends, which are connected to dendrites of other neurons.

Although biological neurons rather simple, complexity comes from networks of billions of neurons, each connected to thousands of other neurons.

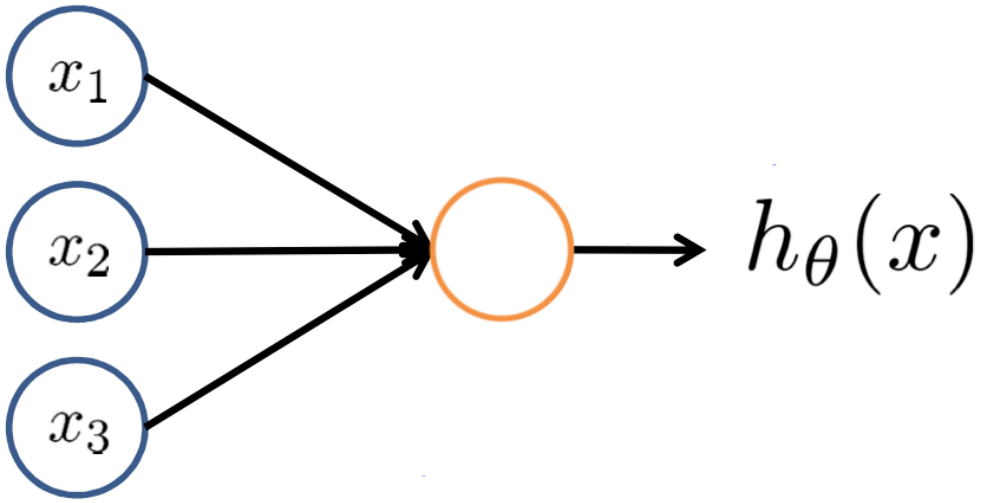
10.2 Artificial neurons (units)

10.2.1 Perceptron



[Image credit]

10.2.2 General logistic unit



Weighted sum:

$$z = \sum_{j=1}^{n} \theta_j x_j = \theta^{\mathrm{T}} x.$$

Activations: $a = h(z)$, for non-linear activation function h.

Generally we refer to as a logistic unit (rather than an artificial neuron) since additional generalities than concepts motivated by biology will be considered.

10.2.3 Examples of activation functions

Step

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

Sigmoid

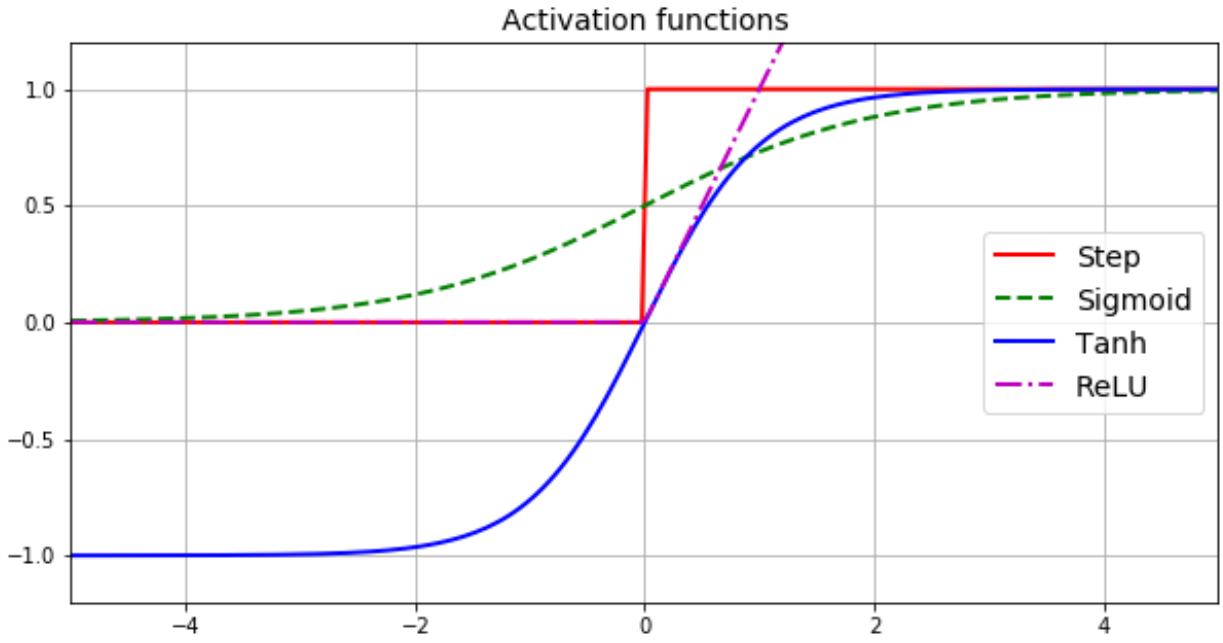
$$a(z) = \frac{1}{1 + \exp(-z)}$$

Hyperboic tangent

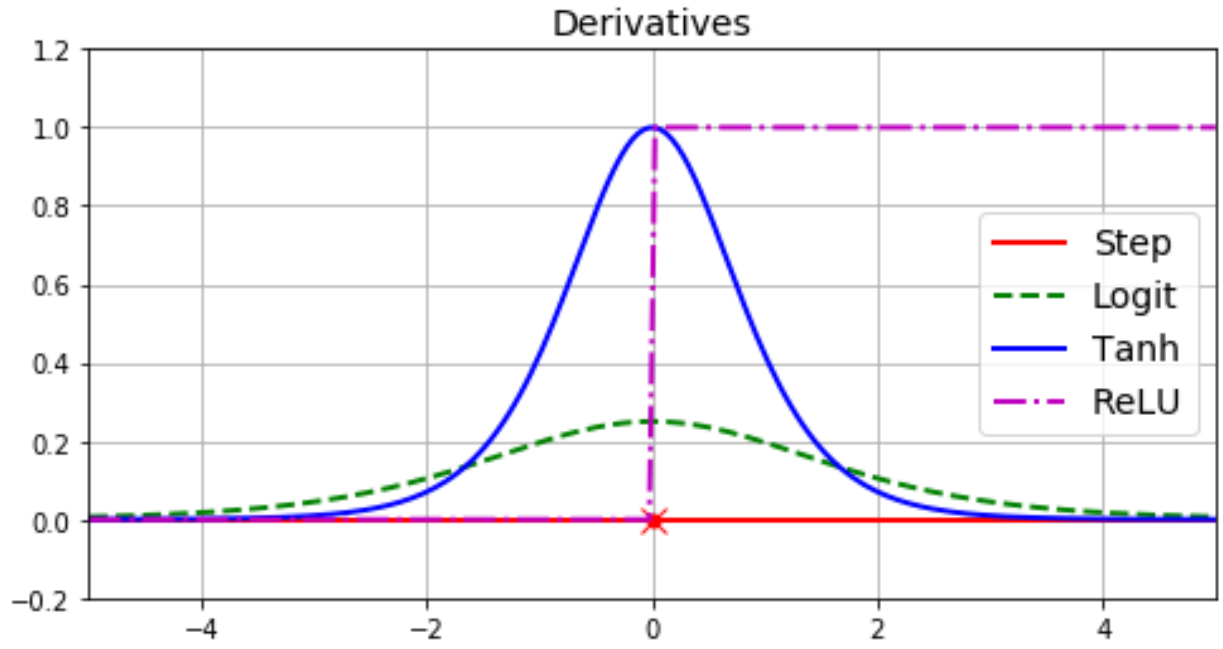
$$a(z) = \tanh(z)$$

Rectified linear unit (ReLU)

$$a(z) = \max(0, z)$$



Gradients of activation functions

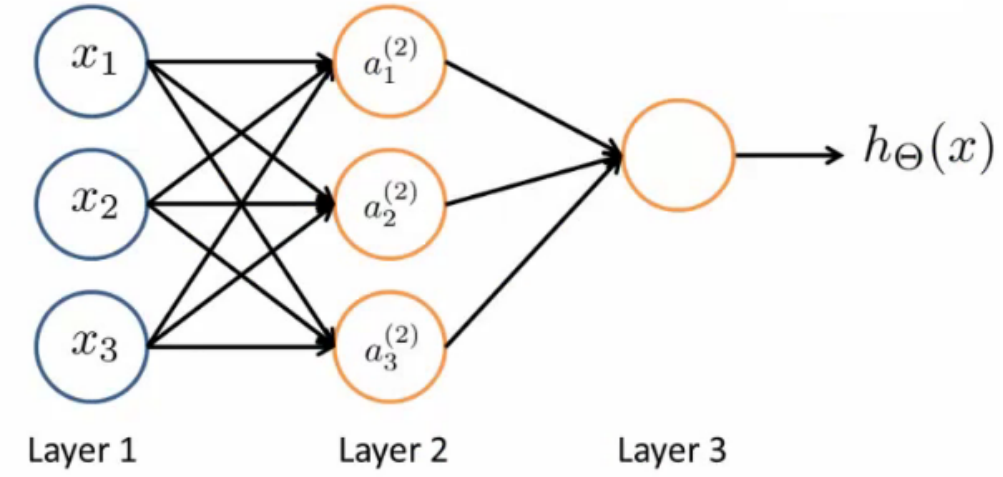


Notice the step function has zero gradient.

Exercises: *You can now complete Exercise 1-2 in the exercises associated with this lecture.*

10.3 Neural network

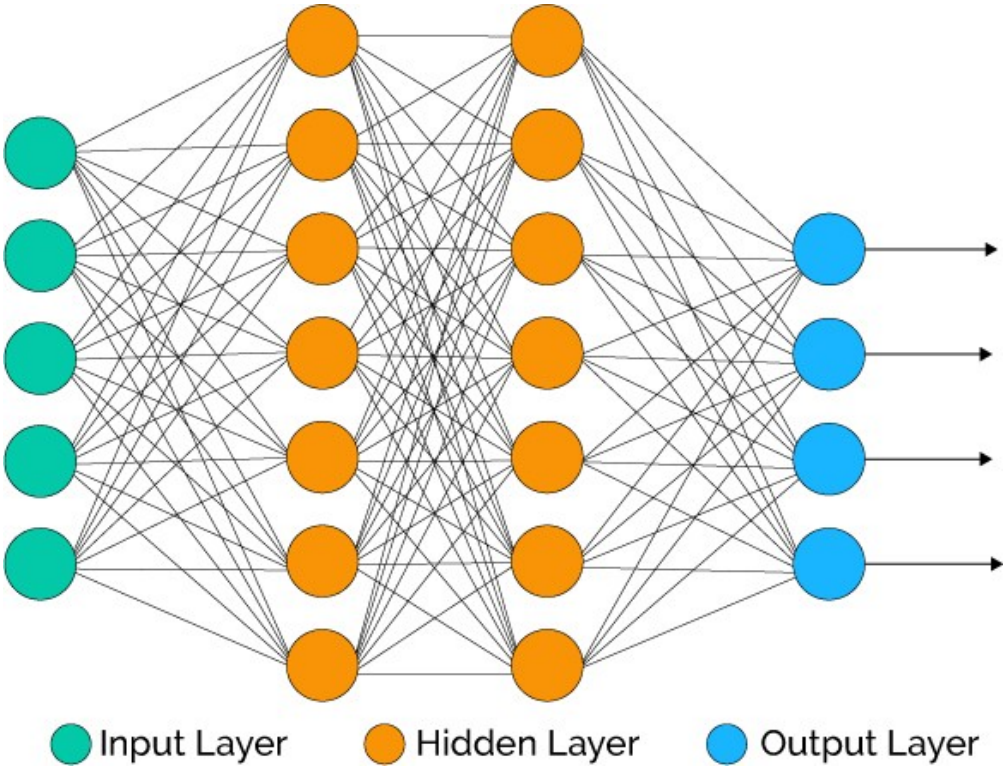
Construct artifical neural network by combining layers of multiple logistic units.



Weighted sums: $z_j = \sum_{i=1}^{n} \theta_{ij} x_i$

Activations: $a_j = h(z_j)$

10.3.1 Architectures and terminology



[Image credit]

Networks can be wide/narrow and deep/shallow.

Here we consider feedforward network only. Other configurations can also be considered, as we will see later in the course.

10.3.2 Universal approximation theorem

The *universal approximation theorem* states that a feedforward network *can* accurately approximate any continuous function from one finite dimensional space to another, given enough hidden units (Hornik et al. 1989, Cybenko 1989).

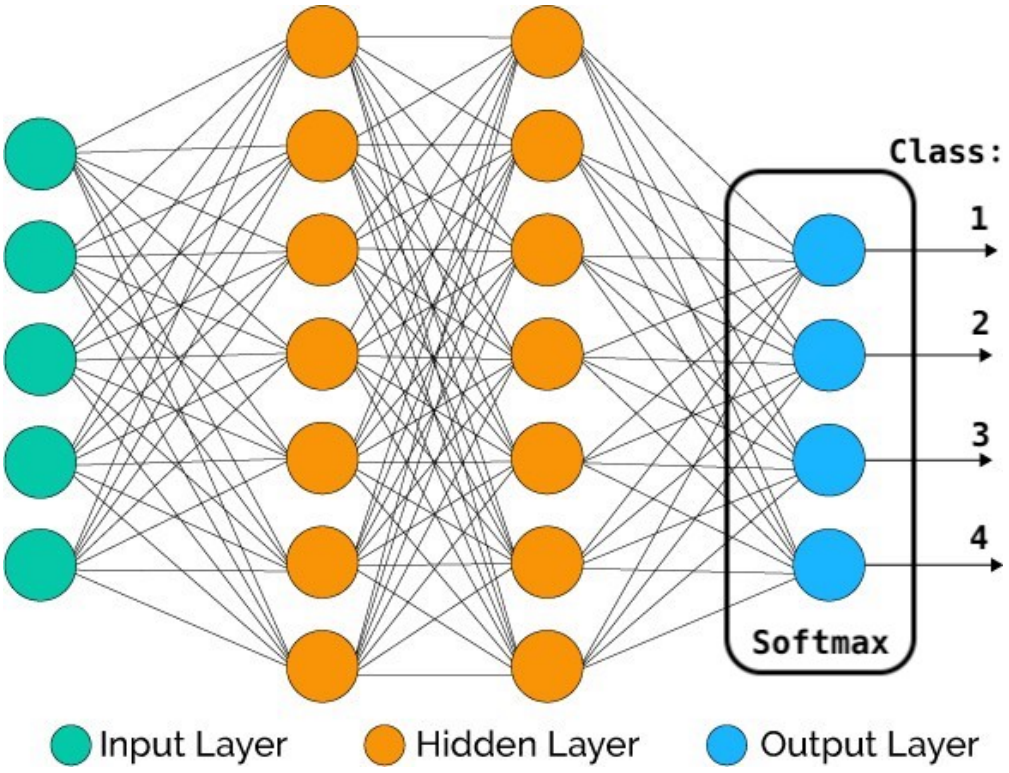
(Some technical caveats that are beyond the scope of this course regarding properties of the mapping and activation functions.)

ANNs thus have the *potential* to be universal approximators.

Universal approximation theorem does *not* provide any guarantee that training finds this representation.

10.4 Multi-class classification

Multi-class classification can be easily performed with an ANN, where each output node corresponds to a certain class.



[Image credit (adapted)]

Set up training data as unit vectors, with 1 for the target class and 0 for all other classes.

10.4.1 Softmax

Map predictions to *"probabilities"* using the softmax function for all output nodes with activiations a_j:

$$\hat{p}_j = \frac{\exp(a_j)}{\sum_{j'} \exp(a_{j'})}.$$

Normalised such that

- $\sum_j \hat{p}_j = 1$
- $0 \leq \hat{p}_j \leq 1$

10.5 Cost functions

Appropriate cost functions depend on whether performing regression or classification. Consider targets $y_j^{(i)}$ and outputs (predictions) of ANN $\hat{p}_j^{(i)}$, for training instance i and output node j.

Typical cost function for regression is the mean square error:

$$\text{MSE}(\Theta) = \frac{1}{m} \sum_i \sum_j \left(\hat{p}_j^{(i)} - y_j^{(i)} \right)^2.$$

Typical cost function for classification is cross-entropy:

$$C(\Theta) = -\frac{1}{m} \sum_i \sum_j y_j^{(i)} \log \left(\hat{p}_j^{(i)} \right).$$

(Although other cost functions are used widely.)

Various forms of regularisation often considered, e.g. ℓ_2 regularisation.

Error surface non-convex, potentially with many local optima. Historically training ANNs has been difficult.

10.6 Backpropagation

10.6.1 Problem

To train ANN's want to exploit gradient of error surface (e.g. for gradient descent algorithms). Therefore need an efficient method to compute gradients.

Backpropagation algorithm developed by Rumelhart, Hinton & Williams (1986) to efficiently compute the gradient of the error surface (i.e. cost function) with respect to each weight of the network.

Gradients then accessible for training.

10.6.2 Overview of backpropagation

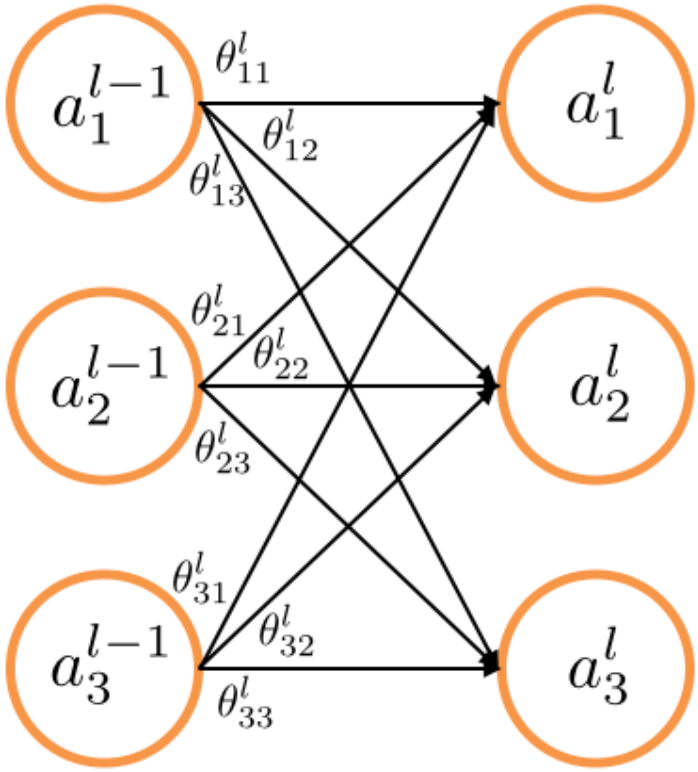
Backpropagation consists of forward and reverse (backwards) passes (hence name).

Consider each training instance. A forward run of the network is applied to compute the output error ϵ. Then errors are backpropagated through the network to compute the rate of change of the error with respect to the weights of the network.

In practice, error gradients $\frac{\partial \epsilon}{\partial z_j}$ are computed and backpropagated, from which error gradients with respect to the weights can be computed $\frac{\partial \epsilon}{\partial \theta_{ij}}$.

Backpropagation algorithm follows from a straightforward application of the chain rule.

10.6.3 Define network architecture and notation



Now make network layer explicit in notation.

Weighted sum: $z_j^l = \sum_i \theta_{ij}^l a_i^{l-1}$, where θ_{ij}^l is the weight between node i at layer $l-1$ and node j at layer l (note that difference conventions are often used, e.g. θ_{ji}^{l-1} for the same connection). Consider L layers.

Activations: $a_i^l = h(z_i^l)$.

10.6.4 Backpropagation calculations

Want to compute

$$\Delta\theta_{ij}^{l} = -\eta \frac{\partial \epsilon}{\partial \theta_{ij}^{l}}.$$

By chain rule:

$$\frac{\partial \epsilon}{\partial \theta_{ij}^{l}} = \frac{\partial \epsilon}{\partial z_j^l} \frac{\partial z_j^l}{\partial \theta_{ij}^{l}} = \frac{\partial \epsilon}{\partial z_j^l} a_i^{l-1} = \delta_i^l a_i^{l-1},$$

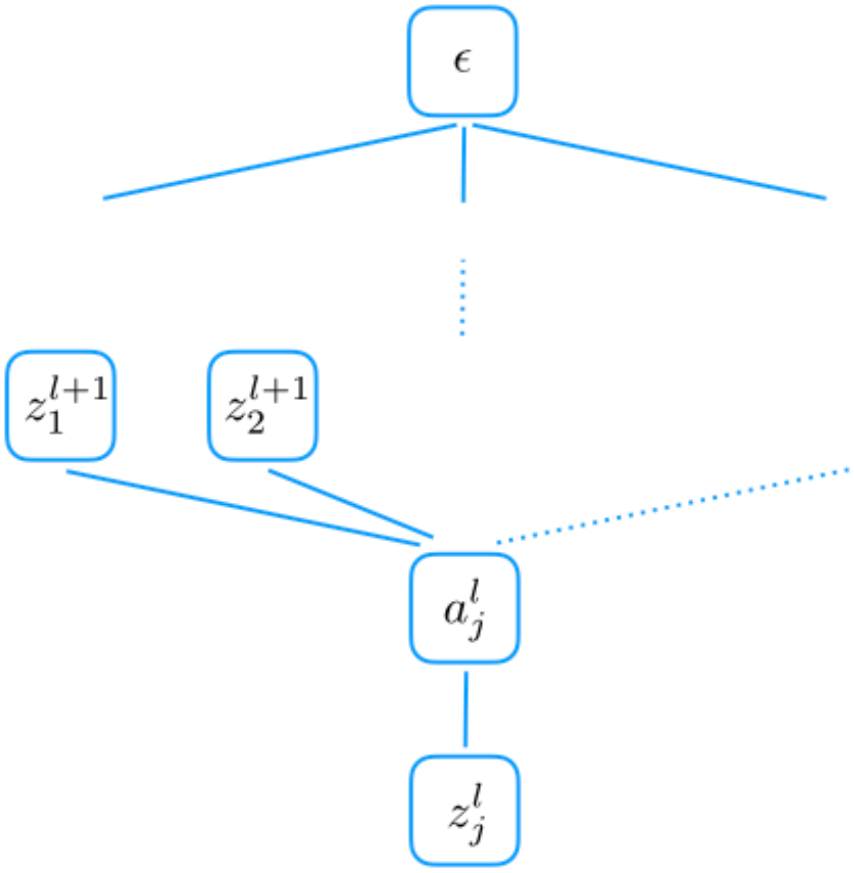
where $\delta_i^l = \frac{\partial \epsilon}{\partial z_j^l}$

$\left(\text{recall } z_j^l = \sum_i \theta_{ij}^l a_i^{l-1}\right)$.

Now need to compute

$$\delta_i^l = \frac{\partial \epsilon}{\partial z_j^l}.$$

Functional dependence



By chain rule again:

$$\delta_i^l = \frac{\partial \epsilon}{\partial z_j^l} = \sum_i \frac{\partial \epsilon}{\partial z_i^{l+1}} \frac{\partial z_i^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial z_j^l} = \sum_i \delta_i^{l+1} \theta_{ji}^{l+1} h'(z_j^l).$$

Note the term $h'(z_j^l)$ is independent of i and so can be moved outside the summation.

Boundary condition:

$$\delta_i^L = \frac{\partial \epsilon}{\partial z_j^L} = \frac{\partial \epsilon}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial \epsilon}{\partial a_j^L} h'(z_j^L).$$

10.6.5 Summary of backpropagation

For current set of weights θ_{ij}^l, compute forward pass through network:

$$z_j^l = \sum_i \theta_{ij}^l a_i^{l-1},$$

$$a_i^l = h(z_i^l).$$

Propagate errors backwards through network:

$$\delta_i^l = \frac{\partial \epsilon}{\partial z_j^l} = \sum_i \delta_i^{l+1} \theta_{ji}^{l+1} h'(z_j^l).$$

Compute derivatives of error with respect to weights:

$$\frac{\partial \epsilon}{\partial \theta_{ij}^l} = \delta_i^l a_i^{l-1}.$$

10.6.6 Training with backpropagation

Backpropagation simply computes derivatives of error with respect to weights.

Still need training algorithm to update weights given derivatives, e.g. $\Delta\theta_{ij}^l = -\eta \frac{\partial \epsilon}{\partial \theta_{ij}^l}$.

Various approaches can be considered:

- Online: update weights after each training instance.
- Full-batch: update weights after full sweep through training data.
- Mini-batch: update weights after a small sample of training cases.

Example

Scikit-learn now supports ANNs but not intended for large scale problems.

Trains using some form of gradient descent, with gradients computed by backpropagation.

```
# disable convergence warning from early stopping
from warnings import simplefilter
from sklearn.exceptions import ConvergenceWarning
simplefilter("ignore", category=ConvergenceWarning)
```

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier

mnist = fetch_openml('mnist_784', version=1, cache=True, parser='auto')
# rescale the data, use the traditional train/test split
X, y = mnist.data / 255., mnist.target
X_train, X_test = X[:60000], X[60000:]
y_train, y_test = y[:60000], y[60000:]

mlp = MLPClassifier(hidden_layer_sizes=(50,), max_iter=10, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1,
                    learning_rate_init=.1)

mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))
```

```
Iteration 1, loss = 0.32009978
Iteration 2, loss = 0.15347534
Iteration 3, loss = 0.11544755
Iteration 4, loss = 0.09279764
Iteration 5, loss = 0.07889367
Iteration 6, loss = 0.07170497
Iteration 7, loss = 0.06282111
Iteration 8, loss = 0.05530788
Iteration 9, loss = 0.04960484
Iteration 10, loss = 0.04645355
Training set score: 0.986800
Test set score: 0.970000
```

CHAPTER

ELEVEN

LECTURE 11: INTRODUCTION TO TENSORFLOW


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:20:52
```

11.1 Overview of TensorFlow

TensorFlow is an open source library developed by Google for numerical computation. It is particularly well suited for large-scale machine learning.

TensorFlow is based on the construction of *computational graphs*. It has evolved considerably since it's open source release in 2015. We will use TF2, which offers many additional features built on top of core features (the most important is `tf.keras` discussed in later lectures).

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
tf.__version__, keras.__version__
```

```
2023-02-04 10:20:52.667368: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
2023-02-04 10:20:53.359087: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror:␣
 ↪libcudart.so.11.0: cannot open shared object file: No such file or directory
2023-02-04 10:20:53.359124: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]␣
 ↪Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2023-02-04 10:20:53.447386: E tensorflow/stream_executor/cuda/cuda_blas.cc:2981]␣
 ↪Unable to register cuBLAS factory: Attempting to register factory for plugin␣
 ↪cuBLAS when one has already been registered
2023-02-04 10:20:54.545527: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror:␣
 ↪libnvinfer.so.7: cannot open shared object file: No such file or directory
```

(continues on next page)

(continued from previous page)

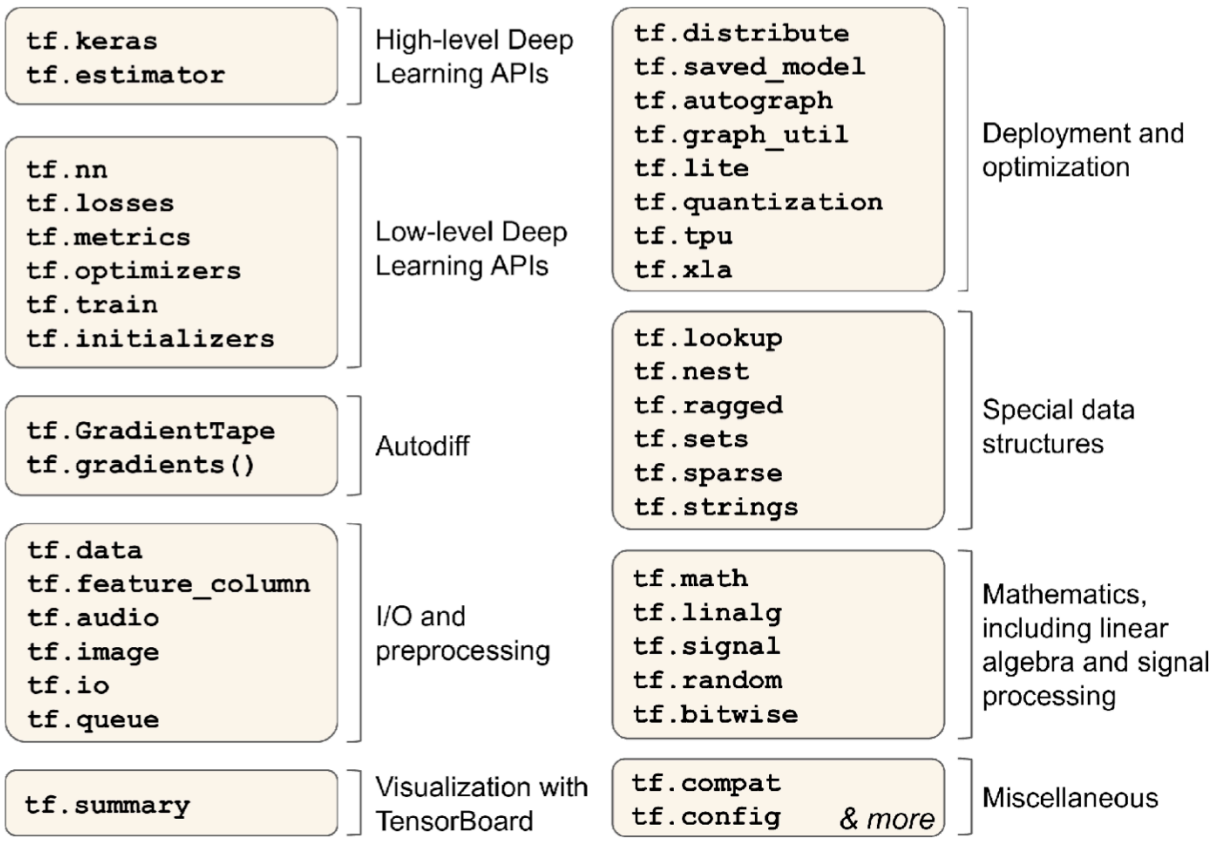
```
2023-02-04 10:20:54.545617: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer_plugin.so.7'; dlerror:␣
 ↪libnvinfer_plugin.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:20:54.545627: W tensorflow/compiler/tf2tensorrt/utils/py_utils.
 ↪cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like␣
 ↪to use Nvidia GPU with TensorRT, please make sure the missing libraries␣
 ↪mentioned above are installed properly.
```

```
('2.10.0', '2.10.0')
```

11.1.1 Features

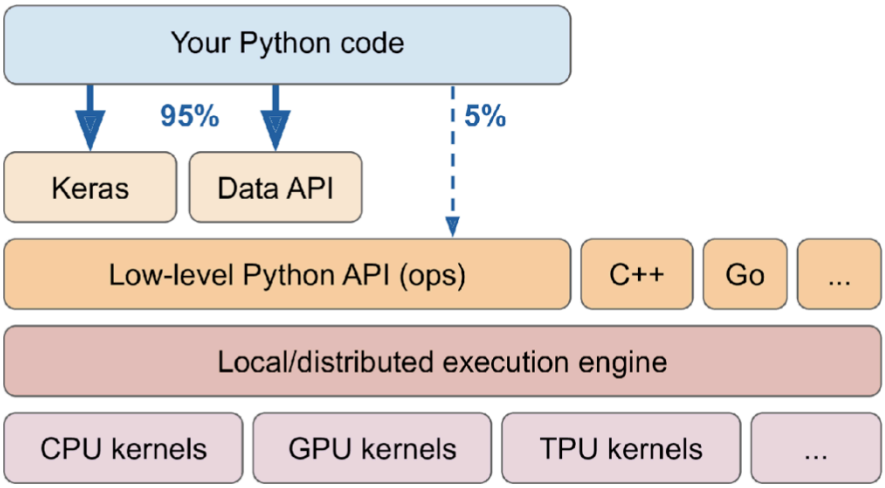
- Similar to `numpy` but with GPU support.
- Supports distributed computing.
- Includes a kind of just-in-time (JIT) compiler to optimise speed and memory usage.
- Computational graphs can be saved and exported.
- Supports autodiff and provides numerous advanced optimisers.

11.1.2 TensorFlow's Python API



[Credit: Geron]

11.1.3 TensorFlow's Architecture



[Credit: Geron]

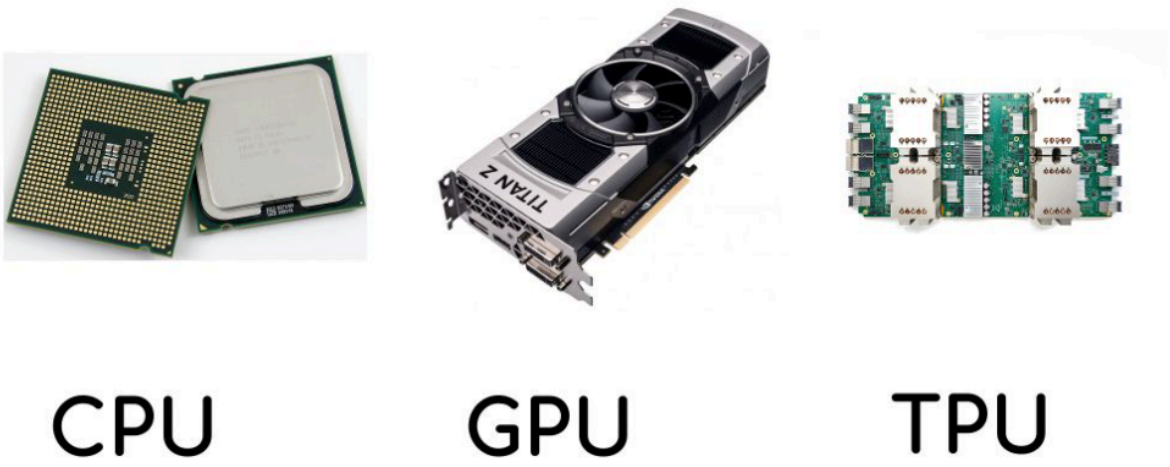
At lowest level TensorFlow is implemented in C++ so that it is highly efficient.

We will focus solely on the python TensorFlow interfaces (typical approach). Most of the time you will simple need to interact with the Keras interface but sometimes you might want to use the low-level python API for greater flexibility.

11.1.4 Hardware

One of the factors responsible for the dramatic recent growth of machine learning is advances in computing power.

In particular, hardware that supports high levels of parallelism.

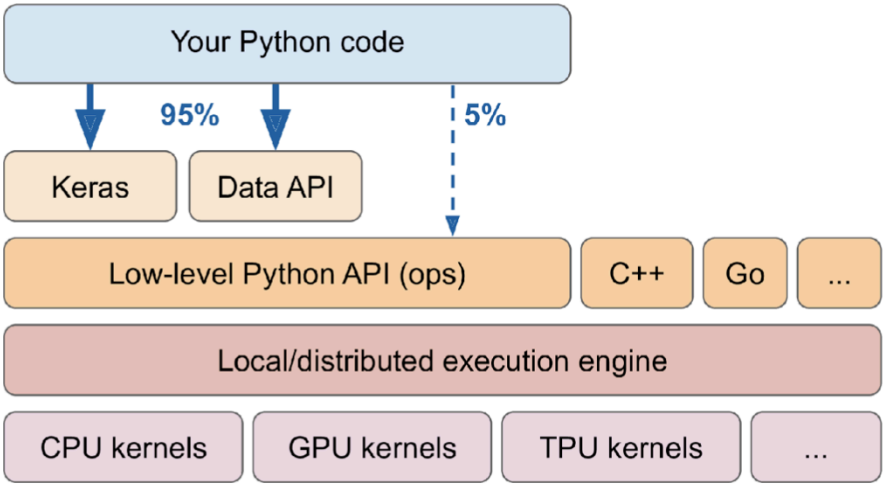


- Central Processing Unit (CPU):

  - General purpose
  - Low latency
  - Low throughput
  - Sequential
- Graphics Processing Unit (GPU)
  - Specialised (for graphics initially)
  - High latency
  - High throughput
  - Parallel execution
- Tensor Processing Unit (TPU)
  - Specialised for matrix operations
  - High latency
  - Very high throughput
  - Extreme parallel execution

In TensorFlow many operations are implemented in low-level kernels, optimised for specific hardware, e.g. CPUs, GPUS, or TPUs.

TensorFlow's execution engine will ensure operations are run efficiently (across multiple machines and devices if set up accordingly).



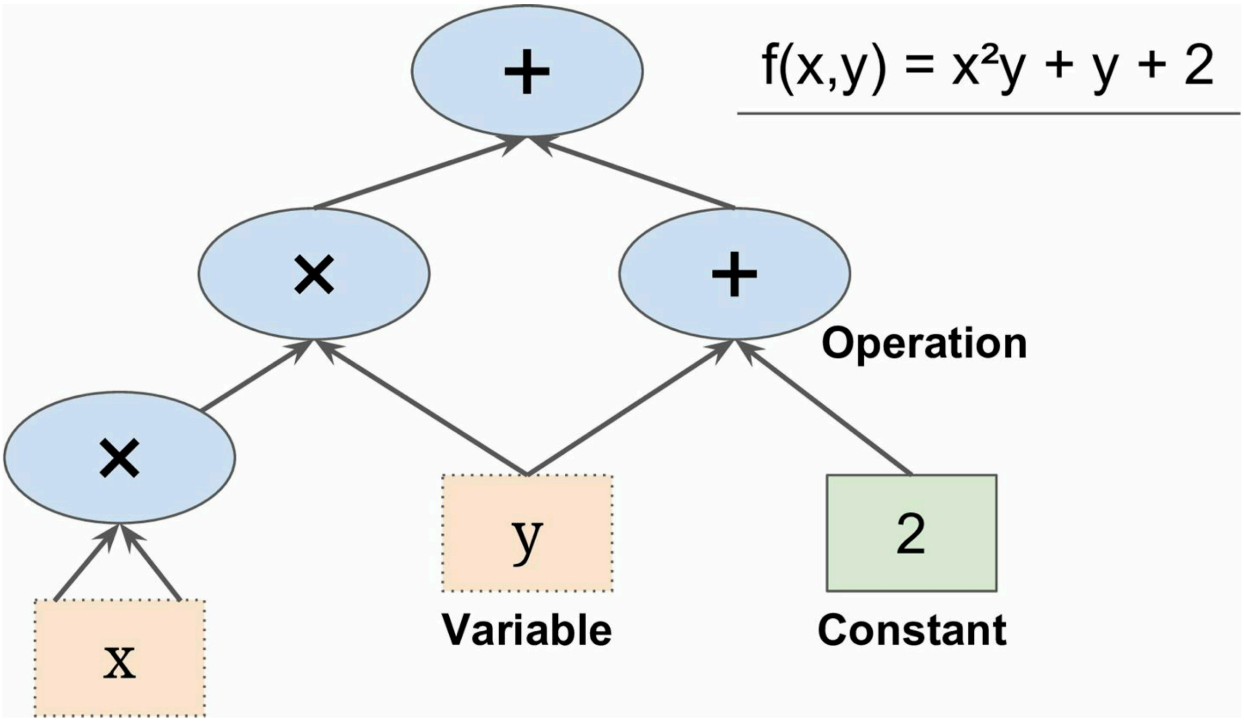
[Credit: Geron]

Aside: chips optimised for machine learning are an active area of development

Google developed TPU.

Graphcore have developed the Intelligence Processing Unit (IPU).

11.1.5 Computational graphs



[Credit: Geron]

User code constructs the computational graph (can be constructed in Python). With TensorFlow 2, graph construction is less explicit and much simpler.

TensorFlow takes computational graph and runs it efficiently via optimized C++ code.

11.1.6 Parallel and distributed computation

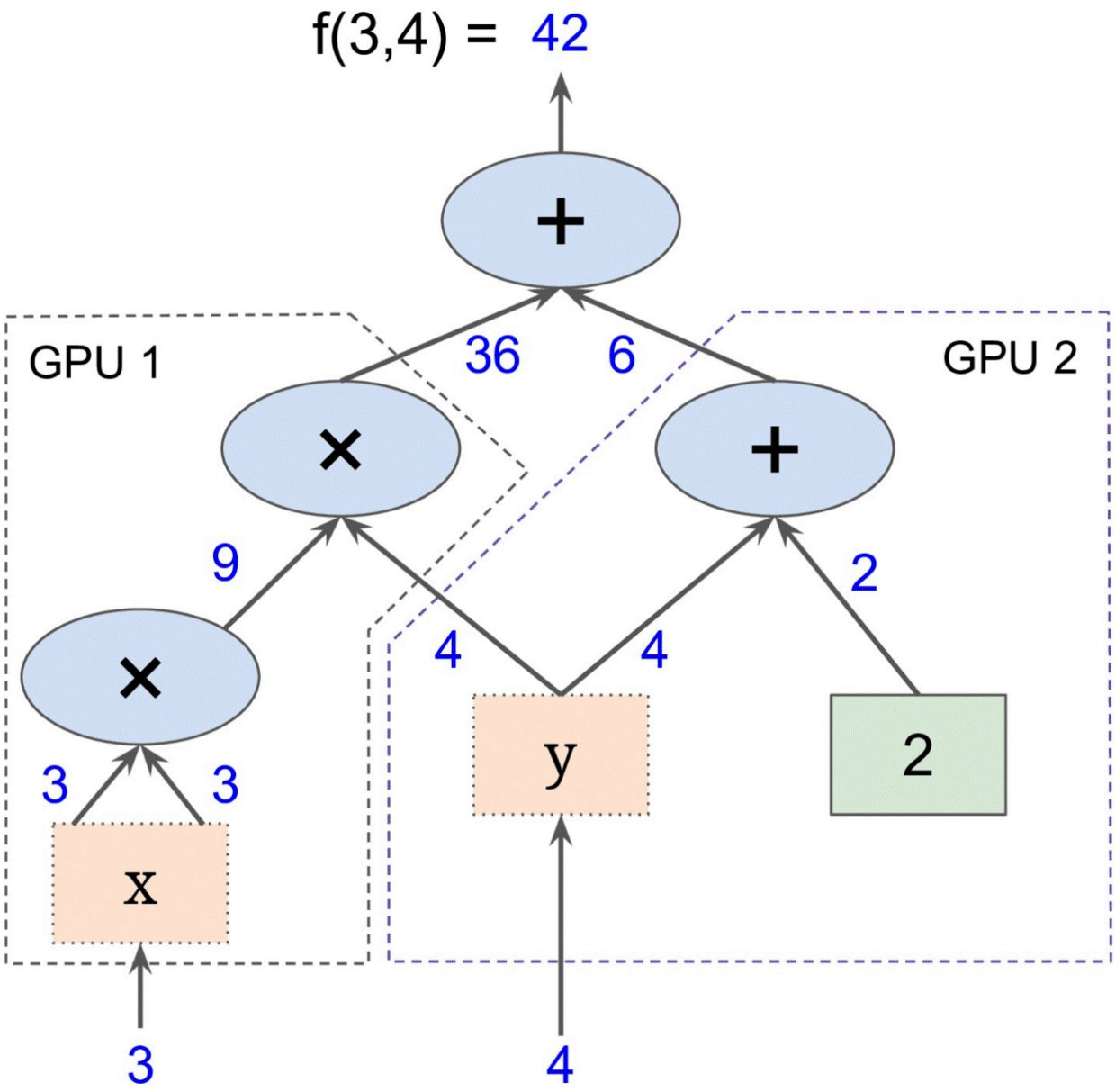


Figure 9-2. Parallel computation on multiple CPUs/GPUs/servers

[Credit: Geron]

Computational graphs can be broken up into different chunks, which are then run in parallel across many CPUs/GPUs/TPUs (or highly distributed systems).

This approach allows TensorFlow to scale to big-data.

11.1.7 Scaling to big-data

For example, TensorFlow can be used to train neural networks with millions of parameters and training sets with billions of training instances.

Provides the infrastructure behind many of Google's large-scale machine learning products, e.g. Google Search, Google Photos, …

11.2 Tensors and operations

TensorFlow API centers around "Tensors" (essentially multi-dimensional arrays of matrices), hence its name.

Similar to numpy `ndarray`.

11.2.1 Tensors

Can construct constant tensors with `tf.constant`.

```
tf.constant([[1., 2., 3.], [4., 5., 6.]]) # 2x3 matrix
```

```
2023-02-04 10:20:56.422610: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.
 ↪1: cannot open shared object file: No such file or directory
2023-02-04 10:20:56.423258: W tensorflow/stream_executor/cuda/cuda_driver.cc:263]␣
 ↪failed call to cuInit: UNKNOWN ERROR (303)
2023-02-04 10:20:56.423336: I tensorflow/stream_executor/cuda/cuda_diagnostics.
 ↪cc:156] kernel driver does not appear to be running on this host (fv-az267-630):␣
 ↪/proc/driver/nvidia/version does not exist
2023-02-04 10:20:56.428339: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

```
tf.constant(42) # scalar
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=42>
```

Tensors have a shape and data type (dtype).

```
t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
t.shape
```

```
TensorShape([2, 3])
```

```
t.dtype
```

```
tf.float32
```

11.2.2 Indexing

Tensor indexing is very similar to numpy.

```
t[:, 1:]
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
```

```
t[..., 1,  tf.newaxis]
```

```
<tf.Tensor: shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

11.2.3 Operations

Variety of tensor operations are possible.

```
t + 10
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
```

```
tf.square(t)
```

```
<tf.Tensor: shape=(2, 3), dtype=float32, numpy=
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)>
```

```
t @ tf.transpose(t) # matrix multiplication
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

11.2.4 Using `keras.backend`

Keras API also includes its own low-level API with similar functionality, which is basically a wrapper for the corresponding TensorFlow operations (more on Keras in next lecture).

```
from tensorflow import keras
K = keras.backend
K.square(K.transpose(t)) + 10
```

```
<tf.Tensor: shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

11.2.5 Tensors and Numpy

Note: From `tf.__version__ == 2.4.0` tensorflow.numpy functionality will be added: https://www.tensorflow.org/api_docs/python/tf/experimental/numpy

Can create a tensor from ndarray.

```
a = np.array([2., 4., 5.])
tf.constant(a)
```

```
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
```

Can convert ndarray to tensor.

```
t.numpy()
```

```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

Can apply numpy operations to tensors and vice versa.

```
np.array(t)
```

```
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
```

```
tf.square(a)
```

```
<tf.Tensor: shape=(3,), dtype=float64, numpy=array([ 4., 16., 25.])>
```

```
np.square(t)
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```

11.2.6 Conflicting Types

TensorFlow does not perform type conversions automatically since they can significantly degrade performance and can easily go unnoticed.

Therefore you cannot add a float to an integer.

```
try:
    tf.constant(2.0) + tf.constant(40)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
cannot compute AddV2 as input #1(zero-based) was expected to be a float tensor but␣
↪is a int32 tensor [Op:AddV2]
```

Similarly, you cannot add a float (32 bit) and a double (64 bit).

```
try:
    tf.constant(2.0) + tf.constant(40., dtype=tf.float64)
except tf.errors.InvalidArgumentError as ex:
    print(ex)
```

```
cannot compute AddV2 as input #1(zero-based) was expected to be a float tensor but␣
↪is a double tensor [Op:AddV2]
```

If you want to consider operations with different types you need to explicitly cast them first.

```
t2 = tf.constant(40., dtype=tf.float64)
tf.constant(2.0) + tf.cast(t2, tf.float32)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=42.0>
```

11.2.7 Variables

Previous tensors we've considered are constant and immutable so they cannot be changed.

We also need tensors that can act as variables that can change over time, for example for weights of a neural network that are regularly updated during training.

```
v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])
v
```

```
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
```

Can be modified in place using the `assign` method.

```
v.assign(2 * v)
```

```
<tf.Variable 'UnreadVariable' shape=(2, 3) dtype=float32, numpy=
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]], dtype=float32)>
```

11.3 TensorFlow Functions

Once TensorFlow has constructed a computational graph, it optimises it (e.g. simplying expressions, pruning unused nodes, etc.).

Consequently, a TensorFlow function will typically run a lot faster than an equivalent numpy function.

`tf.function` can be used to turn a python function into a TensorFlow function.

```
def cube(x):
    return x ** 3
```

```
cube(2)
```

```
8
```

```
tf_cube = tf.function(cube)
tf_cube
```

```
<tensorflow.python.eager.def_function.Function at 0x7ffbc0541910>
```

```
tf_cube(2)
```

```
<tf.Tensor: shape=(), dtype=int32, numpy=8>
```

```
tf_cube(tf.constant(2.0))
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=8.0>
```

When you write custom functionality with a Keras model, Keras will automatically convert your function to a TensorFlow function so typically you will not need to worry about this.

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

11.3.1 Reuse

A TensorFlow function generates a new graph for each unique set of input shapes and data types. The graph is then cached for subsequent use.

This is only the case for tensor arguments.

If you pass numerical python values a new graph will be created for each execution. This could considerably slow down your code and may use up a lot of RAM (for the storage of many computational graphs).

11.4 Gradients

As we have seen when considering training, we often need to compute the gradients to train models, e.g. for gradient descent based approaches. Typically we need to compute the gradient of the cost function with respect to the model weights.

TensorFlow supports automatical differentiation, which allows gradients to be computed automatically.

Consider the following function.

```
def f(w1, w2):
    return 3 * w1 ** 2 + 2 * w1 * w2
```

We will compute gradients analytically, numerically and using TensorFlow's Autodiff functionality at the following point.

```
w1, w2 = 5.0, 3.0
```

11.4.1 Computing gradients analytically

```
def df_dw1(w1, w2):
    return 6 * w1 + 2 * w2
def df_dw2(w1, w2):
    return 2 * w1
```

```
df_dw1(w1, w2)
```

```
36.0
```

```
df_dw2(w1, w2)
```

```
10.0
```

11.4.2 Computing gradients numerically

Compute the gradient by finite differences.

```
eps = 1e-6
(f(w1 + eps, w2) - f(w1, w2)) / eps
```

```
36.000003007075065
```

```
(f(w1, w2 + eps) - f(w1, w2)) / eps
```

```
10.000000003174137
```

Gradients computed are approximate.

Required an extra function evaluation for every gradient. Computationally infeasible for many cases, e.g. large neural networks with hundreds of thousands or millions of parameters (or more!).

11.4.3 Computing gradients with Autodiff

Autodiff builds derivatives of each stage of the computational graph so that gradients can be computed automatically and efficiently.

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)

gradients = tape.gradient(z, [w1, w2])
```

```
gradients
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

Only requires one computation regardless of how many derivatives need to be computed and result does not suffer from any numerical approximations (only limited by machine precision arithmetic).

Persistence

Tape is erased immediately after call to `gradient` method. So will fail if you try to call it twice.

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)
try:
    dz_dw2 = tape.gradient(z, w2)
except RuntimeError as ex:
    print(ex)
```

```
A non-persistent GradientTape can only be used to compute one set of gradients (or␣
↪jacobians)
```

Can make the tape persistent if you need to call it more than once. Then be sure to delete it once done to free resources.

```
with tf.GradientTape(persistent=True) as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)
dz_dw2 = tape.gradient(z, w2) # works now!
del tape
```

```
dz_dw1, dz_dw2
```

```
(<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>)
```

Computing gradients wrt variables and watched tensors

The tape only tracks variables (recall constants are immutable so it does not make sense to compute a gradient with respect to a constant).

If you try to compute the gradient with respect to (wrt) anything other than a variable you will get a None result.

```
c1, c2 = tf.constant(5.), tf.constant(3.)
with tf.GradientTape() as tape:
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2])
gradients
```

```
[None, None]
```

But you can `watch` tensors and then compute gradients with respect to watched tensors as if they were variables.

```
with tf.GradientTape() as tape:
    tape.watch(c1)
    tape.watch(c2)
    z = f(c1, c2)

gradients = tape.gradient(z, [c1, c2])
gradients
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: shape=(), dtype=float32, numpy=10.0>]
```

Stopping gradients propagating

Sometimes you may want to stop gradients propagating through the computational graph.

This can be performed with `tf.stop_gradient`, which allows the function to be evaluated in the forward evaluation pass but not in the reverse gradient pass.

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)

with tf.GradientTape() as tape:
    z = f(w1, w2)

tape.gradient(z, [w1, w2])
```

```
[<tf.Tensor: shape=(), dtype=float32, numpy=30.0>, None]
```

Exercises: *You can now complete Exercise 2 in the exercises associated with this lecture.*

CHAPTER

TWELVE

LECTURE 12: INTRODUCTION TO KERAS


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:21:15
```

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import os
```

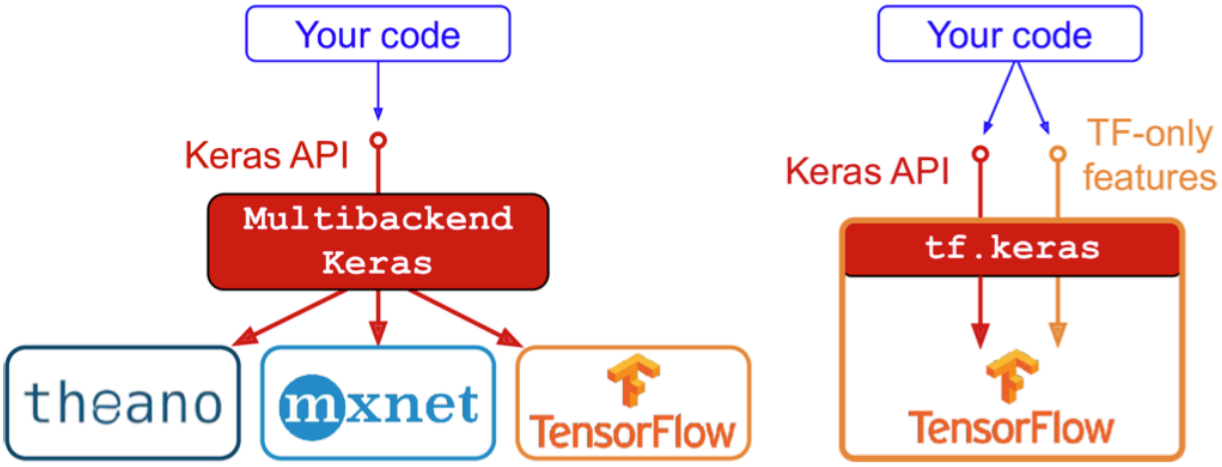
```
2023-02-04 10:21:15.165808: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
2023-02-04 10:21:15.333376: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror:␣
 ↪libcudart.so.11.0: cannot open shared object file: No such file or directory
2023-02-04 10:21:15.333420: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]␣
 ↪Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2023-02-04 10:21:15.371465: E tensorflow/stream_executor/cuda/cuda_blas.cc:2981]␣
 ↪Unable to register cuBLAS factory: Attempting to register factory for plugin␣
 ↪cuBLAS when one has already been registered
2023-02-04 10:21:16.160164: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror:␣
 ↪libnvinfer.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:21:16.160265: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer_plugin.so.7'; dlerror:␣
 ↪libnvinfer_plugin.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:21:16.160276: W tensorflow/compiler/tf2tensorrt/utils/py_utils.
 ↪cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like␣
 ↪to use Nvidia GPU with TensorRT, please make sure the missing libraries␣
 ↪mentioned above are installed properly.
```

12.1 Overview of Keras

Keras is a high-level API for deep learning.

It supports multiple back end deep learning frameworks.

While Keras could be used with TensorFlow 1.0, as of TensorFlow 2.0 it is tightly integrated.



[Credit: Geron]

Keras's API is similar to Facebook's PyTorch library (PyTorch and TensorFlow are the two most popular deep learning libraries). Both are very similar to the Scikit Learn API. So learning Keras will be useful if you want to transition to PyTorch.

Plus, Keras considerably simplifies the use of TensorFlow!

12.2 Sequential API

Keras supports a number of different APIs.

We'll start by considering the *Sequential API*, which is the simplest and simply considers a stack of layers connected sequentially.

12.2.1 Loading data

We'll using the fashion MNIST dataset to illustrate the use of Keras.

Keras includes support to load a number of popular datasets in `keras.datasets`.

Let's use Keras to load fashion MNIST (note that the dataset already includes standard training and test sets).

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
↪train-labels-idx1-ubyte.gz
29515/29515 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
↪train-images-idx3-ubyte.gz
```

(continues on next page)

(continued from previous page)

```
26421880/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
 ↪t10k-labels-idx1-ubyte.gz
5148/5148 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
 ↪t10k-images-idx3-ubyte.gz
4422102/4422102 [==============================] - 0s 0us/step
```

The training set contains 60,000 grayscale images, each 28x28 pixels (similar to the MNIST digit dataset):

```
X_train_full.shape
```

```
(60000, 28, 28)
```

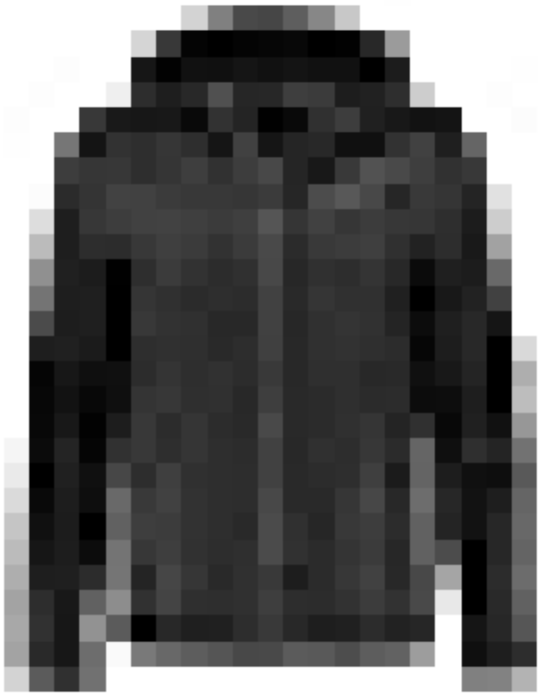
Since no validation set is provided, we'll split the full training set into a validation set and a (smaller) training set.

Let's split the full training set into a validation set and a (smaller) training set (also scale the pixel intensities from 0-255 to 0-1).

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

Plot individual data instance:

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```



The labels are the class IDs (represented as uint8), from 0 to 9:

```
y_train
```

```
array([4, 0, 7, ..., 3, 0, 5], dtype=uint8)
```

With corresponding class names:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

The validation set contains 5,000 images, and the test set contains 10,000 images:

```
X_valid.shape
```

```
(5000, 28, 28)
```

```
X_test.shape
```

```
(10000, 28, 28)
```

Plot a sample of the images in the dataset:

```
n_rows = 4
n_cols = 10
plt.figure(figsize=(n_cols * 1.2, n_rows * 1.2))
for row in range(n_rows):
```

(continues on next page)

(continued from previous page)

```
    for col in range(n_cols):
        index = n_cols * row + col
        plt.subplot(n_rows, n_cols, index + 1)
        plt.imshow(X_train[index], cmap="binary", interpolation="nearest")
        plt.axis('off')
        plt.title(class_names[y_train[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```

12.2.2 Building the model

Let's build a multi-layer perceptron (MLP) with two hidden layers:

```
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"))
```

```
2023-02-04 10:21:21.001836: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.
 ↪1: cannot open shared object file: No such file or directory
2023-02-04 10:21:21.001867: W tensorflow/stream_executor/cuda/cuda_driver.cc:263]␣
 ↪failed call to cuInit: UNKNOWN ERROR (303)
2023-02-04 10:21:21.001890: I tensorflow/stream_executor/cuda/cuda_diagnostics.
 ↪cc:156] kernel driver does not appear to be running on this host (fv-az267-630):␣
 ↪/proc/driver/nvidia/version does not exist
2023-02-04 10:21:21.002205: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
```

Setting up the sequential model consists of the following steps.

1. Set up a sequential model and then we will sequentially add layers to it.
2. Flatten the 28x28 pixel image into a 1D vector. Since it is the first layer, should specify input shape.
3. Add a fully connected dense layer with 300 neurons, including a ReLU activation function.
4. Add another fully connected dense layer with 100 neurons, including a ReLU.
5. Add a final dense layer with 10 neurons so we have one per output class, and include a softmax activation function convert to class probabilities.

Alternatively, the sequential model may be specificed by a list of layers.

```
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")
])
```

Can get a summary of the model using the `summary` method.

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 flatten_1 (Flatten)          (None, 784)               0

 dense_3 (Dense)              (None, 300)               235500

 dense_4 (Dense)              (None, 100)               30100

 dense_5 (Dense)              (None, 10)                1010

=================================================================
Total params: 266,610
Trainable params: 266,610
Non-trainable params: 0
_________________________________________________________________
```

Can extract list of layers from `model.layers`.

```
model.layers
```

```
[<keras.layers.reshaping.flatten.Flatten at 0x7fb291867e20>,
 <keras.layers.core.dense.Dense at 0x7fb291702cd0>,
 <keras.layers.core.dense.Dense at 0x7fb290639190>,
 <keras.layers.core.dense.Dense at 0x7fb290639670>]
```

12.2.3 Compiling the model

Once the model is created we still need to specify the loss function and the optimizer to use to train the model. This is set by *compiling* the model.

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

Since we have a classification problem we will use a categorical cross-entropy loss, with a stochastic gradient descent optimiser.

We will also specify an extra metric to compute during training and evlaution, in this case the accuracy.

You can find further details regarding Keras loss functions, optimisers and metrics in the Keras API.

- Keras loss functions
- Keras optimisers
- Keras metrics

12.2.4 Fitting the model

Call `model.fit` to fit the model, given training and validation data and number of epochs.

The loss and extra metrics will be evaluated on the validation data at the end of each epoch.

```
history = model.fit(X_train, y_train, epochs=10,
                    validation_data=(X_valid, y_valid))
```

```
Epoch 1/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.7265 -␣
 ↪accuracy: 0.7605 - val_loss: 0.5511 - val_accuracy: 0.7956
Epoch 2/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4875 -␣
 ↪accuracy: 0.8289 - val_loss: 0.4500 - val_accuracy: 0.8458
Epoch 3/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4455 -␣
 ↪accuracy: 0.8431 - val_loss: 0.4463 - val_accuracy: 0.8464
Epoch 4/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.4197 -␣
 ↪accuracy: 0.8526 - val_loss: 0.4196 - val_accuracy: 0.8576
Epoch 5/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3988 -␣
 ↪accuracy: 0.8589 - val_loss: 0.3836 - val_accuracy: 0.8658
Epoch 6/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3836 -␣
 ↪accuracy: 0.8655 - val_loss: 0.4207 - val_accuracy: 0.8530
Epoch 7/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3703 -␣
 ↪accuracy: 0.8685 - val_loss: 0.3687 - val_accuracy: 0.8686
Epoch 8/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3587 -␣
 ↪accuracy: 0.8717 - val_loss: 0.3593 - val_accuracy: 0.8736
Epoch 9/10
1719/1719 [==============================] - 5s 3ms/step - loss: 0.3467 -␣
 ↪accuracy: 0.8766 - val_loss: 0.3453 - val_accuracy: 0.8748
```

(continues on next page)

(continued from previous page)

```
Epoch 10/10
1719/1719 [==============================] - 4s 3ms/step - loss: 0.3379 -␣
 ↪accuracy: 0.8791 - val_loss: 0.3457 - val_accuracy: 0.8776
```

12.2.5 Model history

The fit method returns the model history, which contains the training parameters (`history.parms`) and a dictionary containing the loss and additional metrics computed during training (`history.history`).

```
history.params
```

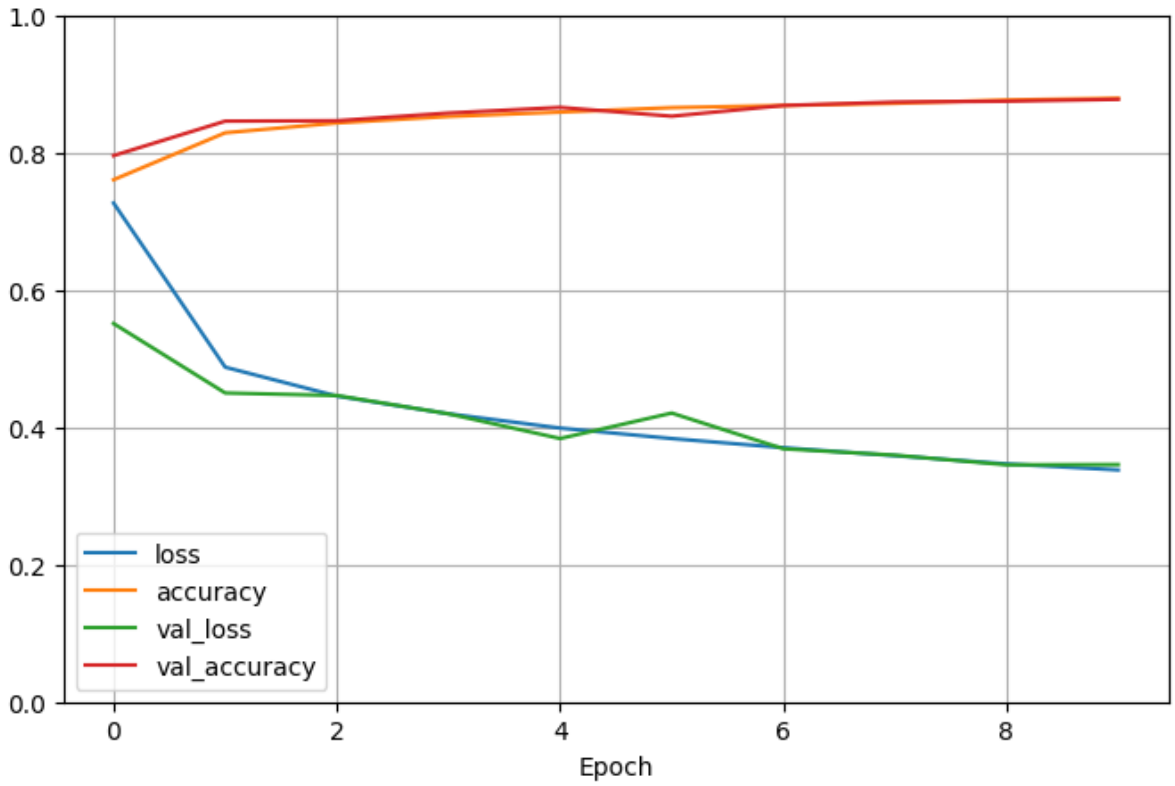
```
{'verbose': 1, 'epochs': 10, 'steps': 1719}
```

```
history.history.keys()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

Let's plot the history.

```
import pandas as pd
pd.DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.xlabel("Epoch")
plt.show()
```



Note that the validation error is computed at the end of each epoch, whereas the training error is computed using a running mean during each epoch.

12.2.6 Evaluating on the test set

We can evaluate the model on the test set using `model.evaluate`.

```
model.evaluate(X_test, y_test)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.3818 - accuracy:␣
↪0.8649
```

```
[0.3817661702632904, 0.8648999929428101]
```

12.2.7 Predictions

We can make predictions on new data using `model.predict`.

Consider the first three images in the test set.

```
X_new = X_test[:3]
```

Estimating class probabilities:

```
y_proba = model.predict(X_new)
y_proba.round(2)
```

```
1/1 [==============================] - 0s 107ms/step
```

```
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.05, 0.  , 0.09, 0.  , 0.86],
       [0.  , 0.  , 0.99, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 1.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ]],
      dtype=float32)
```

Making class predictions:

```
y_pred = np.argmax(model.predict(X_new), axis=-1)
y_pred
```

```
1/1 [==============================] - 0s 20ms/step
```

```
array([9, 2, 1])
```

```
np.array(class_names)[y_pred]
```

```
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
```

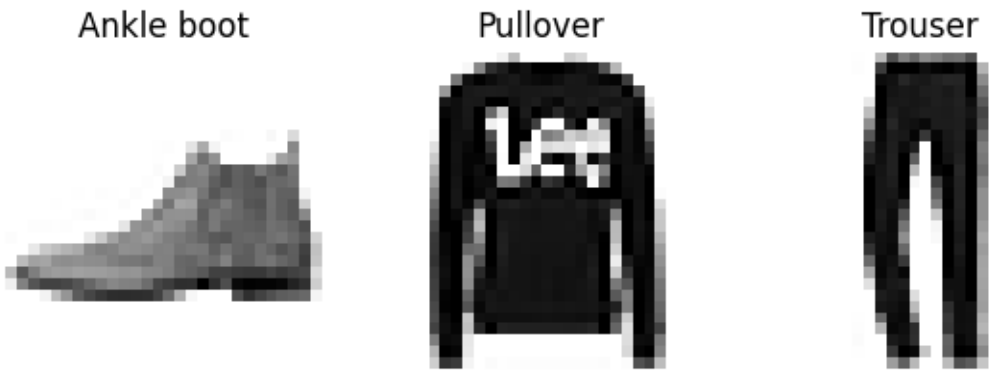
Check with test set targets:

```
y_new = y_test[:3]
y_new
```

```
array([9, 2, 1], dtype=uint8)
```

Let's view the images considered for prediction.

```
plt.figure(figsize=(7.2, 2.4))
for index, image in enumerate(X_new):
    plt.subplot(1, 3, index + 1)
    plt.imshow(image, cmap="binary", interpolation="nearest")
    plt.axis('off')
    plt.title(class_names[y_test[index]], fontsize=12)
plt.subplots_adjust(wspace=0.2, hspace=0.5)
plt.show()
```



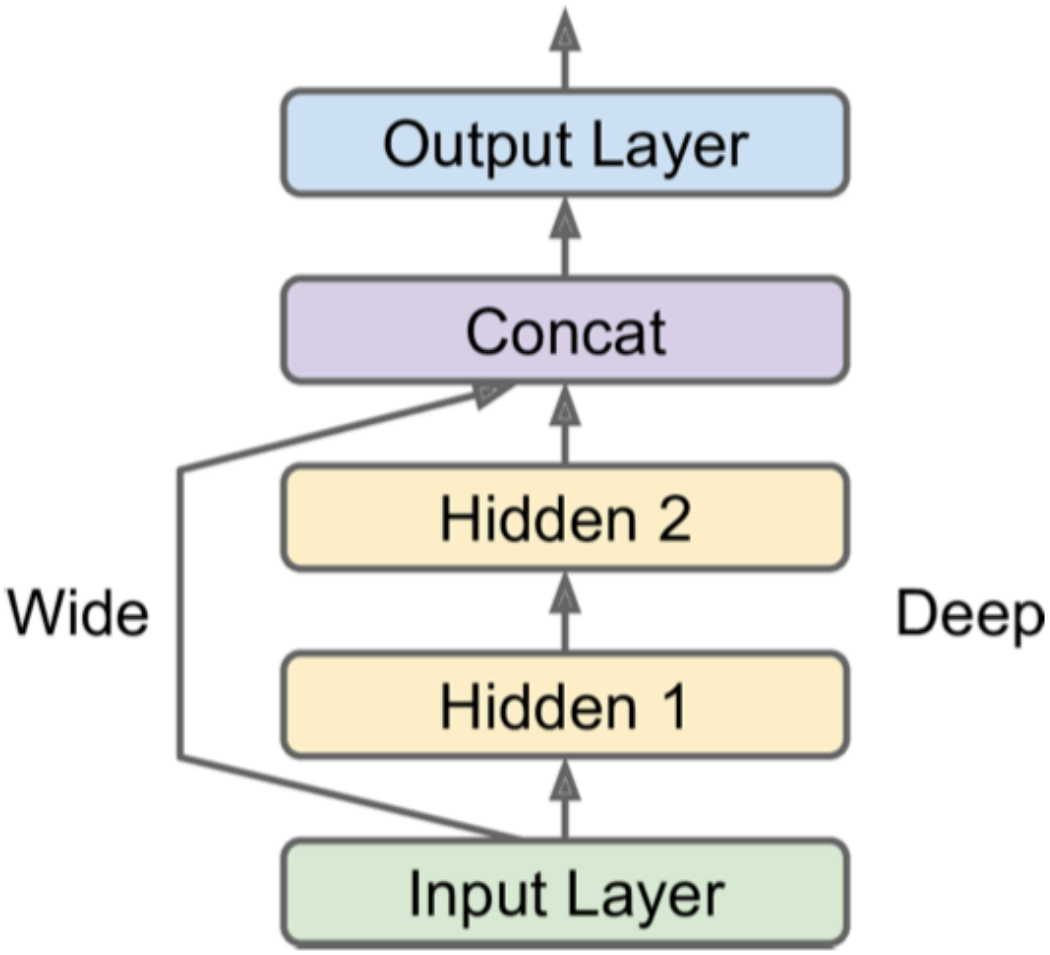
Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

12.3 Functional API

While the *Sequential API* is very easy to use and many models are sequential in nature, it is also somewhat limited.

The *Functional API* allows more complex models to be built.

For example, a Wide & Deep neural network (see paper) connects all or part of the inputs directly to the output layer.



[Credit: Geron]

12.3.1 Loading data

We will illustrate the use of the Functional API to build this model, using the California housing dataset as a regression 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()
```

The dataset contains 8 features (displayed below) contained in `housing.data` that can be used to predict house prices (in units of $100,000), which are contained in `housing.targets`.

```
housing.feature_names
```

```
['MedInc',
 'HouseAge',
 'AveRooms',
```

(continues on next page)

(continued from previous page)

```
 'AveBedrms',
 'Population',
 'AveOccup',
 'Latitude',
 'Longitude']
```

Split the data into training, validation and test sets.

```
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)
```

Scale features to standardise.

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

12.3.2 Building the model

```
input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.models.Model(inputs=[input_], outputs=[output])
```

Setting up the functional model consists of the following steps.

1. Set up the input of the model.
2. Create a dense layer with 30 neurons that takes `_input` as the input, and includes a ReLU activation function.
3. Create a dense layer with 30 neurons that takes the output of the previous layer as an input, and includes a ReLU activation function.
4. Concatenate the input and the output of the second hidden layer.
5. Create a dense layer that takes the concatenated data as an input and outputs a single value (for the regression model).
6. Build the model specifying the inputs and outputs (of which there can be multiple).

Note that each component is called like a function, mapping inputs to outputs, hence the name of the *Functional API*.

```
model.summary()
```

```
Model: "model"
_________________________________________________________________________________
↪_________________
 Layer (type)                 Output Shape              Param #     Connected to
=================================================================================
```

(continues on next page)

(continued from previous page)

```
 input_1 (InputLayer)           [(None, 8)]          0           []

 dense_6 (Dense)                (None, 30)           270         ['input_1[0][0]']

 dense_7 (Dense)                (None, 30)           930         ['dense_6[0][0]']

 concatenate (Concatenate)      (None, 38)           0           ['input_1[0][0]',
                                                                  'dense_7[0][0]']

 dense_8 (Dense)                (None, 1)            39          [
↪'concatenate[0][0]']

==================================================================================================
Total params: 1,239
Trainable params: 1,239
Non-trainable params: 0
__________________________________________________________________________________________________
 ↪________________
```

12.3.3 Compiling the model

Once the model is created we again need to *compile* the mode to specify the loss function and the optimizer to use to train the model.

```
model.compile(loss="mean_squared_error", optimizer=keras.optimizers.SGD(lr=1e-3))
```

```
/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/keras/
 ↪optimizers/optimizer_v2/gradient_descent.py:111: UserWarning: The `lr` argument␣
 ↪is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

12.3.4 Fitting the model

Again we fit the model, given training and validation data and number of epochs, and compute the loss on the validation data at the end of each epoch.

```
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

12.3.5 Evaluating on the test set

We evaluate the model on the test set as before.

```
mse_test = model.evaluate(X_test, y_test)
```

```
162/162 [==============================] - 0s 1ms/step - loss: 0.3985
```

12.3.6 Predictions

We again make predictions on new data (the first three instances of the test set) using `model.predict`.

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

```
1/1 [==============================] - 0s 60ms/step
```

```
array([[0.7423509],
       [1.5806545],
       [3.546597 ]], dtype=float32)
```

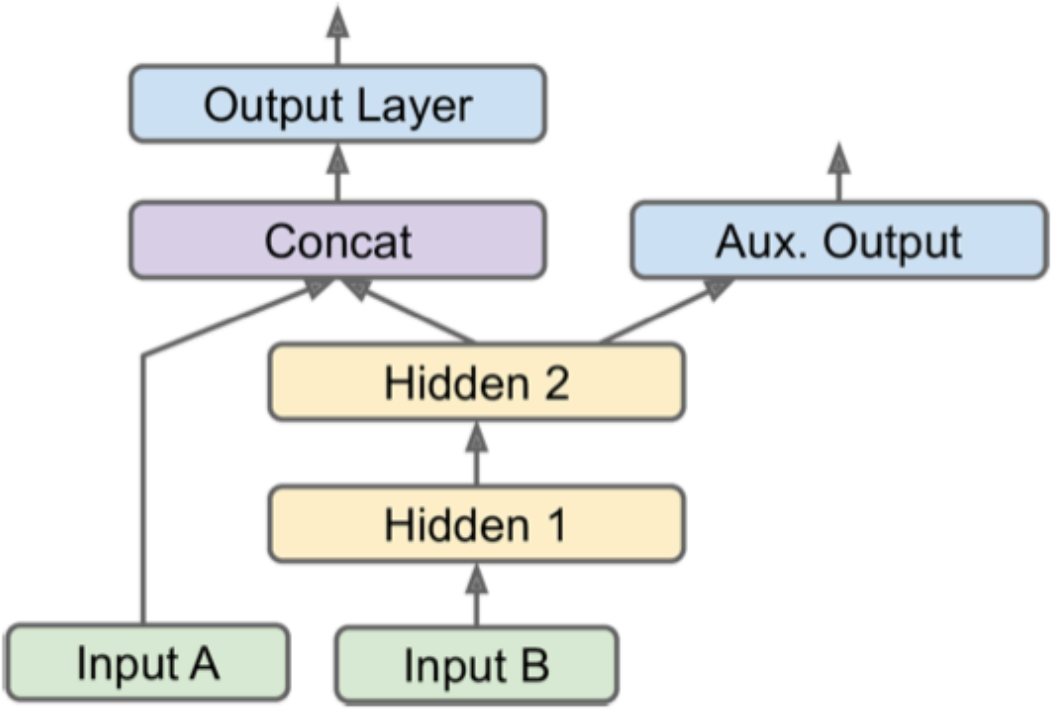
12.4 Subclassing API

Both the *Sequential API* and the *Functional API* support static models: they involving defining a model and then feeding it data for training or inference.

The *Subclassing API* allows you to subclass Keras models (and other objects such as layers). This supports dynamic models and allows you to construct alternative model objects that can be widely used elsewhere.

12.4.1 Building the model

Let's use the *Subclassing API* to build a Wide & Deep neural network with two inputs and two outputs as an example.



[Credit: Geron]

Firstly, let's spit our data so we have two sets of inputs.

```
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]
```

A model class can then be written to implement the model, inheriting from `keras.models.Model`.

```
class WideAndDeepModel(keras.models.Model):
    def __init__(self, units=30, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(units, activation=activation)
        self.hidden2 = keras.layers.Dense(units, activation=activation)
        self.main_output = keras.layers.Dense(1)
        self.aux_output = keras.layers.Dense(1)

    def call(self, inputs):
        input_A, input_B = inputs
        hidden1 = self.hidden1(input_B)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_A, hidden2])
        main_output = self.main_output(concat)
        aux_output = self.aux_output(hidden2)
        return main_output, aux_output

model = WideAndDeepModel(30, activation="relu")
```

The model must have `__init__` and `call` methods. Within `__init__` you need to instantiate the layers that you need, before you make use of them within `call`.

Since the `call` method is implemented directly, there is a great deal of flexibility to implement complex models.

The model can then be used like other Keras model and you can compile and fit it as usual.

Note that for bespoke models implemented this way, Keras cannot easily inspect it. Thus, it is not so straightforward, e.g., to save the model or to perform type and shape checking.

12.4.2 Compiling the model

```
model.compile(loss="mse", loss_weights=[0.9, 0.1], optimizer=keras.optimizers.
↪SGD(lr=1e-3))
```

12.4.3 Fitting the model

```
history = model.fit((X_train_A, X_train_B), (y_train, y_train), epochs=10,
                    validation_data=((X_valid_A, X_valid_B), (y_valid, y_valid)))
```

12.4.4 Evaluating and making predictions

```
total_loss, main_loss, aux_loss = model.evaluate((X_test_A, X_test_B), (y_test, y_
↪test))
y_pred_main, y_pred_aux = model.predict((X_new_A, X_new_B))
```

12.5 Saving and restoring models

Keras can be used to easily save and restore models when using the *Sequential API* of the *Functional API*.

Models can of course be saved and restored when using the *Subclassing API* but it is necessary to use underlying TensorFlow methods and is not quite so straightforward (not covered further here).

Let's build and train a simple model and then save and restore it.

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=[8]),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
])
```

```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
history = model.fit(X_train, y_train, epochs=10, validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
```

12.5.1 Saving

Saving the model is simply one line.

```
model.save("my_keras_model.h5")
```

The model is then saved using the HDF5 format.

The model's achitecture, values of the model parameters, the optimiser, hyperparameters, and state are all saved.

12.5.2 Restoring

Restoring the model is also just one line.

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

The model can then be used as usual, e.g. to make predictions.

```
model.predict(X_new)
```

```
1/1 [==============================] - 0s 54ms/step
```

```
array([[0.5763352],
       [1.5386071],
       [3.1901321]], dtype=float32)
```

12.5.3 Checkpointing

Since training may take a long time, you might like to save the model weights as training progresses so that you do not lose the results if the machine crashes or disconnects.

Model weights can also be saved and loaded in a single line.

```
model.save_weights("my_keras_weights.ckpt")
```

```
model.load_weights("my_keras_weights.ckpt")
```

```
<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x7fb29177e4f0>
```

12.6 TensorBoard

TensorBoard is an interactive visualisation tool that you can use to review learning curves, analyse training statistics and inspect the architecture of your model.

12.6.1 Logging

Let's log a training run and model and then view in TensorBoard.

```
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()
run_logdir
```

```
'./my_logs/run_2023_02_04-10_22_41'
```

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

```
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu", input_shape=[8]),
    keras.layers.Dense(30, activation="relu"),
    keras.layers.Dense(1)
```

(continues on next page)

(continued from previous page)

```
])
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
```

To log training as it progresses we need to set up `callback`.

The `fit` methods allows a list of `callbacks` to be specified, which can then be called at the start and end of each epoch.

Let's set up two callbacks:

- One to checkpoint the model as we go, saving the best model considered so far, as evaluated on the validation set.
- One to log TensorBoard data.

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5", save_best_
↪only=True)
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
```

The callbacks are then simply passed to the `fit` method.

```
history = model.fit(X_train, y_train, epochs=30,
                    validation_data=(X_valid, y_valid),
                    callbacks=[checkpoint_cb, tensorboard_cb])
```

12.6.2 Visualisation

TensorBoard then needs to be run to visualise the logs. It simply runs in a browser.

TensorBoard can be launched from your terminal by:

```
tensorboard --logdir=./my_logs --port=6001
```

We can then launch TensorBoard in a browser at localhost:6001.

Let's switch to TensorBoard to view the logs and inspect the computational graph of the model.

CHAPTER

THIRTEEN

LECTURE 13: TRAINING DEEP NEURAL NETWORKS


Run in colab

```
import datetime
now = datetime.datetime.now()
print("Last executed: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Last executed: 2023-02-04 10:23:39
```

```
# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
def reset_state(seed=42):
    tf.keras.backend.clear_session()
    tf.random.set_seed(seed)
    np.random.seed(seed)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

13.1 Vanishing and exploding gradients

Training typically relies on gradients.

Vanishing gradients problem: For deep networks, gradients in lower layers can become very small. Hence, corresponding weights are not updated during training.

Exploding gradients problem: In some situations (typically recurrent neural networks) gradients can become very large. Hence, weight updates are very large and the training algorithm may not converge.

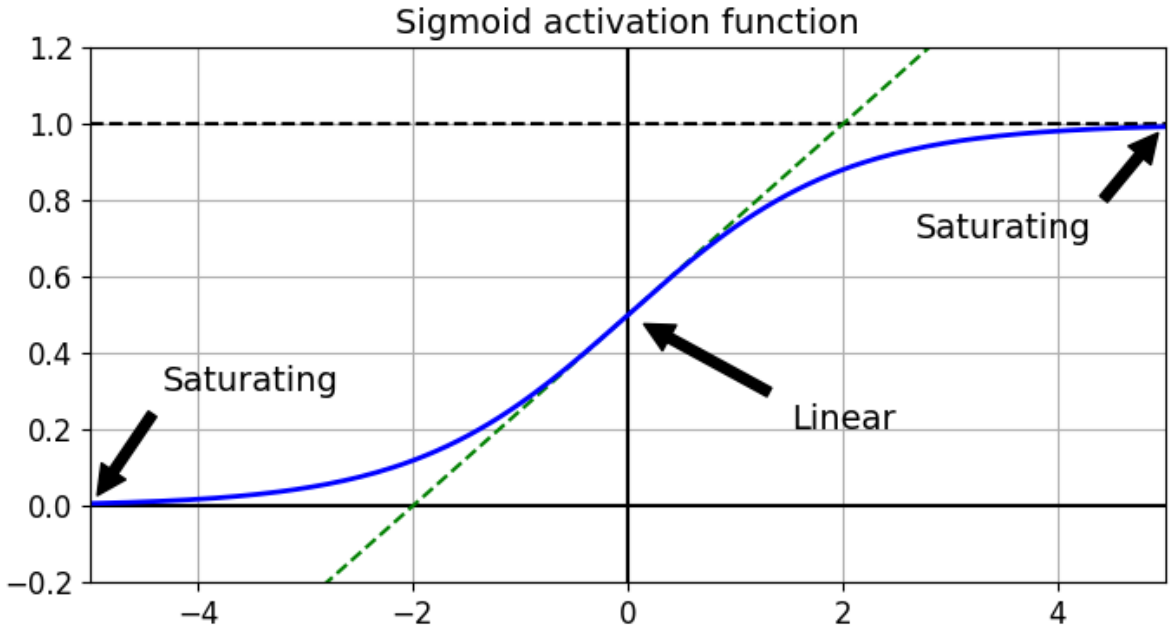
In general deep neural networks can suffer from *unstable gradients*.

13.1.1 Problematic activation functions

One common cause of vanishing gradients in the past was the use of the sigmoid activation function (and unit Gaussian initialisation).

```
def logit(z):
    return 1 / (1 + np.exp(-z))
```

```
z = np.linspace(-5, 5, 200)

plt.figure(figsize=(8,4))
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [1, 1], 'k--')
plt.plot([0, 0], [-0.2, 1.2], 'k-')
plt.plot([-5, 5], [-3/4, 7/4], 'g--')
plt.plot(z, logit(z), "b-", linewidth=2)
props = dict(facecolor='black', shrink=0.1)
plt.annotate('Saturating', xytext=(3.5, 0.7), xy=(5, 1), arrowprops=props,␣
 ↪fontsize=14, ha="center")
plt.annotate('Saturating', xytext=(-3.5, 0.3), xy=(-5, 0), arrowprops=props,␣
 ↪fontsize=14, ha="center")
plt.annotate('Linear', xytext=(2, 0.2), xy=(0, 0.5), arrowprops=props, fontsize=14,␣
 ↪ha="center")
plt.grid(True)
plt.title("Sigmoid activation function", fontsize=14)
plt.axis([-5, 5, -0.2, 1.2]);
```



Variance of outputs grows at each layer. Final layers essentially saturate. Gradients on final layers then very small and when propagate gradients back with back-propagation then get vanishing gradients.

13.1.2 Weight initialisation

To avoid this problem need signals and gradients to *not* decay as propagating through network.

Avoid decaying signals/gradients by promoting equal variance at outputs and inputs of layer.

Can be promoted by random initialisation of weights to follow Gaussian with standard deviation:

$$\text{Sigmoid activation:} \quad \sigma = \sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}} \tag{13.1}$$

$$\text{Hyperbolic tangent activation:} \quad \sigma = 4\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}} \tag{13.2}$$

$$\text{ReLU activation:} \quad \sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}} \tag{13.3}$$

$$\tag{13.4}$$

where n_{inputs} and n_{outputs} are the number of input and output nodes, respectively, for the layer.

There are a lot of different weight initialisation strategies.

Weight initialisation in TensorFlow

```
import tensorflow as tf
from tensorflow import keras
```

```
2023-02-04 10:23:40.500156: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
2023-02-04 10:23:40.661228: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror:␣
 ↪libcudart.so.11.0: cannot open shared object file: No such file or directory
2023-02-04 10:23:40.661252: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]␣
 ↪Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2023-02-04 10:23:40.698990: E tensorflow/stream_executor/cuda/cuda_blas.cc:2981]␣
 ↪Unable to register cuBLAS factory: Attempting to register factory for plugin␣
 ↪cuBLAS when one has already been registered
2023-02-04 10:23:41.530957: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror:␣
 ↪libnvinfer.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:23:41.531049: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer_plugin.so.7'; dlerror:␣
 ↪libnvinfer_plugin.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:23:41.531060: W tensorflow/compiler/tf2tensorrt/utils/py_utils.
 ↪cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like␣
 ↪to use Nvidia GPU with TensorRT, please make sure the missing libraries␣
 ↪mentioned above are installed properly.
```

```
[name for name in dir(keras.initializers) if not name.startswith("_")]
```

```
['Constant',
 'GlorotNormal',
 'GlorotUniform',
```

(continues on next page)

(continued from previous page)

```
 'HeNormal',
 'HeUniform',
 'Identity',
 'Initializer',
 'LecunNormal',
 'LecunUniform',
 'Ones',
 'Orthogonal',
 'RandomNormal',
 'RandomUniform',
 'TruncatedNormal',
 'VarianceScaling',
 'Zeros',
 'constant',
 'deserialize',
 'get',
 'glorot_normal',
 'glorot_uniform',
 'he_normal',
 'he_uniform',
 'identity',
 'lecun_normal',
 'lecun_uniform',
 'ones',
 'orthogonal',
 'random_normal',
 'random_uniform',
 'serialize',
 'truncated_normal',
 'variance_scaling',
 'zeros']
```

Can often simply set initialiser when defining layer.

```
reset_state()

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

```
<keras.layers.core.dense.Dense at 0x7f326c839850>
```

Or can set up a `VarianceScaling` object directly.

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

```
<keras.layers.core.dense.Dense at 0x7f31fdbb4340>
```

13.1.3 Non-saturating activation functions

ReLU activation behaves much better than the sigmoid in deep networks since it does not saturate for positive values (and it is fast to compute).

However, the ReLU does suffer from the *dying neuron* problem.

In this senario neurons effectively die and only output zero. The neuron is unlikely to come back to life since the gradient of the ReLU activation function is zero for negative inputs.

Leaky ReLU

The *leaky ReLU* avoids this problem and is defined by

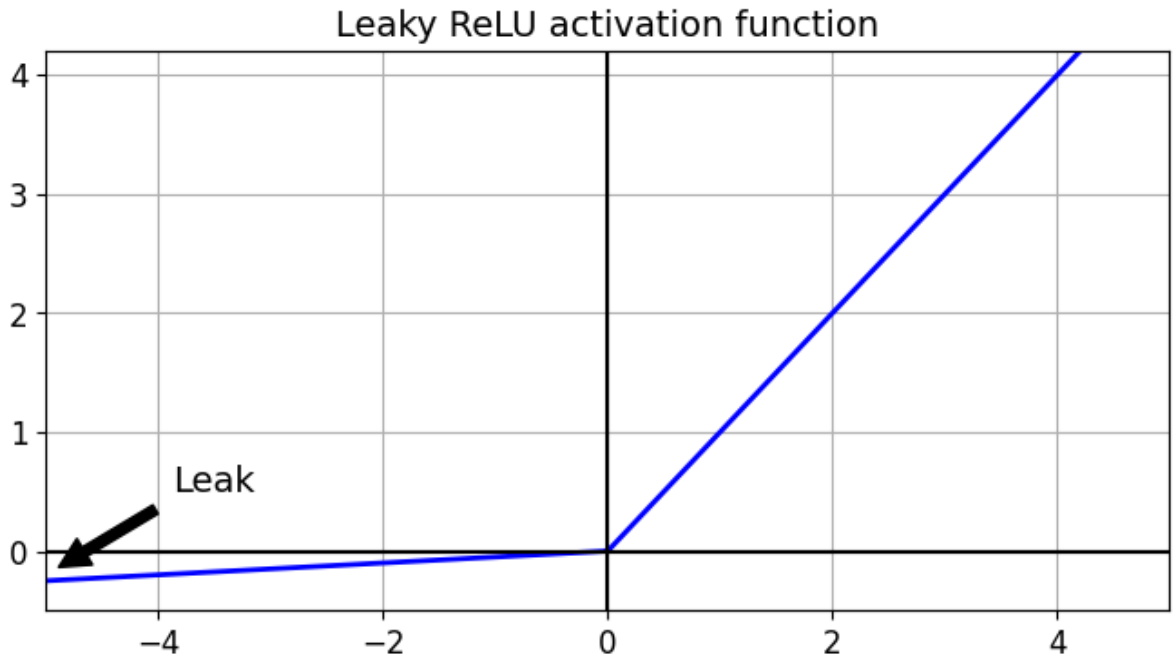
$$\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z),$$

where the hyperparameter α defines how much the leaky ReLU leaks (typically $\alpha = 0.01$).

Let's plot the Leaky ReLU activation function for $\alpha = 0.05$.

```
def leaky_relu(z, alpha=0.01):
    return np.maximum(alpha*z, z)
```

```
z = np.linspace(-5, 5, 200)

plt.figure(figsize=(8,4))
plt.plot(z, leaky_relu(z, 0.05), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([0, 0], [-0.5, 4.2], 'k-')
plt.grid(True)
props = dict(facecolor='black', shrink=0.1)
plt.annotate('Leak', xytext=(-3.5, 0.5), xy=(-5, -0.2), arrowprops=props, fontsize=14,
 ↪ ha="center")
plt.title("Leaky ReLU activation function", fontsize=14)
plt.axis([-5, 5, -0.5, 4.2]);
```



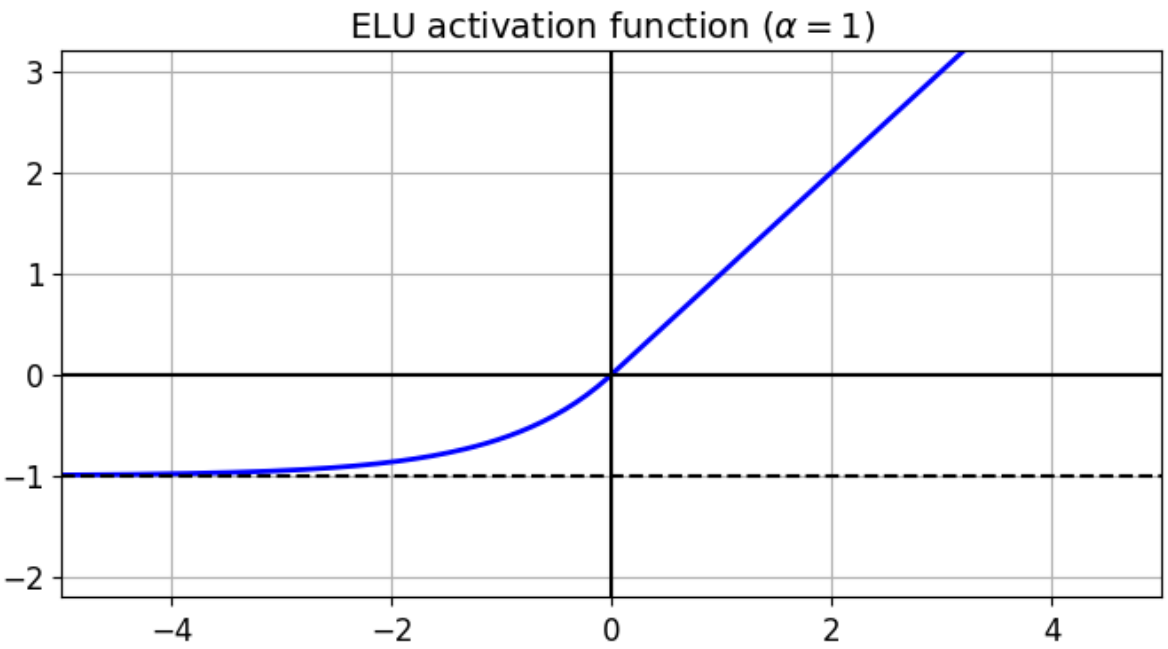
ELU

Another alternative is the *exponental linear unit* (ELU).

```
def elu(z, alpha=1):
    return np.where(z < 0, alpha * (np.exp(z) - 1), z)
```

Let's plot the ELU activation function for $\alpha = 1$.

```
plt.figure(figsize=(8,4))
plt.plot(z, elu(z), "b-", linewidth=2)
plt.plot([-5, 5], [0, 0], 'k-')
plt.plot([-5, 5], [-1, -1], 'k--')
plt.plot([0, 0], [-2.2, 3.2], 'k-')
plt.grid(True)
plt.title(r"ELU activation function ($\alpha=1$)", fontsize=14)
plt.axis([-5, 5, -2.2, 3.2]);
```



Properties:

- Non-zero gradient for $z < 0$ to avoid dying neuron issue.
- Smooth so gradients well defined.
- But is slower to compute.

Activations functions in TensorFlow

TensorFlow supports a lot of activation functions.

```
[m for m in dir(keras.activations) if not m.startswith("_")]
```

```
['deserialize',
 'elu',
 'exponential',
 'gelu',
 'get',
 'hard_sigmoid',
 'linear',
 'relu',
 'selu',
 'serialize',
 'sigmoid',
 'softmax',
 'softplus',
 'softsign',
 'swish',
 'tanh']
```

Can again simply set when definiting layer or can construct directly.

```
reset_state()
keras.layers.Dense(10, activation="elu", name="hidden1")
```

```
<keras.layers.core.dense.Dense at 0x7f31f4a2ad00>
```

```
reset_state()
keras.layers.Dense(10, activation=keras.layers.Activation("elu"), name="hidden1")
```

```
<keras.layers.core.dense.Dense at 0x7f31f4a31880>
```

13.2 Batch normalisation

While weight normalisation can reduce gradient problems at the beginning of training, it does not guarantee that these problems won't resurface during training.

Batch normalisation adds normalisation during training to address these issues.

Consists of zero-centering and normalising inputs just before the activation function, followed by shifting and scaling the result. The shift and scale are considered additional parameters that are learnt during training.

This approach allows training to select the appropriate scale and shift (mean) for each layer.

The mean and standard deviation of the unnormalised inputs are computed for each mini-batch, hence the name *batch normalisation*.

When the trained network is applied to the test set there are no batches, so instead a running mean and standard deviation computed on the *training* set are used.

13.2.1 Batch normalisation in TensorFlow

```
reset_state()

import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

model = keras.models.Sequential([
keras.layers.Flatten(input_shape=[28, 28]), keras.layers.BatchNormalization(),
keras.layers.Dense(n_hidden1, activation="elu", kernel_initializer="he_normal"),␣
 ↪keras.layers.BatchNormalization(),
keras.layers.Dense(n_hidden2, activation="elu", kernel_initializer="he_normal"),␣
 ↪keras.layers.BatchNormalization(),
keras.layers.Dense(n_outputs, activation="softmax")
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 batch_normalization (BatchN  (None, 784)              3136
 ormalization)

 dense (Dense)               (None, 300)               235500

 batch_normalization_1 (Batc  (None, 300)              1200
 hNormalization)

 dense_1 (Dense)             (None, 100)               30100

 batch_normalization_2 (Batc  (None, 100)              400
 hNormalization)

 dense_2 (Dense)             (None, 10)                1010

=================================================================
Total params: 271,346
Trainable params: 268,978
Non-trainable params: 2,368
_________________________________________________________________
```

```
2023-02-04 10:23:43.397438: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.
 ↪1: cannot open shared object file: No such file or directory
2023-02-04 10:23:43.397472: W tensorflow/stream_executor/cuda/cuda_driver.cc:263]␣
 ↪failed call to cuInit: UNKNOWN ERROR (303)
2023-02-04 10:23:43.397497: I tensorflow/stream_executor/cuda/cuda_diagnostics.
 ↪cc:156] kernel driver does not appear to be running on this host (fv-az267-630):␣
 ↪/proc/driver/nvidia/version does not exist
2023-02-04 10:23:43.397737: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
```

```
[(var.name, var.trainable) for var in model.layers[1].variables]
```

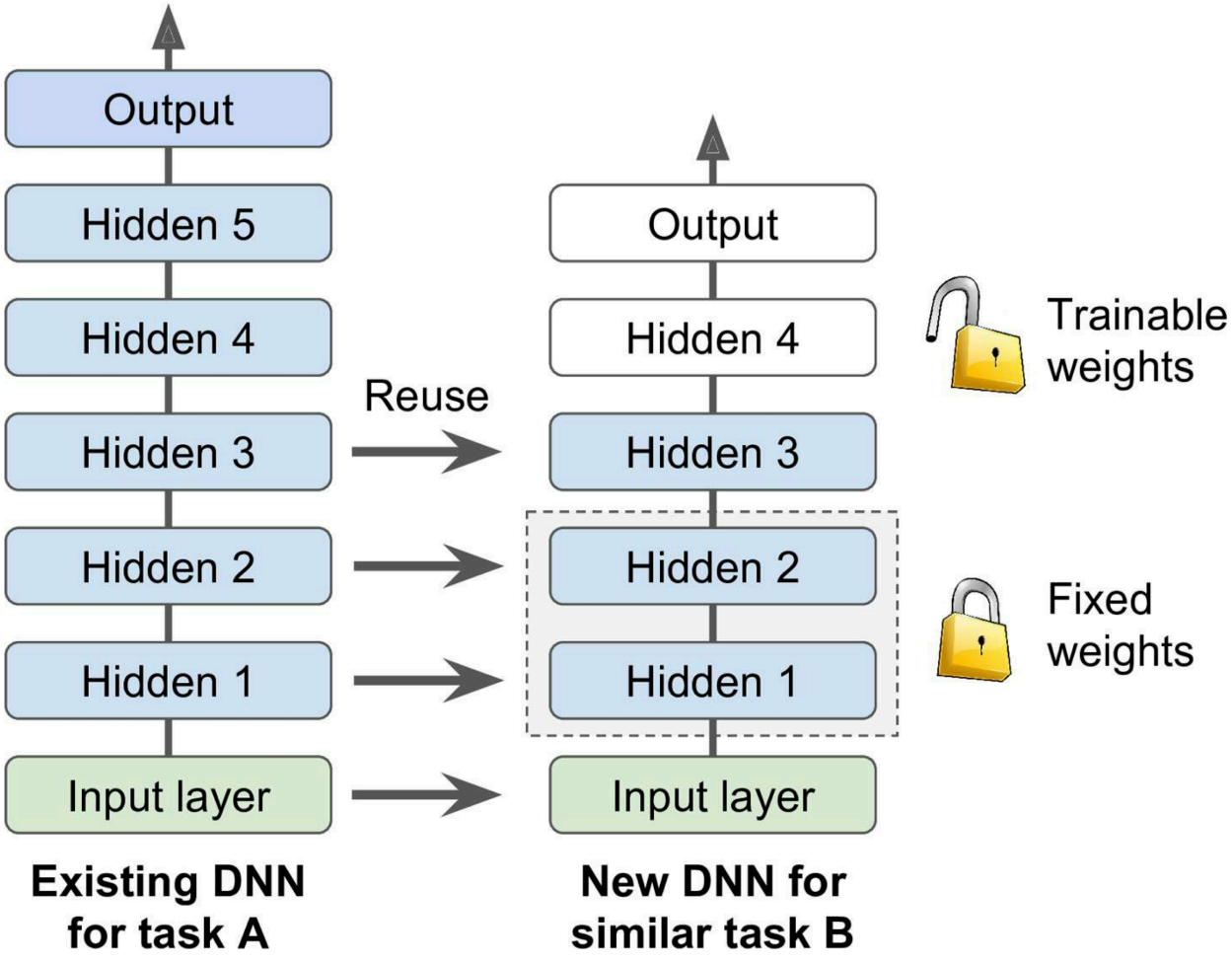
```
[('batch_normalization/gamma:0', True),
 ('batch_normalization/beta:0', True),
 ('batch_normalization/moving_mean:0', False),
 ('batch_normalization/moving_variance:0', False)]
```

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

13.3 Pretraining and transfer learning

A deep network trained for one task can often be adapted for a similar task.

Reuse lower layers of network trained for another task.



[Credit: Geron]

For transfer learning to be successful the data must have similar low-level features.

13.3.1 Reusing a Keras model

Let's work through a transfer learning example.

Split the fashion MNIST training set into two:

- `X_train_A`: all images of all items, except sandals and shirts (classes 5 and 6).
- `X_train_B`: first 200 images of sandals or shirts.

The validation set and the test set are split similarly, but without restricting the number of images.

Dataset B corresponds to a simple problem (binary classification) but we only have a small number of training instances.

Dataset A corresponds to a more difficult problem (classification between 8 classes) but we have much more data.

We will attempt to transfer knowledge from setting A to B, since classes in set A (sneakers, ankle boots, coats, t-shirts, etc.) are somewhat similar to classes in set B (sandals and shirts).

Aside: Note that only patterns that occur in the same location can be reused since we are using `Dense` layers (CNNs will be much more effective in tranferring information detected anywhere in the image due to their translational equivariance properties, as we'll see in the CNN lecture).

Set up data

```
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_
↪data()
X_train_full = X_train_full / 255.0
X_test = X_test / 255.0
X_valid, X_train = X_train_full[:5000], X_train_full[5000:]
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]

def split_dataset(X, y):
    y_5_or_6 = (y == 5) | (y == 6) # sandals or shirts
    y_A = y[~y_5_or_6]
    y_A[y_A > 6] -= 2 # class indices 7, 8, 9 should be moved to 5, 6, 7
    y_B = (y[y_5_or_6] == 6).astype(np.float32) # binary classification task: is it a␣
↪shirt (class 6)?
    return ((X[~y_5_or_6], y_A),
            (X[y_5_or_6], y_B))

(X_train_A, y_train_A), (X_train_B, y_train_B) = split_dataset(X_train, y_train)
(X_valid_A, y_valid_A), (X_valid_B, y_valid_B) = split_dataset(X_valid, y_valid)
(X_test_A, y_test_A), (X_test_B, y_test_B) = split_dataset(X_test, y_test)
X_train_B = X_train_B[:200]
y_train_B = y_train_B[:200]
```

```
X_train_A.shape, X_train_B.shape
```

```
((43986, 28, 28), (200, 28, 28))
```

```
y_train_A[:30]
```

```
array([4, 0, 5, 7, 7, 7, 4, 4, 3, 4, 0, 1, 6, 3, 4, 3, 2, 6, 5, 3, 4, 5,
       1, 3, 4, 2, 0, 6, 7, 1], dtype=uint8)
```

```
y_train_B[:30]
```

```
array([1., 1., 0., 0., 0., 0., 1., 1., 1., 0., 0., 1., 1., 0., 0., 0., 0.,
       0., 0., 1., 1., 0., 0., 1., 1., 0., 1., 1., 1., 1.], dtype=float32)
```

Define, compile, fit and save model on dataset A

```
reset_state()
model_A = keras.models.Sequential()
model_A.add(keras.layers.Flatten(input_shape=[28, 28]))
for n_hidden in (300, 100, 50, 50, 50):
    model_A.add(keras.layers.Dense(n_hidden, activation="selu"))
model_A.add(keras.layers.Dense(8, activation="softmax"))
```

```
model_A.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, 50)                5050

 dense_3 (Dense)             (None, 50)                2550

 dense_4 (Dense)             (None, 50)                2550

 dense_5 (Dense)             (None, 8)                 408

=================================================================
Total params: 276,158
Trainable params: 276,158
Non-trainable params: 0
_________________________________________________________________
```

```
model_A.compile(loss="sparse_categorical_crossentropy",
                optimizer=keras.optimizers.SGD(lr=1e-3),
                metrics=["accuracy"])
```

```
/opt/hostedtoolcache/Python/3.8.16/x64/lib/python3.8/site-packages/keras/
 ↪optimizers/optimizer_v2/gradient_descent.py:111: UserWarning: The `lr` argument␣
 ↪is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

```
history = model_A.fit(X_train_A, y_train_A, epochs=20,
                      validation_data=(X_valid_A, y_valid_A))
```

```
Epoch 1/20
1375/1375 [==============================] - 5s 3ms/step - loss: 0.5524 -␣
 ↪accuracy: 0.8215 - val_loss: 0.3863 - val_accuracy: 0.8647
Epoch 2/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.3496 -␣
 ↪accuracy: 0.8789 - val_loss: 0.3267 - val_accuracy: 0.8901
```

(continues on next page)

(continued from previous page)

```
Epoch 3/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.3170 -␣
 ↪accuracy: 0.8898 - val_loss: 0.3016 - val_accuracy: 0.8959
Epoch 4/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2984 -␣
 ↪accuracy: 0.8967 - val_loss: 0.2879 - val_accuracy: 0.9053
Epoch 5/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2852 -␣
 ↪accuracy: 0.9012 - val_loss: 0.2785 - val_accuracy: 0.9063
Epoch 6/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2751 -␣
 ↪accuracy: 0.9041 - val_loss: 0.2755 - val_accuracy: 0.9056
Epoch 7/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2666 -␣
 ↪accuracy: 0.9078 - val_loss: 0.2673 - val_accuracy: 0.9068
Epoch 8/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2596 -␣
 ↪accuracy: 0.9111 - val_loss: 0.2595 - val_accuracy: 0.9118
Epoch 9/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2537 -␣
 ↪accuracy: 0.9123 - val_loss: 0.2550 - val_accuracy: 0.9116
Epoch 10/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2484 -␣
 ↪accuracy: 0.9151 - val_loss: 0.2562 - val_accuracy: 0.9138
Epoch 11/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2438 -␣
 ↪accuracy: 0.9162 - val_loss: 0.2487 - val_accuracy: 0.9131
Epoch 12/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2395 -␣
 ↪accuracy: 0.9180 - val_loss: 0.2472 - val_accuracy: 0.9143
Epoch 13/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2361 -␣
 ↪accuracy: 0.9192 - val_loss: 0.2436 - val_accuracy: 0.9170
Epoch 14/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2324 -␣
 ↪accuracy: 0.9202 - val_loss: 0.2402 - val_accuracy: 0.9193
Epoch 15/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2295 -␣
 ↪accuracy: 0.9216 - val_loss: 0.2409 - val_accuracy: 0.9178
Epoch 16/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2261 -␣
 ↪accuracy: 0.9229 - val_loss: 0.2362 - val_accuracy: 0.9170
Epoch 17/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2236 -␣
 ↪accuracy: 0.9236 - val_loss: 0.2440 - val_accuracy: 0.9160
Epoch 18/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2205 -␣
 ↪accuracy: 0.9248 - val_loss: 0.2447 - val_accuracy: 0.9133
Epoch 19/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2182 -␣
 ↪accuracy: 0.9253 - val_loss: 0.2331 - val_accuracy: 0.9188
Epoch 20/20
1375/1375 [==============================] - 4s 3ms/step - loss: 0.2160 -␣
 ↪accuracy: 0.9270 - val_loss: 0.2356 - val_accuracy: 0.9165
```

We achieve an accuracy ~92%, which is reasonable.

```
model_A.save("my_model_A.h5")
```

Repeat on dataset B

```
model_B = keras.models.Sequential()
model_B.add(keras.layers.Flatten(input_shape=[28, 28]))
for n_hidden in (300, 100, 50, 50, 50):
    model_B.add(keras.layers.Dense(n_hidden, activation="selu"))
model_B.add(keras.layers.Dense(1, activation="sigmoid"))
```

```
model_B.compile(loss="binary_crossentropy",
                optimizer=keras.optimizers.SGD(lr=1e-3),
                metrics=["accuracy"])
```

```
history = model_B.fit(X_train_B, y_train_B, epochs=20,
                      validation_data=(X_valid_B, y_valid_B))
```

```
Epoch 1/20
7/7 [==============================] - 1s 48ms/step - loss: 0.6065 - accuracy: 0.
 ↪6700 - val_loss: 0.4980 - val_accuracy: 0.7424
Epoch 2/20
7/7 [==============================] - 0s 16ms/step - loss: 0.4511 - accuracy: 0.
 ↪8100 - val_loss: 0.3888 - val_accuracy: 0.8671
Epoch 3/20
7/7 [==============================] - 0s 15ms/step - loss: 0.3506 - accuracy: 0.
 ↪8650 - val_loss: 0.3238 - val_accuracy: 0.9016
Epoch 4/20
7/7 [==============================] - 0s 16ms/step - loss: 0.2899 - accuracy: 0.
 ↪9250 - val_loss: 0.2773 - val_accuracy: 0.9300
Epoch 5/20
7/7 [==============================] - 0s 16ms/step - loss: 0.2447 - accuracy: 0.
 ↪9350 - val_loss: 0.2431 - val_accuracy: 0.9381
Epoch 6/20
7/7 [==============================] - 0s 16ms/step - loss: 0.2101 - accuracy: 0.
 ↪9650 - val_loss: 0.2177 - val_accuracy: 0.9503
Epoch 7/20
7/7 [==============================] - 0s 15ms/step - loss: 0.1858 - accuracy: 0.
 ↪9700 - val_loss: 0.1985 - val_accuracy: 0.9554
Epoch 8/20
7/7 [==============================] - 0s 17ms/step - loss: 0.1664 - accuracy: 0.
 ↪9750 - val_loss: 0.1825 - val_accuracy: 0.9584
Epoch 9/20
7/7 [==============================] - 0s 16ms/step - loss: 0.1510 - accuracy: 0.
 ↪9900 - val_loss: 0.1690 - val_accuracy: 0.9635
Epoch 10/20
7/7 [==============================] - 0s 15ms/step - loss: 0.1377 - accuracy: 0.
 ↪9850 - val_loss: 0.1584 - val_accuracy: 0.9675
Epoch 11/20
7/7 [==============================] - 0s 16ms/step - loss: 0.1274 - accuracy: 0.
 ↪9900 - val_loss: 0.1490 - val_accuracy: 0.9686
Epoch 12/20
7/7 [==============================] - 0s 16ms/step - loss: 0.1181 - accuracy: 0.
 ↪9900 - val_loss: 0.1412 - val_accuracy: 0.9716
```

(continues on next page)

(continued from previous page)

```
Epoch 13/20
7/7 [==============================] - 0s 19ms/step - loss: 0.1104 - accuracy: 0.
 ↪9900 - val_loss: 0.1342 - val_accuracy: 0.9726
Epoch 14/20
7/7 [==============================] - 0s 17ms/step - loss: 0.1034 - accuracy: 0.
 ↪9900 - val_loss: 0.1280 - val_accuracy: 0.9757
Epoch 15/20
7/7 [==============================] - 0s 19ms/step - loss: 0.0973 - accuracy: 0.
 ↪9950 - val_loss: 0.1222 - val_accuracy: 0.9767
Epoch 16/20
7/7 [==============================] - 0s 18ms/step - loss: 0.0917 - accuracy: 0.
 ↪9900 - val_loss: 0.1172 - val_accuracy: 0.9797
Epoch 17/20
7/7 [==============================] - 0s 17ms/step - loss: 0.0867 - accuracy: 0.
 ↪9950 - val_loss: 0.1127 - val_accuracy: 0.9807
Epoch 18/20
7/7 [==============================] - 0s 16ms/step - loss: 0.0823 - accuracy: 0.
 ↪9950 - val_loss: 0.1089 - val_accuracy: 0.9828
Epoch 19/20
7/7 [==============================] - 0s 18ms/step - loss: 0.0784 - accuracy: 0.
 ↪9950 - val_loss: 0.1051 - val_accuracy: 0.9828
Epoch 20/20
7/7 [==============================] - 0s 18ms/step - loss: 0.0748 - accuracy: 0.
 ↪9950 - val_loss: 0.1020 - val_accuracy: 0.9807
```

We achieve an accuracy ~97% since this is an easier problem (binary classification).

However, we could do better by transferring information from setting A.

13.3.2 Freezing lower layers

The lower layers of the first network have already learnt low-level features for the first task, so they can be reused as they are.

That is, we freeze their weights so that they are not altered during subsequent training of the new network.

We will take all layers from model A and then add a final output layer for our binary classification problem.

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1]) # Reuse all layers except␣
 ↪output.
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Note that `model_B_on_A` and `model_A` now share layers. When you train on `model_B_on_A` that will also impact `model_A`.

To avoid this you can clone a model.

Let's freeze all layers except the final dense output layer.

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

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=keras.optimizers.SGD(lr=1e-3),
                     metrics=["accuracy"])
```

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                           validation_data=(X_valid_B, y_valid_B))
```

```
Epoch 1/4
7/7 [==============================] - 1s 45ms/step - loss: 1.0335 - accuracy: 0.
↪3350 - val_loss: 0.9698 - val_accuracy: 0.4016
Epoch 2/4
7/7 [==============================] - 0s 18ms/step - loss: 0.9193 - accuracy: 0.
↪4050 - val_loss: 0.8752 - val_accuracy: 0.4270
Epoch 3/4
7/7 [==============================] - 0s 16ms/step - loss: 0.8248 - accuracy: 0.
↪4550 - val_loss: 0.7916 - val_accuracy: 0.4767
Epoch 4/4
7/7 [==============================] - 0s 30ms/step - loss: 0.7416 - accuracy: 0.
↪5100 - val_loss: 0.7166 - val_accuracy: 0.5254
```

Even with just one trained layer and a few epochs, our model is starting to learn the new problem.

Now let's unfreeze the lower layers and train the full model to fine-tune it.

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

model_B_on_A.compile(loss="binary_crossentropy",
                     optimizer=keras.optimizers.SGD(lr=1e-3),
                     metrics=["accuracy"])
```

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,
                           validation_data=(X_valid_B, y_valid_B))
```

```
Epoch 1/16
7/7 [==============================] - 1s 43ms/step - loss: 0.5843 - accuracy: 0.
↪6700 - val_loss: 0.4748 - val_accuracy: 0.8316
Epoch 2/16
7/7 [==============================] - 0s 17ms/step - loss: 0.3954 - accuracy: 0.
↪9050 - val_loss: 0.3687 - val_accuracy: 0.9097
Epoch 3/16
7/7 [==============================] - 0s 16ms/step - loss: 0.3061 - accuracy: 0.
↪9600 - val_loss: 0.3040 - val_accuracy: 0.9432
Epoch 4/16
7/7 [==============================] - 0s 17ms/step - loss: 0.2503 - accuracy: 0.
↪9650 - val_loss: 0.2605 - val_accuracy: 0.9544
Epoch 5/16
7/7 [==============================] - 0s 18ms/step - loss: 0.2121 - accuracy: 0.
↪9750 - val_loss: 0.2298 - val_accuracy: 0.9645
Epoch 6/16
7/7 [==============================] - 0s 16ms/step - loss: 0.1847 - accuracy: 0.
↪9800 - val_loss: 0.2065 - val_accuracy: 0.9675
Epoch 7/16
7/7 [==============================] - 0s 16ms/step - loss: 0.1639 - accuracy: 0.
↪9850 - val_loss: 0.1885 - val_accuracy: 0.9716
Epoch 8/16
7/7 [==============================] - 0s 16ms/step - loss: 0.1476 - accuracy: 0.
↪9900 - val_loss: 0.1740 - val_accuracy: 0.9767
Epoch 9/16
```

(continues on next page)

(continued from previous page)

```
7/7 [==============================] - 0s 18ms/step - loss: 0.1346 - accuracy: 0.
↪9950 - val_loss: 0.1610 - val_accuracy: 0.9787
Epoch 10/16
7/7 [==============================] - 0s 15ms/step - loss: 0.1231 - accuracy: 0.
↪9950 - val_loss: 0.1512 - val_accuracy: 0.9787
Epoch 11/16
7/7 [==============================] - 0s 16ms/step - loss: 0.1143 - accuracy: 0.
↪9950 - val_loss: 0.1426 - val_accuracy: 0.9787
Epoch 12/16
7/7 [==============================] - 0s 17ms/step - loss: 0.1065 - accuracy: 0.
↪9950 - val_loss: 0.1352 - val_accuracy: 0.9787
Epoch 13/16
7/7 [==============================] - 0s 17ms/step - loss: 0.0997 - accuracy: 0.
↪9950 - val_loss: 0.1285 - val_accuracy: 0.9787
Epoch 14/16
7/7 [==============================] - 0s 17ms/step - loss: 0.0937 - accuracy: 0.
↪9950 - val_loss: 0.1223 - val_accuracy: 0.9807
Epoch 15/16
7/7 [==============================] - 0s 15ms/step - loss: 0.0880 - accuracy: 0.
↪9950 - val_loss: 0.1172 - val_accuracy: 0.9807
Epoch 16/16
7/7 [==============================] - 0s 17ms/step - loss: 0.0834 - accuracy: 0.
↪9950 - val_loss: 0.1128 - val_accuracy: 0.9828
```

```
model_B.evaluate(X_test_B, y_test_B)
```

```
63/63 [==============================] - 0s 2ms/step - loss: 0.0986 - accuracy: 0.
↪9880
```

```
[0.0986466035246849, 0.9879999756813049]
```

```
model_B_on_A.evaluate(X_test_B, y_test_B)
```

```
63/63 [==============================] - 0s 2ms/step - loss: 0.1014 - accuracy: 0.
↪9890
```

```
[0.10140985995531082, 0.9890000224113464]
```

13.3.3 Model gardens

Many trained Tensor Flow models are available at https://github.com/tensorflow/models.

13.4 Improved optimizers

Although standard (stochastic) gradient descent is very effective it can still be slow for deep networks.

There are a number of more advanced optimizers that provide improvements, e.g.:

- Momentum optimization
- Nesterov accelerated gradient
- AdaGrad
- RMSProp
- Adam optimization
- …

Recall gradient descent, with cost function $J(\theta)$ and gradients $\nabla_\theta J(\theta)$, proceeds simply by updating the weights θ by taking a step η (learning rate) in the direction of the gradient:

$$\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$$

13.4.1 Momentum optimization

Momentum optimization uses the gradients to modify a momentum vector and uses the momentum to update the weights:

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

Gradient is used as an acceleration rather than speed. Can help to traverse plateaus and to avoid local minima.

The additional hyperparameter β is introduced as a friction term to avoid the momentum growing too large (typically $\beta \sim 0.9$).

13.4.2 Nesterov accelerated gradient

Nesterov accelerated gradient is a variant of momentum optimization where the gradient is computed further ahead in the direction of the momentum:

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

In general the momentum will be pointing toward the optimum and so Nesterov modification typically provides an improvement over standard momentum optimization.

13.4.3 AdaGrad

AdaGrad scales down the gradient vector along the steepest direction by incorporating a gradient squared term:

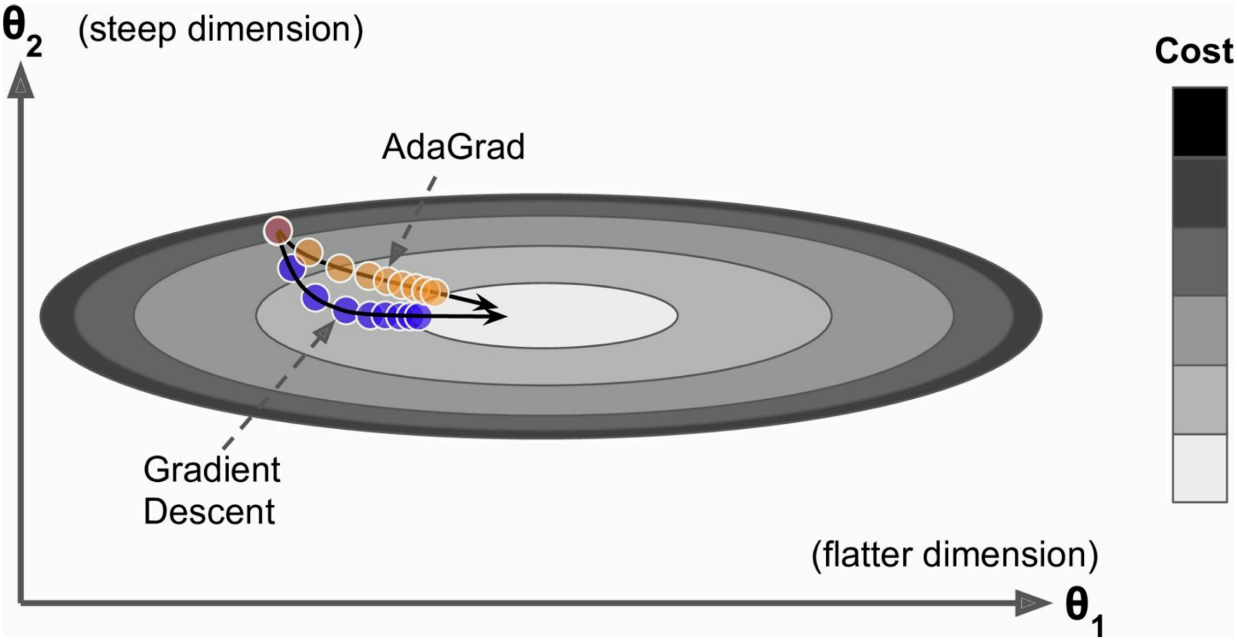
1. $s \leftarrow s + \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s + \epsilon}$

Note that $\otimes$ and $\oslash$ are elementwise multiplication and division, respectively.

The parameter ϵ is introduced for numerical stability (typically $\epsilon \sim 10^{-10}$).

Basically, AdaGrad correspondings to an *adaptive learning rate* where the learning rate is decayed faster for steep directions.

Consequently, it requires much less tuning of the learning rate η.



[Credit: Geron]

13.4.4 RMSProp

RMSProp extends AdaGrad by introducing an exponential decay in the accumulated squared gradient:

1. $s \leftarrow \beta s + (1 - \beta) \nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta) \oslash \sqrt{s + \epsilon}$

(Typically $\beta \sim 0.9$.)

Avoids the problem where AdaGrad slows down too fast and so doesn't converge to the global optimum.

13.4.5 Adam optimization

Adam optimization combines momentum and RMSProp:

1. $m \leftarrow \beta_1 m + (1-\beta_1)\nabla_\theta J(\theta)$
2. $s \leftarrow \beta_2 s + (1-\beta_2)\nabla_\theta J(\theta) \otimes \nabla_\theta J(\theta)$
3. $m \leftarrow \frac{m}{1-\beta_1^t}$, where t is the iteration number
4. $s \leftarrow \frac{s}{1-\beta_2^t}$, where t is the iteration number
5. $\theta \leftarrow \theta - \eta m \oslash \sqrt{s+\epsilon}$

Steps 3 and 4 are introduced to boost m and s at the beginnning of training (since they are initialised to 0 they can otherwise be low at the beginning).

(Typically $\beta_1 \sim 0.9$, $\beta_2 \sim 0.999$.)

13.5 Regularization

Deep networks have many parameters (sometimes millions) and so are prone to overfitting.

Regularization therefore becomes increasingly important.

13.5.1 Early stopping

A simple regularization strategy is to end training early, e.g. when performance on validation set starts to degrade.

Although early stopping works well, other regularisation techniques can lead to better performance.

13.5.2 ℓ_2 and ℓ_1 regularization

Tikhonov regularization adopts ℓ_2 regularising term (also called *Ridge regression*):

$$R(\theta) = \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\theta^{\mathrm{T}}\theta.$$

Lasso regularization adopts ℓ_1 regularising term:

$$R(\theta) = \sum_{j=1}^{n}\left|\theta_j\right|.$$

Elastic net regularization provides a mix of Tikhonov and Lasso regularization, controlled by mix ratio r:

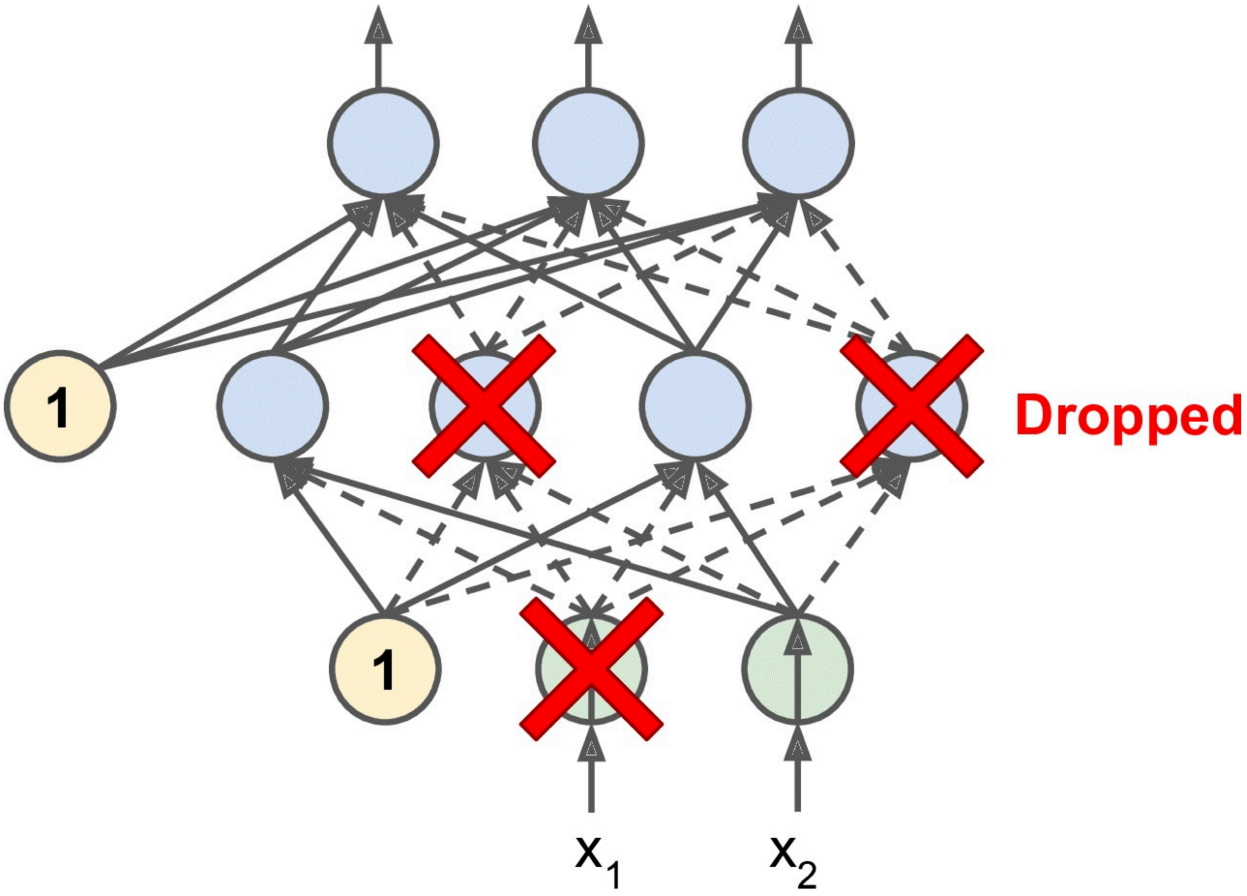
$$R(\theta) = r\sum_{j=1}^{n}\left|\theta_j\right| + \frac{1-r}{2}\sum_{j=1}^{n}\theta_j^2.$$

- For $r = 0$, corresponds to Tikhonov regularization.
- For $r = 1$, corresponds to Lasso regularization.

13.5.3 Dropout

Dropout is a very popular and effective regularlisation technique developed by Geoff Hinton in 2012.

Dropout involves simply dropping each neuron for a given training set with probability p.



[Credit: Geron]

Dropout encourages each neuron to be as effective as possible individually and not to rely heavily on a few nearby neurons but to consider all input neurons carefully.

The probability p is called the *dropout rate* (typically $p \sim 0.5$).

After training the neurons don't get dropped.

The number of inputs of active neurons is lower when dropout is applied during training, than when the network is applied during testing.

For example, if $p = 0.5$, on average there are half as many input neurons during training than when testing. During testing each neuron will get an input signal (approximately) twice as large as during training.

It is important to account for this difference.

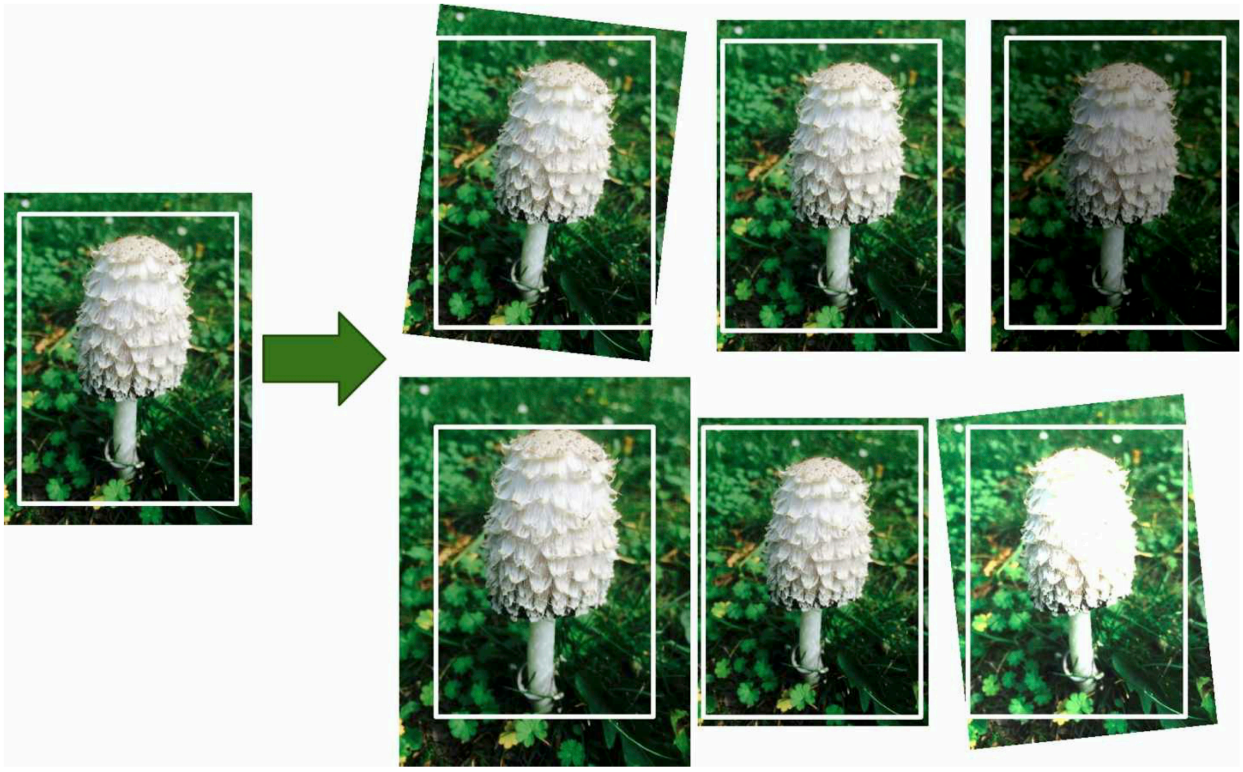
To compensate, after training each neurons input weights are multiplied by the keep probability $1 - p$ before applying the network to test data.

13.5.4 Data augmentation

Data augmentation can be applied both as a regularization technique and to increase the volume of the training set.

Essentially, new training instances are created from the original training set.

For example, for images, data augmentation can be performed by rotating, shifting, scaling, flipping, changing the contrast, …, of the original images in the training data-set.



[Credit: Geron]

Appropriate data augmentation strategies depend on the type of data under consideration.

Typically training instances are generated on the fly to avoid additional storage requirements.

Tensor Flow has built in functionality for many transformations for image data, making data augmentation for image data straightforward.

CHAPTER

FOURTEEN

LECTURE 14: CONVOLUTIONAL NEURAL NETWORKS


Run in colab

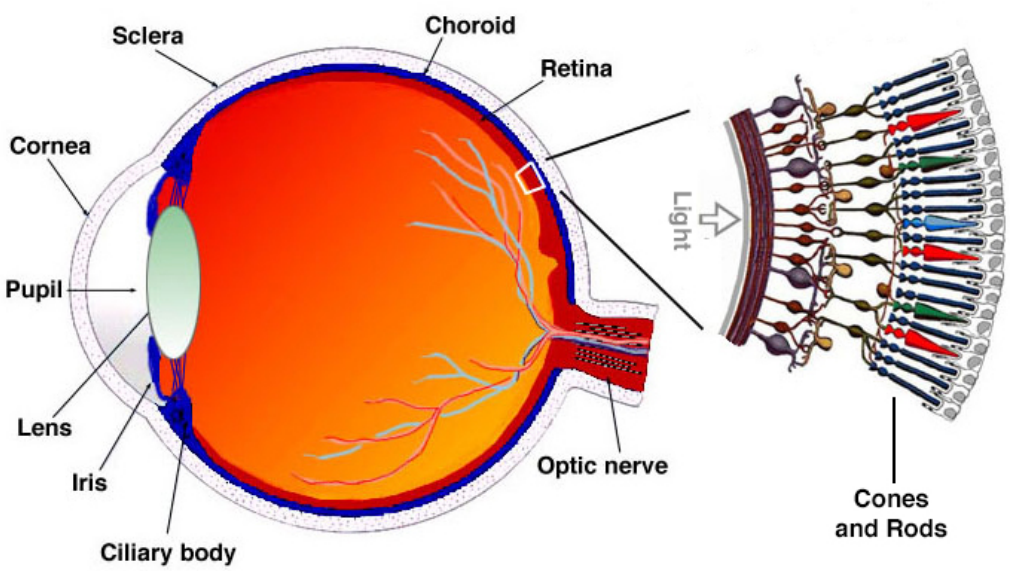
```
import datetime
now = datetime.datetime.now()
print("Version: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Version: 2023-02-04 10:28:01
```

```
# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
def reset_state(seed=42):
    tf.keras.backend.clear_session()
    tf.random.set_seed(seed)
    np.random.seed(seed)

# To plot pretty figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

14.1 Motivation

14.1.1 Visual cortex



[Image source]

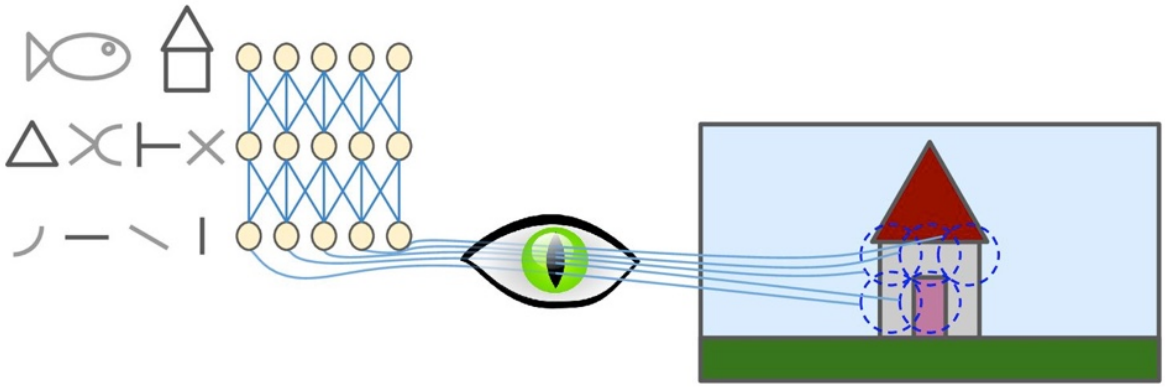
Neurons in the visual cortex have a small local receptive field, i.e. only react to limited region of visual field.

Receptive fields of neurons overlap and together cover full visual field.

14.1.2 Architecture of the visual cortex

Some neurons only sensitive to stimuli of certain structure, e.g. horizonal lines.

Some neurons have larger receptive fields, and are sensitive to stimuli that are combinations of lower-level patterns.



[Credit: Geron]

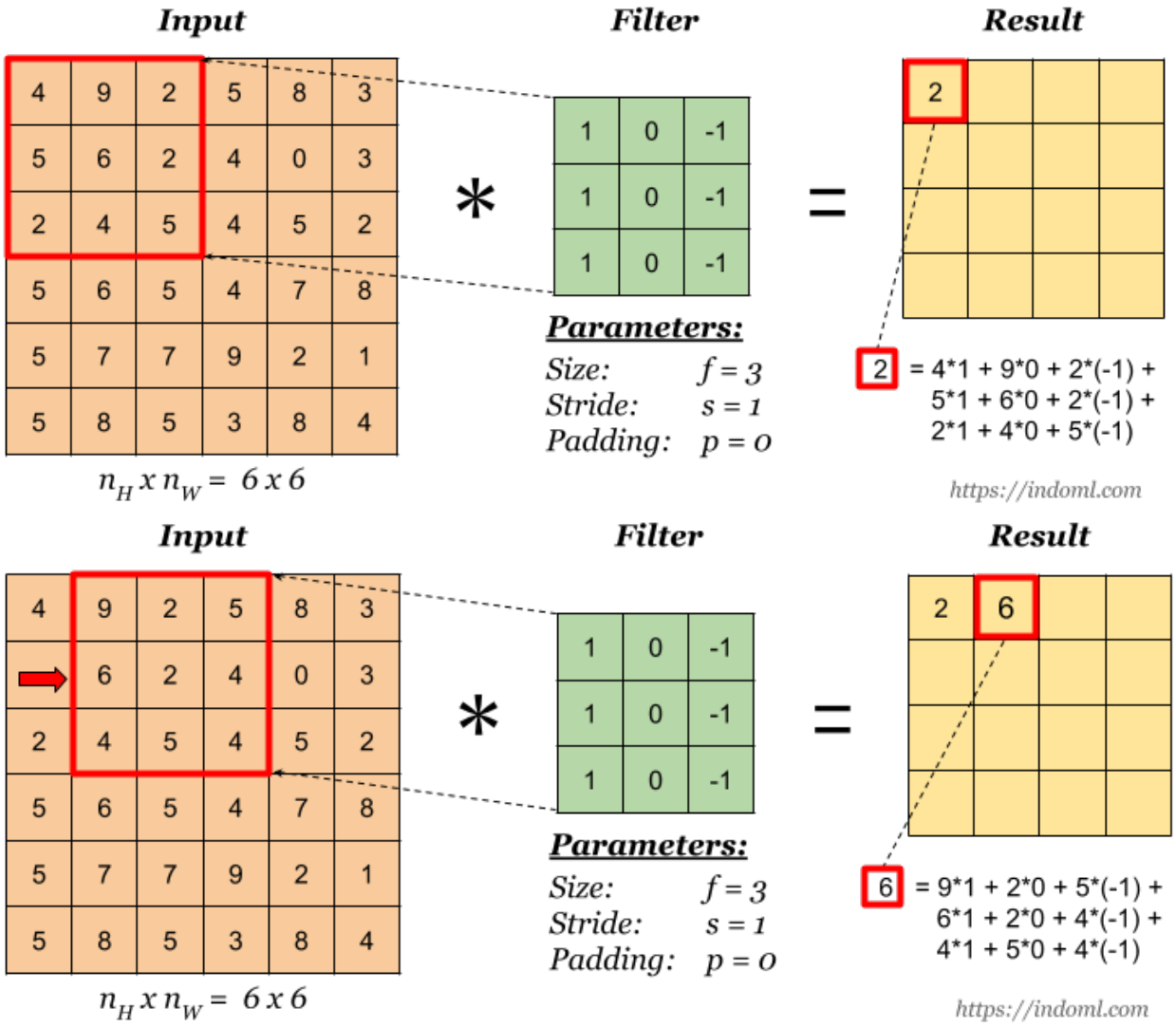
Corresponding architecture can detect complex patterns in full visual field, which inspires design of convolutional neural networks (CNNs).

14.2 Convolution

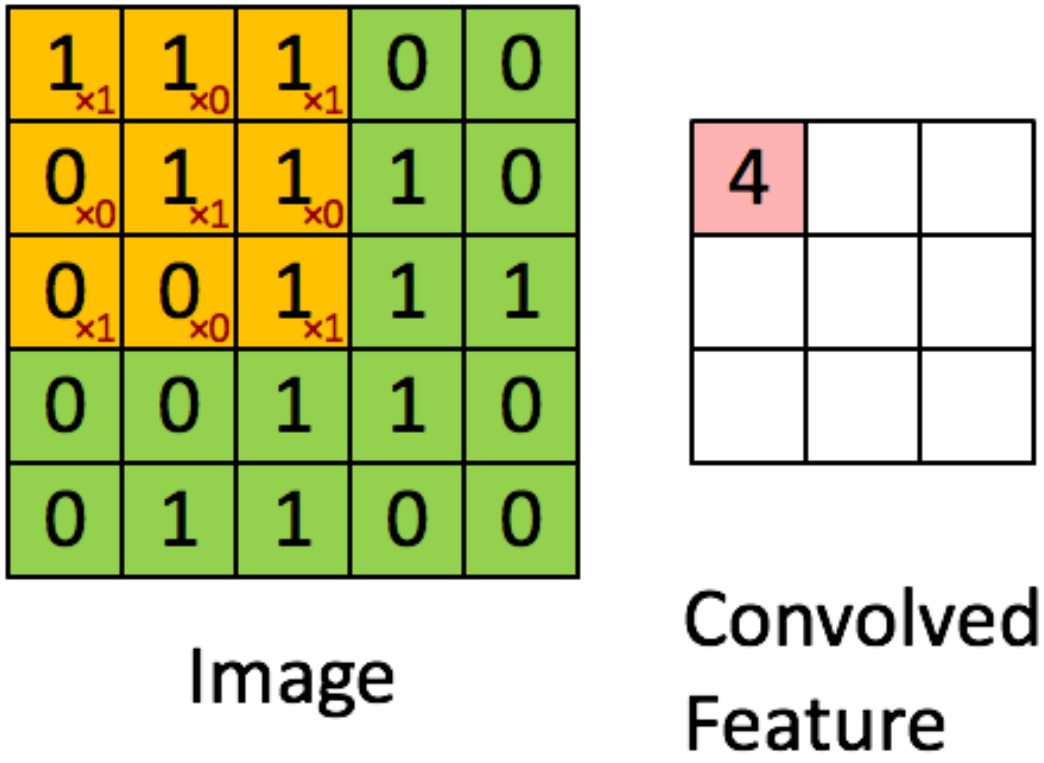
Core building block of CNNs is convolution.

14.2.1 Graphical description

Convolution involves passing a filter (kernel) over an image and taking the sum of the product of terms for all positions.



Animation of convolution



[Animation source]

14.2.2 Mathematical description

Convolution output is given by

$$z_{i,j} = \sum_{u,v} x_{u,v} w_{u-i,v-j},$$

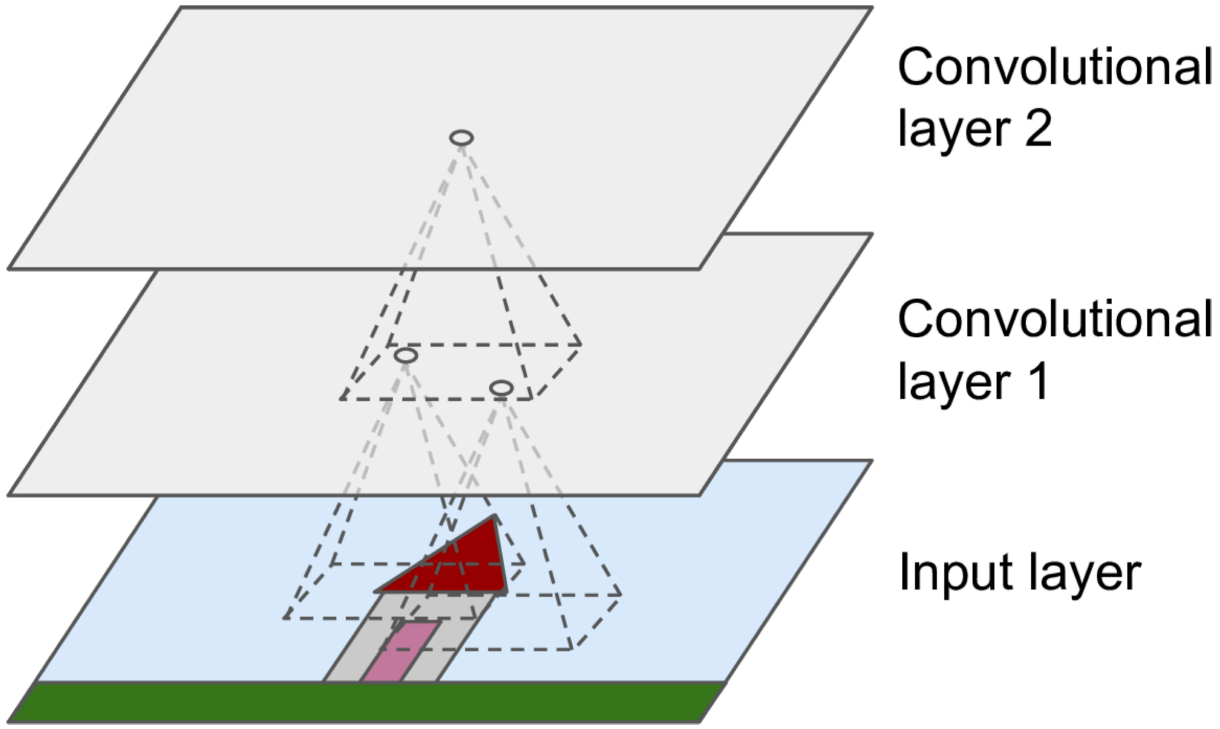
where x is the input image, w is the filter (kernel) and i (u) and j (v) denote row and column indices, respectively. (Note that the kernel is not reflected as is typical in the usual mathematical definition of convolution.)

14.2.3 Advantages

- Localisation: Capture local structure.
- Efficiency: weight sharing results in dramatic reduction in number of weights (parameters) compared to fully-connected neural network.
- Translational equivariance: Feature space behavies "nicely" under a translation of the input. In a sense, learn general features rather than features for all locations of image.
- Composition: Can compose convolutions to extract more complex features.

14.3 Convolutional layers

Define neural network layers using convolutions.



[Credit: Geron]

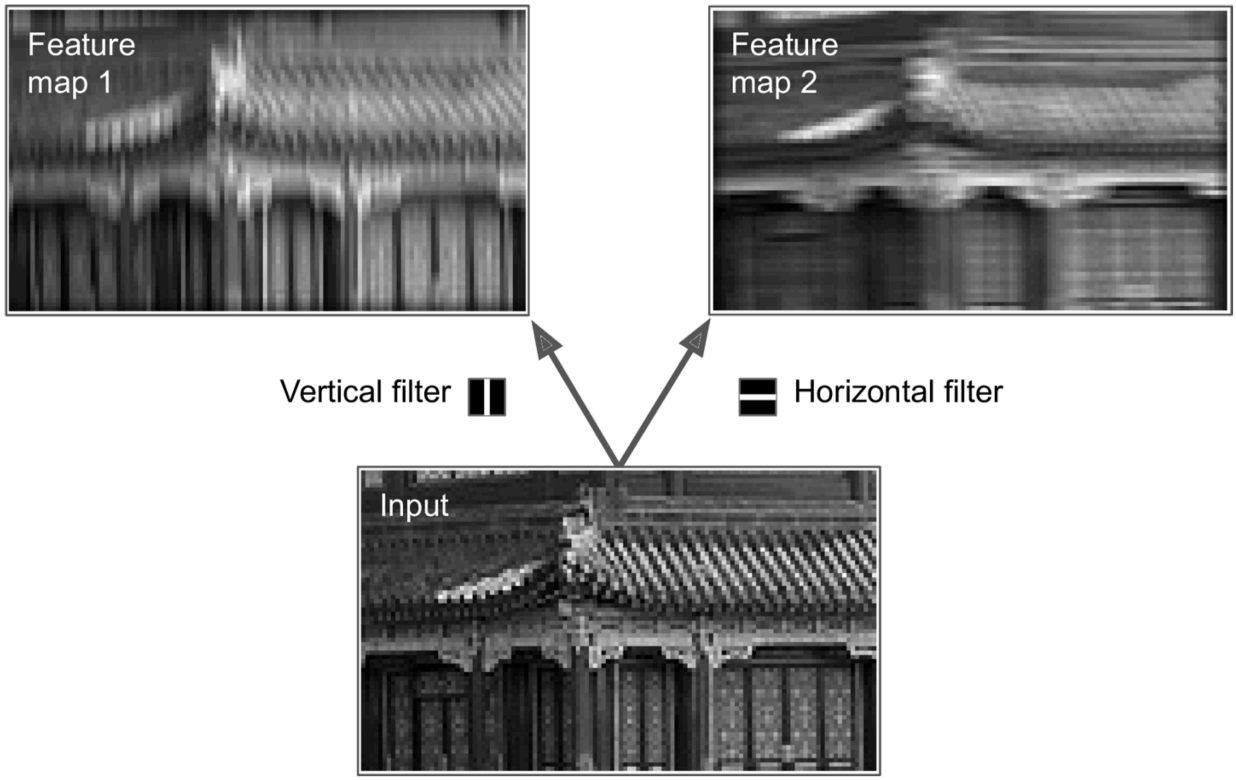
Allows the network to focus on small low-level features in the first layer and use these to constuct higher-level features in next layer, and so on.

Results in a hierarchical representation that is common in images of the real-world.

14.3.1 Filters and features

Filters pull out corresponding features from image.

Output of convolutional layer is typically called a *feature map*.

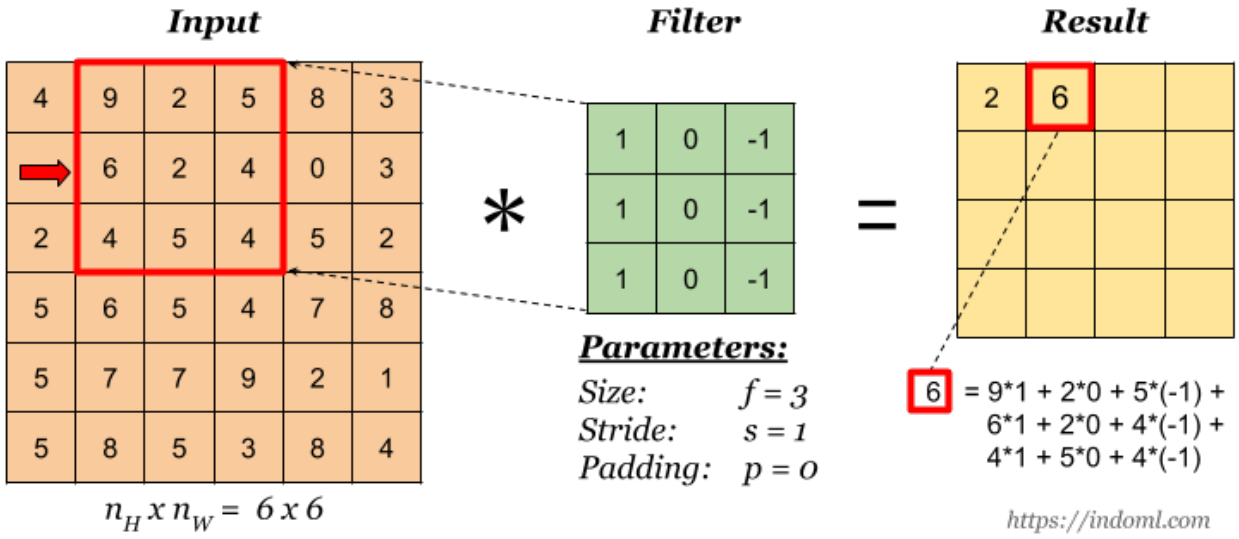


[Credit: Geron]

During training, effective filters to extract representative features are learned.

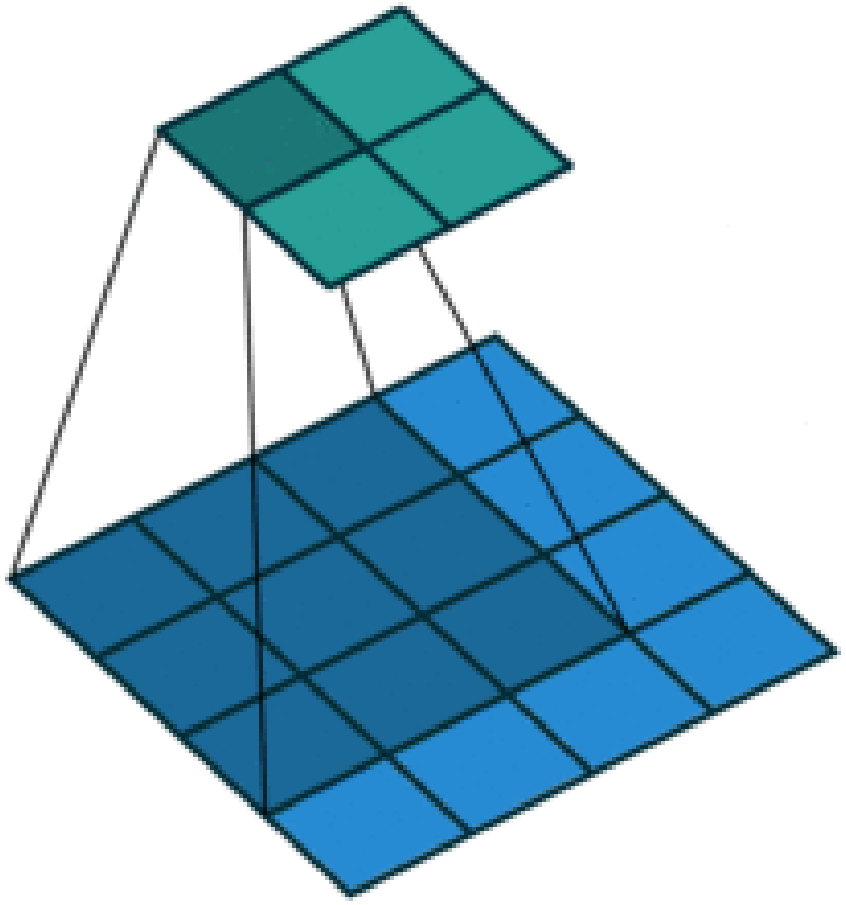
14.3.2 Padding

Nominally, convolution results in an output image that is *smaller* than the input.



Various modifications to the nominal convolution are often considered.

Animation of convolution (no padding, no stride)

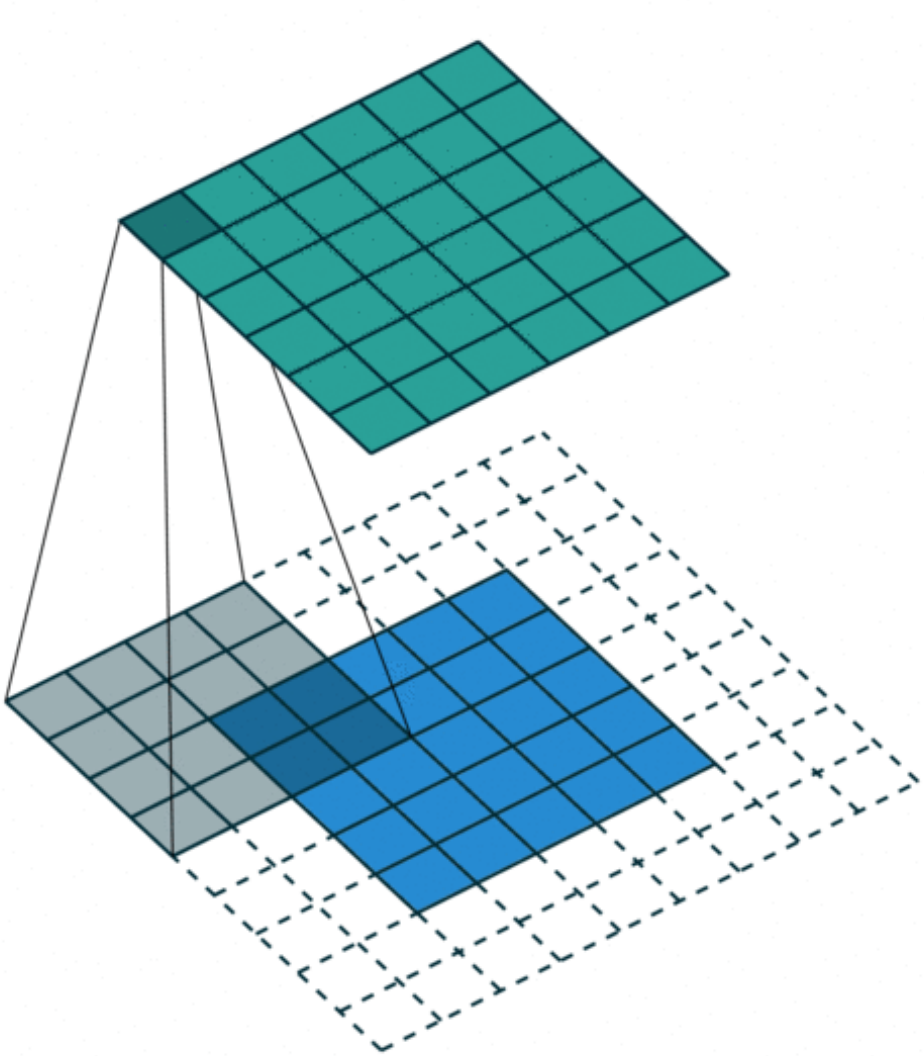


[Animation source]

Blue pixels denote the input image, grey the convolutional kernel as it moves over the image, and green pixels denote the output.

Padding often introduced to control output size.

Animation of convolution (padding, no stride)

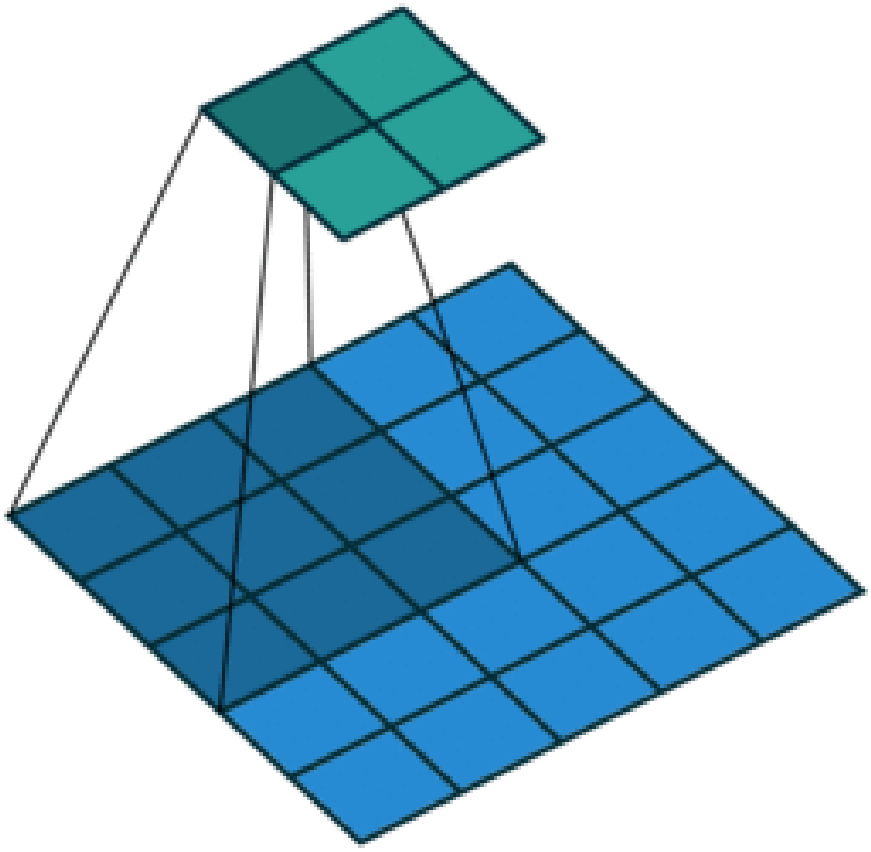


[Animation source]

Uncoloured pixels denote padding with zeros.

14.3.3 Stride

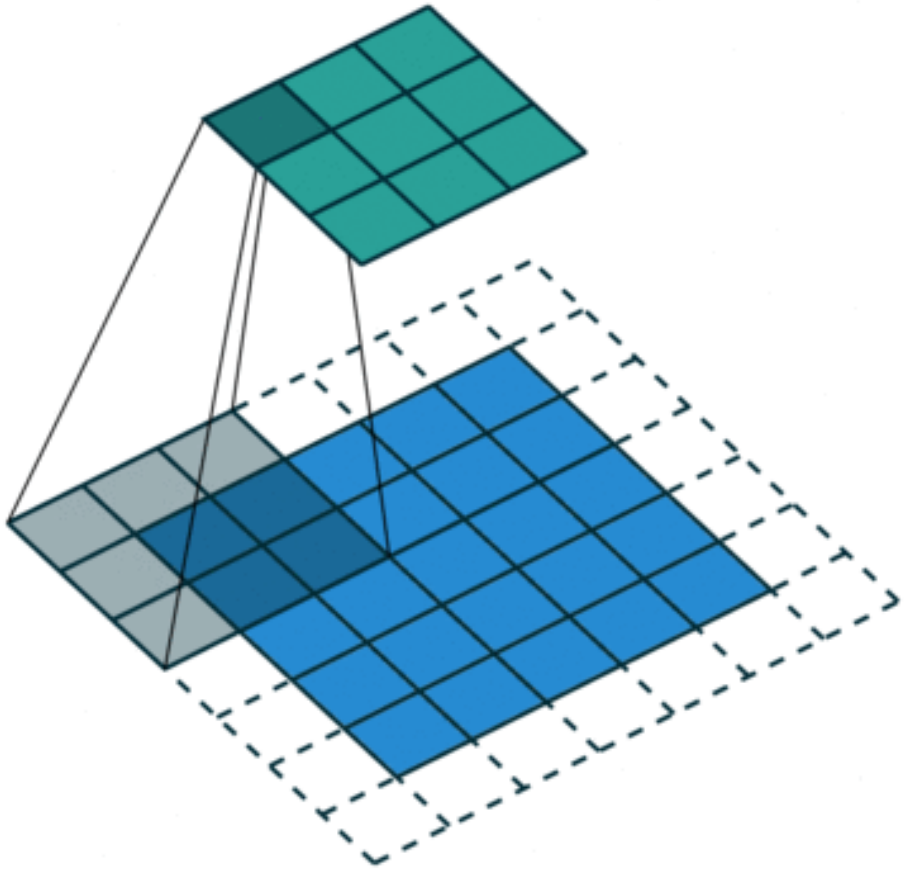
Animation of convolution (no padding, stride)



[Animation source]

14.3.4 Padding and stride

Animation of convolution (padding, stride)

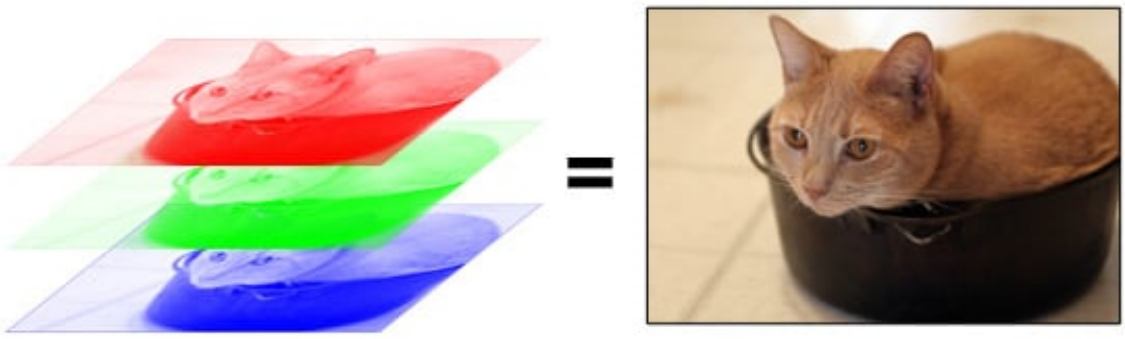


[Animation source]

14.3.5 Channels

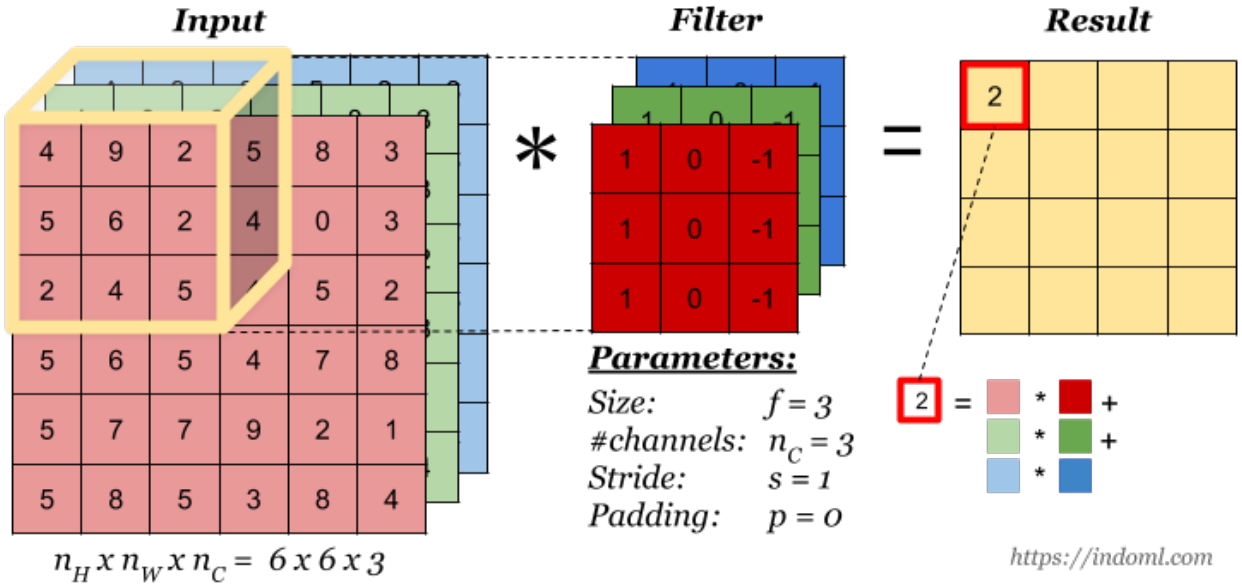
Multiple input channels

Input images often have multiple channels, e.g. red, green and blue colour channels.



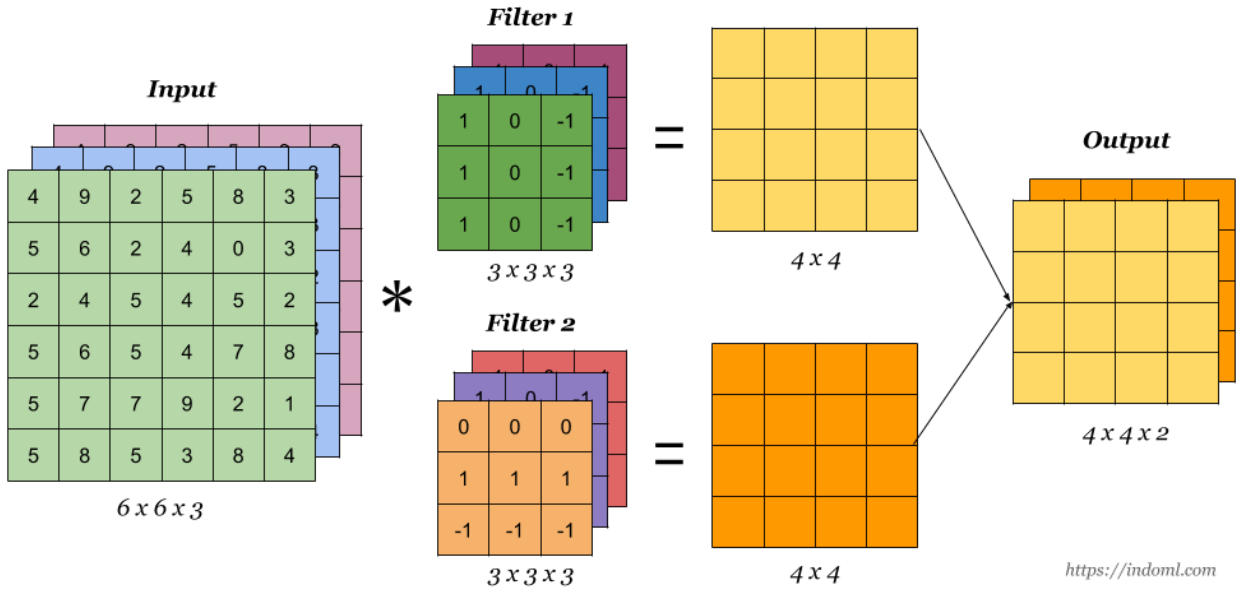
[Image source]

Support multiple input channels by defining a filter for each channel and summing result.



Multiple input and output channels

Add a set of filters for each desired output channel.



Mathematical description

Convolution output is given by

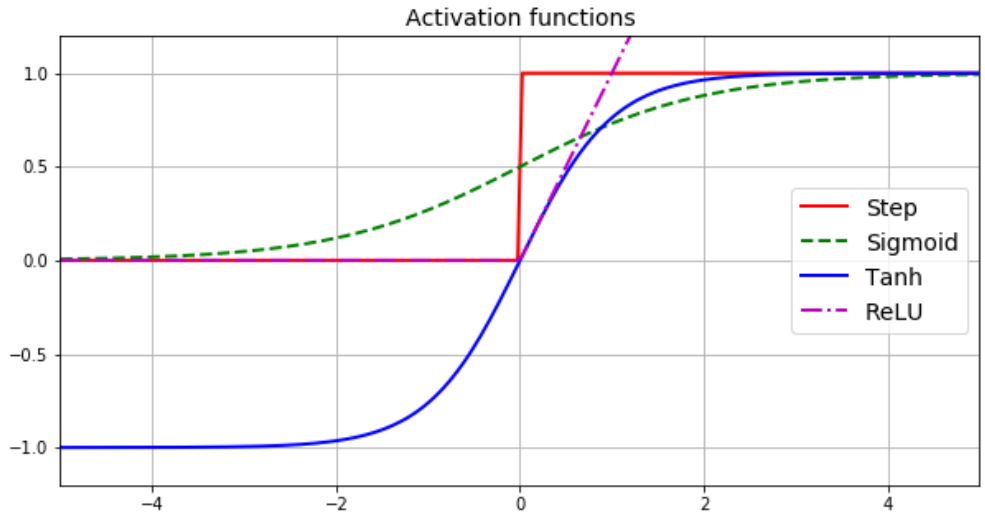
$$z_{i,j,k_{\text{out}}} = \sum_{u,v,k_{\text{in}}} x_{u,v,k_{\text{in}}} w_{u-i,v-j,k_{\text{in}},k_{\text{out}}},$$

where x is the input image, w is the filter (kernel) and i (u) and j (v) denote row and column indices, respectively. The input channel index is denoted k_{in} and the output channel index k_{out}.

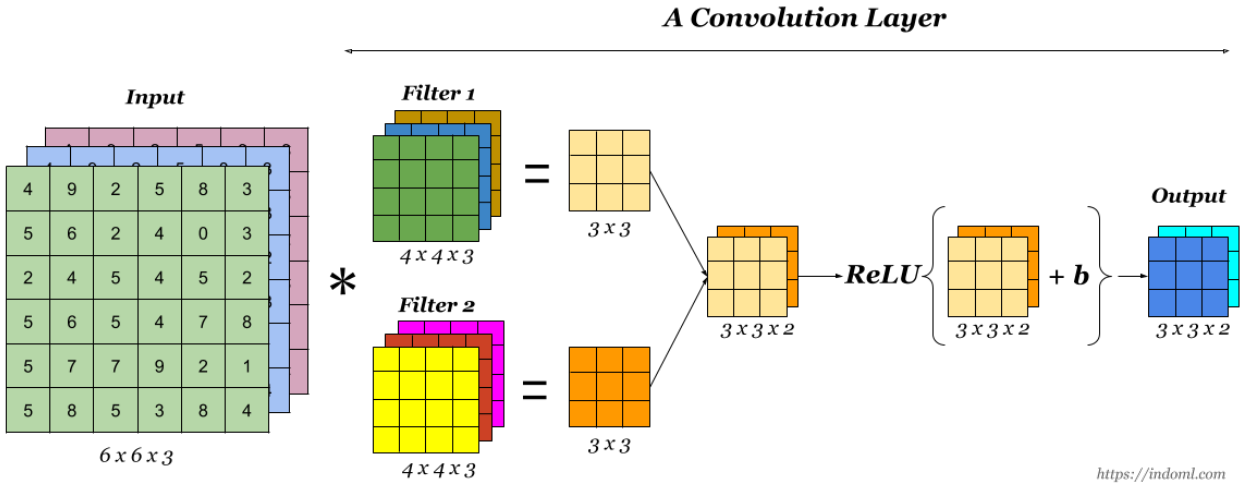
Notice that the filter is now 4-dimensional. We quickly add a lot of additional parameters.

14.3.6 Non-linear activations

Convolutions usually followed by pointwise activation functions to introduce non-linearity (cf. fully-connected neural networks).



14.3.7 Full convolutional layer



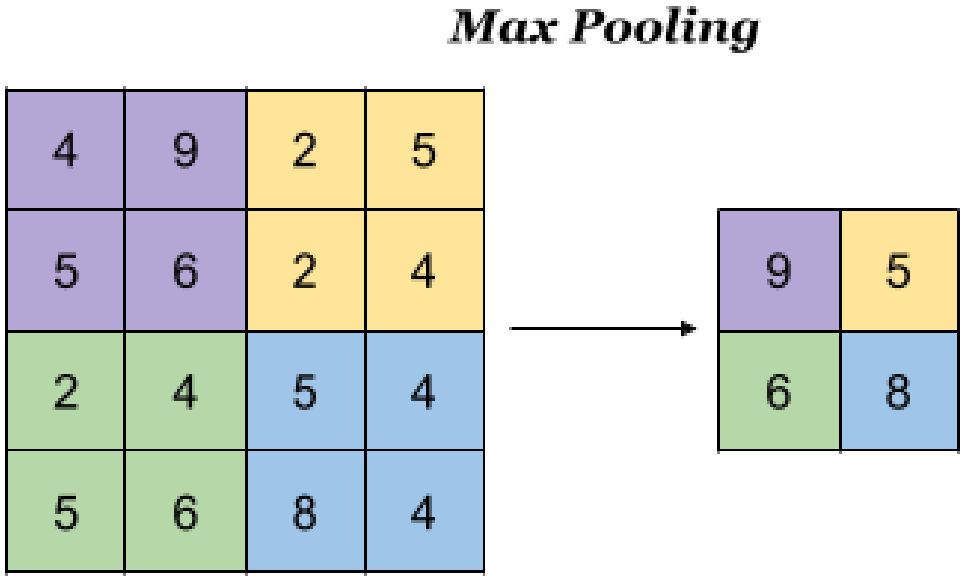
14.4 Pooling layers

Pooling layers are used to downsample to reduce the computational load, memory usage and number of parameters.

Retains equivariance to large translation and can introduce invariance to small translations.

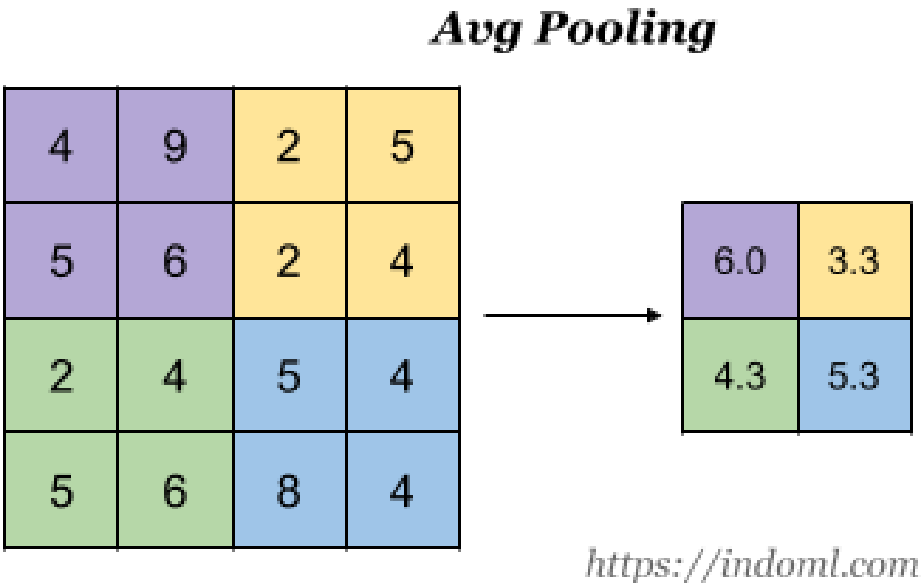
14.4.1 Max pooling

Take maximum over parent cell size, translating cell over image.



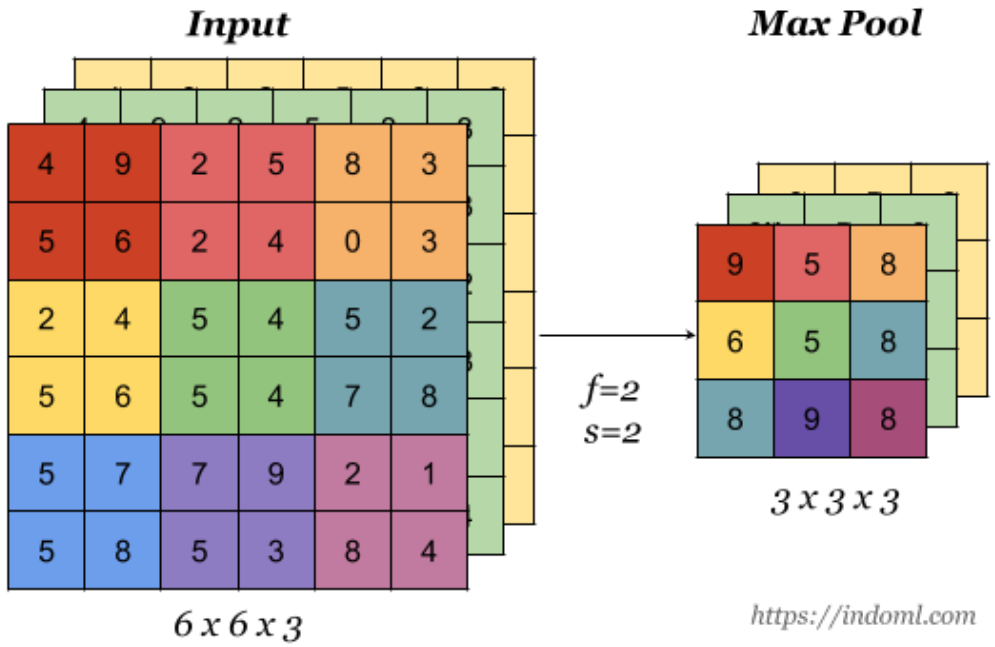
14.4.2 Average pooling

Take average over parent cell size, translating cell over image.



14.4.3 Multiple input channels

When considering multiple input channels, pooling is performed separately for each channel.

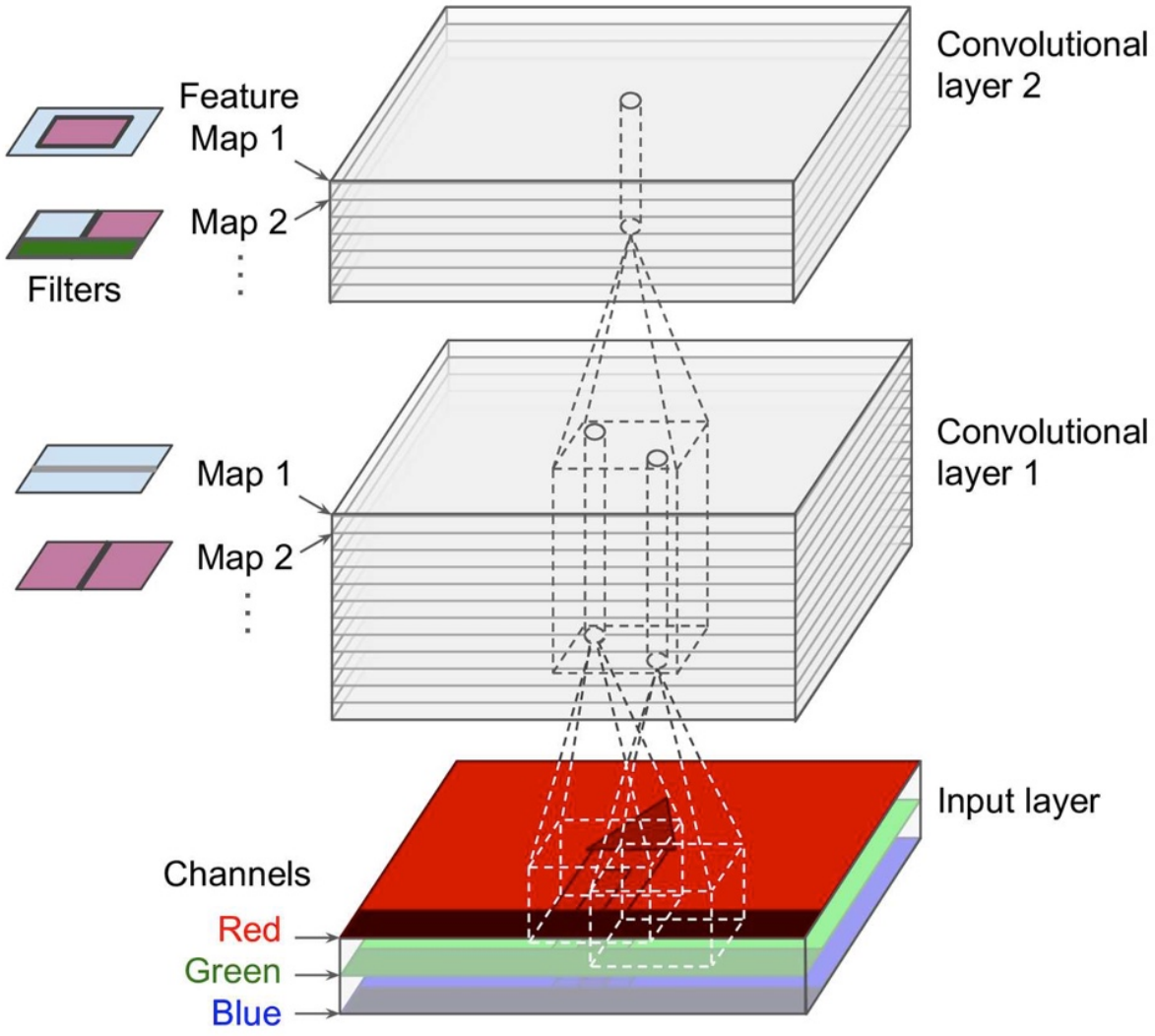


14.5 CNN architectures

14.5.1 Stacking multiple feature maps

Convolutional layers typically stacked with multiple input and output channels, leading to multiple feature maps.

Neuron's receptor field extends across all previous layers' feature maps.



[Credit: Geron]

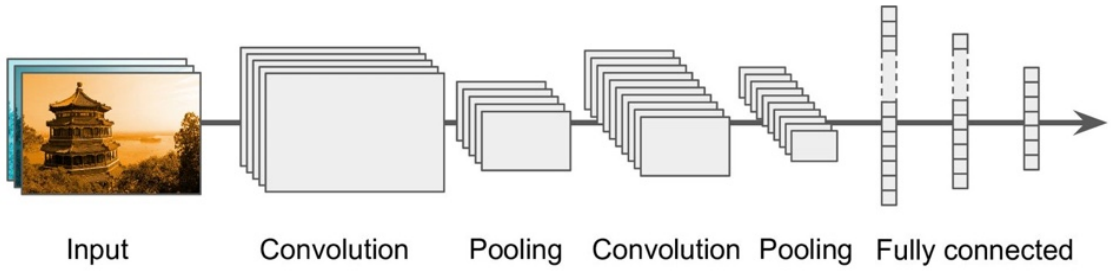
14.5.2 Basic CNN architecture

Basic CNN architecture typically consists of combining the following layers:

- Convolutions
- Non-linear activations
- Pooling

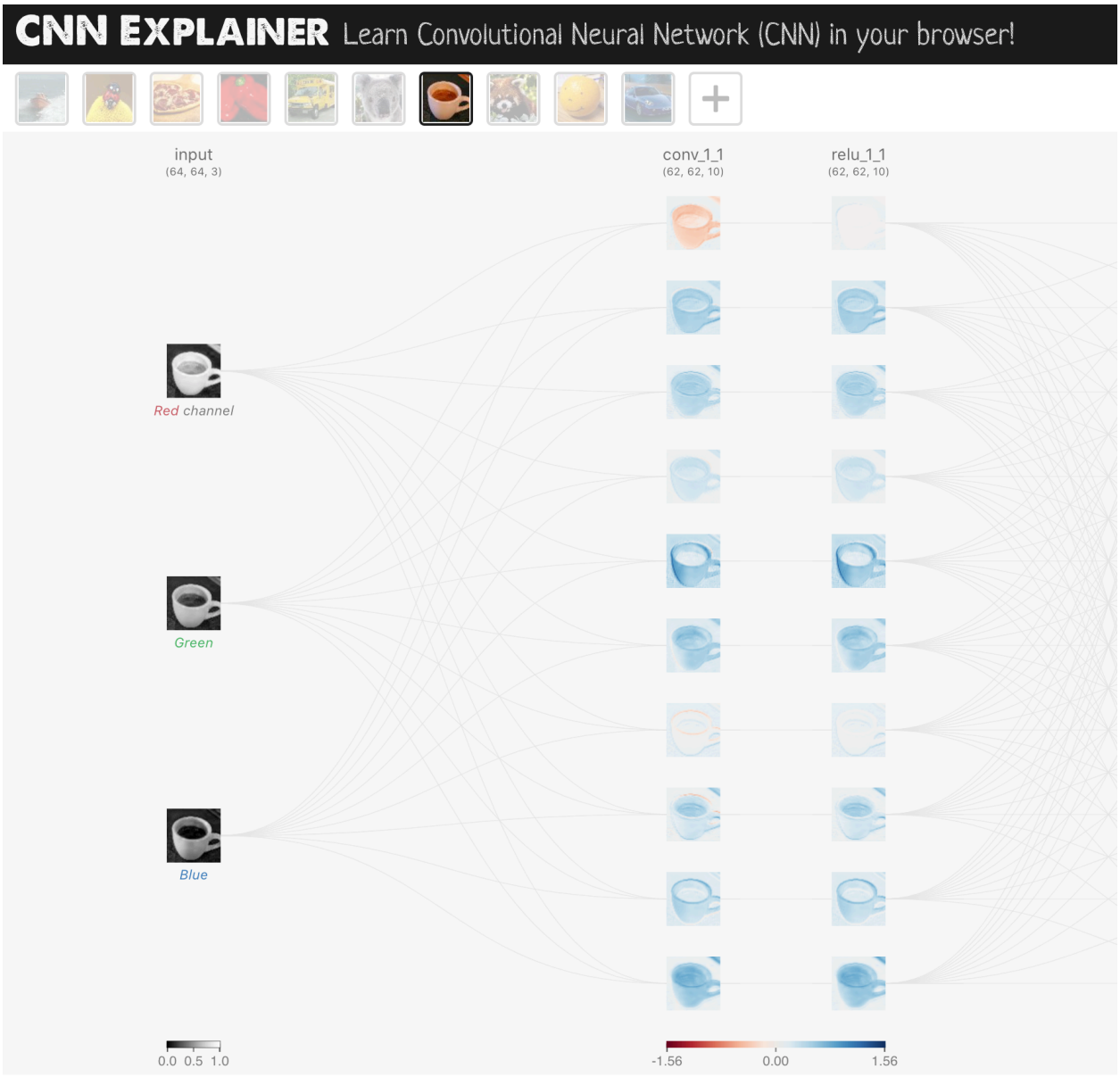
And repeating.

Final layers are then added on that are tailored to the problem at hand (often fully connected layers).



[Credit: Geron]

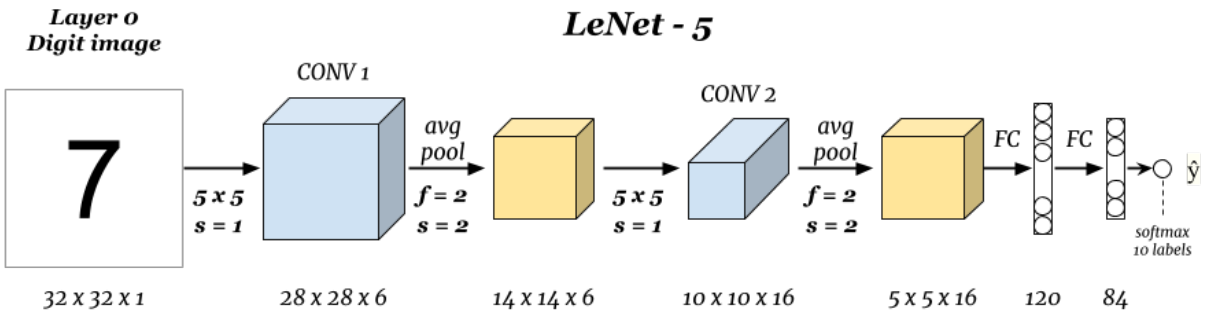
14.5.3 CNN explainer



The CNN explainer is a great visualisation tool, allowing you to look inside a CNN to visualise the layers making up a network.

14.5.4 LeNet architecture

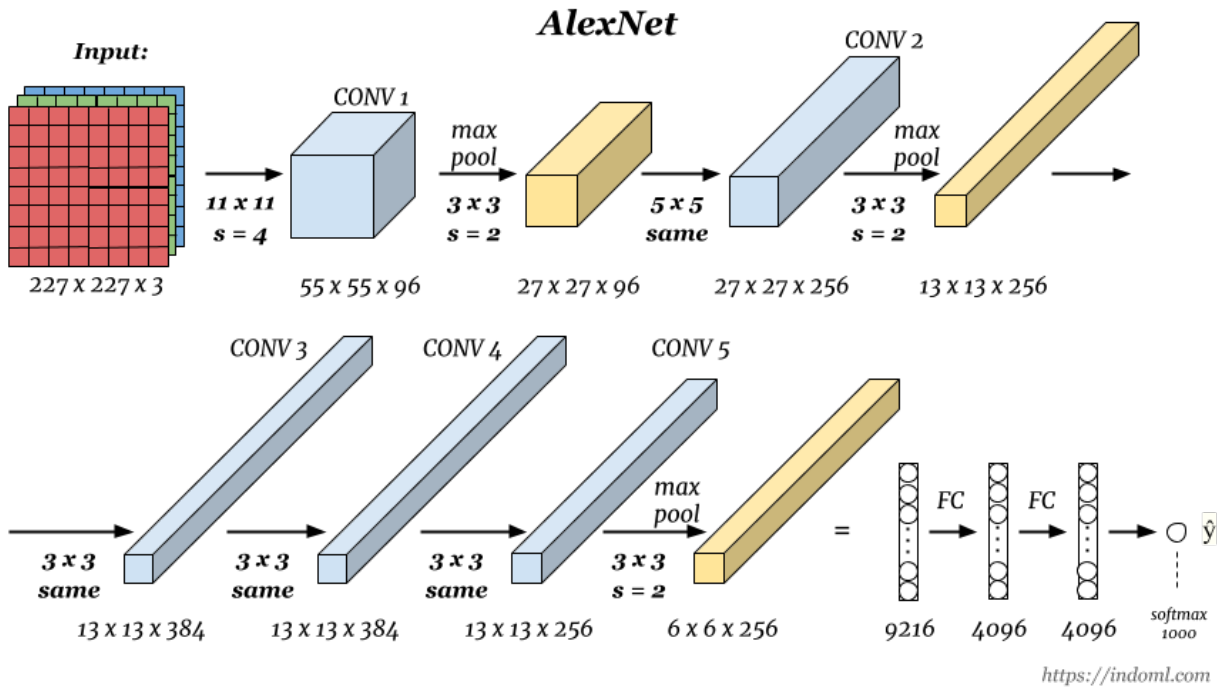
Convolutional layers first introduced by Lecun et al. in 1998 for digital classification problem.



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	–	–	–

14.5.5 AlexNet

AlexNet, which helped to intitiate the deep learning revolution in 2012, was based on a CNN architecture and showed a significant improvement in performance on the ImageNet benchmark problem compared to the state-of-the-art at the time.

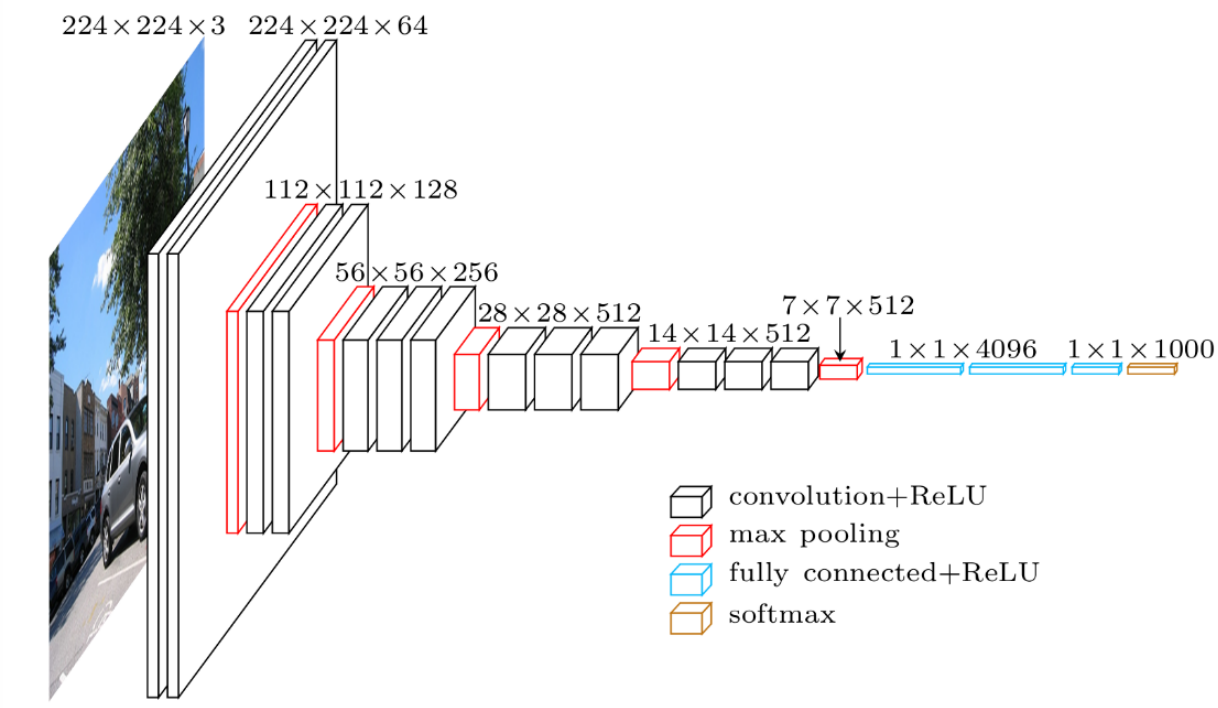


Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully connected	–	1,000	–	–	–	Softmax
F10	Fully connected	–	4,096	–	–	–	ReLU
F9	Fully connected	–	4,096	–	–	–	ReLU
S8	Max pooling	256	6×6	3×3	2	`valid`	–
C7	Convolution	256	13×13	3×3	1	`same`	ReLU
C6	Convolution	384	13×13	3×3	1	`same`	ReLU
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	–	–	–	–

14.5.6 VGG

The VGG network followed soon afterwards, making another significant improvement in performance, while simplifying the architecture.

VGG-16 uses 3x3 convolutions only and max pooling layers that step down by a factor of two at each stage.



14.6 Implementing CNNs in TensorFlow

14.6.1 Load and set up data

Let's consider fashion MNIST again.

```
import tensorflow as tf
from tensorflow import keras
```

```
2023-02-04 10:28:01.765182: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
2023-02-04 10:28:01.927988: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dlerror:␣
 ↪libcudart.so.11.0: cannot open shared object file: No such file or directory
2023-02-04 10:28:01.928012: I tensorflow/stream_executor/cuda/cudart_stub.cc:29]␣
 ↪Ignore above cudart dlerror if you do not have a GPU set up on your machine.
2023-02-04 10:28:01.965645: E tensorflow/stream_executor/cuda/cuda_blas.cc:2981]␣
 ↪Unable to register cuBLAS factory: Attempting to register factory for plugin␣
 ↪cuBLAS when one has already been registered
```

(continues on next page)

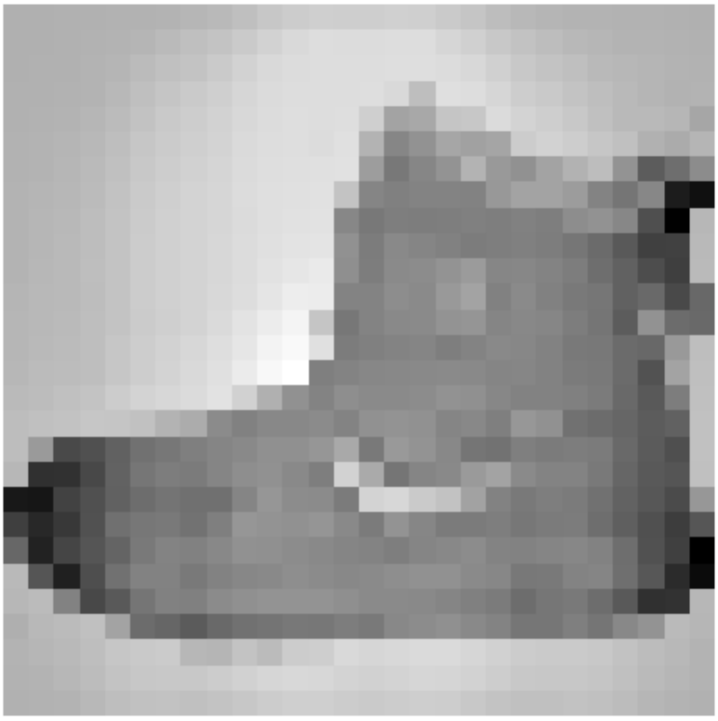
(continued from previous page)

```
2023-02-04 10:28:02.793060: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer.so.7'; dlerror:␣
 ↪libnvinfer.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:28:02.793145: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libnvinfer_plugin.so.7'; dlerror:␣
 ↪libnvinfer_plugin.so.7: cannot open shared object file: No such file or directory
2023-02-04 10:28:02.793154: W tensorflow/compiler/tf2tensorrt/utils/py_utils.
 ↪cc:38] TF-TRT Warning: Cannot dlopen some TensorRT libraries. If you would like␣
 ↪to use Nvidia GPU with TensorRT, please make sure the missing libraries␣
 ↪mentioned above are installed properly.
```

```
# Load fashion MNIST data
(X_train_full, y_train_full), (X_test, y_test) = keras.datasets.fashion_mnist.load_
 ↪data()
X_train, X_valid = X_train_full[:-30000], X_train_full[-30000:]
y_train, y_valid = y_train_full[:-30000], y_train_full[-30000:]

# Standardize
X_mean = X_train.mean(axis=0, keepdims=True)
X_std = X_train.std(axis=0, keepdims=True) + 1e-7
X_train = (X_train - X_mean) / X_std
X_valid = (X_valid - X_mean) / X_std
X_test = (X_test - X_mean) / X_std

# Add final channel axis (one channel)
X_train = X_train[..., np.newaxis]
X_valid = X_valid[..., np.newaxis]
X_test = X_test[..., np.newaxis]
```

14.6.2 Plot example data instance

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.imshow(X_train[0], cmap="binary")
plt.axis('off')
plt.show()
```



14.6.3 Build CNN model

```
model = keras.models.Sequential([

    keras.layers.Conv2D(filters=4, kernel_size=7, activation="relu", padding="same",
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(pool_size=2),

    keras.layers.Flatten(),
    keras.layers.Dense(units=8, activation='relu'),
    keras.layers.Dense(units=10, activation='softmax'),
])
```

```
2023-02-04 10:28:05.232591: W tensorflow/stream_executor/platform/default/dso_
 ↪loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.
 ↪1: cannot open shared object file: No such file or directory
2023-02-04 10:28:05.232625: W tensorflow/stream_executor/cuda/cuda_driver.cc:263]␣
 ↪failed call to cuInit: UNKNOWN ERROR (303)
2023-02-04 10:28:05.232648: I tensorflow/stream_executor/cuda/cuda_diagnostics.
 ↪cc:156] kernel driver does not appear to be running on this host (fv-az267-630):␣
 ↪/proc/driver/nvidia/version does not exist
2023-02-04 10:28:05.232962: I tensorflow/core/platform/cpu_feature_guard.cc:193]␣
 ↪This TensorFlow binary is optimized with oneAPI Deep Neural Network Library␣
 ↪(oneDNN) to use the following CPU instructions in performance-critical␣
 ↪operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate␣
 ↪compiler flags.
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 28, 28, 4)         200

 max_pooling2d (MaxPooling2D  (None, 14, 14, 4)        0
 )

 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 8)                 6280

 dense_1 (Dense)             (None, 10)                90

=================================================================
Total params: 6,570
Trainable params: 6,570
Non-trainable params: 0
_________________________________________________________________
```

14.6.4 Compile and fit model

```
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam", metrics=[
 ↪"accuracy"])
history = model.fit(X_train, y_train, epochs=3, validation_data=(X_valid, y_valid))
model.evaluate(X_test, y_test)
```

```
Epoch 1/3
938/938 [==============================] - 18s 18ms/step - loss: 0.7368 -␣
 ↪accuracy: 0.7487 - val_loss: 0.4763 - val_accuracy: 0.8381
Epoch 2/3
938/938 [==============================] - 17s 18ms/step - loss: 0.4436 -␣
 ↪accuracy: 0.8452 - val_loss: 0.4543 - val_accuracy: 0.8430
Epoch 3/3
938/938 [==============================] - 17s 18ms/step - loss: 0.3985 -␣
 ↪accuracy: 0.8612 - val_loss: 0.4125 - val_accuracy: 0.8576
313/313 [==============================] - 2s 6ms/step - loss: 0.4324 - accuracy:␣
 ↪0.8454
```

```
[0.4324043393135071, 0.8453999757766724]
```

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

CHAPTER

FIFTEEN

LECTURE 15: DEEP CNN ARCHITECTURES


Run in colab

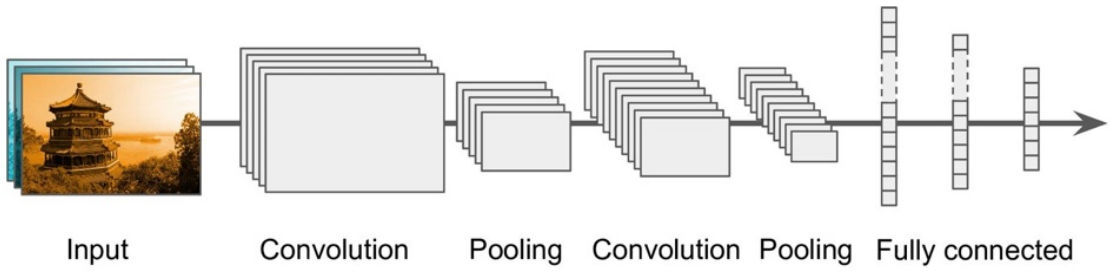
```
import datetime
now = datetime.datetime.now()
print("Version: " + now.strftime("%Y-%m-%d %H:%M:%S"))
```

```
Version: 2023-02-04 10:34:33
```

15.1 Classical CNN architecture

15.1.1 General CNN architecture

- (convolution, activation, pooling) $\times N_1$
- (fully connected layer) $\times N_2$



[Credit: Geron]

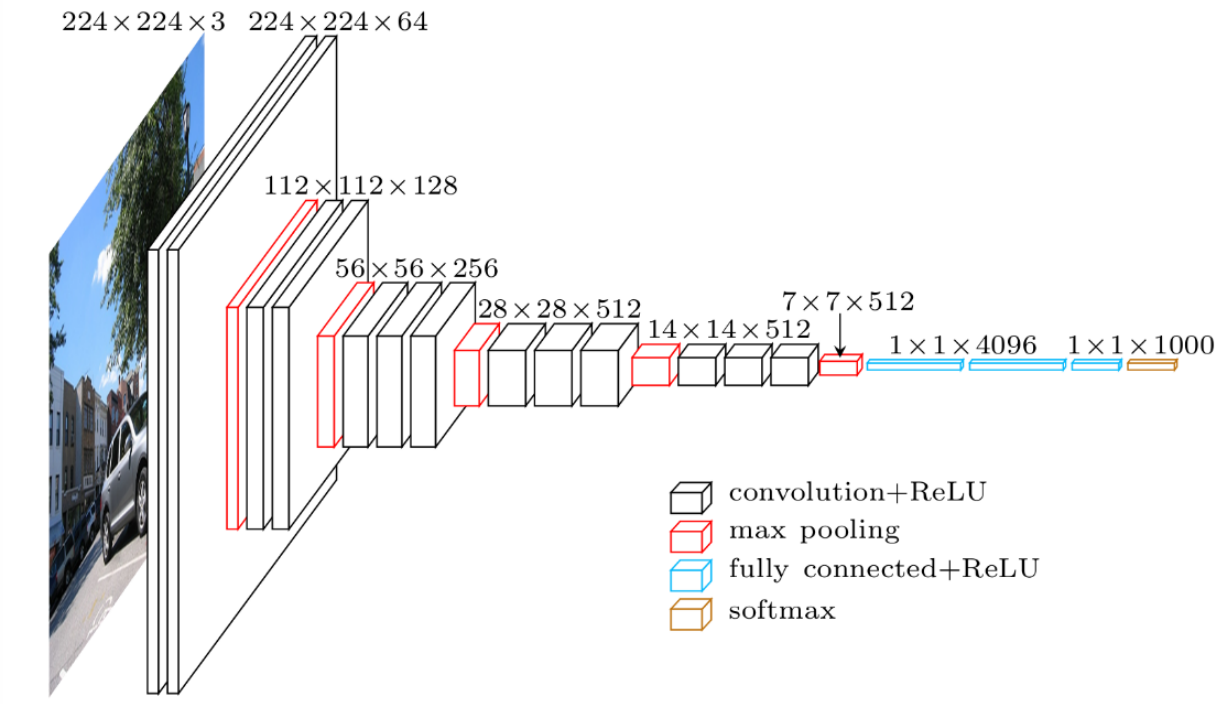
15.1.2 Decreasing resolution and increasing number of channels

In typical architectures decrease image resolution and increase number of channels as progress deeper in the network.

Decreasing image resolution (with the same convolutional kernel size), acts to increase the size of the receptor field of neurons as progress deeper in the network.

Increasing number of channels (i.e. filters), provides larger feature set (and is computationally possible since image resolution decreased).

For example VGG-16



Networks becoming very deep, e.g. VGG-16 has 138 million parameters.

Even the techniques we have discussed for training deep networks can begin to struggle.

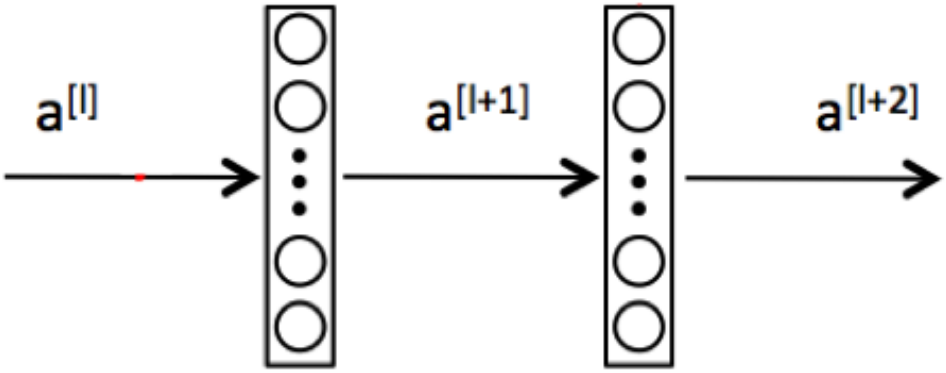
15.2 ResNet

ResNets (residual networks) were introduced to mitigate problems of training deep networks.

Introduce skip connections, which are a common feature of many of the latest cutting-edge deep learning architectures being developed today.

15.2.1 Standard neural network block

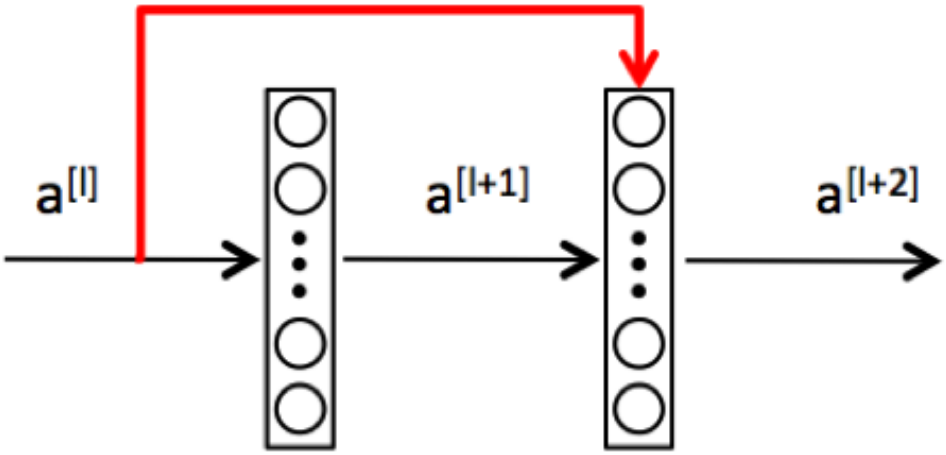
Recall the classical neural network block.



[Credit (modified)]

15.2.2 Residual block

ResNets introduce a residual block, with a connection that skips a layer and connects the activations of one layer to another layer deeper in the network.



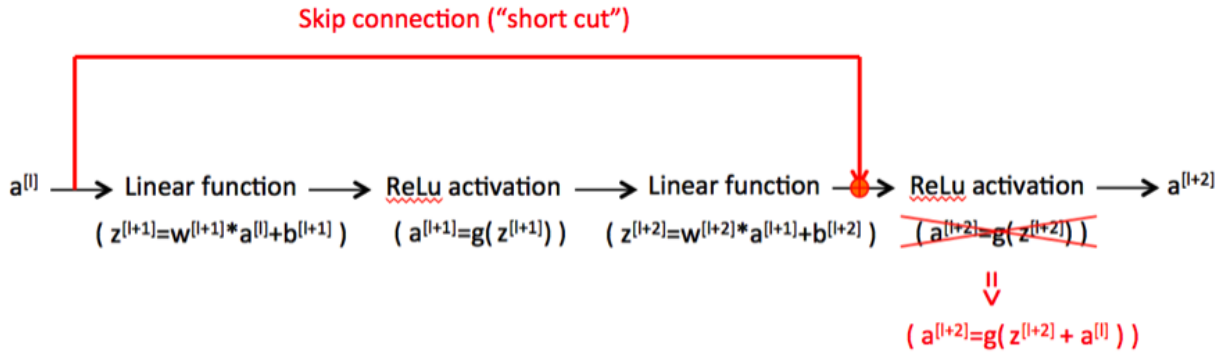
[Credit]

Information can flow directly from $a^{[l]}$ to $a^{[l+2]}$, and so can more easily flow deeper in network.

The connection is drawn as connecting *into* the subsequent layer (rather than after it) since the connection is typically made *before* the non-linear activation function, e.g. ReLU.

15.2.3 Residual connection

The residual connection involves *adding* the earlier activation, which is typically added before the activation.

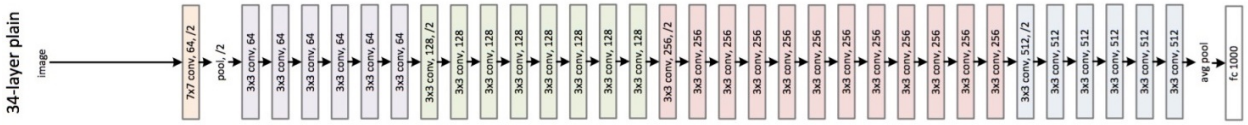


[Credit]

15.2.4 Residual network architecture

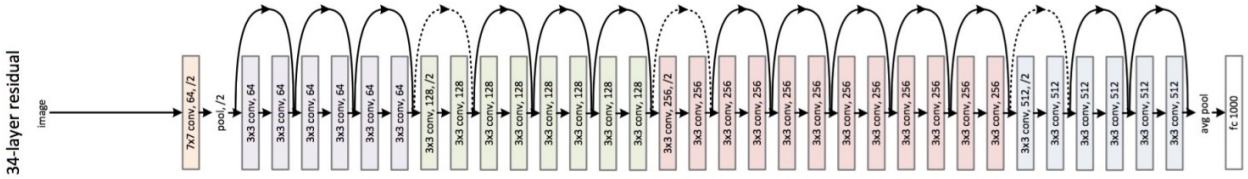
ResNet architectures constructed by concatenating residual blocks.

Standard architecture



[Credit: He et al.]

ResNet architecture



[Credit: He et al.]

In order to add activations at different levels, must have compatible shapes.

ResNets architectures often include operations that preseve the shape of activations. When shapes are not preservered a suitable adjustment is made in the skip connection, e.g. downsampling.

15.2.5 Why are ResNets effective?

ResNets revise the computation of the next activation as follows:

$$a^{[l+2]} = g(z^{[l+2]}) \rightarrow a^{[l+2]} = g(z^{[l+2]} + a^{[l]}).$$

Expanding this out in terms of the intermediate activation:

$$\begin{aligned} a^{[l+2]} &= g(z^{[l+2]} + a^{[l]}) \\ &= g(W^{[l+2]}a^{[l+1]} + b^{[l+2]} + a^{[l]}). \end{aligned}$$

Relatively easy for network to learn $W^{[l+2]} = 0$ and $b^{[l+2]} = 0$ (particularly with small weight initialisation and regularisation).

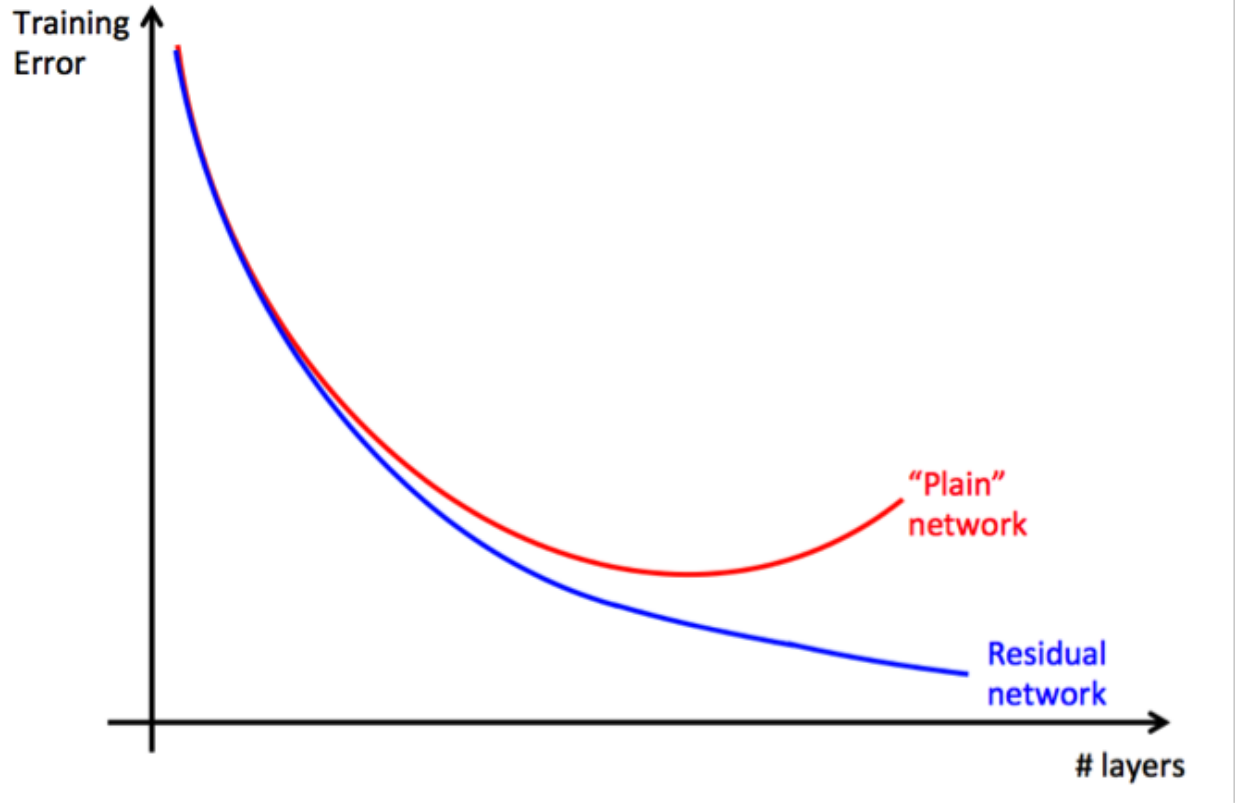
Then, for ReLU, $a^{[l+2]} = g(a^{[l]}) = a^{[l]}$.

So adding additional blocks generally shouldn't hinder performance and has the potential to further improve performance (each block can learn a residual to improve performance or leave essentially unchanged).

15.2.6 Performance for deep networks

Due to issues with training deep standard networks, performance generally starts to decrease if the network get too deep.

For ResNets, increasing depth generally continues to lead to improving performance.



[Credit]

15.2.7 Skip connections

The residual connection is an example of a skip connection.

In ResNets, the connection is made by *adding* activations. Alternatively, one could also concatenate layers to allow information to flow deeper into the network more easily.

Skip connections are a useful concept used widely in many cutting-edge architectures.

Will see concept again later in this lecture when we look at UNets.

Exercises: *You can now complete Exercise 1 in the exercises associated with this lecture.*

15.3 Inception

15.3.1 Motivation for Inception

What size convolutional kernel should we use?

Can decide empirically by cross-validation but could also use many at once and let network decide how to combine.

This is the general idea behind the Inception module.

Leverages 1x1 convolutions.

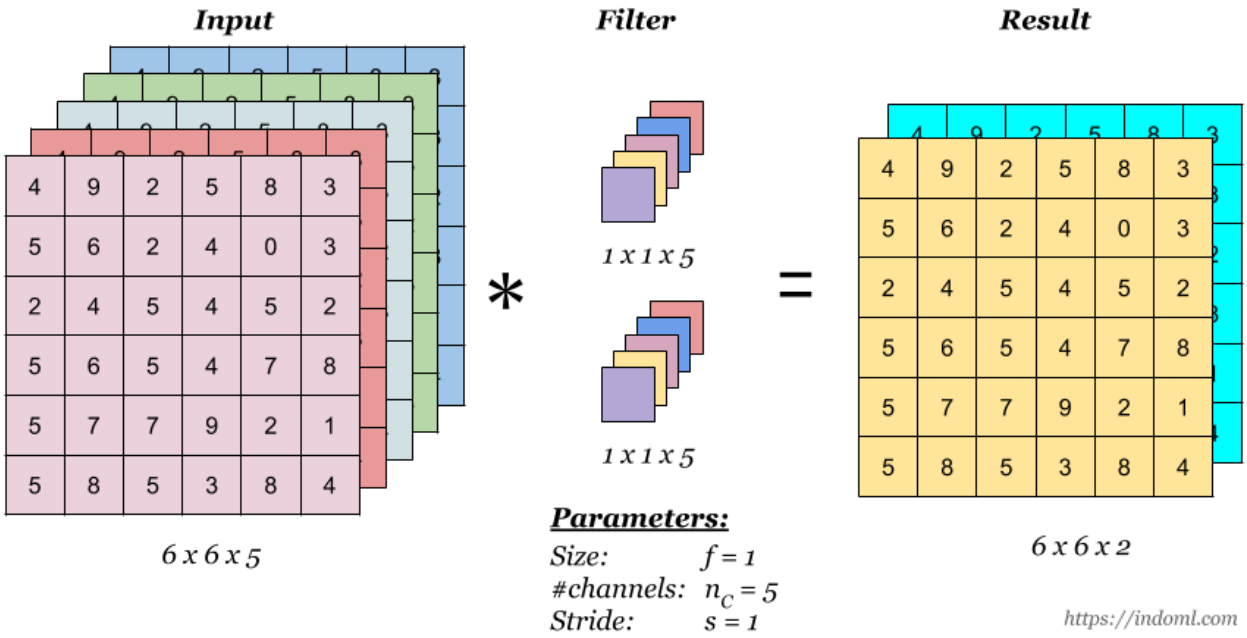
15.3.2 1x1 convolution

1x1 convolution is a powerful layer used in many cutting-edge deep learning architectures.

But isn't a 1x1 convolution just multiplying by a number?

It is for a single channel but not when considering multiple input and output channels.

Graphical illustration of 1x1 convolution



When have multiple input channels, weighting, summation, and activation applied across channels.

Acts like a fully-connected neural network across channels. Sometimes called *network in a network*.

Then repeat with multiple filters to create multiple output channels.

1x1 convolution to control channel size

1x1 convolutions often used to control the number of channels at intermediate points in a network, e.g. as a channel bottleneck (as well see shortly in the Inception module).

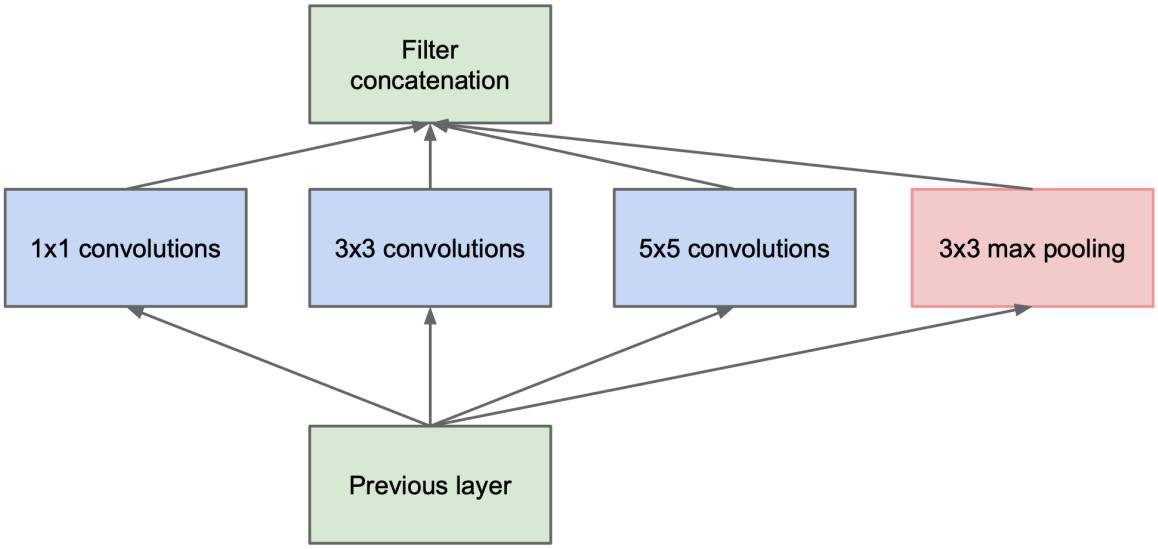
15.3.3 Inception module

We saw how 1x1 convolutions can be considered as a *network in a network*.

Need to go deeper!

General Inception module

Consider multiple kernel sizes at once and a pooling layer. Then concatenate outputs.

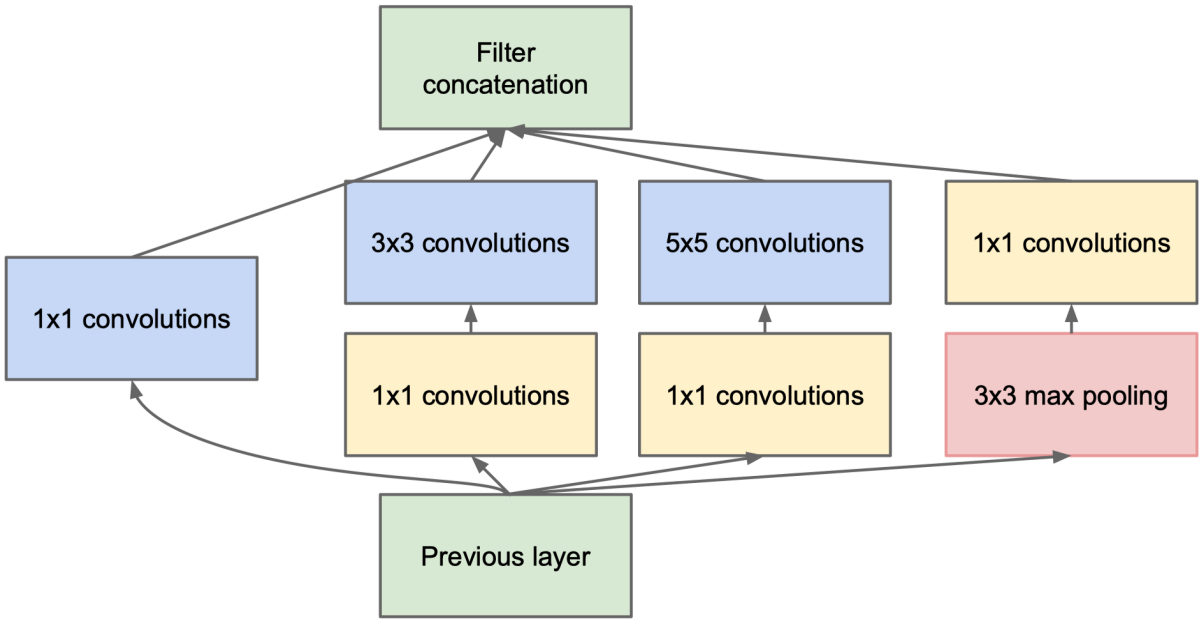


[Credit: Szegedy et al.]

This architecture can quickly become computationally demanding.

Inception module with 1x1 convolutions

Include 1x1 convolutions as a channel bottleneck to reduce computational cost.

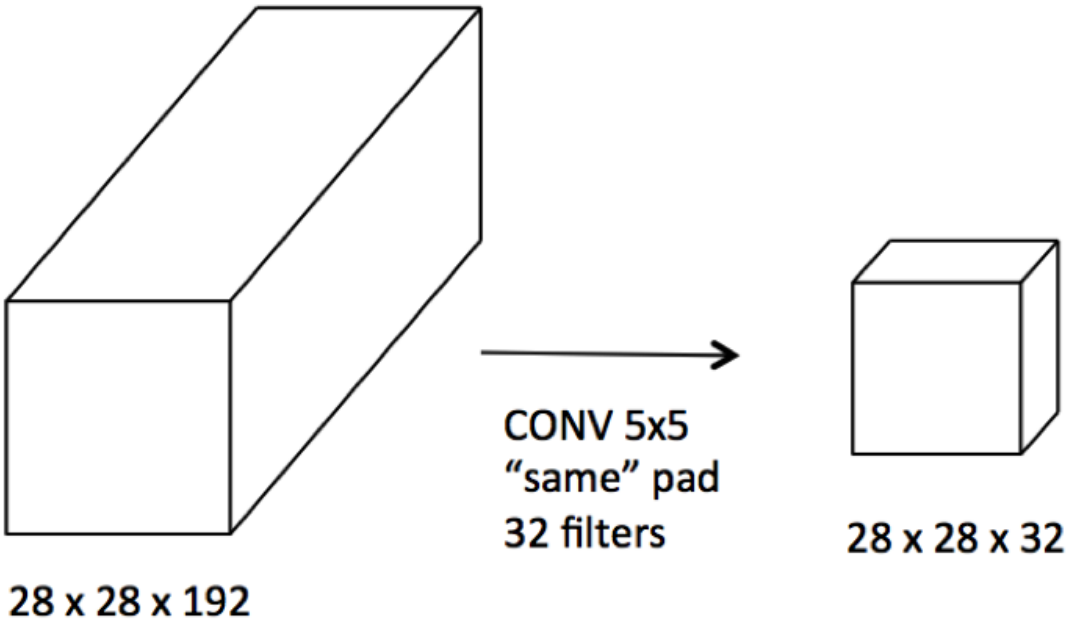


[Credit: Szegedy et al.]

Example of Inception module computational costs

Consider a 28x28 input feature map, with 192 channels. Require an output map with resolution 28x28 and 32 channels ("same" convolution so input and output resolutions the same).

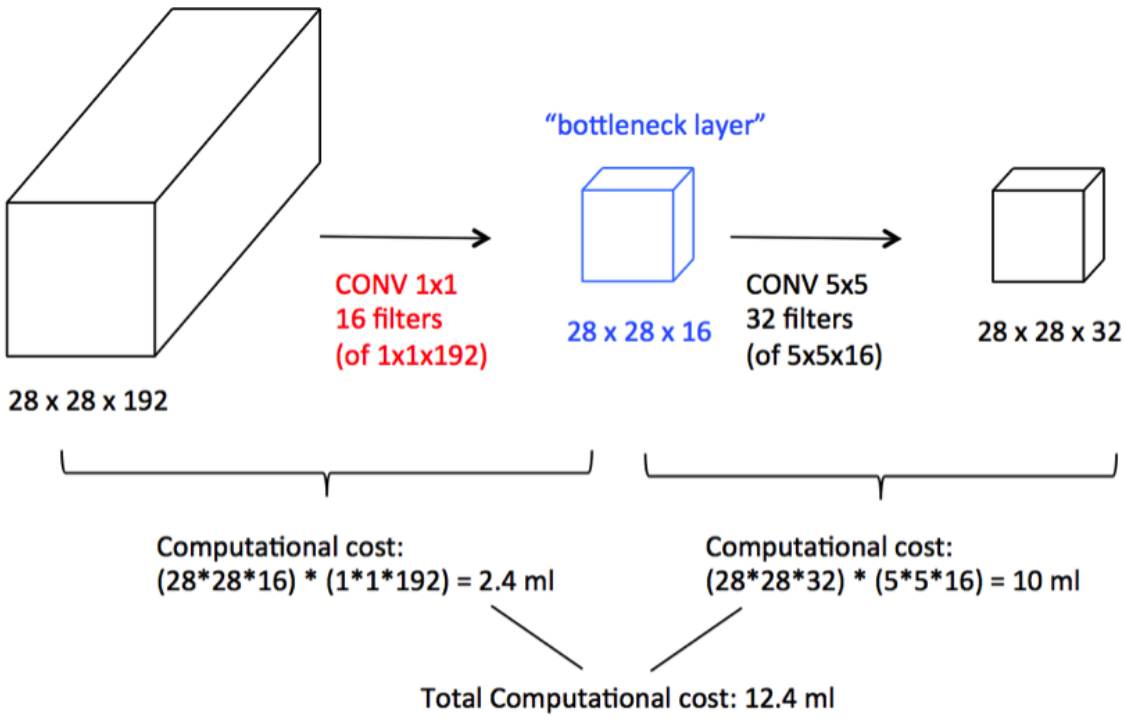
Standard convolutional layer



[Credit]

Number of flops = (28 x 28 x 32) x (5 x 5 x 192) = 120,422,400 = 120 million

1x1 convolution bottleneck



[Credit]

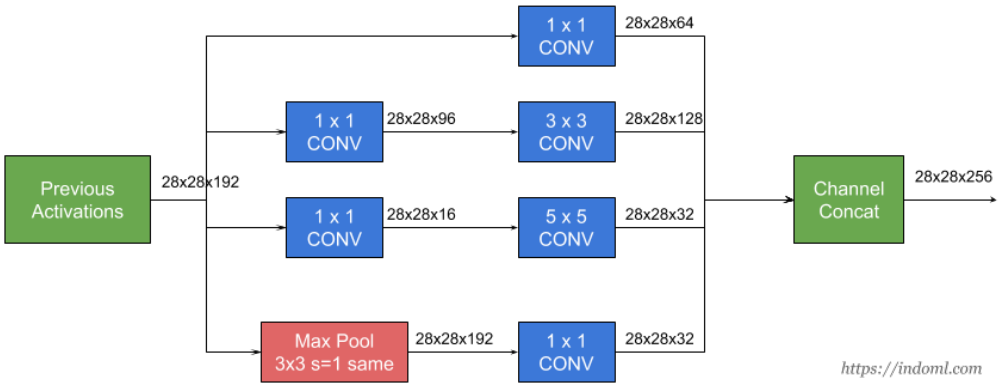
Number of flops = (28 x 28 x 16) x (1 x 1 x 192) + (28 x 28 x 32) x (5 x 5 x 16) = 2,408,448 + 10,035,200 = 12 million

Generally, performance is not significantly degraded (within reason).

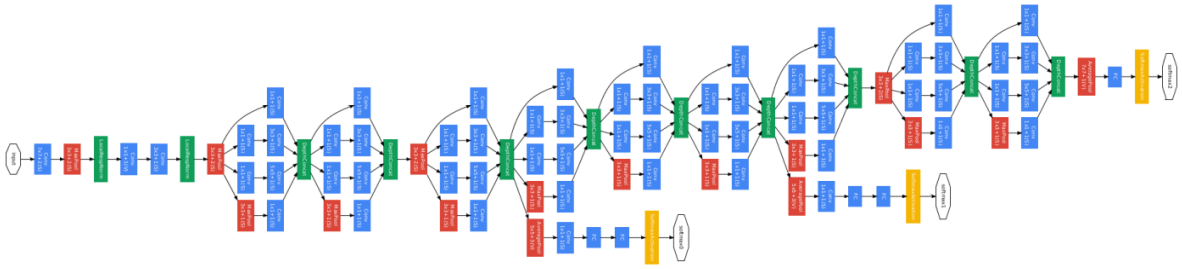
15.3.4 Inception network / GoogLeNet architecture

Overall Inception network architecture (also called GoogLeNet, cf. LeNet) involves combining multiple Inception modules.

Inception module (from above but drawn sideways)



Inception network



[Credit: Szegedy et al.]

15.4 MobileNet

15.4.1 Motivation for MobileNet

The architectures we've seen above are computationally demanding (even when leveraging 1x1 convolution channel bottlenecks).

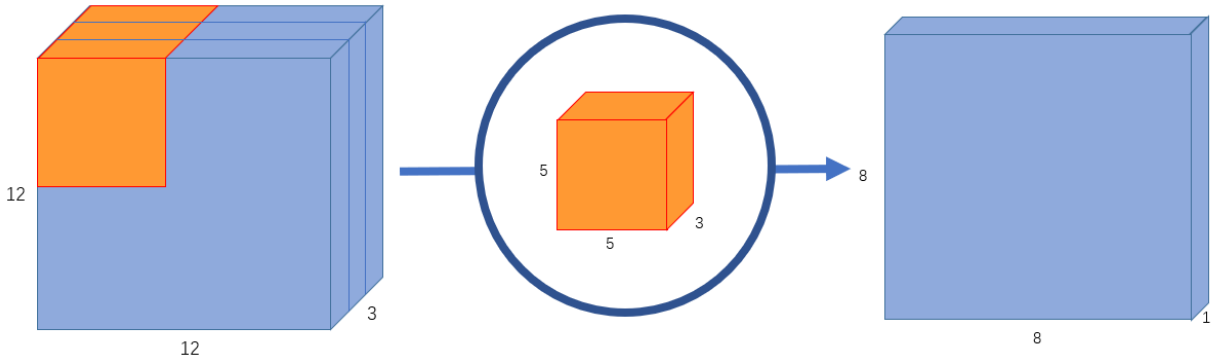
MobileNet architecture provides a more computationally efficient architecture, for example for low cost deployment on mobile devices (hence its name).

Based on *depthwise separable convolution*, which includes a *depthwise convolution*, followed by a *pointwise convolution*.

15.4.2 Recap standard convolution

Multiple input channels, single output channel

Consider 5x5 kernel with no padding and stride of one.

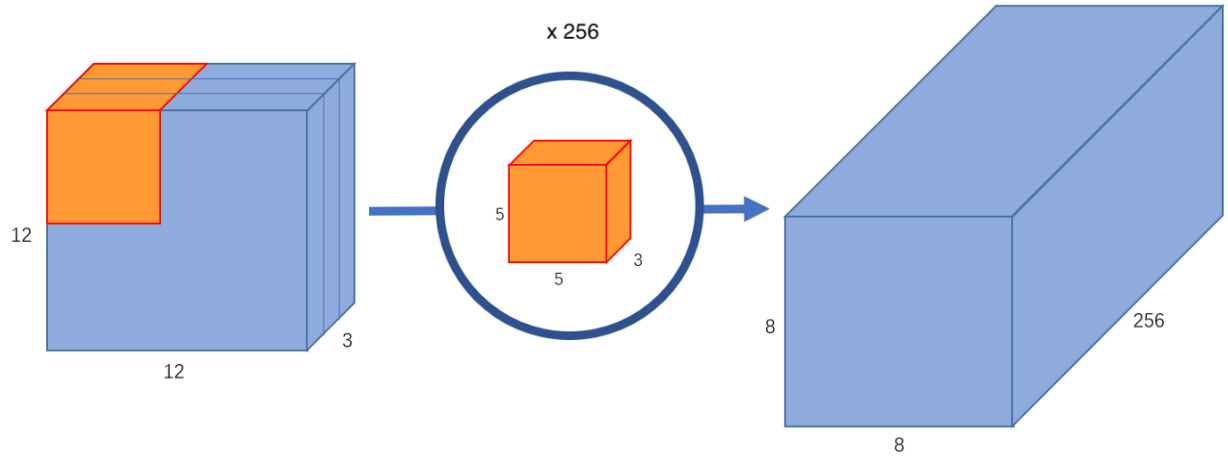


[Credit]

Number of flops = (8 x 8) x (5 x 5 x 3) = 4,800 (no padding)

Multiple input channels, multiple output channels

Repeat the above for each output channel.

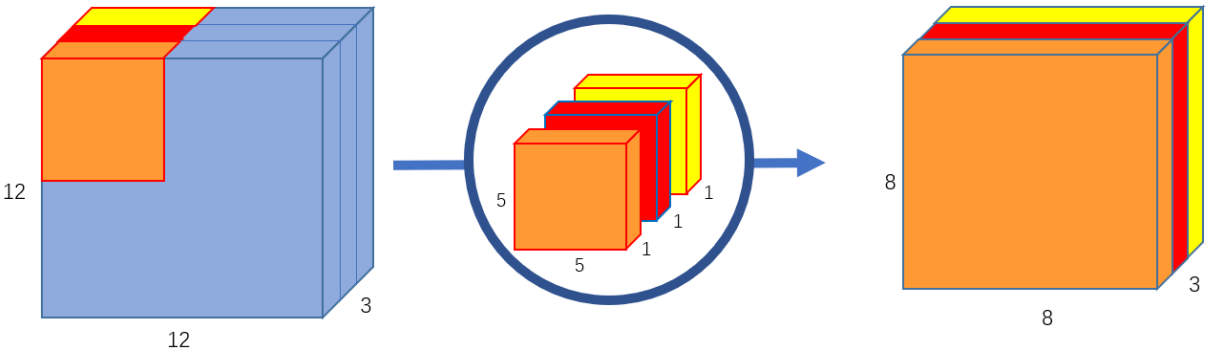


[Credit]

Number of flops = (8 x 8 x 256) x (5 x 5 x 3) = 1,228,800 = 1.2 million

15.4.3 Depthwise convolution

One filter for each channel (no summation over channels). Number of input and output channels are the same.



[Credit]

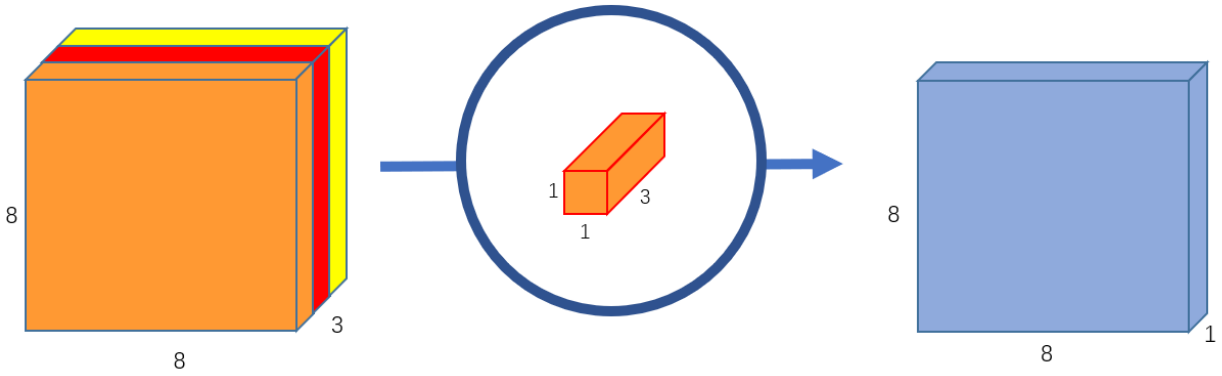
Number of flops = (8 x 8) x (5 x 5) x 3 = 4,800

Computation cost reduced to setting of a single output channel.

But no mixing across channels and number of output channels must be identical to input.

15.4.4 Pointwise convolution

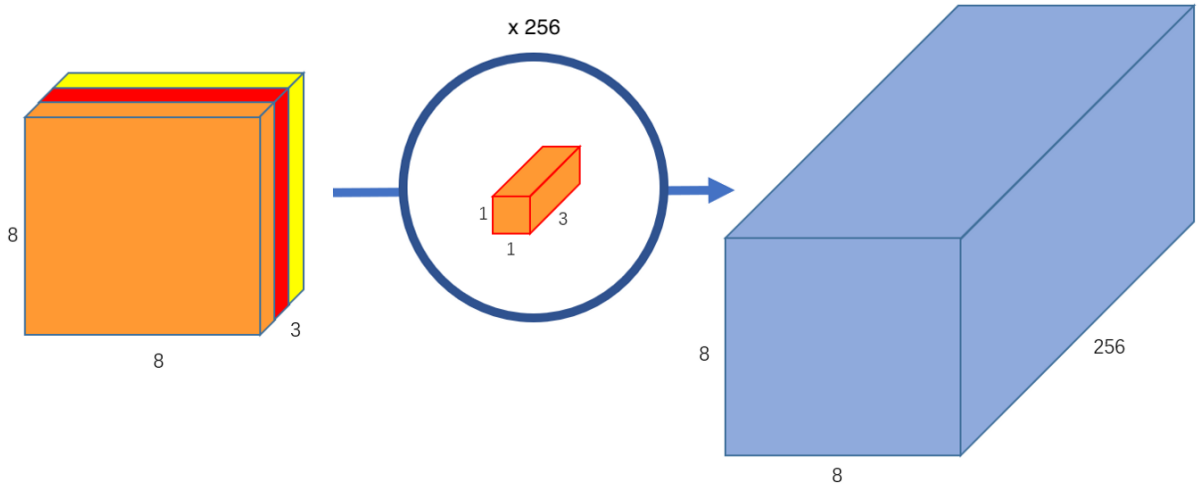
Introduce mixing of output channels using pointwise convolutions (1x1 convolutions).



[Credit]

Number of flops = (8 x 8) x (1 x 1 x 3) = 192

Can also control number of output channels.

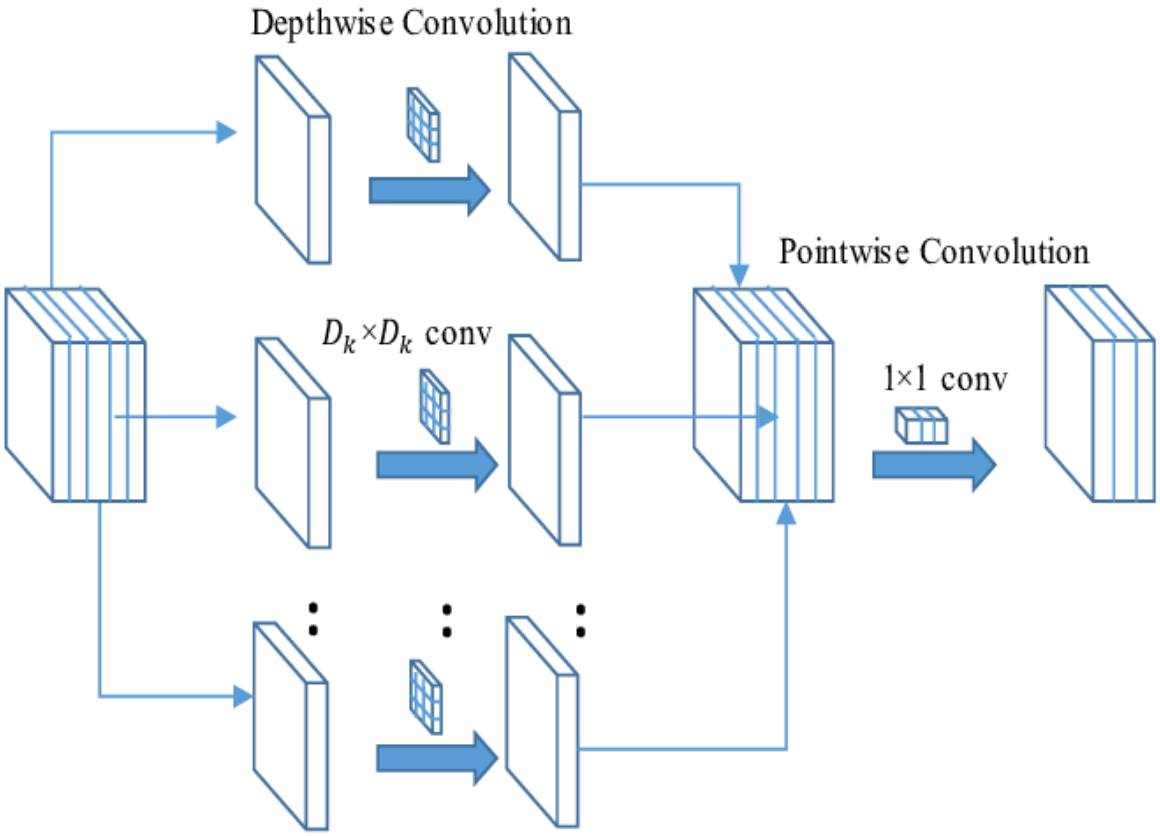


[Credit]

Number of flops = (8 x 8 x 256) x (1 x 1 x 3) = 49,152

15.4.5 Depthwise separable convolutions

Depthwise separable convolutions include depthwise convolution, followed by pointwise convolution. Separable since we separate the spatial and channel mixing.



[Credit]

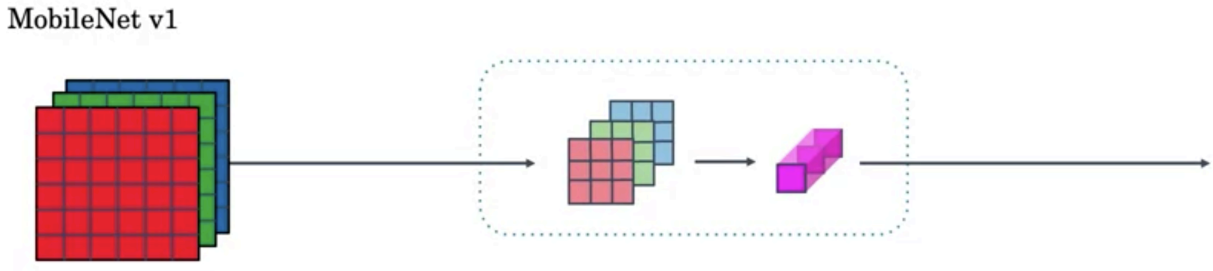
In the example considered above, 1.2 million flops (standard convolution) → 4,800 (depthwise convolutions) + 49,152 (pointwise convolutions) = 53,952 (same output resolution and number of channels).

Generally, performance is not significantly degraded (within reason).

15.4.6 MobileNet architectures

MobileNet v1

Use depthwise separable convolutions as building block for architecture.

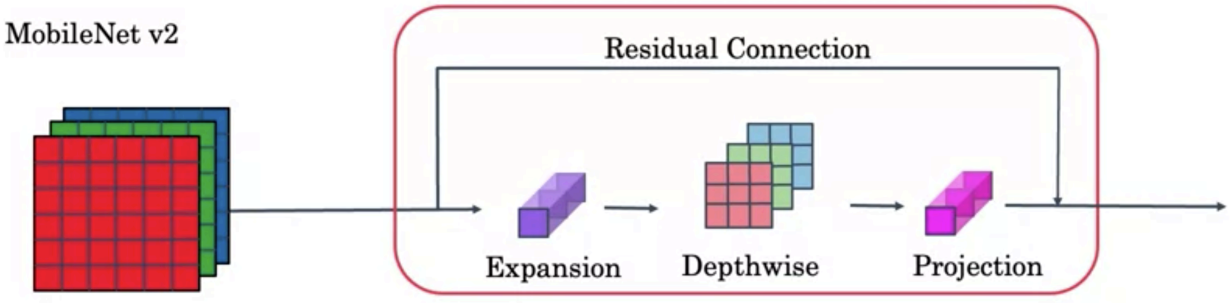


[Credit]

MobileNet v2

Add a pointwise 1x1 convolution before the depthwise separable convolution to increase number of channels in the intermediate stage (results in inverted channel bottleneck).

Also include residual connection (same as ResNet).



[Credit]

15.5 UNet

15.5.1 Motivation for UNet

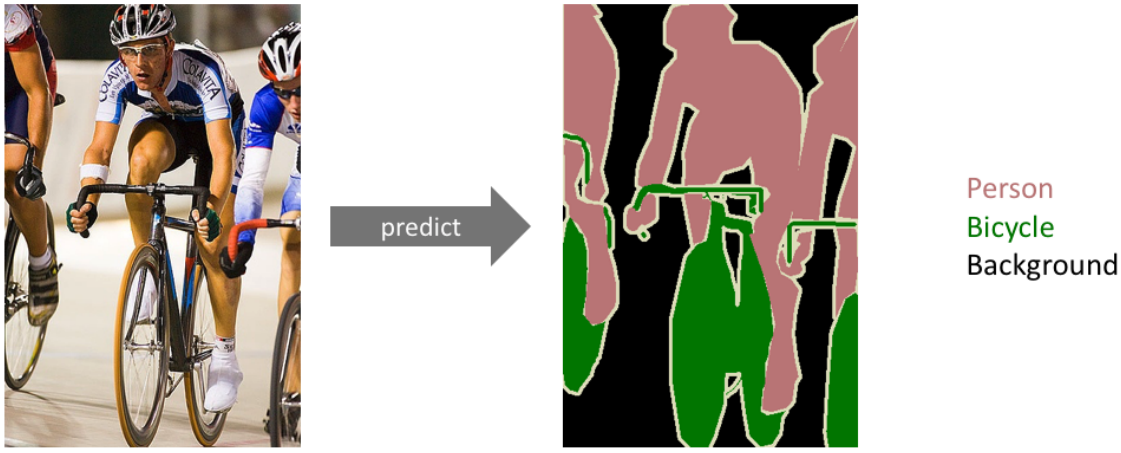
So far we've considered problems where the ouputs of the machine learning model are very low resolution, e.g. classification, low-dimensional regression.

For many problems we require high-resolution outputs for dense predictions.

We need to modify architectures to support dense predictions.

Semantic segmentation

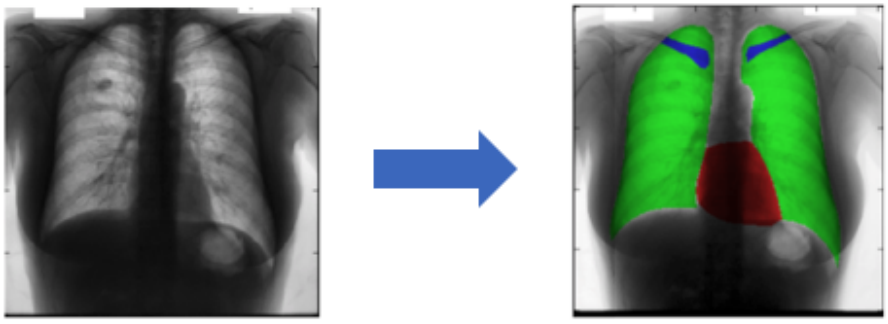
Semantic segmentation is a common type of problem where we require a high-resolution output with dense predictions. Goal is to predict a class for every single pixel in an image.



[Credit]

Segmentation of medical images

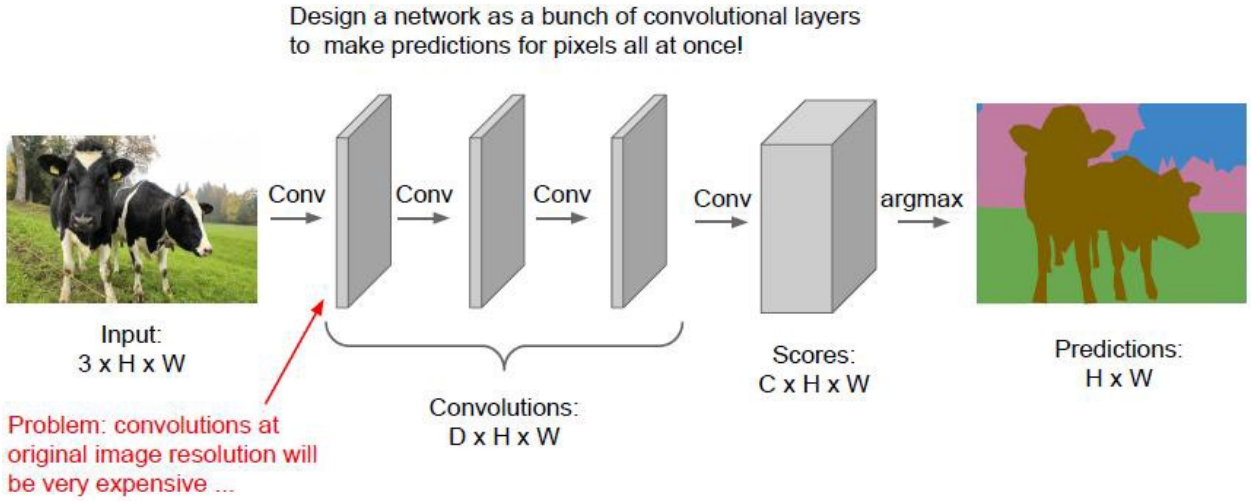
The UNet architecture was initially proposed for segmentation of medical images but has proven useful widely.



[Credit]

15.5.2 General approach

Naive approach is to adopt standard architectures and stay at high-resolutions

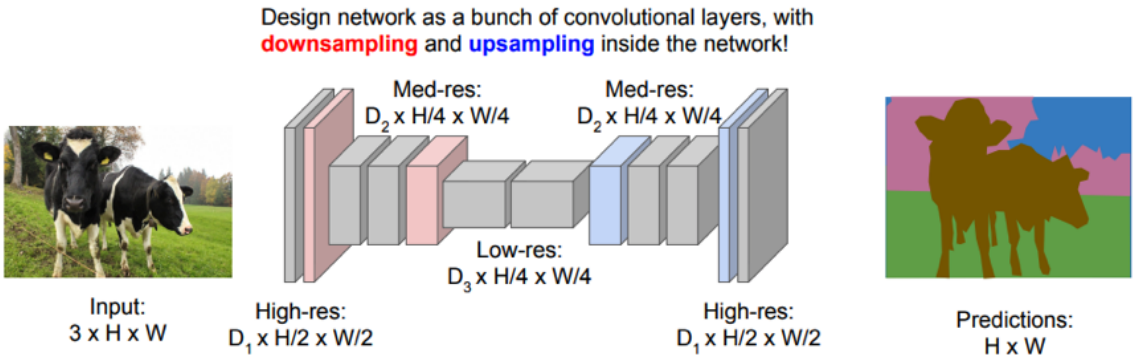


[Credit]

If don't reduce image resolutions through network, then we need very large kernels deeper into the network to have larger receptive fields. Also difficult to increase number of channels due to computational costs.

Becomes extremely computationally demanding.

Alternative approach is to reduce image resolution through network as usual but then include subsequent layers to increase resolution.



Solution: Make network deep and *work at a lower spatial resolution* for many of the layers.

[Credit]

Require an upsampling layer.

15.5.3 Transpose convolution

Transpose convolutional layer provides a way to upsample images.

Mathematical representation

Recall the convolution output is given by

$$z_{i,j} = \sum_{u,v} w_{u-i,v-j} x_{u,v},$$

where x is the input image, w is the filter (kernel) and i (u) and j (v) denote row and column indices, respectively.

Can represent convolution in matrix form:

$$\mathsf{W}\boldsymbol{x} = \begin{bmatrix} [w_{u-0}] & & & \\ & [w_{u-1}] & & \\ & & \ddots & \\ & & & [w_{u-m+1}] \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Convolution applied by matrix multiplication with W.

Consider multiplication of transpose of W:

$$\mathsf{W}^{\mathrm{T}}\boldsymbol{z} = \begin{bmatrix} [w_{u-0}] & & & \\ & [w_{u-1}] & & \\ & & \ddots & \\ & & & [w_{u-m+1}] \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_{m-1} \end{bmatrix} = \begin{bmatrix} [w_{u-0}] \\ \\ \\ \end{bmatrix} z_0 + \begin{bmatrix} \\ [w_{u-1}] \\ \\ \end{bmatrix} z_1 + \ldots + \begin{bmatrix} \\ \\ \\ [w_{u-m+1}] \end{bmatrix} z_{m-1}$$

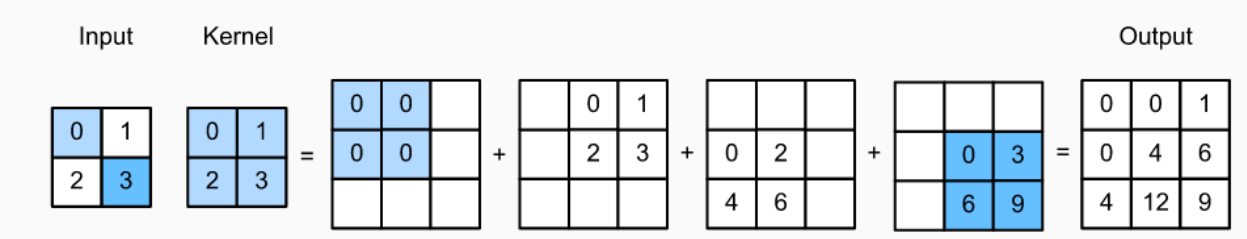
Can see transpose convolution involves placing shifted kernels down on *output*, weighting by input at shifted position and summing.

Contrast with convolution, which involves placing shifted kernels down on *input*, weighting by inputs that overlap and summing.

Graphical representation

Consider input, kernel and output shape.

Transpose convolution is given by placing shifted kernels down on *output*, weighting by input at shifted position and summing.



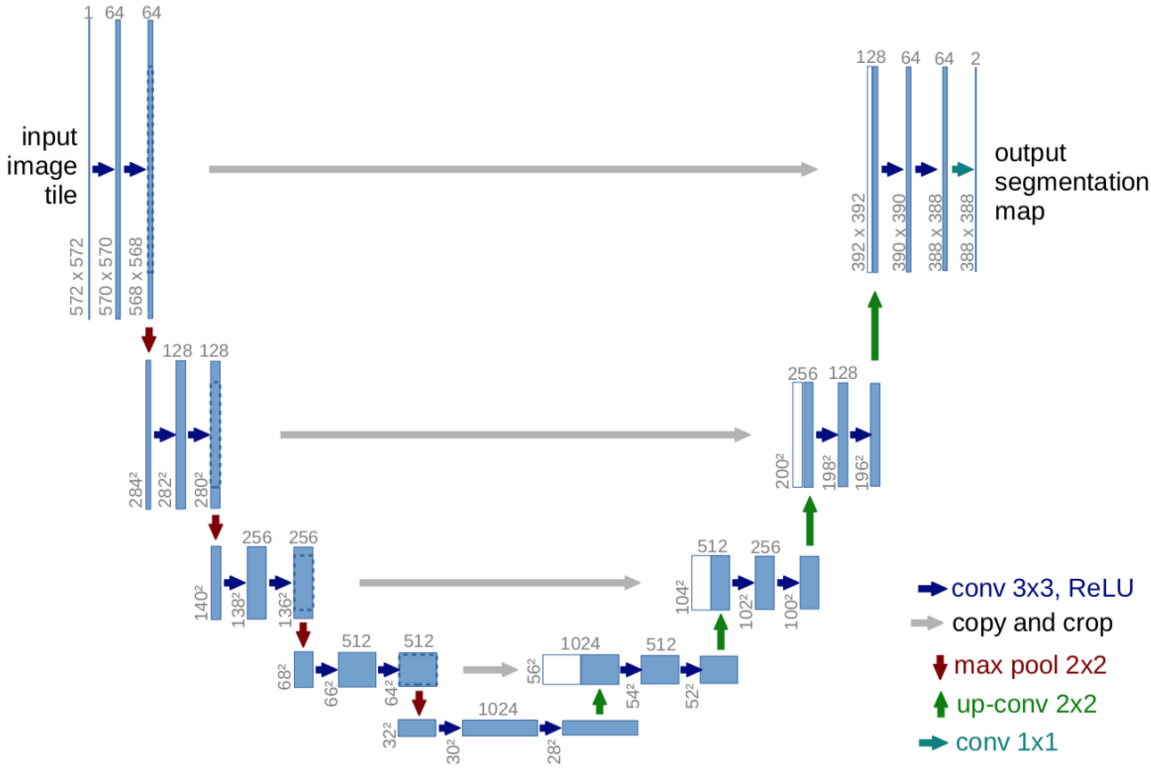
[Credit]

15.5.4 UNet architecture

UNet architecture leverages standard convolutions and pooling for downsampling stages.

Then adopts transpose convolution for upsampling stages.

Also includes skip connections to copy higher resolution feature maps from the downsampling path to the upsampling path.



[Credit]