

Data Science & ML Course

Lesson #25 Deep Learning Fundamentals II

Ivanovitch Silva
December, 2018



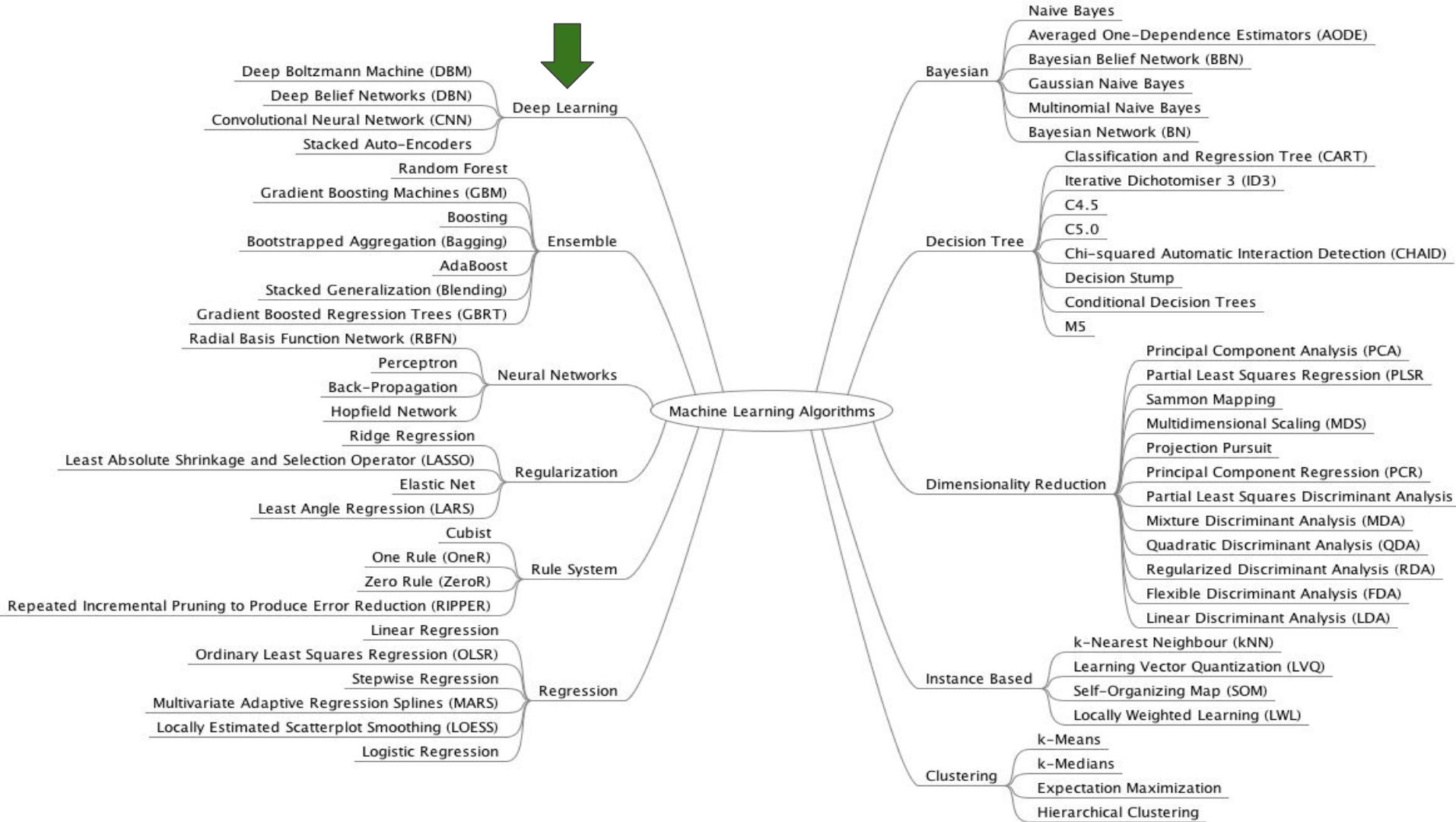
Update from repository

```
git clone https://github.com/ivanovitchm/datascience2machinelearning.git
```

Or

```
git pull
```





Agenda

1. Introduction
2. Mathematical building blocks of neural networks
3. Getting started with neural networks
4. Classifying movie reviews: a binary classification example (section 3.4)
5. Classifying newswires: a multiclass classification problem (section 3.5)
6. Predicting houses price: a regression problem (section 3.6)

#1 Introduction

ARTIFICIAL INTELLIGENCE

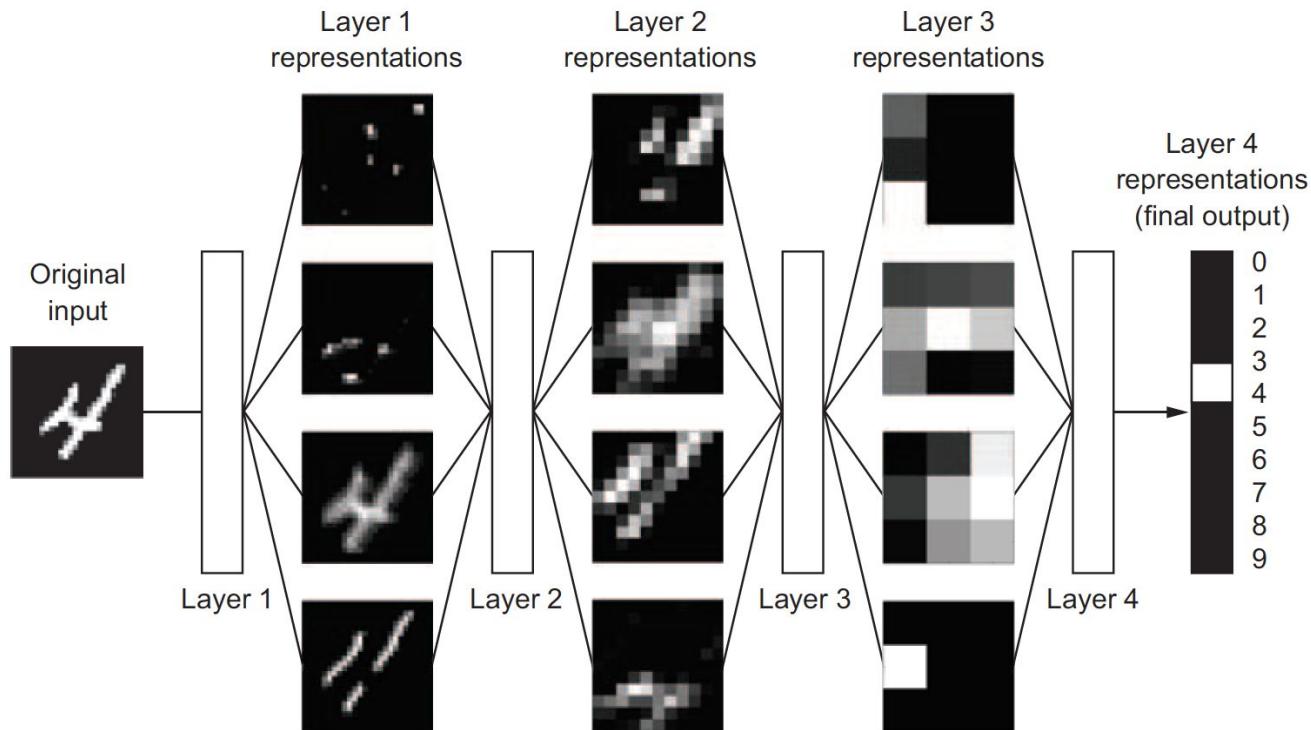
Early artificial intelligence stirs excitement.



Symbolic AI (rules)



What is Deep Learning?

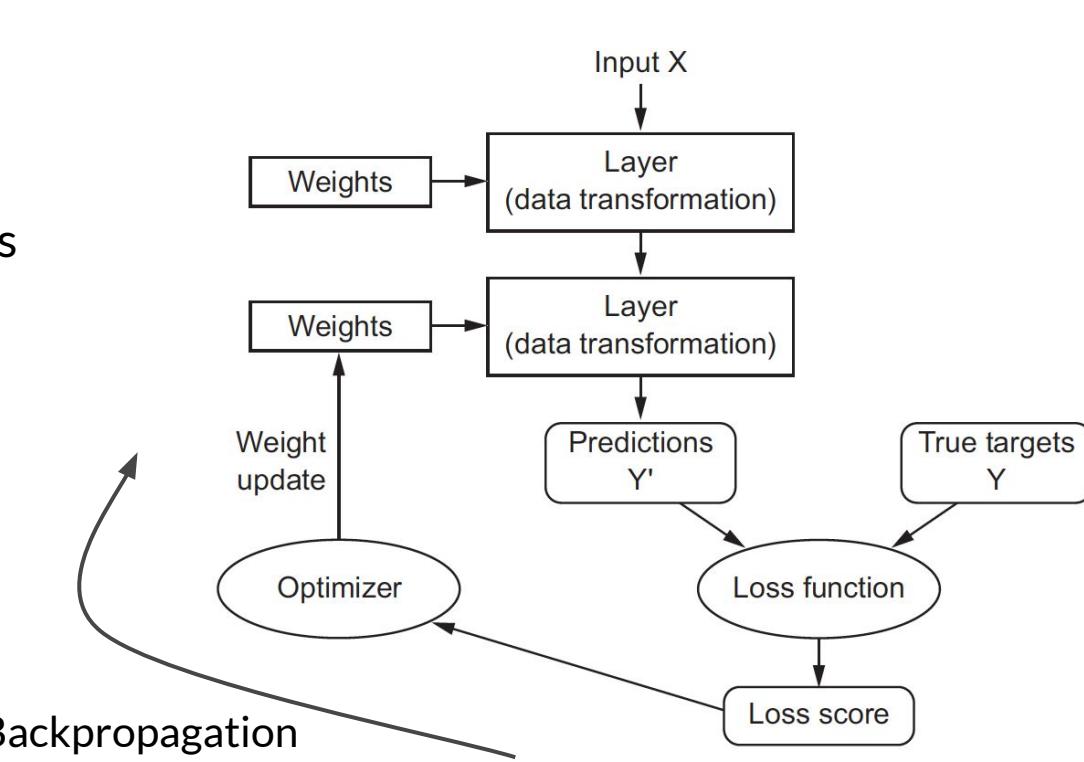


“Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [...] The key aspect of deep learning is that these layers are not designed by human engineers: they are learned from data using a general-purpose learning procedure”

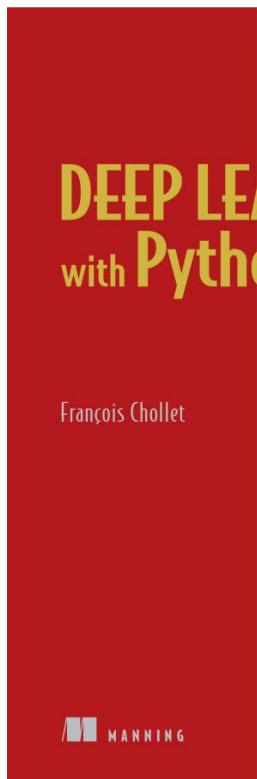
[Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, Nature 2015](#)

Understanding how DL works

Finding the right values
of weights which
minimize the error



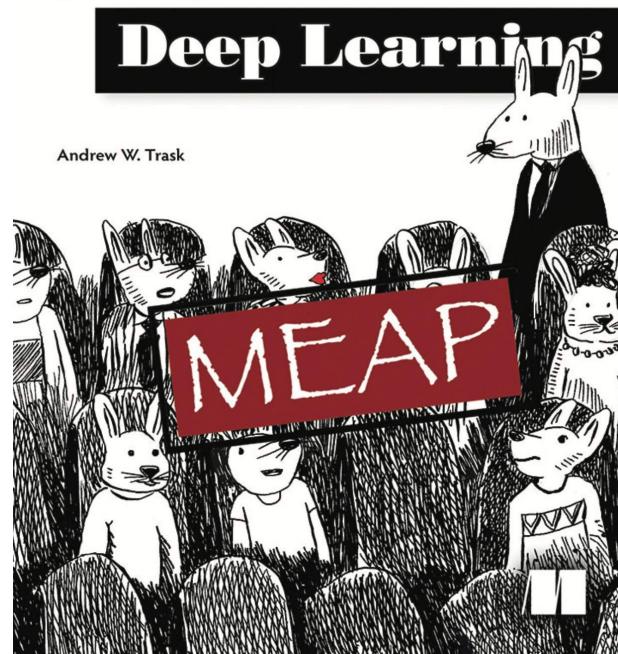
References



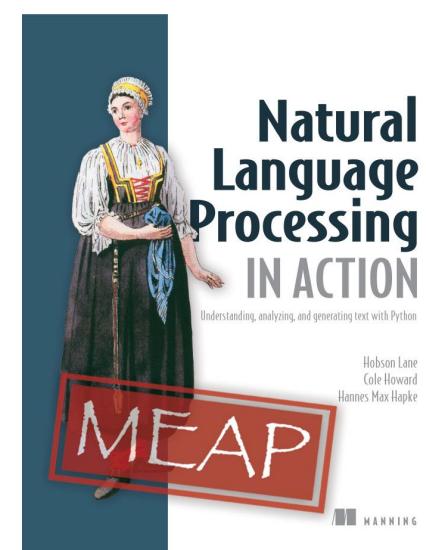
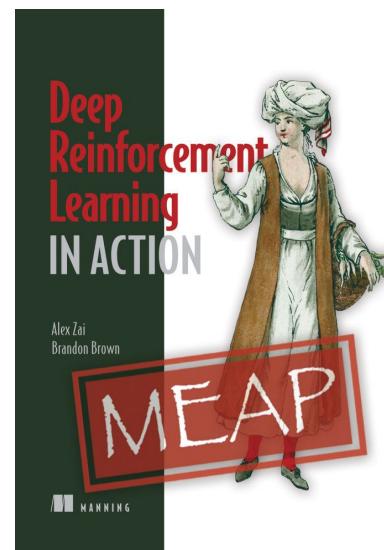
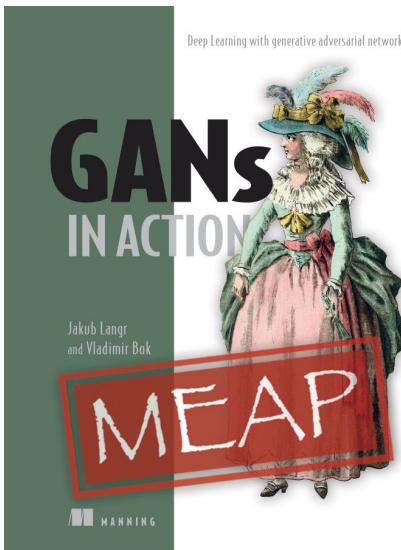
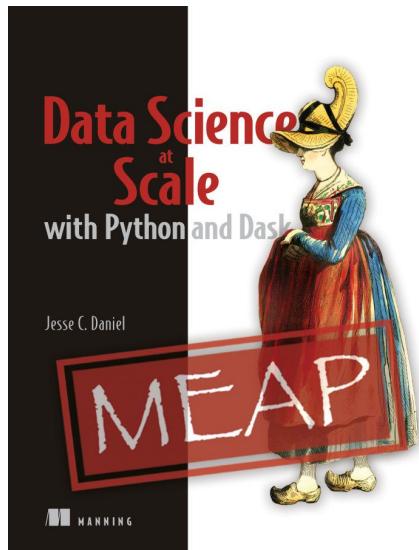
grokking

Deep Learning

Andrew W. Trask



References





Deep Learning Specialization

Deep Learning is transforming multiple industries. This five-course specialization will help you understand Deep Learning fundamentals, apply them, and build a career in AI.



Course 1

Neural Networks
and Deep Learning

Course 2

Improving Deep
Neural Networks

Course 3

Structuring
Machine Learning
Projects

Course 4

Convolutional
Neural Networks

Course 5

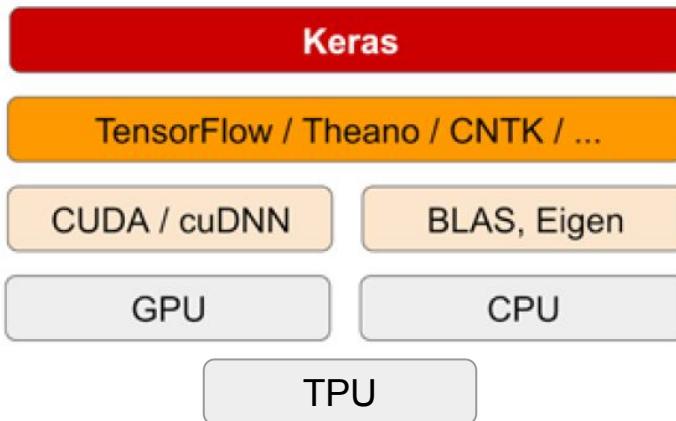
Sequence Models

AI SUPER- POWERS CHINA, SILICON VALLEY, AND THE NEW WORLD ORDER

KAI-FU LEE



Introduction to Keras



Keras has the following key features:

- It allows the same code to run seamlessly on CPU, GPU or TPU.
- Support for: convolutional networks, recurrent networks, and any combination of both.
- It supports arbitrary network architectures
- It can be freely used in commercial projects (MIT License).
- It's compatible with any version of Python from 2.7 to 3.6

Install Keras

```
conda install -c conda-forge tensorflow keras
```

#2 Mathematical building blocks of neural networks

Mathematical building blocks of neural networks



1. A first example of a neural network
2. Tensors and tensor operations
3. How neural networks learn via backpropagation and gradient descent

Google Colaboratory Config.

```
1 import keras
2 from keras import backend as K
3
4 print('Using Keras version:', keras.__version__,
5       '\nbackend:', K.backend())
```

Using Keras version: 2.2.4
backend: tensorflow

Gen RAM Free: 12.5 GB		Proc size: 1.1 GB	
GPU RAM Free: 11271MB		Used: 170MB	Util 1% Total 11441MB

Notebook settings

Runtime type

Python 3

Hardware accelerator

GPU

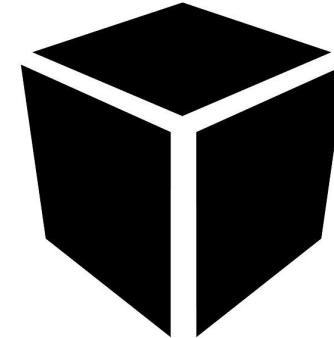


A first look at a neural network

MNIST Database



1. Loading the data
2. Define the network architecture
3. Compilation step
4. Preparing data
5. Train the network
6. Evaluate the network



Loading the Data

```
1 # Loading the MNIST dataset in Keras
2 from keras.datasets import mnist
3
4 # The images are encoded as Numpy arrays,
5 # and the labels are an array of digits, ranging
6 # from 0 to 9. The images and labels have a one-to-one correspondence.
7
8 (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
1 # first ten train labels
2 train_labels[:10].
```

```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4], dtype=uint8)
```

```
Type of train_images <class 'numpy.ndarray'>
Type of train_labels <class 'numpy.ndarray'>
```

```
Shape of train data: images (60000, 28, 28), labels (60000,)
Shape of test data: images (10000, 28, 28), labels (10000,)
```



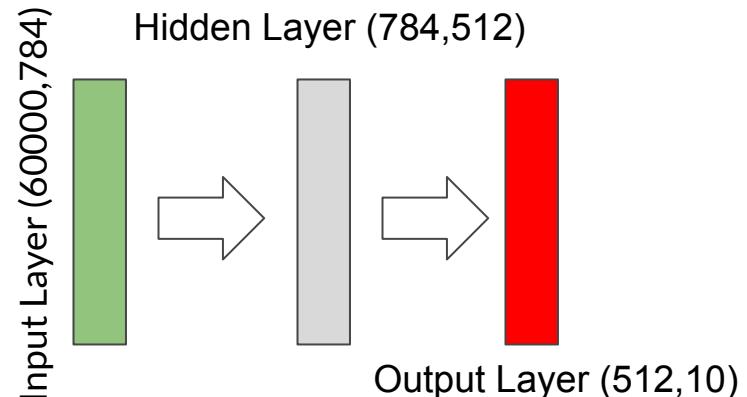
The network architecture

```
from keras import models
from keras import layers

# The network architecture
network = models.Sequential()

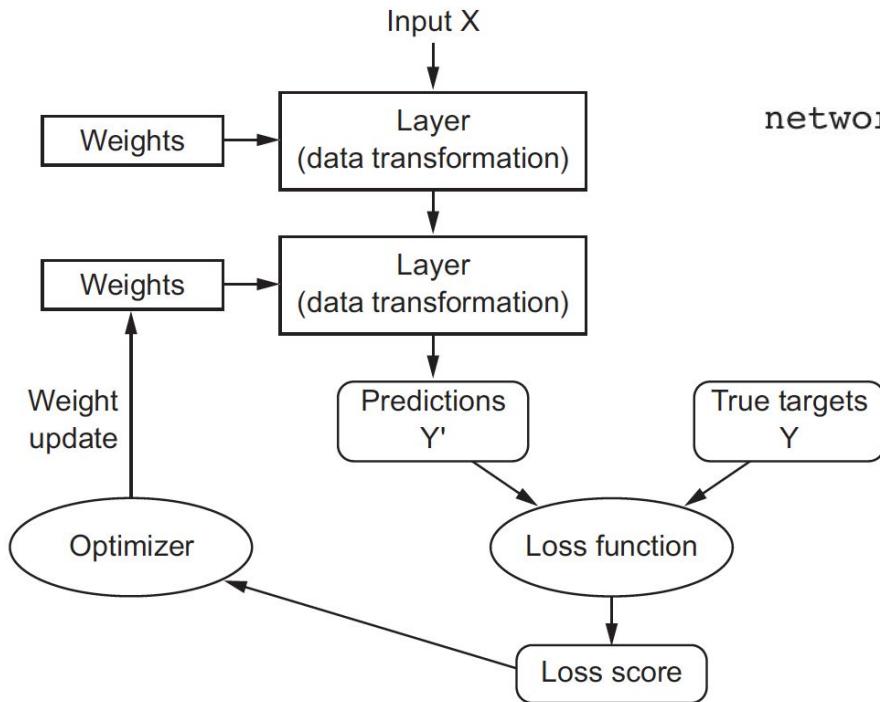
# Hidden layer
network.add(layers.Dense(512,
                        activation='relu',
                        input_shape=(28 * 28,)))

# Output layer
network.add(layers.Dense(10, activation='softmax'))
```



#1 parameters = 784×512 weights + 512 bias
#2 parameters = 512×10 weights + 10 bias

The compilation step



```
network.compile(optimizer='rmsprop',  
                 loss='categorical_crossentropy',  
                 metrics=[ 'accuracy' ])
```

Preparing data

```
1 train_images = train_images.reshape((60000, 28 * 28))  
2 train_images = train_images.astype('float32') / 255  
3  
4 test_images = test_images.reshape((10000, 28 * 28))  
5 test_images = test_images.astype('float32') / 255
```

```
1 from keras.utils import to_categorical  
2  
3 train_labels = to_categorical(train_labels)  
4 test_labels = to_categorical(test_labels)
```

```
array([[0., 0., 0., 0., 1., 0., 0., 0., 0.],  
       [1., 0., 0., 0., 0., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 1., 0., 0., 0., 0.],  
       [0., 1., 0., 0., 0., 0., 0., 0., 0.],  
       [0., 0., 0., 0., 0., 0., 0., 0., 1.]], dtype=float32)
```



Train the network

```
1 | network.fit(train_images, train_labels, epochs=5, batch_size=128).
```

```
Epoch 1/5
60000/60000 [=====] - 2s 39us/step - loss: 0.2565 - acc: 0.9258
Epoch 2/5
60000/60000 [=====] - 2s 41us/step - loss: 0.1035 - acc: 0.9697
Epoch 3/5
60000/60000 [=====] - 2s 41us/step - loss: 0.0679 - acc: 0.9799
Epoch 4/5
60000/60000 [=====] - 2s 41us/step - loss: 0.0490 - acc: 0.9849
Epoch 5/5
60000/60000 [=====] - 2s 41us/step - loss: 0.0370 - acc: 0.9888
<keras.callbacks.History at 0x7f266efecd30>
```

Network Summary

```
1 | network.summary()
```

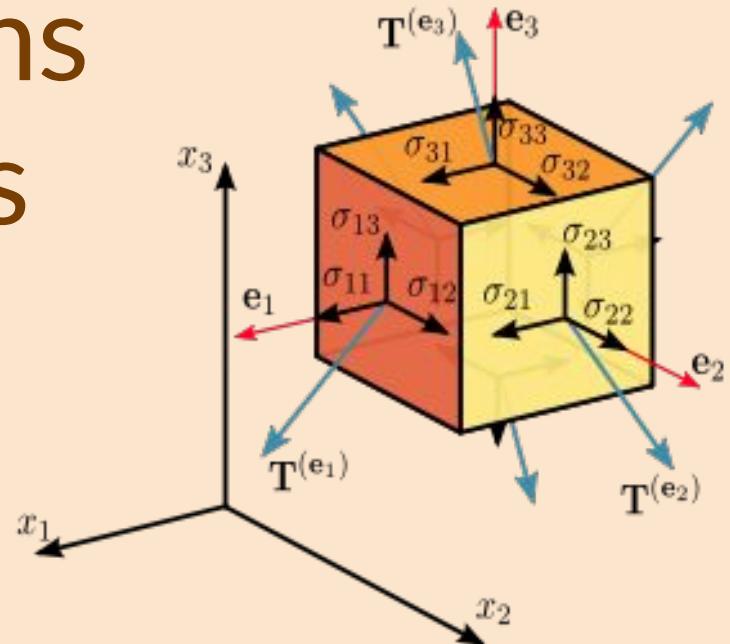
Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 512)	401920
dense_2 (Dense)	(None, 10)	5130
<hr/>		
Total params: 407,050		
Trainable params: 407,050		
Non-trainable params: 0		

Evaluate the network

```
1 test_loss, test_acc = network.evaluate(test_images, test_labels)
2 print('test_acc:', test_acc)
```

```
10000/10000 [=====] - 0s 46us/step
test_acc: 0.9777
```

data representations for neural networks



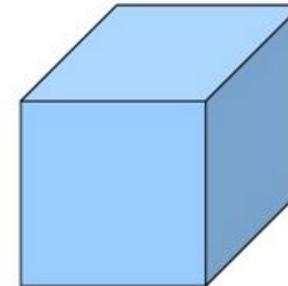
Multidimensional numpy arrays - TENSOR



1d-tensor



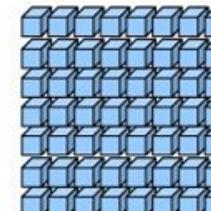
2d-tensor



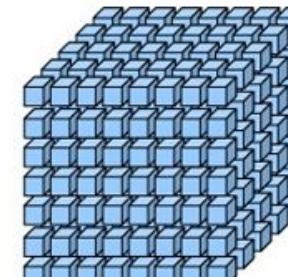
3d-tensor



4d-tensor



5d-tensor



6d-tensor

Tensors - Dimensions

```
1 scalar = np.array(12)  
2 scalar.ndim
```

0

```
1 tensor1D = np.array([12, 3, 6, 14])  
2 tensor1D.ndim
```

1

```
1 tensor2D = np.array([[5, 78, 2, 34, 0],  
2 [6, 79, 3, 35, 1],  
3 [7, 80, 4, 36, 2]])  
4 tensor2D.ndim
```

2

```
1 tensor3D = np.array([[[5, 78, 2, 34, 0],  
2 [6, 79, 3, 35, 1],  
3 [7, 80, 4, 36, 2]  
4 ],  
5 [[[5, 78, 2, 34, 0],  
6 [6, 79, 3, 35, 1],  
7 [7, 80, 4, 36, 2]  
8 ],  
9 [[[5, 78, 2, 34, 0],  
10 [6, 79, 3, 35, 1],  
11 [7, 80, 4, 36, 2]]  
12 .1])  
13 tensor3D.ndim
```

3



Real-world examples of data tensors

Vector data - the most common case

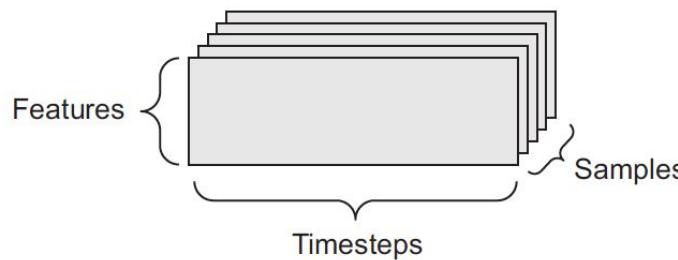
An actuarial dataset of people, where we consider each person's

- Age
- ZIP code
- income

An entire dataset of 100,000 people can be stored in a 2D tensor of shape $(100000, 3)$.

Real-world examples of data tensors

Sequence data



A dataset of tweets

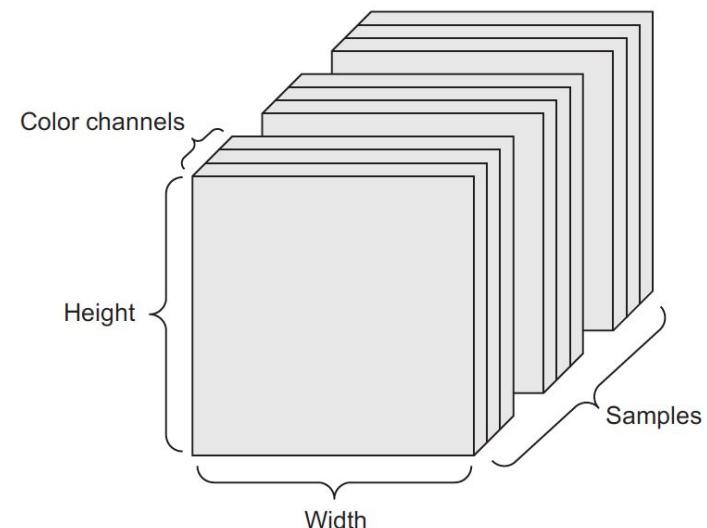
- Each tweet as a sequence of 280 characters
- An alphabet of 128 unique characters. In this setting, each character can
- Then each tweet can be encoded as a 2D tensor of shape $(280, 128)$

A dataset of 1 million tweets can be stored in a tensor of shape $(1000000, 280, 128)$.

Real-world examples of data tensors

- A batch of 128 grayscale images of size 256 x 256 could thus be stored in a tensor of shape (128, 256, 256, 1)
- A batch of 128 color images could be stored in a tensor of shape (128, 256, 256, 3)

Image data



Real-world examples of data tensors

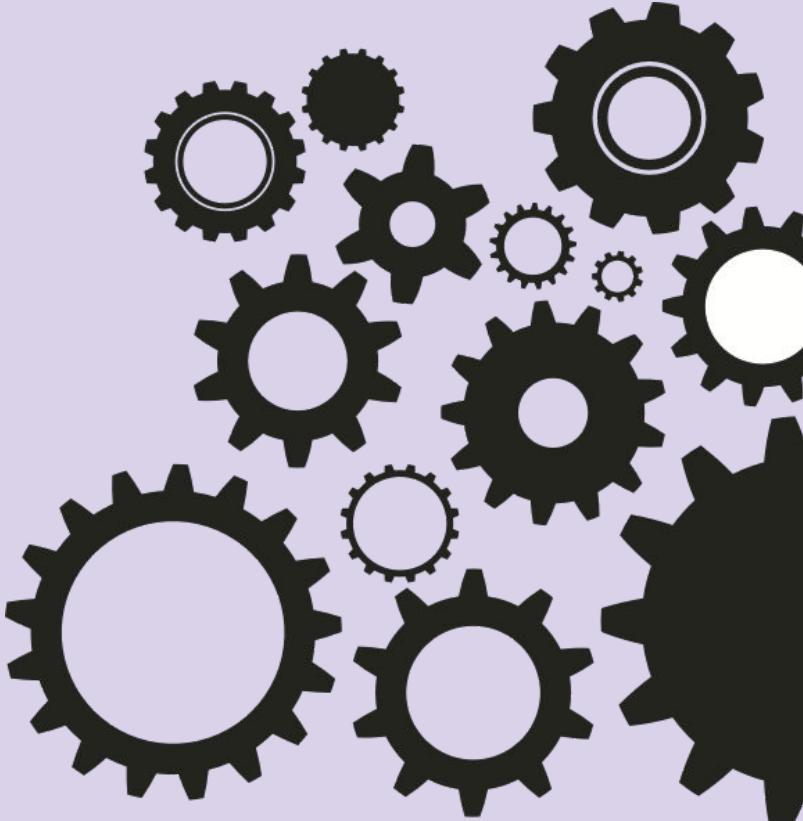
Video data is one of the few types of real-world data for which you'll need **5D tensors**.

Video data

- 60-second, 144 x 256 YouTube video clip
 - 4 frames per second - 240 frames
- A batch of four such video clips
 - $(4, 240, 144, 256, 3)$ - 106,168,320 values!

If the dtype of the tensor was float32, then each value would be stored in 32 bits, so the tensor would represent **405 MB**.

the gears of neural networks: tensor operations



Tensor Operations

- Add tensors
- Multiply tensors
- Reshape tensors
- Broadcasting
- Element-wise
- Relu - max(x,0)
- And so on.

```
keras.layers.Dense(512, activation='relu')
```



```
output = relu(dot(W, input) + b)
```

- We have three tensor operations here:
 - a **dot product** (dot) between the input tensor and a tensor named W;
 - Tensor product operation
 - an **addition (+)** between the resulting 2D tensor and a vector b (1D);
 - Broadcast operation
 - a **relu** operation. $\text{relu}(x)$ is $\max(x, 0)$.
 - Element-wise operation

Tensor Operations

```
1 ''' Tensor product (similar to a matrix multiplication)
2 (a, b, c, d) . (d,) -> (a, b, c)
3 (a, b, c, d) . (d, e) -> (a, b, c, e)
4 '''
5
6 tensor2D = np.array([[1,-1],
7                      [0,2]])
8 tensor1D = np.array([1, 0])
9
10 # Tensor product
11 print(np.dot(tensor2D,tensor1D))
12
```

[1 0]



Tensor Operations

```
1 tensor2D = np.array([[1,-1],  
2                      [0,2]])  
3 tensor1D = np.array([1, 0]).  
4  
5 # Element-wise operation  
6 print(tensor2D * tensor1D)
```

```
[[1 0]  
[0 0]]
```

```
1 ''' Element-wise operation  
2 tensor2D = |1,-1|    ->  relu(tensor2D)   -> |1,0|  
3          |0, 2|  
4 ...  
5  
6 tensor2D = np.array([[1,-1],[0,2]])  
7 # before relu()  
8 print(tensor2D)  
9  
10 # after relu()  
11 print(np.maximum(tensor2D,0))
```

```
[[ 1 -1]  
 [ 0  2]]  
[[1 0]  
[0 2]]
```

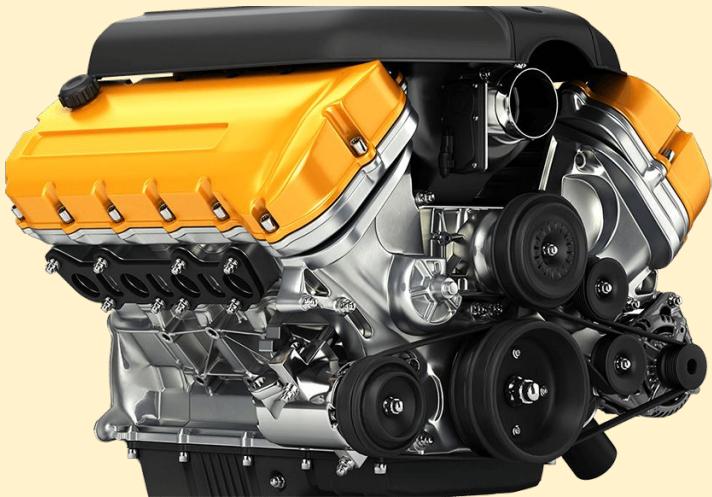
Tensor Operations

```
1 ''' Broadcast operation
2 What happens with addition when the shapes
3 of the two tensors being added differ?
4 '''
5
6 tensor2D = np.array([[1,-1],[0,2]])
7 tensor1D = np.array([1, 0])
8
9 tensor2D + tensor1D
10
```

```
array([[ 2, -1],
       [ 1,  2]])
```



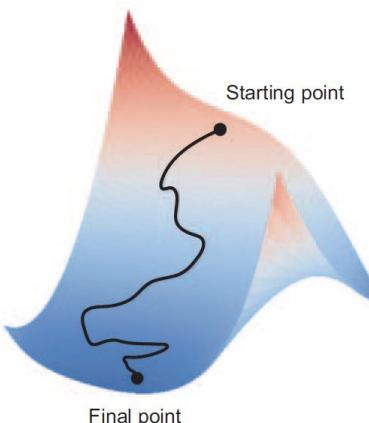
the engine of
neural networks:
gradient-based
optimization



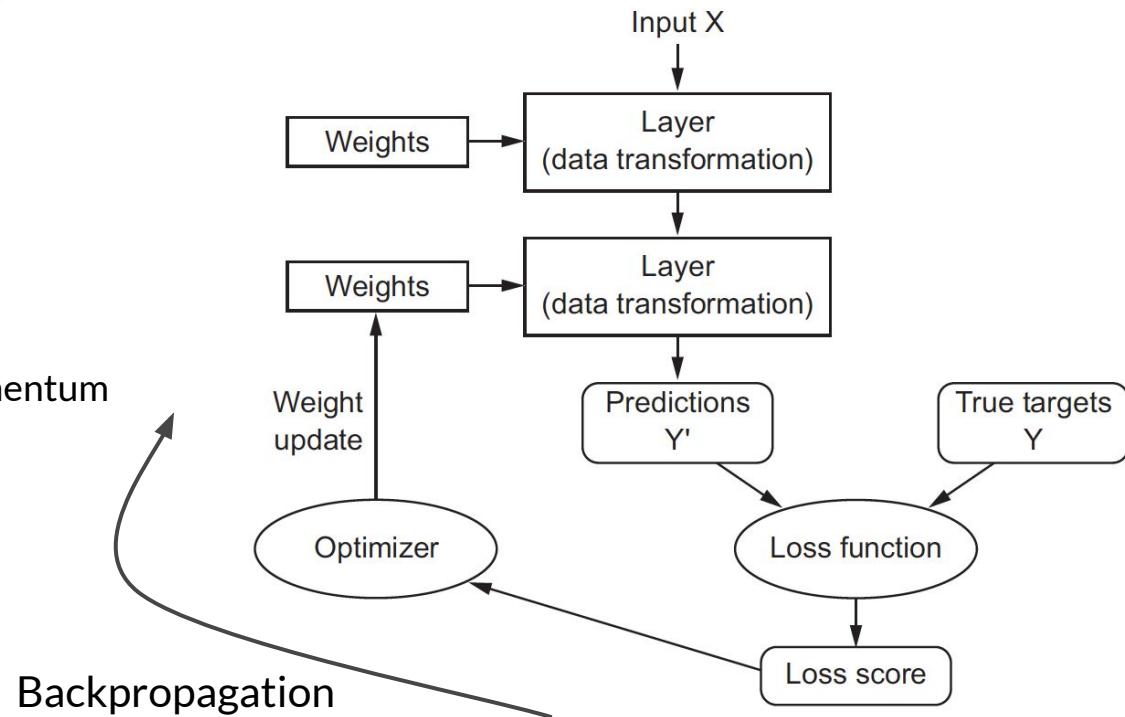
Gradient based optimization

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=[ 'accuracy'])

network.fit(train_images,
            train_labels, epochs=5,
            batch_size=128)
```



- SGD with momentum
- Adagrad
- RMSProp
- Others



#3 Getting started with neural networks

Getting started with Neural Network

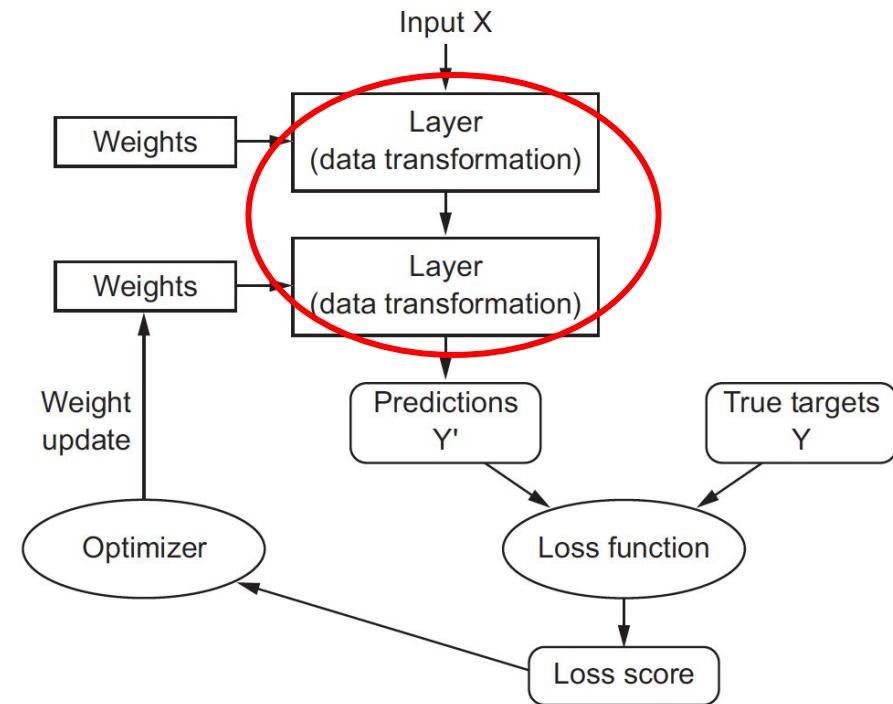
This section covers

- Core components of neural networks
- An introduction to Keras
- Using neural networks to solve basic classification and regression problems
 - Classifying movie reviews as positive or negative (**binary classification**)
 - Classifying news wires by topic (**multiclass classification**)
 - Estimating the price of a house, given real-estate data (**regression**)

Anatomy of a neural network

Different layers are appropriate for different tensor formats and different types of data processing.

- **fully connected or dense layers**
 - simple vector data, stored in 2D tensors of shape
- **recurrent layers such as an LSTM layer**
 - Sequence data, stored in 3D tensors of shape
- **2D convolution layers (Conv2D)**
 - image data, stored in 4D tensors



Loss functions and optimizers

Choosing the right objective function for the right problem is extremely important.

- **binary crossentropy**
 - two-class classification
- **categorical crossentropy**
 - many-class classification problem
- **mean squared error**
 - regression problem
- **connectionist temporal classification (CTC)**
 - sequence-learning problem

Case Study #1

Classifying movie reviews: a binary classification example

IMDB Movie reviews sentiment classification

Dataset of 25,000 movies reviews from IMDB, labeled by sentiment (positive/negative). Reviews have been preprocessed, and each review is encoded as a **sequence** of word indexes (integers). For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".

As a convention, "0" does not stand for a specific word, but instead is used to encode any unknown word.

IMDB Dataset

```
1 # loading the IMDB dataset
2
3 from keras.datasets import imdb
4
5 # num_words=10000 means you'll only keep the
6 # top 10,000 most frequently occurring words
7 #in the training data.
8 (train_data, train_labels),
9 (test_data, test_labels.) = imdb.load_data(num_words=10000)
```

IMDB Dataset (train & test)

```
1 # The variables train_data and test_data are lists of reviews;  
2 # each review is a list of word indices (encoding a sequence of words)  
3 print(train_data[0][:20])  
4 print(len(train_data[0]))
```

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256, 5, 25]  
218
```

```
1 # train_labels and test_labels are lists of 0s and 1s,  
2 # where 0 stands for negative and 1 stands for positive  
3 train_labels[0]
```

1

Preparing the data

You can't feed lists of integers into a neural network. You have to turn your lists into tensors

```
1 # Encoding the integer sequences into a binary matrix (25000,10000)
2 import numpy as np
3
4 def vectorize_sequences(sequences, dimension=10000):
5     results = np.zeros((len(sequences), dimension))
6     for i, sequence in enumerate(sequences):
7         results[i, sequence] = 1.
8     return results
9
10 x_train = vectorize_sequences(train_data)
11 x_test = vectorize_sequences(test_data)
```

Building your network

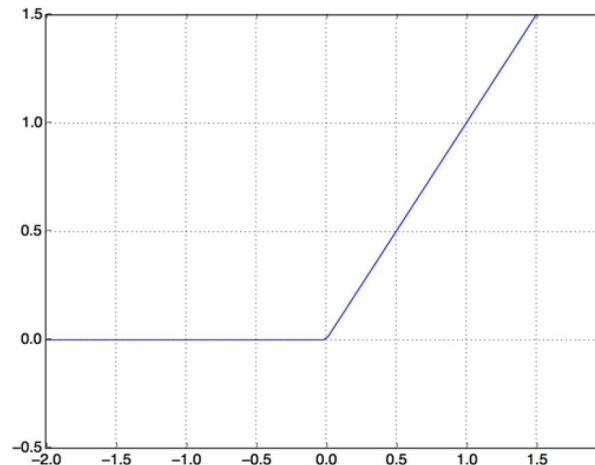
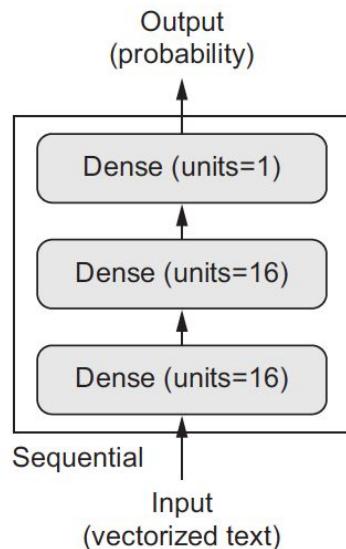


Figure 3.4 The rectified linear unit function

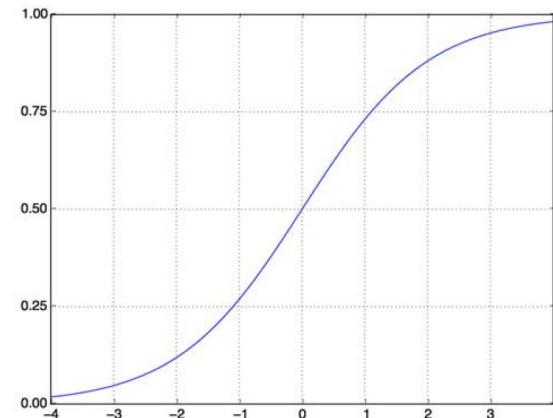
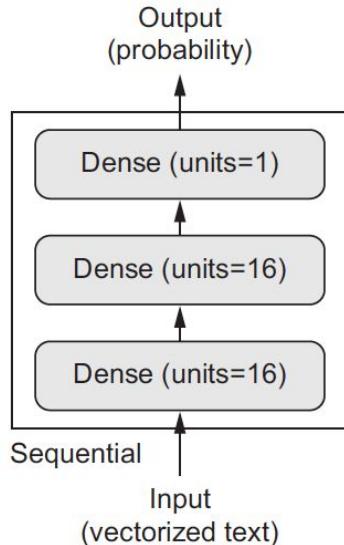


Figure 3.5 The sigmoid function

Building your network



```
1 from keras import models
2 from keras import layers
3
4 # the model definition
5 model = models.Sequential()
6 model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
7 model.add(layers.Dense(16, activation='relu'))
8 model.add(layers.Dense(1, activation='sigmoid'))
9
10 # compile the model
11 model.compile(optimizer='rmsprop',
12                 loss='binary_crossentropy',
13                 metrics=['accuracy'])
```

Validating your approach

```
1 # Setting aside a validation set
2 x_val = x_train[:10000]
3 partial_x_train = x_train[10000:]
4
5 y_val = y_train[:10000]
6 partial_y_train = y_train[10000:]
```

```
1 # Training your model
2 history = model.fit(partial_x_train,
3                      partial_y_train,
4                      epochs=20,
5                      batch_size=512,
6                      validation_data=(x_val, y_val))
```



Validating your approach



Evaluate the model

```
1 # this approach achieves an accuracy of 85%.  
2 # With state-of-the-art approaches,  
3 # you should be able to get close to 95%  
4 results = model.evaluate(x_test, y_test)  
5 results
```

```
25000/25000 [=====] - 2s 98us/step  
[0.7686490219664573, 0.84988]
```

Using a trained network to generate predictions

```
1 | model.predict(x_test).
```

```
array([[0.00540271],  
       [0.9999999 ],  
       [0.9111773 ],  
       ...,  
       [0.00160069],  
       [0.00567658],  
       [0.4761478 ]], dtype=float32)
```

```
1 | model.predict_classes(x_test)
```

```
array([[0],  
       [1],  
       [1],  
       ...,  
       [0],  
       [0],  
       [0]]], dtype=int32)
```

Further experiments

1. You used **two hidden layers**. Try using one or three hidden layers, and see how doing so affects validation and test accuracy.
2. Try using layers with **more hidden units** or fewer hidden units: 32 units, 64 units, and so on.
3. Try using the **mse loss** function instead of **binary_crossentropy**.
4. Try using the **tanh activation** (an activation that was popular in the early days of neural networks) instead of **relu**.

```
1 # the model definition
2 # layer_1,layer_2
3
4 for hidden in hidden_units:
5     for activations in activations_funct:
6         for losses in loss_funct:
7             model = models.Sequential()
8             model.add(layers.Dense(hidden, activation=activations,
9                               input_shape=(10000,)))
10            model.add(layers.Dense(1, activation='sigmoid'))
11
12            # compile the model
13            model.compile(optimizer='rmsprop', loss=losses, metrics=[ 'accuracy' ])
14
15            # Training your model
16            history = model.fit(partial_x_train,
17                                partial_y_train,
18                                epochs=20,
19                                batch_size=512,
20                                validation_data=(x_val, y_val))
21            training.append(history)
22            results.append(model.evaluate(x_test, y_test))
```

```
1 # parameters to be evaluated
2
3 hidden_units = [16,32,64]
4 activations_funct = ['relu','tanh']
5 loss_funct = ['binary_crossentropy',
6                 'mean_squared_error']
7 training = []
8 results = []
```

Model	Hidden Unit	Loss Function	Evaluation (accuracy)
relu,sigmoid(1)	16	binary_cross	0.8574
relu,sigmoid(1)	16	mse	0.8551
tanh,sigmoid(1)	16	binary_cross	0.8513
tanh,sigmoid(1)	16	mse	0.8515
relu,sigmoid(1)	32	binary_cross	0.8500
relu,sigmoid(1)	32	mse	0.8538
tanh,sigmoid(1)	32	binary_cross	0.8460
tanh,sigmoid(1)	32	mse	0.8469
relu,sigmoid(1)	64	binary_cross	0.8470
relu,sigmoid(1)	64	mse	0.8502
tanh,sigmoid(1)	64	binary_cross	0.8464
tanh,sigmoid(1)	64	mse	0.7906

Section 3.4



Classifying newswires: a multiclass classif. problem

- You'll build a network to **classify Reuters newswires** into **46 mutually exclusive topics**
- the problem is more specifically an instance of **single-label, multiclass classification**.
- If each data point could belong to multiple categories (in this case, topics), you'd be facing a **multilabel, multiclass classification problem**.

```
1 # Loading the Reuters dataset
2
3 from keras.datasets import reuters
4
5 (train_data, train_labels),
6 (test_data, test_labels) = reuters.load_data(num_words=10000)
```

Classifying newswires: train & test sets

```
1 # 8982 train data samples
2 print(len(train_data))
3
4 # 2246 test data samples
5 print(len(test_data))
```

8982

2246

```
1 # print a train data sample
2 print(train_data[3][:10])
```

[1, 4, 686, 867, 558, 4, 37, 38, 309, 2276]



Classifying newswires: data preparation

```
1 # Encoding the data
2
3 import numpy as np
4
5 def vectorize_sequences(sequences, dimension=10000):
6     results = np.zeros((len(sequences), dimension))
7     for i, sequence in enumerate(sequences):
8         results[i, sequence] = 1.
9     return results
10
11 x_train = vectorize_sequences(train_data)
12 x_test = vectorize_sequences(test_data)
```

```
1 from keras.utils.np_utils import to_categorical
2
3 one_hot_train_labels = to_categorical(train_labels)
4 one_hot_test_labels = to_categorical(test_labels)
```



Classifying newswires: network architecture

```
1 # Model definition
2
3 from keras import models
4 from keras import layers
5
6 model = models.Sequential()
7 model.add(layers.Dense(64, activation='relu',
8                      input_shape=(10000,)))
9 model.add(layers.Dense(64, activation='relu'))
10 model.add(layers.Dense(46, activation='softmax'))
```

```
1 # Compiling the model
2 model.compile(optimizer='rmsprop',
3                 loss='categorical_crossentropy',
4                 metrics=['accuracy'])
```

- You end the network with a Dense layer of size 46.
- The last layer uses a softmax activation.
- The 46 scores will sum to 1.

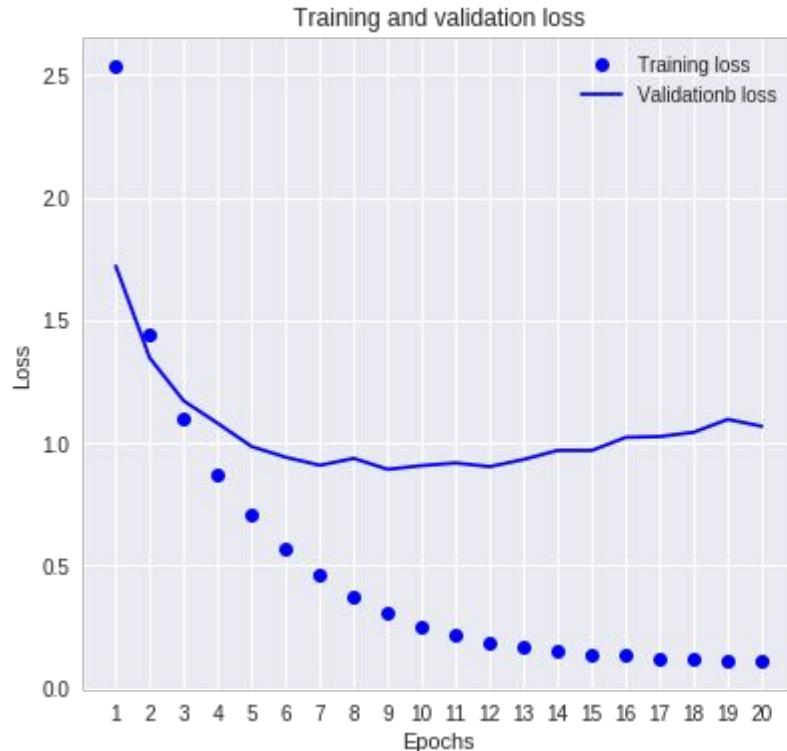
Classifying newswires: validation & training

```
1 # Setting aside a validation set
2
3 x_val = x_train[:1000]
4 partial_x_train = x_train[1000:]
5
6 y_val = one_hot_train_labels[:1000]
7 partial_y_train = one_hot_train_labels[1000:]
```

```
1 # Training the model
2
3 history = model.fit(partial_x_train,
4                      partial_y_train,
5                      epochs=20,
6                      batch_size=512,
7                      validation_data=(x_val, y_val))
```



Classifying newswires: validation & training



The importance of having sufficiently large intermediate layers

```
1 model = models.Sequential()
2 model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
3 model.add(layers.Dense(4, activation='relu'))
4 model.add(layers.Dense(46, activation='softmax'))
5
6 model.compile(optimizer='rmsprop',
7                 loss='categorical_crossentropy',
8                 metrics=['accuracy'])
9
10 model.fit(partial_x_train,
11            partial_y_train,
12            epochs=20,
13            batch_size=512,
14            validation_data=(x_val, y_val))
```

```
1 # evaluating the model
2 model.evaluate(x_test,one_hot_test_labels)
2246/2246 [=====] - 0s 109us/step
[1.469466186482675, 0.6821015138553915]
```

Epoch 19/20

7982/7982 [=====] - 1s 81us/step - loss: 0.6534 - acc: 0.8349 - val_loss: 1.3740 - val_acc: 0.7090

Epoch 20/20

7982/7982 [=====] - 1s 81us/step - loss: 0.6294 - acc: 0.8399 - val_loss: 1.4038 - val_acc: 0.7090



Section 3.5



Predicting house prices: a regression examples

```
1 # Loading the Boston housing dataset
2
3 from keras.datasets import boston_housing
4
5 (train_data, train_targets),
6 (test_data, test_targets) = boston_housing.load_data()
7
8 # Training sample size
9 print(train_data.shape)
10
11 # Test sample size
12 print(test_data.shape)
```

(404, 13)

(102, 13)

```
1 # Normalizing the data
2
3 from sklearn import preprocessing
4
5 # create a scaler to fit train_data
6 scaler = preprocessing.StandardScaler().fit(train_data)
7
8 # feature-wise normalization over train_data
9 train_data_scaled = scaler.transform(train_data)
10
11 # Scaled train data has zero mean and unit variance
12 print(train_data_scaled.mean(axis=0))
13 print(train_data_scaled.std(axis=0))
14
15 # Note that the quantities used for normalizing the
16 # test data are computed using the training data.
17 # You should never use in your workflow any quantity
18 # computed on the test data,
19 # even for something as simple as data normalization.
20 test_data_scaled = scaler.transform(test_data)
```

Preparing the data

```
[-1.01541438e-16  1.09923072e-17  1.74337992e-15 -1.26686340e-16
 -5.25377321e-15  6.41414864e-15  2.98441140e-16  4.94653823e-16
  1.12671149e-17 -1.98136337e-16  2.36686358e-14  5.95679996e-15
  6.13920356e-16]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```



```
1 # Model definition
2
3 from keras import models
4 from keras import layers
5
6 def build_model():
7     model = models.Sequential()
8     model.add(layers.Dense(64, activation='relu',
9                           input_shape=(train_data_scaled.shape[1],)))
10    model.add(layers.Dense(64, activation='relu'))
11
12    # The network ends with a single unit and no activation (it will be a linear layer).
13    # This is a typical setup for scalar regression (a regression where you're trying
14    # to predict a single continuous value).
15    model.add(layers.Dense(1))
16
17    # compile the network with the mse loss function—mean squared error,
18    # the square of the difference between the predictions and the targets.
19    # This is a widely used loss function for regression problems.
20    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])
21
22    # You're also monitoring a new metric during training: mean absolute error (MAE).
23    # It's the absolute value of the difference between the predictions and the targets.
24    return model
```

Building your network

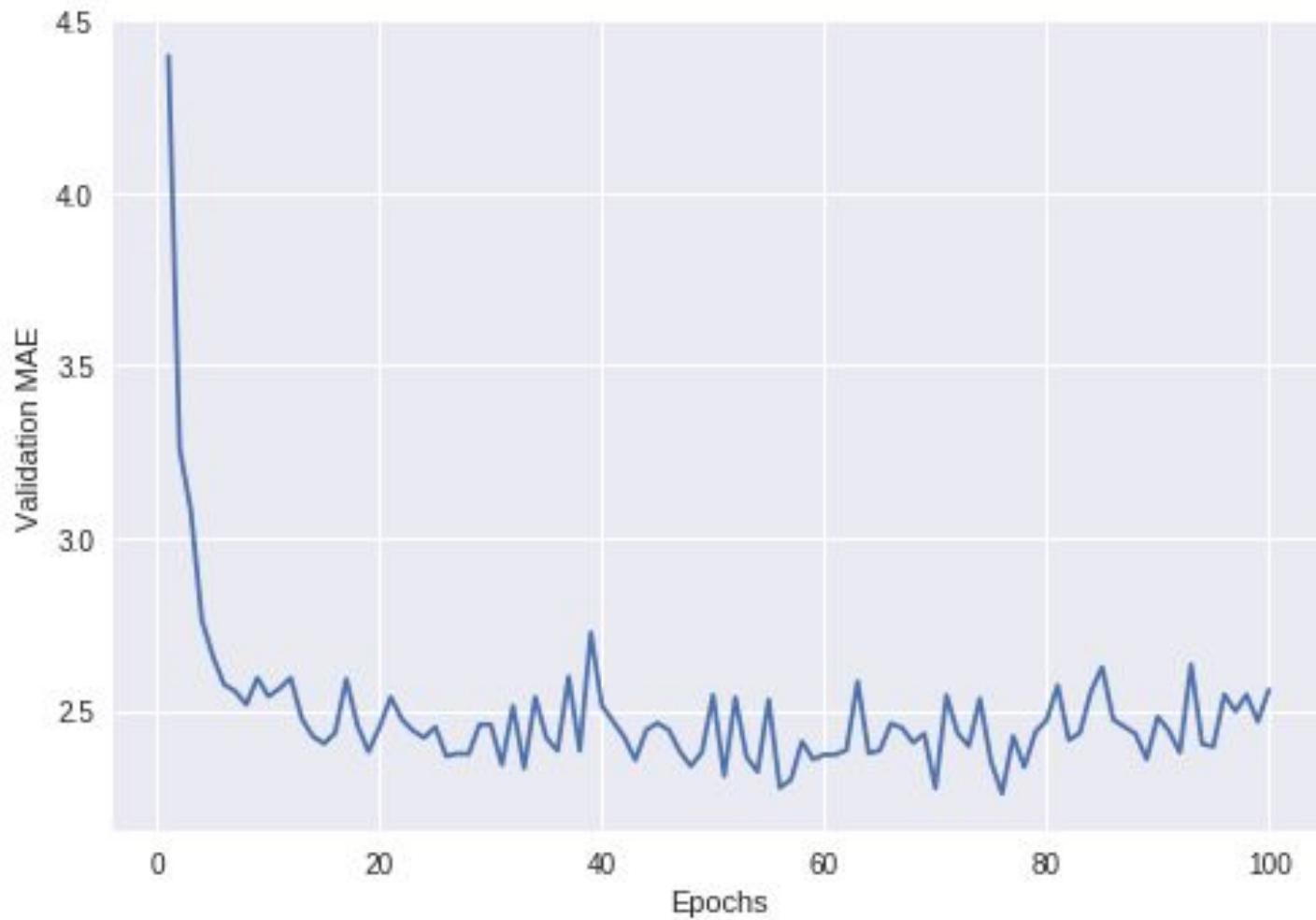
```
1 from sklearn.model_selection import KFold
2
3 # fix random seed for reproducibility
4 seed = 7
5 np.random.seed(seed)
6
7 # define k-fold cross validation test
8 kf = KFold(n_splits=4, random_state=seed)
9
10 all_mae_histories = []
11
12 for train, test in kf.split(train_data_scaled, train_targets):
13     # create model
14     model = build_model()
15     history = model.fit(train_data_scaled[train],
16                          train_targets[train],
17                          epochs=100,
18                          batch_size=1,
19                          verbose=1,
20                          validation_data=(train_data_scaled[test],
21                                           train_targets[test]))
22     mae_history = history.history['val_mean_absolute_error'].
23     all_mae_histories.append(mae_history)
```

```
1 # mae for each k-fold step
2 np.mean(all_mae_histories, axis=1)
```

```
array([2.07212125, 2.5140706 , 2.66233936, 2.70220025])
```

Validate using k-fold





Section 3.6

