EEC1509 - Machine Learning

Lesson #12 Deep Learning Fundamentals I

Ivanovitch Silva

December, 2018

Update repository

```
git clone https://github.com/ivanovitchm/EEC1509_MachineLearning.git
```

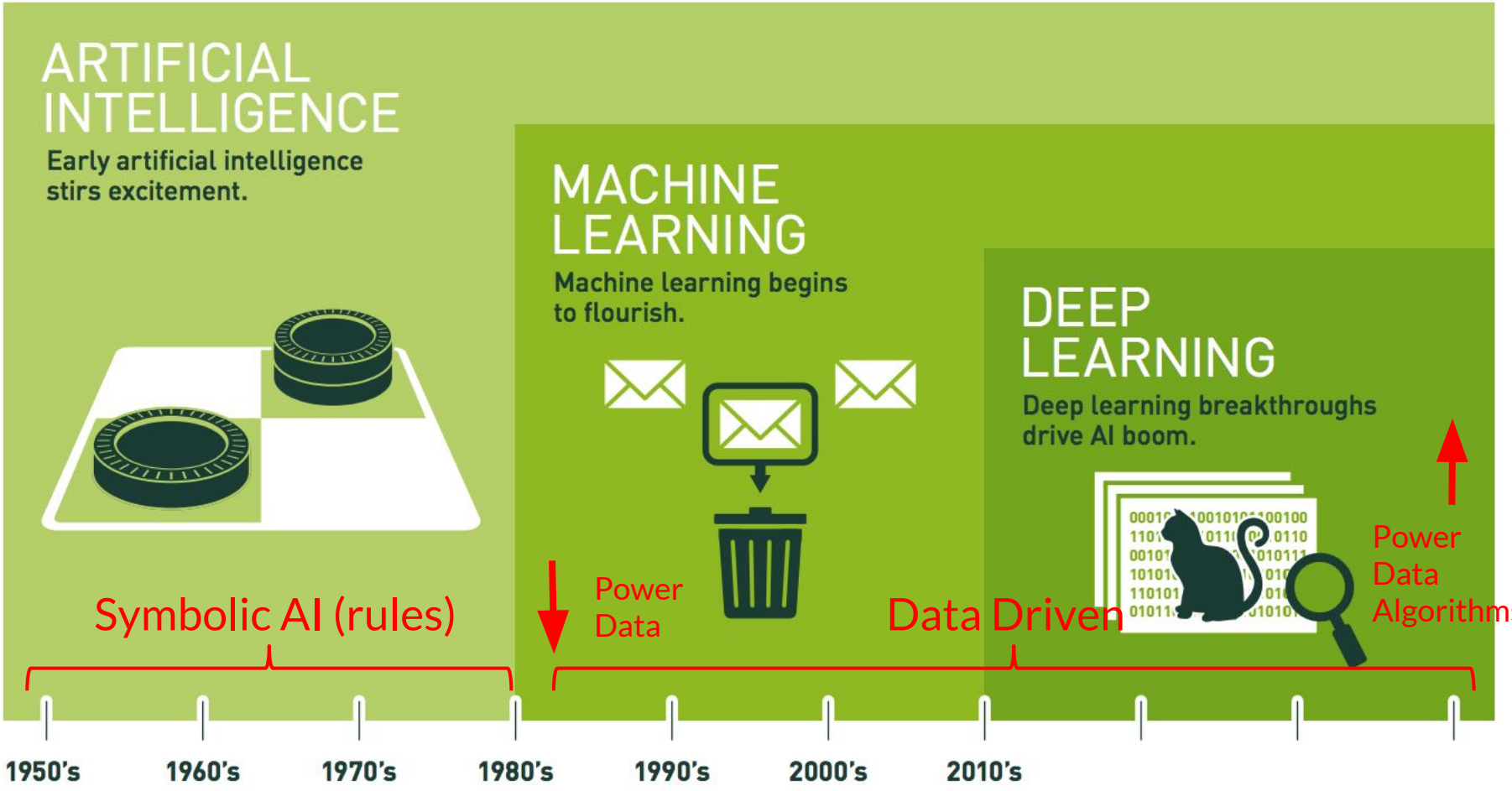
Ou ....

```
git pull
```

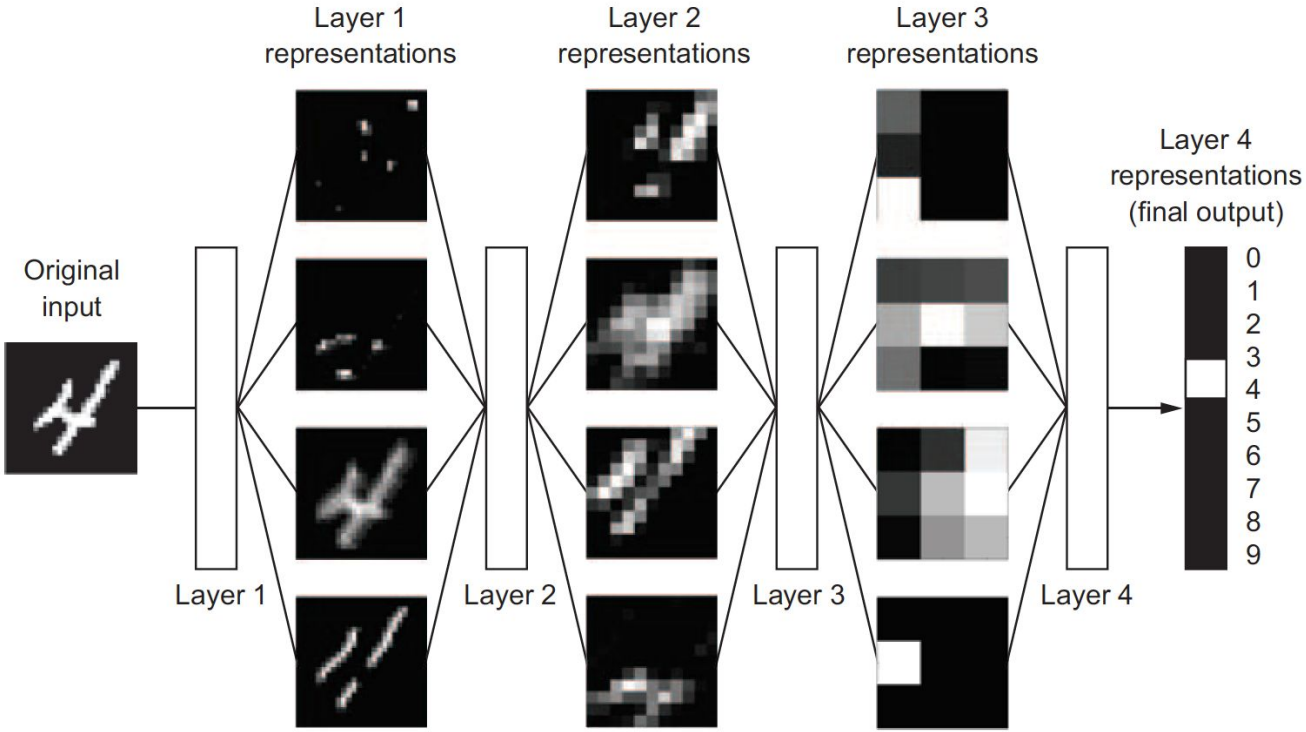
Agenda

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

#1 Introduction


ARTIFICIAL INTELLIGENCE
Early artificial intelligence stirs excitement.
MACHINE LEARNING
Machine learning begins to flourish.
DEEP LEARNING
Deep learning breakthroughs drive AI boom.
Symbolic AI (rules)
Power
Data
Data Driven
Power
Data
Algorithms
1950's
1960's
1970's
1980's
1990's
2000's
2010's

What is Deep Learning?

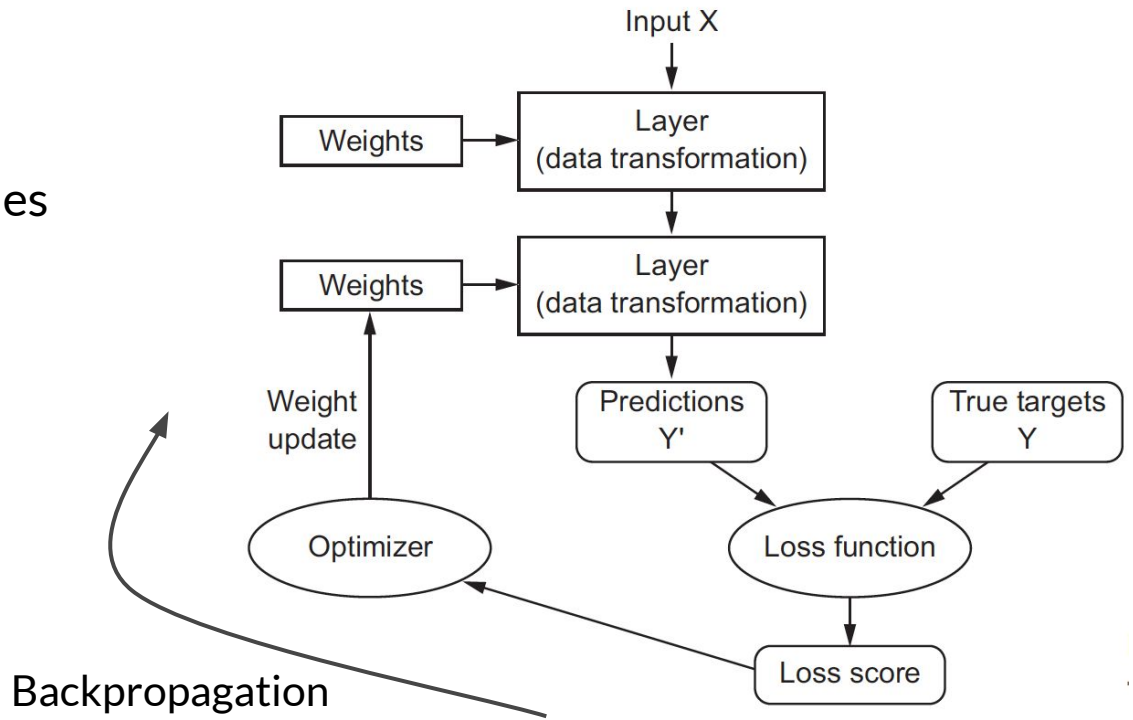


"Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [. . .] 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"

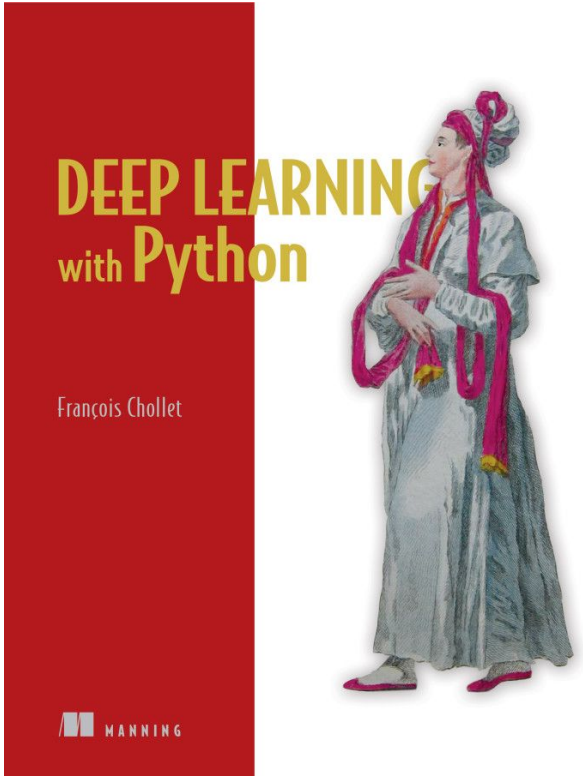
Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, Nature 2015

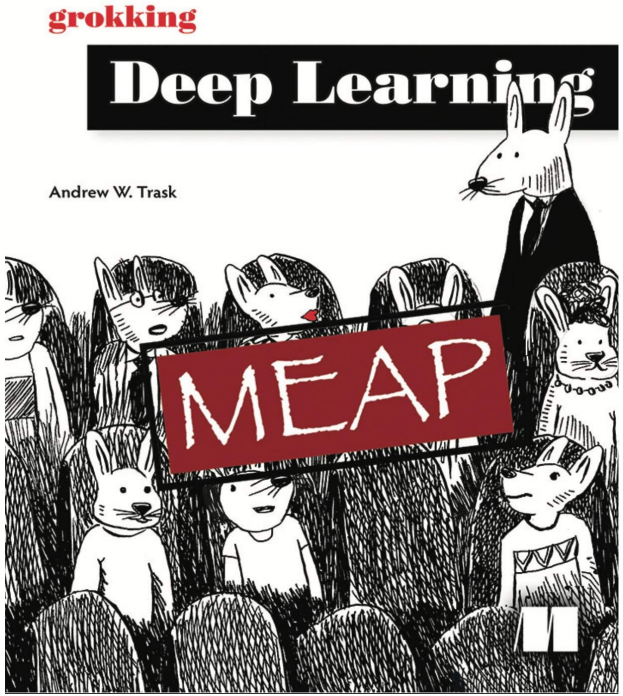
Understanding how DL works

Finding the right values of weights which minimize the error



References


DEEP LEARNING
with Python
François Chollet
MANNING


grokking
Deep Learning
Andrew W. Trask
MEAP

#2 Mathematical building blocks of neural networks

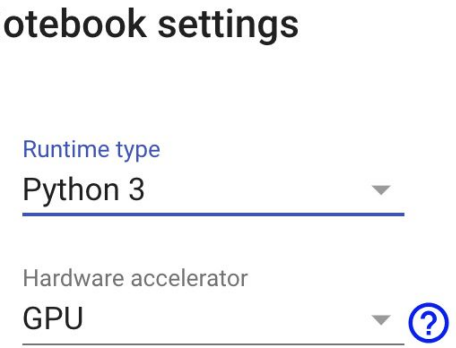
Mathematical building blocks of neural networks

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.

```
import keras
from keras import backend as K

print('Using Keras version:', keras.__version__,
      '\nbackend:', K.backend())
```

```
Using Keras version: 2.2.4
backend: tensorflow
```

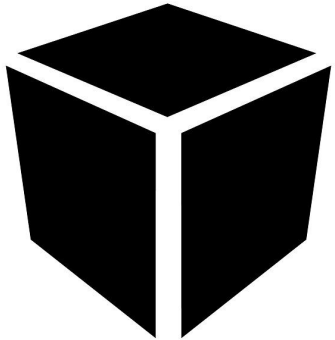



```
Gen RAM Free: 12.5 GB  | Proc size: 1.1 GB
GPU RAM Free: 11271MB  | Used: 170MB | Util   1% |  Total 11441MB
```

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

```
# Loading the MNIST dataset in Keras
from keras.datasets import mnist

# The images are encoded as Numpy arrays,
# and the labels are an array of digits, ranging
# from 0 to 9. The images and labels have a one-to-one correspondence.

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

```
# first ten train labels
train_labels[:10]
```

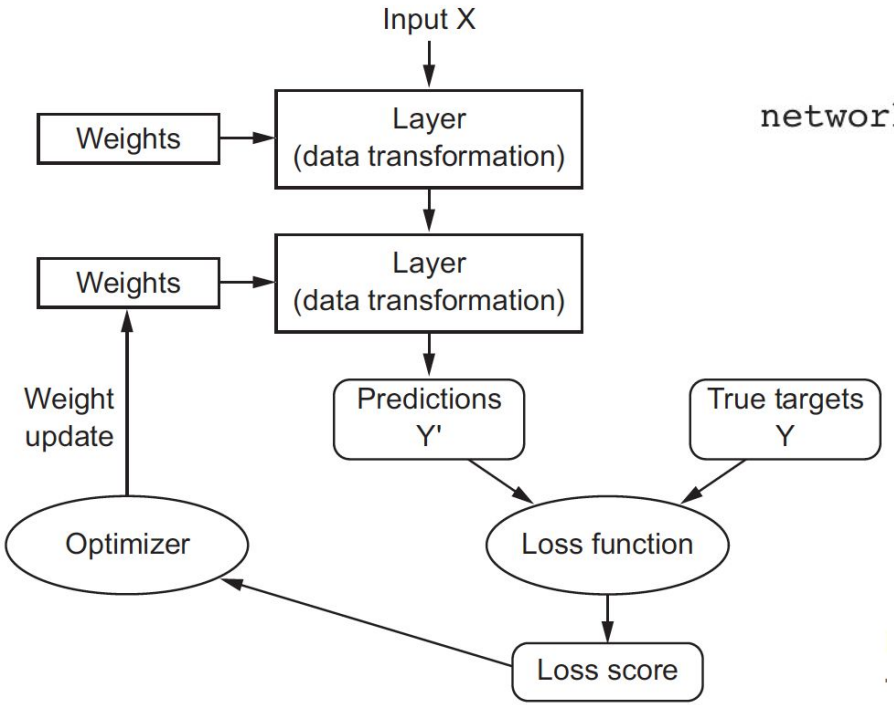
```
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4], dtype=uint8)
```

```
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()

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

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

The compilation step




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

Preparing data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype('float32') / 255
```

```
from keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

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

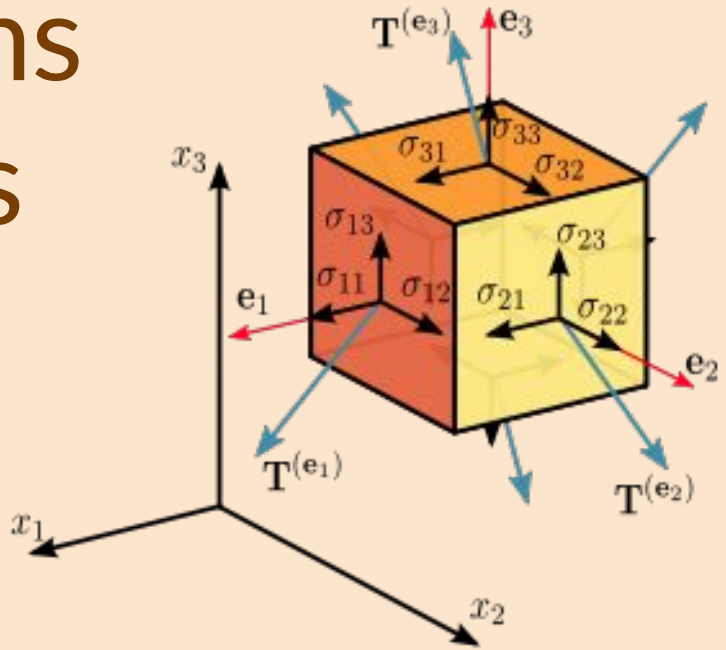
```
Epoch 1/5
60000/60000 [==============================] - 2s 39us/step - loss: 0.2565 - acc: 0.9258
Epoch 2/5
60000/60000 [==============================] - 2s 41us/step - loss: 0.1035 - acc: 0.9697
Epoch 3/5
60000/60000 [==============================] - 2s 41us/step - loss: 0.0679 - acc: 0.9799
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>
```

Evaluate the network

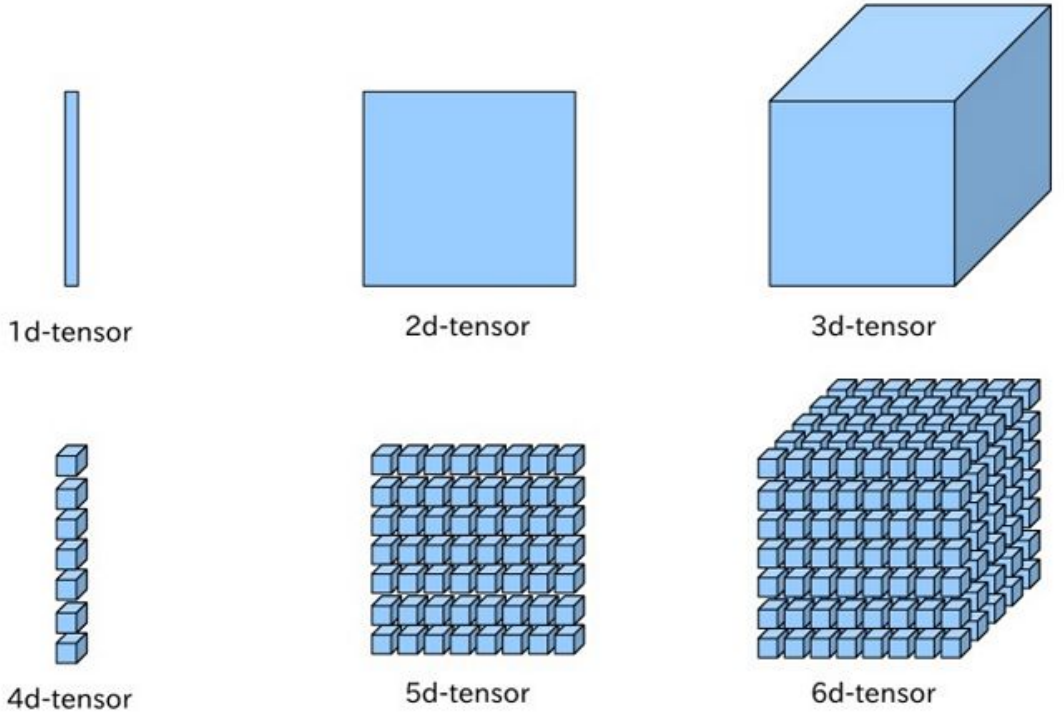
```
1 test_loss, test_acc = network.evaluate(test_images, test_labels)
2 print('test_acc:', test_acc)
```

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

data representations for neural networks



Multidimensional numpy arrays - TENSOR



Tensors - Dimensions

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

0

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

1

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

2

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

3

Real-world examples of data tensors

Vector data - the most common case

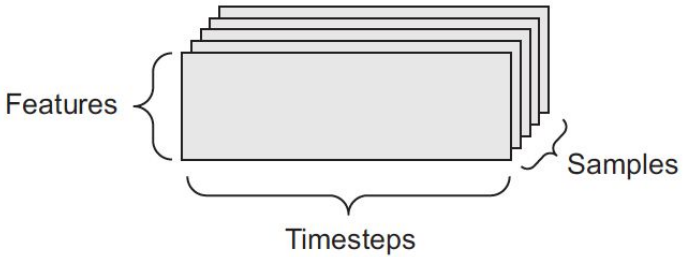
An actuarial dataset of people, where we consider each person's

- 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

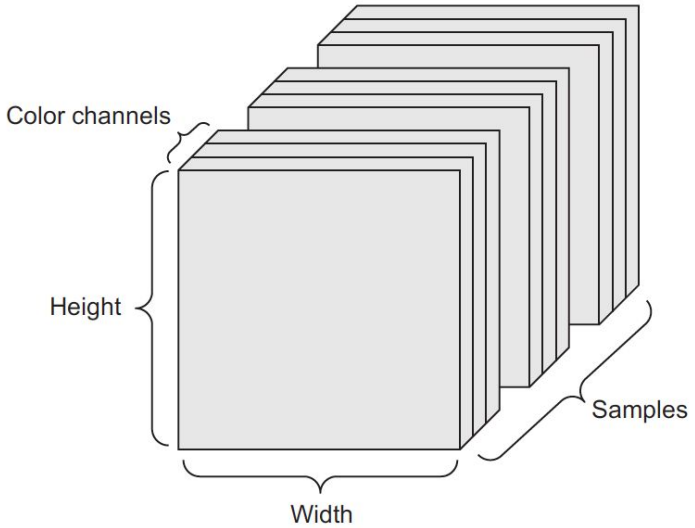
- Each tweet as a sequence of 280 characters
- An alphabet of 128 unique characters. In this setting, each character can
- Then each tweet can be encoded as a 2D tensor of shape (280, 128)

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

Real-world examples of data tensors

Image data

- 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)



Real-world examples of data tensors

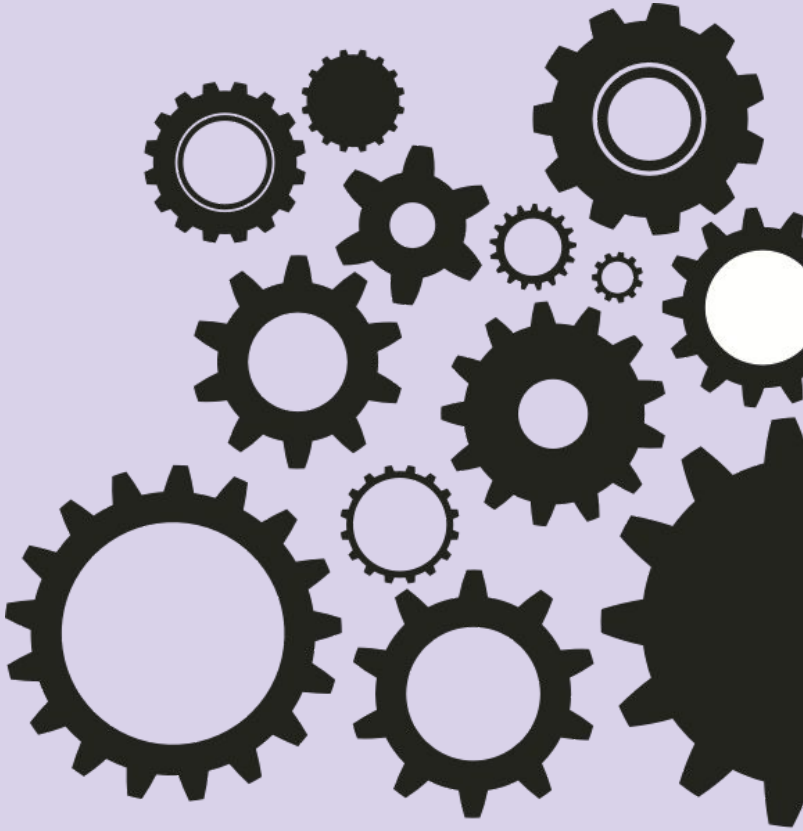
Video data

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

- 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. relu(x) is max(x, 0).
    - Element-wise operation

Tensor Operations

```
''' Tensor product (similar to a matrix multiplication)
(a, b, c, d) . (d,) -> (a, b, c)
(a, b, c, d) . (d, e) -> (a, b, c, e)
'''

tensor2D = np.array([[1,-1],
                     [0,2]])
tensor1D = np.array([1, 0])

# Tensor product
print(np.dot(tensor2D,tensor1D))

```

```
[1 0]
```

Tensor Operations

```
tensor2D = np.array([[1,-1],
                     [0,2]])
tensor1D = np.array([1, 0])

# Element-wise operation
print(tensor2D * tensor1D)
```

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

```
''' Element-wise operation
tensor2D =  |1,-1|   ->   relu(tensor2D)   ->  |1,0|
            |0, 2|                             |0,2|
'''

tensor2D = np.array([[1,-1],[0,2]])
# before relu()
print(tensor2D)

# after relu()
print(np.maximum(tensor2D,0))
```

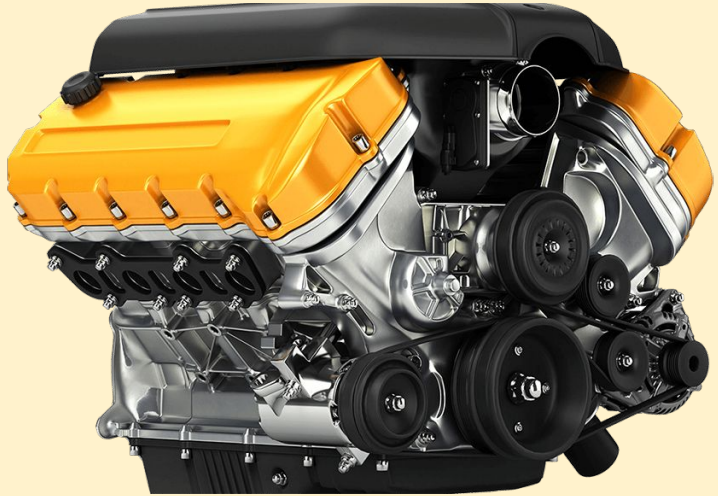
```
[[ 1 -1]
 [ 0  2]]
[[1 0]
 [0 2]]
```

Tensor Operations

```
''' Broadcast operation
What happens with addition when the shapes
of the two tensors being added differ?
'''

tensor2D = np.array([[1,-1],[0,2]])
tensor1D = np.array([1, 0])

tensor2D + tensor1D

```

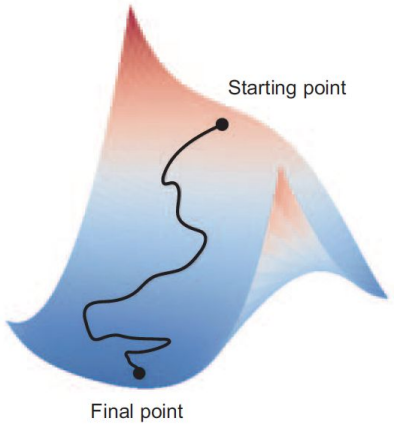
```
array([[ 2, -1],
       [ 1,  2]])
```

the engine of neural networks: gradient-based optimization

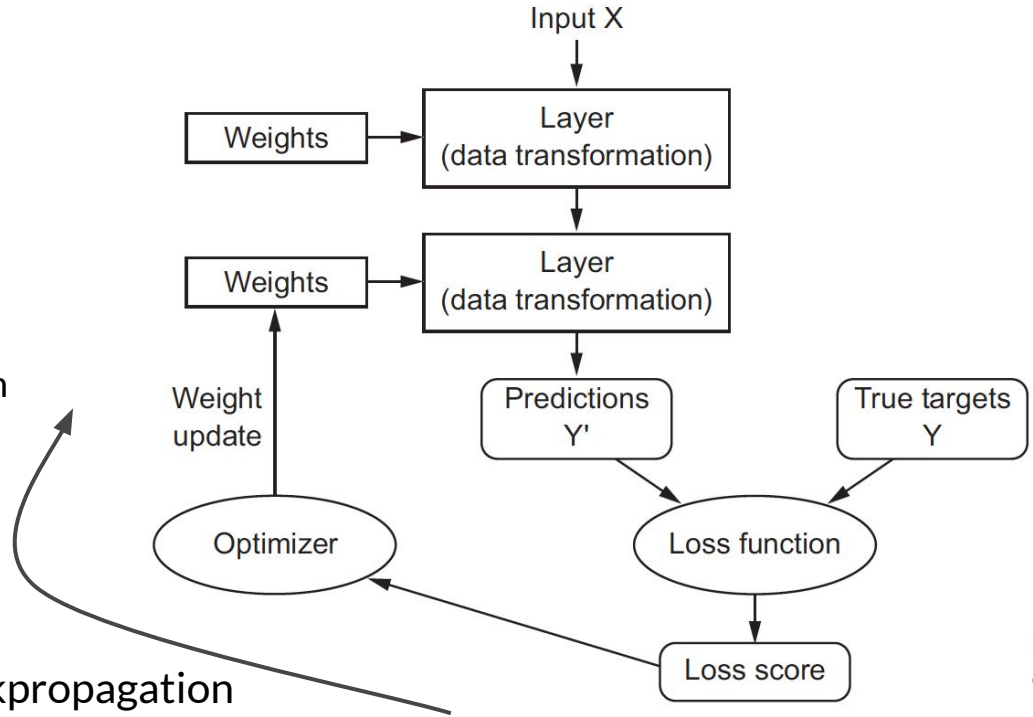


Gradient based optimization

```
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



Backpropagation

#3 Getting started with neural networks

Getting started with Neural Network

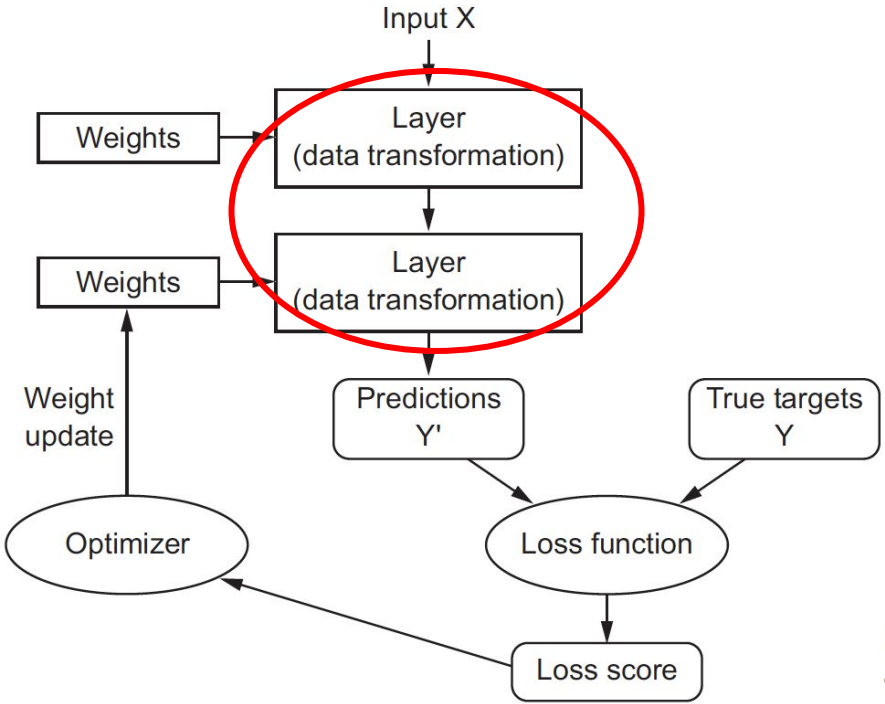
This section covers

- 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

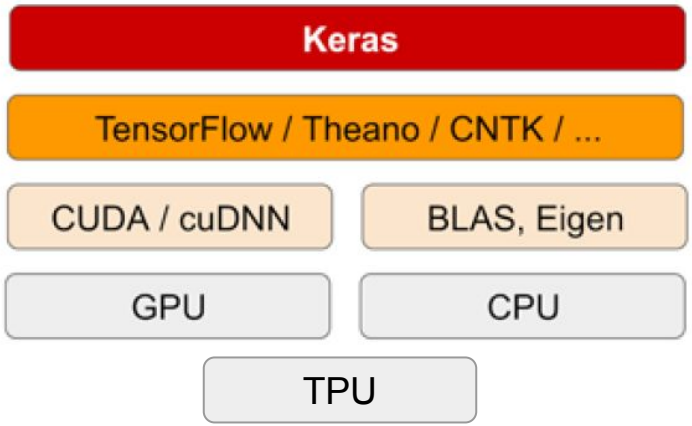


Loss functions and optimizers

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

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.**
- It's compatible with any version of Python from 2.7 to 3.6

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

```
# loading the IMDB dataset

from keras.datasets import imdb

# num_words=10000 means you'll only keep the
# top 10,000 most frequently occurring words
#in the training data.
(train_data, train_labels),
(test_data, test_labels) = imdb.load_data(num_words=10000)
```

IMDB Dataset (train & test)

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

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

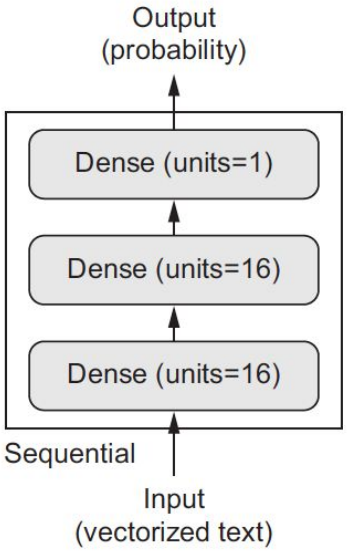
```
1
```

Preparing the data

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

```
# Encoding the integer sequences into a binary matrix (25000,10000)
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

Building your network



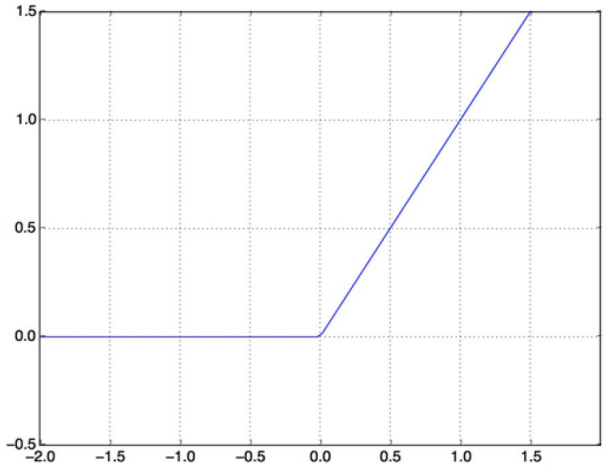


Figure 3.4 The rectified linear unit function

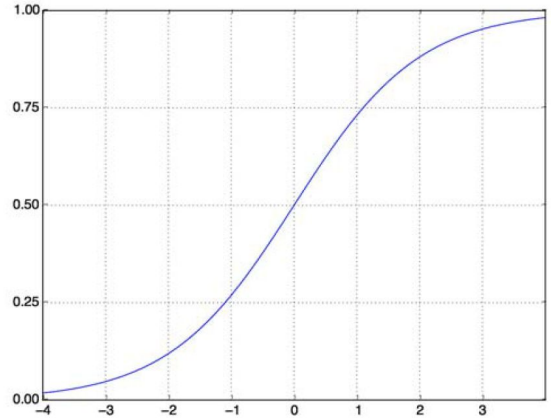
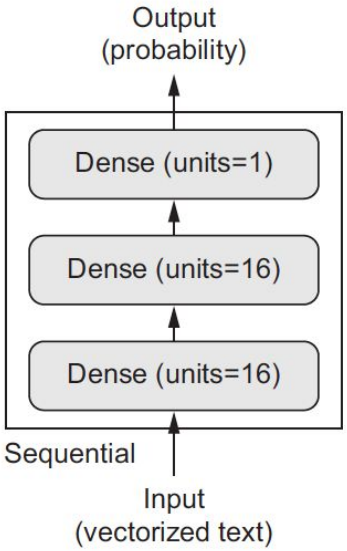


Figure 3.5 The sigmoid function

Building your network



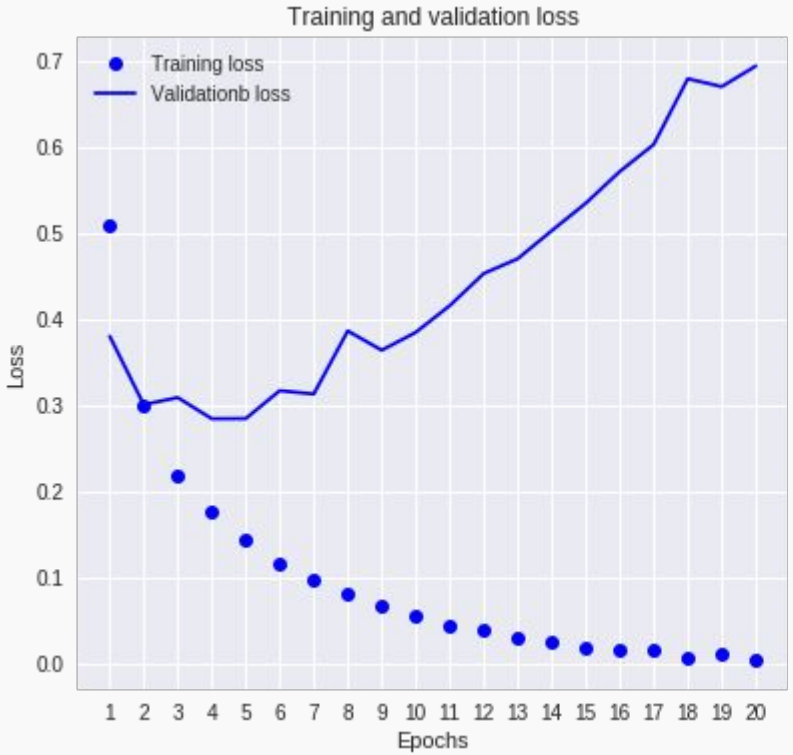

```
from keras import models
from keras import layers

# the model definition
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

# compile the model
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

Validating your approach

```
# Setting aside a validation set
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

```
# Training your model
history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Validating your approach



Evaluate the model

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

```
# parameters to be evaluated

hidden_units = [16,32,64]
activations_funct = ['relu','tanh']
loss_funct = ['binary_crossentropy',
              'mean_squared_error']
training = []
results = []
```

```
# the model definition
# layer_1,layer_2

for hidden in hidden_units:
    for activations in activations_funct:
        for losses in loss_funct:
            model = models.Sequential()
            model.add(layers.Dense(hidden, activation=activations,
                                   input_shape=(10000,)))
            model.add(layers.Dense(1, activation='sigmoid'))

            # compile the model
            model.compile(optimizer='rmsprop',loss=losses,metrics=['accuracy'])

            # Training your model
            history = model.fit(partial_x_train,
                                partial_y_train,
                                epochs=20,
                                batch_size=512,
                                validation_data=(x_val, y_val))
            training.append(history)
            results.append(model.evaluate(x_test, y_test))
```

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**.

```
# Loading the Reuters dataset

from keras.datasets import reuters

(train_data, train_labels),
(test_data, test_labels) = reuters.load_data(num_words=10000)
```

Classifying newswires: train & test sets

```
# 8982 train data samples
print(len(train_data))

# 2246 test data samples
print(len(test_data))
```

```
8982
2246
```

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

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

Classifying newswires: data preparation

```
# Encoding the data

import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results

x_train = vectorize_sequences(train_data)
x_test = vectorize_sequences(test_data)
```

```
from keras.utils.np_utils import to_categorical

one_hot_train_labels = to_categorical(train_labels)
one_hot_test_labels = to_categorical(test_labels)
```

Classifying newswires: network architecture

```
# Model definition

from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(64, activation='relu',
                       input_shape=(10000,)))
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))
```

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

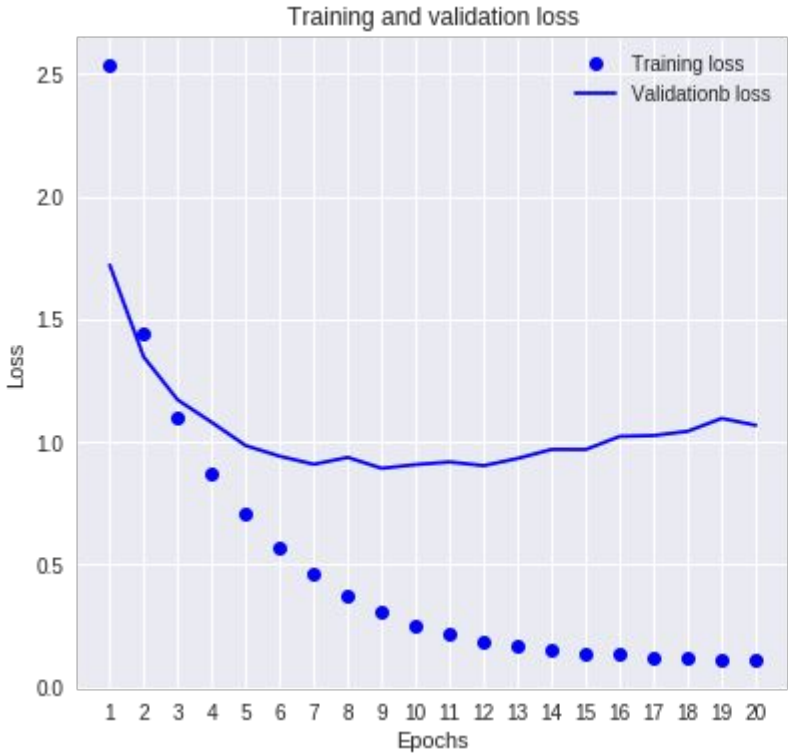
- You end the network with a Dense layer of size 46.
- The last layer uses a softmax activation.
- The 46 scores will sum to 1.

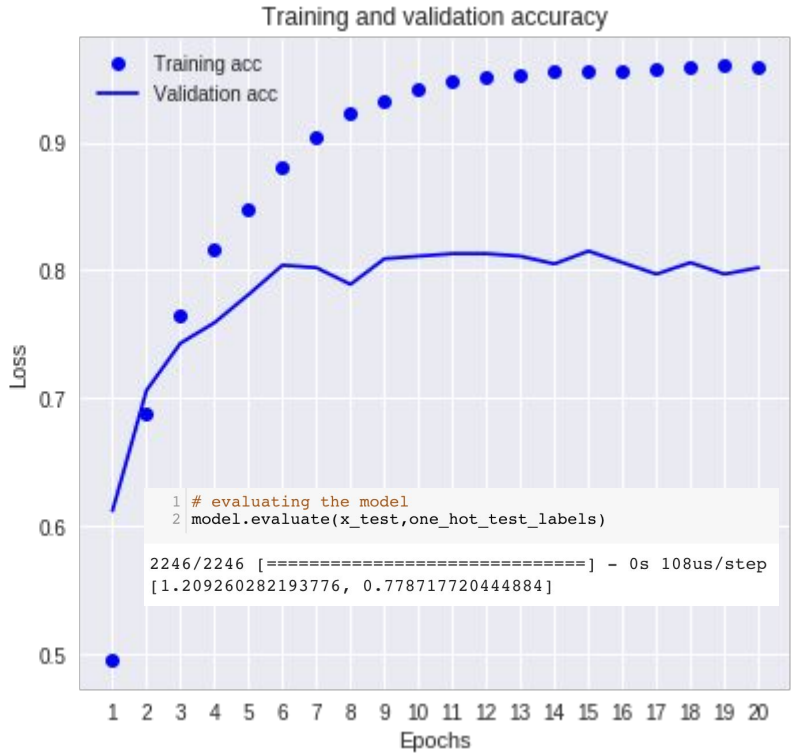
Classifying newswires: validation & training

```
# Setting aside a validation set

x_val = x_train[:1000]
partial_x_train = x_train[1000:]

y_val = one_hot_train_labels[:1000]
partial_y_train = one_hot_train_labels[1000:]
```

```
# Training the model

history = model.fit(partial_x_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(x_val, y_val))
```

Classifying newswires: validation & training





The importance of having sufficiently large intermediate layers

```
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(46, activation='softmax'))

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

model.fit(partial_x_train,
          partial_y_train,
          epochs=20,
          batch_size=512,
          validation_data=(x_val, y_val))
```

```
# evaluating the model
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

```
# Loading the Boston housing dataset

from keras.datasets import boston_housing

(train_data, train_targets),
(test_data, test_targets) = boston_housing.load_data()

# Training sample size
print(train_data.shape)

# Test sample size
print(test_data.shape)
```

```
(404, 13)
(102, 13)
```

Preparing the data

```
# Normalizing the data

from sklearn import preprocessing

# create a scaler to fit train_data
scaler = preprocessing.StandardScaler().fit(train_data)

# feature-wise normalization over train_data
train_data_scaled = scaler.transform(train_data)

# Scaled train data has zero mean and unit variance
print(train_data_scaled.mean(axis=0))
print(train_data_scaled.std(axis=0))

# Note that the quantities used for normalizing the
# test data are computed using the training data.
# You should never use in your workflow any quantity
# computed on the test data,
# even for something as simple as data normalization.
test_data_scaled = scaler.transform(test_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. 1. 1.]
```

Building your network

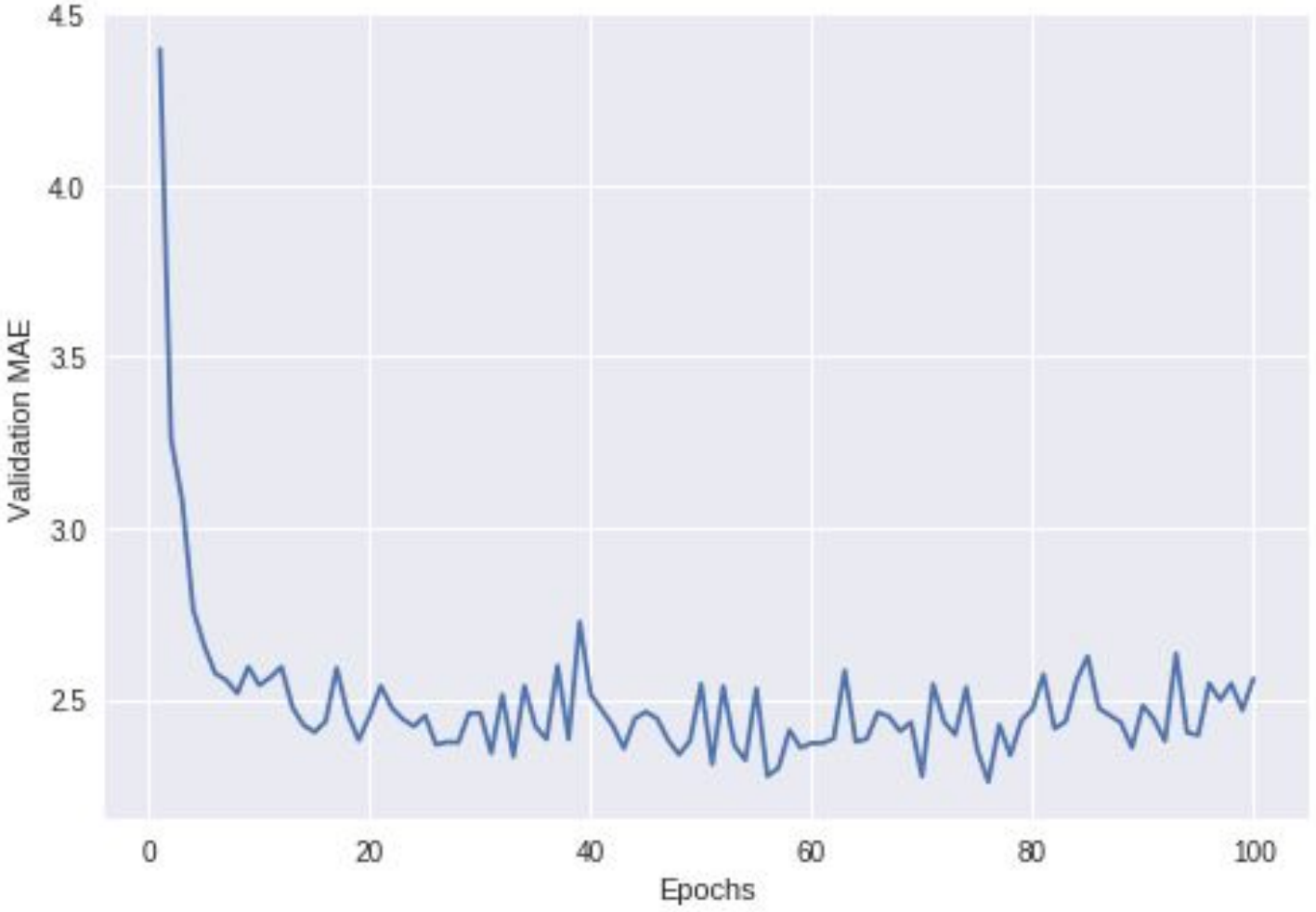
```
# Model definition

from keras import models
from keras import layers

def build_model():
    model = models.Sequential()
    model.add(layers.Dense(64, activation='relu',
                           input_shape=(train_data_scaled.shape[1],)))
    model.add(layers.Dense(64, activation='relu'))

    # The network ends with a single unit and no activation (it will be a linear layer).
    # This is a typical setup for scalar regression (a regression where you're trying
    # to predict a single continuous value).
    model.add(layers.Dense(1))

    # compile the network with the mse loss function—mean squared error,
    # the square of the difference between the predictions and the targets.
    # This is a widely used loss function for regression problems.
    model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

    # You're also monitoring a new metric during training: mean absolute error (MAE).
    # It's the absolute value of the difference between the predictions and the targets.
    return model
```

Validate using k-fold

```
from sklearn.model_selection import KFold

# fix random seed for reproducibility
seed = 7
np.random.seed(seed)

# define k-fold cross validation test
kf = KFold(n_splits=4, random_state=seed)

all_mae_histories = []

for train, test in kf.split(train_data_scaled, train_targets):
    # create model
    model = build_model()
    history = model.fit(train_data_scaled[train],
              train_targets[train],
              epochs=100,
              batch_size=1,
              verbose=1,
              validation_data=(train_data_scaled[test],
                               train_targets[test]))
    mae_history = history.history['val_mean_absolute_error']
    all_mae_histories.append(mae_history)
```

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

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


4.5
4.0
3.5
3.0
2.5
Validation MAE
0
20
40
60
80
100
Epochs

Section 3.6