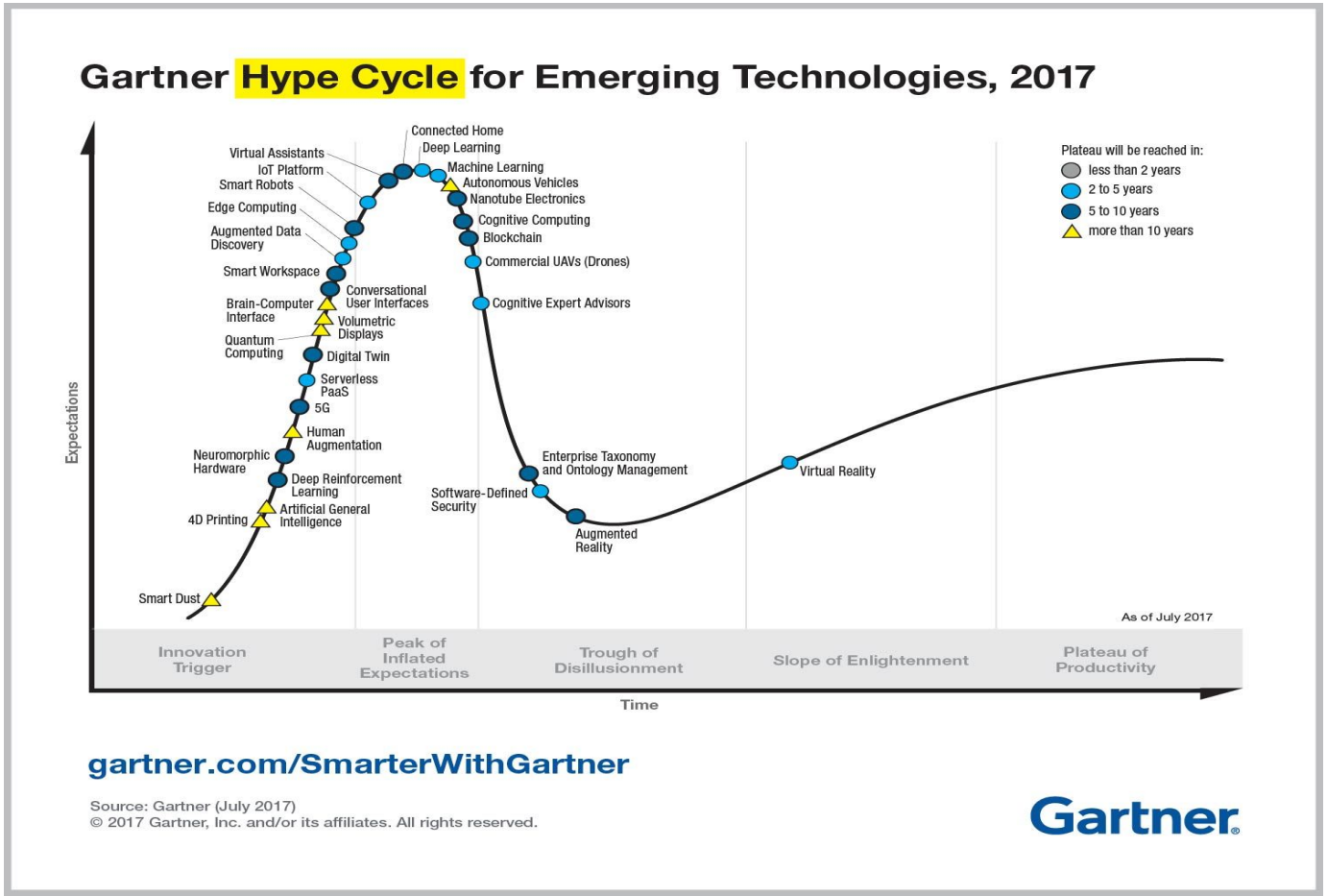
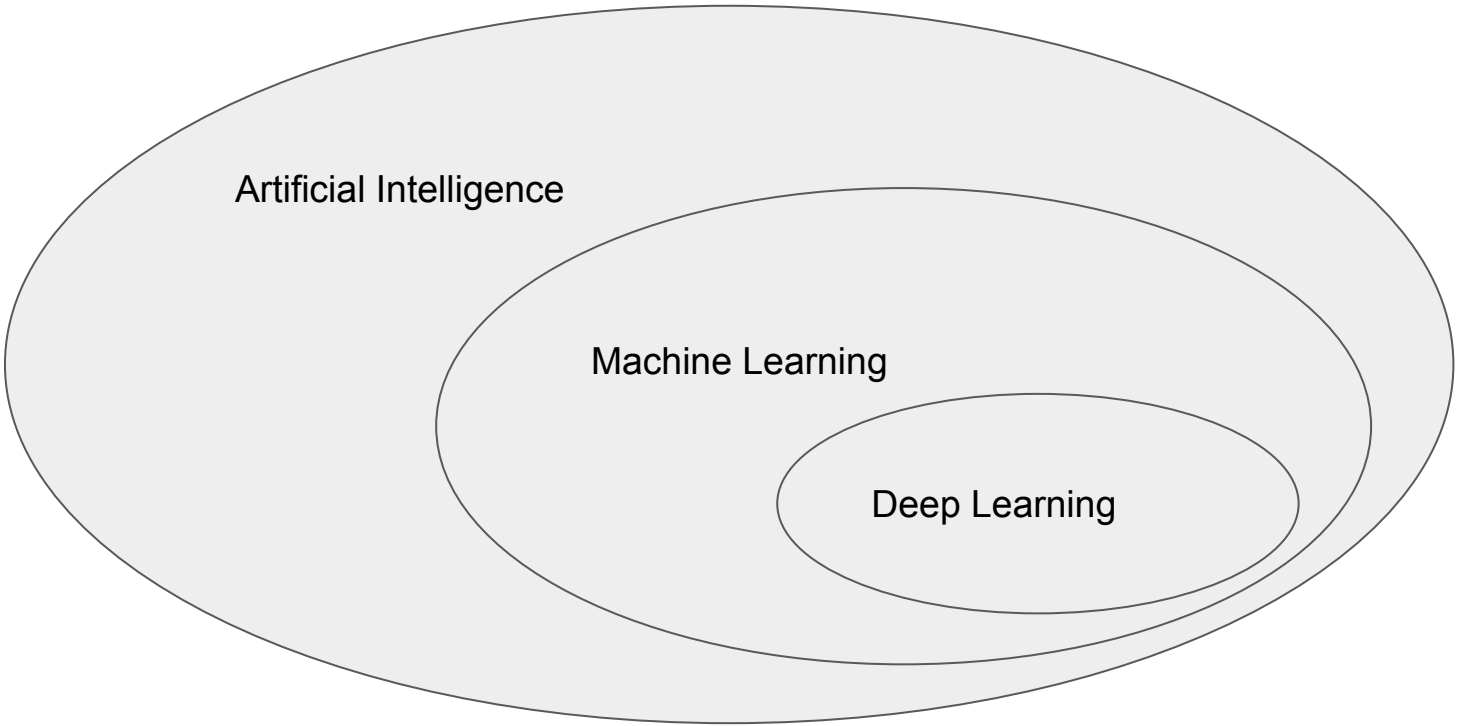
TX00DH43-3001
Introduction to Deep Learning

peter.hjort@metropolia.fi

Welcome to the Peak of Inflated Expectations!

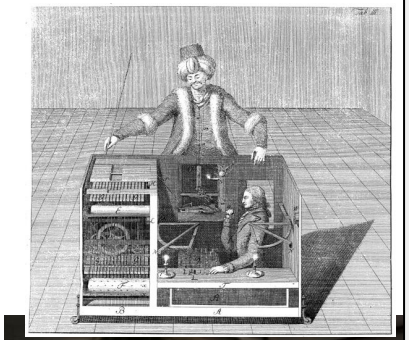


Deep learning in context

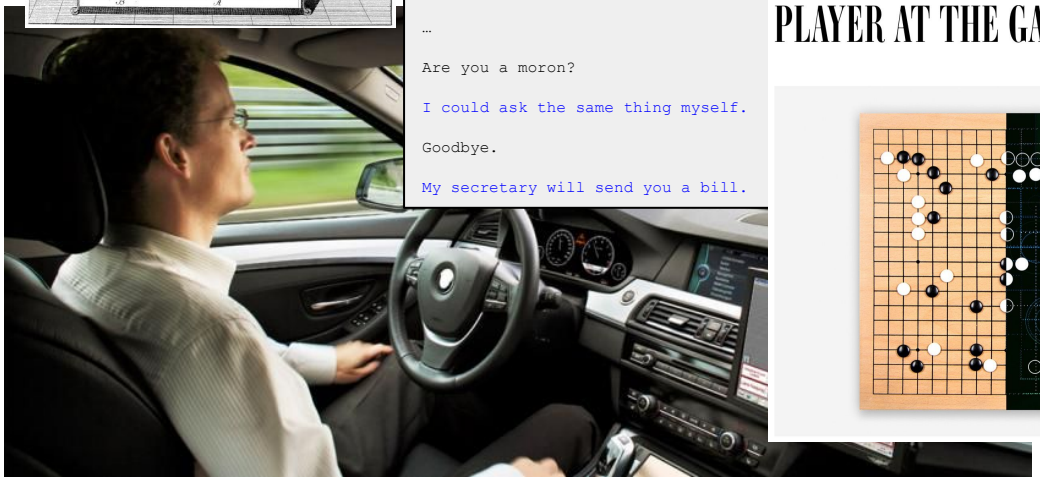


Artificial Intelligence

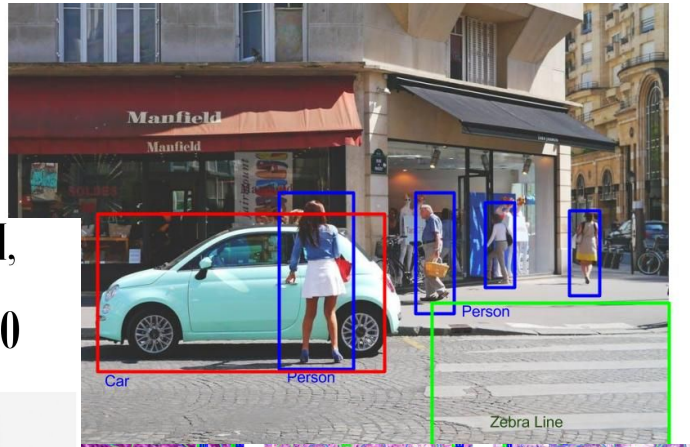
Anything machines can be made to do that humans thought only they can do.

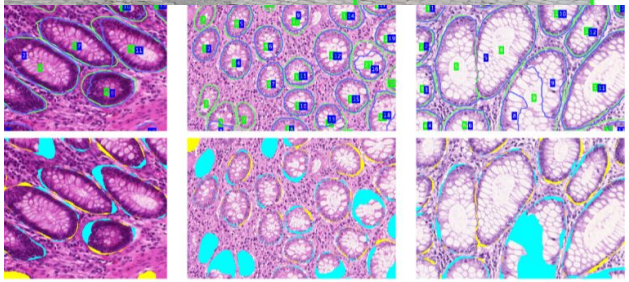


```
I am the psychotherapist.  Please, describe your problems.  Each time
you are finished talking, type RET twice.

Is P = NP going to be solved this year?

What do you think?

I think not, but it would great to hear that your thoughts on this are.

Is it because it would great to hear that my thoughts on this are that
you came to me?

Exactly, my friend.

Why do you say exactly your friend?

…

Are you a moron?

I could ask the same thing myself.

Goodbye.

My secretary will send you a bill.
```



IN A HUGE BREAKTHROUGH, GOOGLE'S AI BEATS A TOP PLAYER AT THE GAME OF GO





(d) benign (e) benign (f) malignant

Machine Learning

Traditional programming. Human invents how results are computed from data.

Machine learning. System learns how input and results are related.

Machine Learning types

- **Supervised learning** ← focus in this course
  - Labeled data, learn how to associate data with label
- **Unsupervised learning**
  - Clustering; divide input into categories
  - Dimensionality reduction; identify input features that depend on other features and remove those
- **Self-supervised learning**
  - Supervised learning without need for human-made labeling; text generation by learning to predict next word based on previous words
- **Reinforcement learning**
  - Select actions to maximize reward based on information available; automatic Atari video game playing to maximize score, based on screen pixels

“Shallow” models

Examples - linear regression, polynomial regression; fit a function to inputs and use the fitted function to make predictions with never seen inputs.

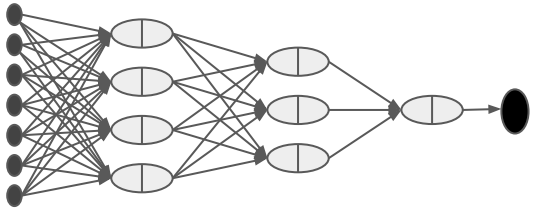
Number of parameters depends on

- Model complexity (for example, 3rd order polynomial has 4 parameters)
- Input dimensionality (# of input features)

Typically requires that input is selected and processed/transformed carefully; **feature engineering**.

More expressive/clever models exist; support vector machines (SVMs) etc.

Deep Learning



Use of multilayer (deep) networks of artificial neurons with **high number of parameters** to learn input/output relationship → artificial neural network (ANN).

Can learn **complex** input/output relationships.

Can often learn an efficient **representation** of input → reduces need for feature engineering (which is very much needed in more shallow machine learning models). Learned representation can be **hierarchical**.

Usually needs **large training material volumes** and takes significant computational resources to train.

Why deep learning now?

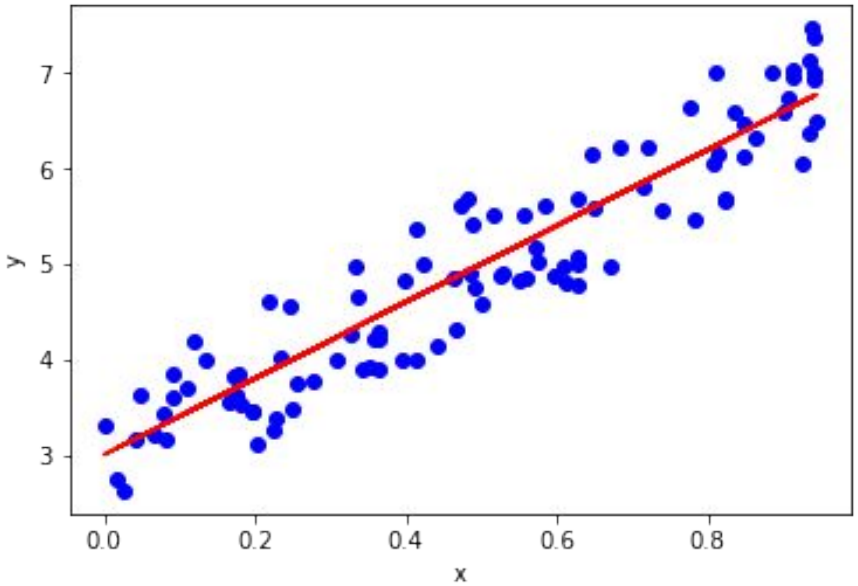
Algorithms - better understanding of ANN initialization, cost functions, network components, and cost function optimization. This field of science / engineering is far from mature, though.

Hardware - highly parallel computing available for mass-market prices. Video card with 1280 cores and 6G memory for about 250€ is a powerful tool.

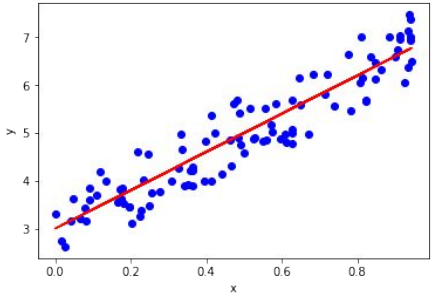
Datasets - datasets from diverse sectors available. Image recognition: ImagetNet (1.4 m annotated images), text processing: large corpuses for most languages, etc.

Linear regression

Let’s start with linear model. Classic example: fit a straight line to a set of given observations as well as possible.



Linear regression



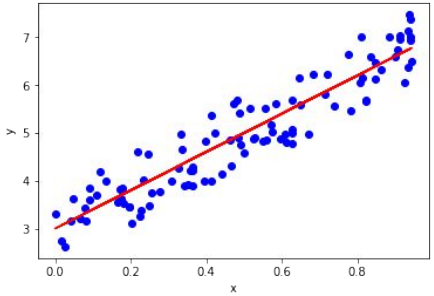
Our **model** is a straight line: $\mathbf{y = w_0 + w_1 * x}$

So, the model has two **parameters** (w_0 and w_1) and we want to find values for them so that a straight line fits to data as well as possible.

What is as well as possible? In this case, **mean squared error** (mse) is typically used as the **loss** value to be minimized.

Error = $\mathbf{1/m * (\sum_{i=1,...,m}(y^{(i)} - y\_pred^{(i)})^2)}$ where m is the number of data points (samples), $y\_pred^{(i)} = w_0 + w_1 * x^{(i)}$ is the prediction (or estimate) given by the model for data point $x^{(i)}$, and $y^{(i)}$ true value for corresponding y.

Linear regression



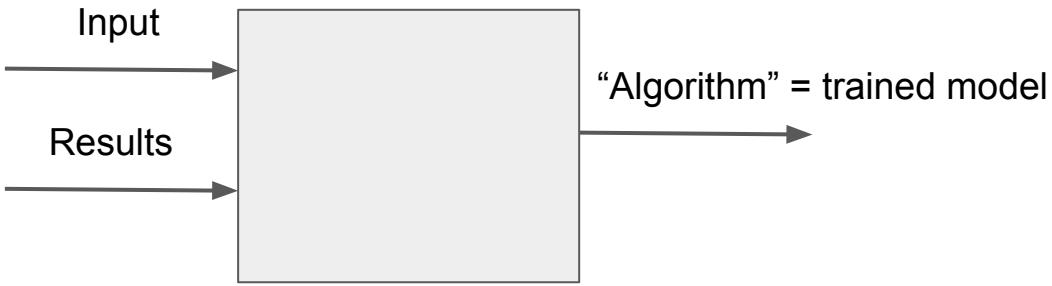
To minimize mse =

$$\mathbf{1/m * \sum_{i=1,...,m}(y^{(i)} - y\_pred^{(i)})^2 = 1/m * \sum_{i=1,...,m}(y^{(i)} - (w_0 + w_1 * x^{(i)}))^2}$$

observe that the function is convex → minimum is found at point where derivative = 0. (Note: we are **minimizing with respect to w_0 and w_1**.) To find the minimum, an iterative method called stochastic gradient descent (sgd) can be used (more on this and other optimization methods later). (In this case, the problem can also be solved in closed form but that is in general not the case).

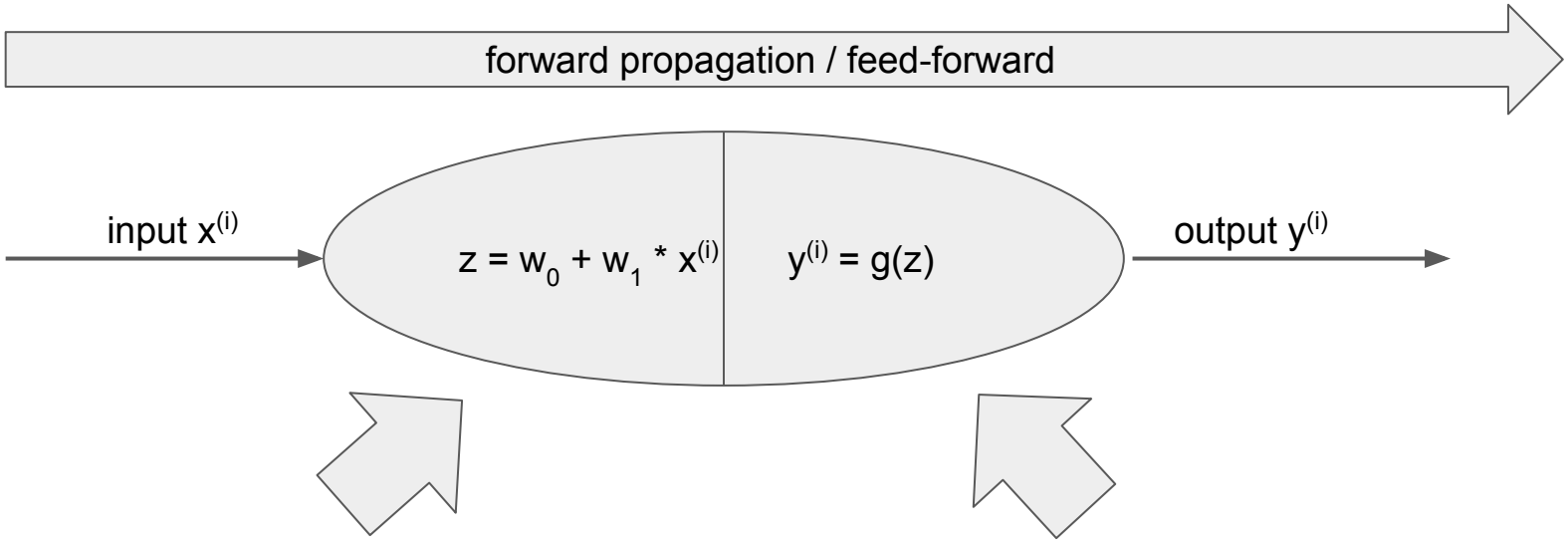
Artificial neural network for the regression example

Inputs for the example are $x = [x^{(1)}, \ldots, x^{(m)}]$ and results $y = [y^{(1)}, \ldots, y^{(m)}]$.



We now want to find the “algorithm”, ie. rule by which we can compute a result for a previously unseen input. We need to make decision on the structure of the ANN; in this case we use a simple architecture. After the decision on structure we train the model - find suitable values for the model parameters by comparing results of known samples to predicted values.

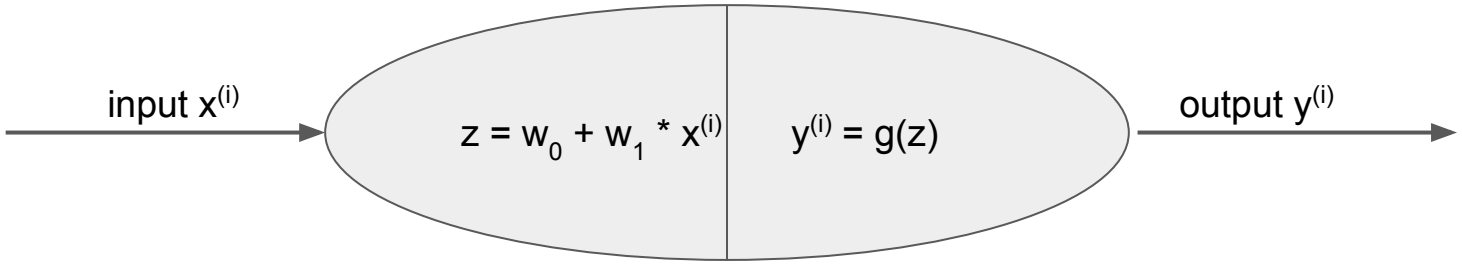
A single artificial neuron single input network



Linear part of the AN. Applies (kernel) **weights** (the parameters w_0 and w_1 that are to be learned) to input.

Non-linear part; computes **activation function** g(z) which is output by the AN. Several functions are commonly used; more later. In out example case g is identity function g(z) = z. (So in this case activation function is linear.)

Single neuron network in Keras



Example in file `linregr.ipynb`

Build a **model** that has a sequence of **layers**. Here we have only one.

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(1,)))
```

Add a layer to the model.

Layer is fully connected (dense), output dimension is 1, and input dimension is 1. No activation function defined: default is identity function.

Specify loss and optimization

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(1,)))

model.compile(optimizer='sgd', loss='mse', metrics=['mse'])
```

Specify optimizer; the way in which suitable model weights are searched for.

Specify function for computing the loss ie. what will be optimized.

Specify function(s) for computing the metric of how the model performing. This is shown when the model is trained.

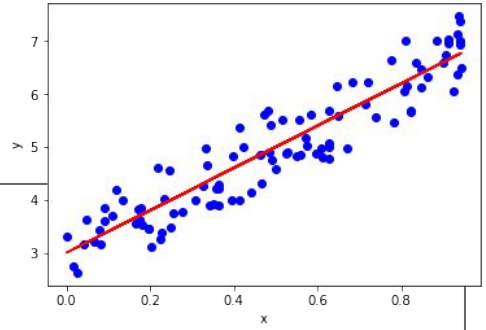
Train the model

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(1,)))

model.compile(optimizer='sgd', loss='mse', metrics=['mse'])

hist = model.fit(x, y, epochs=10, batch_size=16)
```

Train the model by providing it all the x values and corresponding y values, ie. (x,y) pairs from which it is hoped to learn the weights (parameters) in the network.

Make 10 iterations with the whole training material in batches of 16 (x,y) pairs (more on batches when we talk about optimization).

Use the model




```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(1,)))

model.compile(optimizer='sgd', loss='mse', metrics=['mse'])

hist = model.fit(x, y, epochs=40, batch_size=16)

p = model.predict(np.array([3]))

model.get_weights()

model.summary()
```

- Make a prediction
- Take a look at the weights (note: makes sense only in simple cases like this)
- Print a summary of the model (can also be done before compiling and fitting)

After running sgd on 2000 data points 40 times we get w_0 = 3.012 and w_1 = 3.986. (Values used for creating the data were 3 and 4)

Data representation

In the linear regression example we had m **training samples** of inputs and results. Both inputs and results were scalar values and their **shape** as **numpy** arrays was (2000, 1), ie. we had 2000 samples, and each sample was one-dimensional.

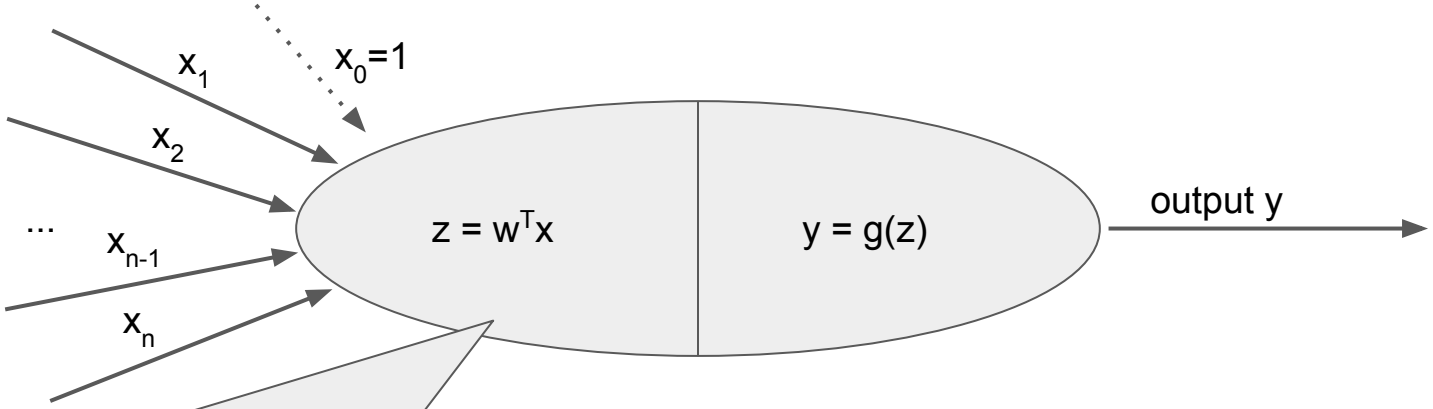
If we'd have three input features, the shape of input would be (2000, 3). For a 64 x 64 RGB image, input shape would be (m, 64, 64, 3). We call the numpy arrays where we store the information **tensors**. A tensor is defined by:

- Rank, or # of axis. For linear regression example rank = 2, for image = 4
- Shape, or # of dimensions for each axis.
- Data type of elements. Usually float32 (or float64) in deep learning applications

Manipulating tensors with numpy

Take a look at exaples in file
`numpyexamples.ipynb`

Neuron with vector input



z is computed by multiplying x_i by corresponding weight w_i and summing over all n multiplications: $z = \sum_{i=0,\ldots,n} w_i * x_i$.

(With w and x interpreted as vectors, this is their dot, or inner, product marked as $w \cdot x$, or $w^T x$, or `numpy.dot(w,x)`.)

Note that $x_0=1$ means that w_0 is the **bias** for the neuron. Sometimes notation $z = w^T x + b$, where b is bias, is used.

Linear regression: Boston housing dataset

Publicly available dataset that has 13-dimensional data on houses and their values (see https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names). The data is available in `keras.datasets`. The set is divided into training and test sets (more on this later), and can be loaded quite nicely:

```
from keras.datasets import boston_housing

(x_train, y_train), (x_test, y_test) =
boston_housing.load_data()
```

Example is in file
bostonhousinglinear.ipynb

```
x_train.shape → (404, 13)
x_test.shape → (404,)

x_train[78] → [1.71710000e-01 2.50000000e+01 5.13000000e+00 0.00000000e+00 4.53000000e-01 5.96600000e+00
  9.34000000e+01 6.81850000e+00 8.00000000e+00   2.84000000e+02   1.97000000e+01 3.78080000e+02 1.44400000e+01]
y_train[78] → 16.0
```

Input normalization

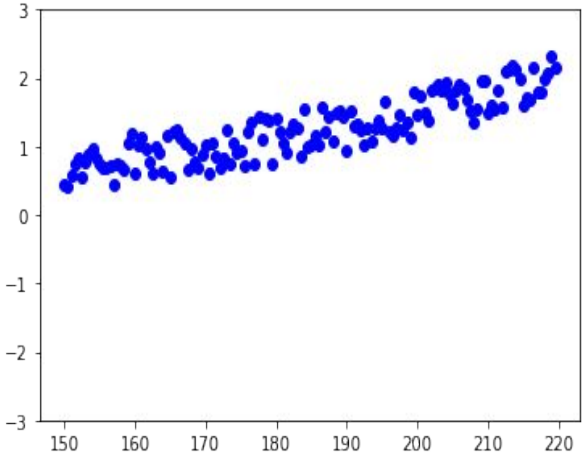
ANN optimization works best (or, often, at all) when input feature value ranges are close to each other and relatively close to 0. This is because of the way optimization works (more later). In practise input values are often scaled to ranges close to -1...1 or 0...1.

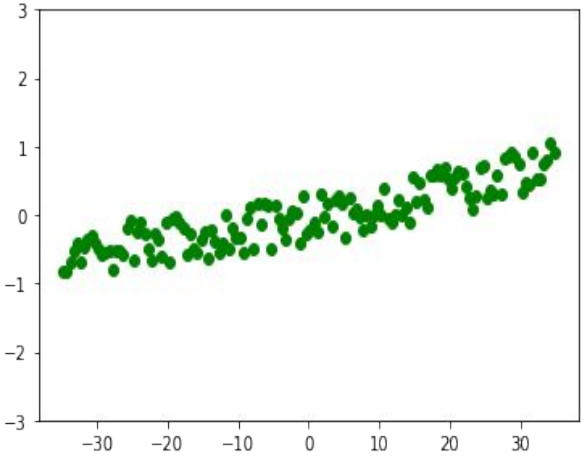
This can be done for 8-bit pixel values by dividing with 255. For other data, centering to 0 and scaling with standard deviation (or 2 * std) is a popular strategy.

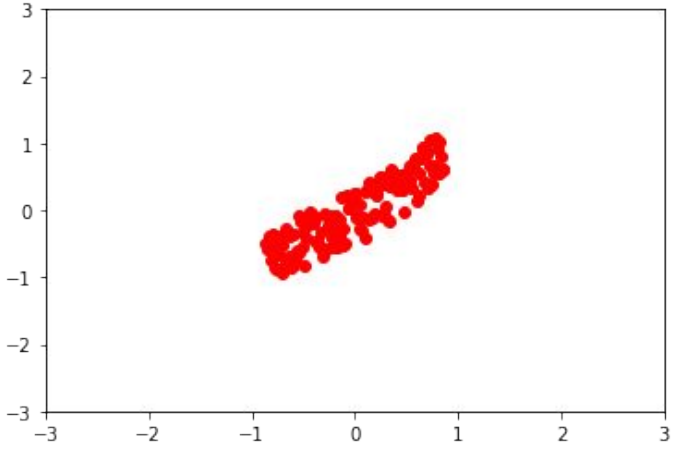
With numpy this is quite straightforward:

```
mean = x_train.mean(axis=0)
std = x_train.std(axis=0)

x_train -= mean
x_train /= std
```

Input normalization








```
x -= x.mean(axis=0)
```

```
x /= (2 * x.std(axis=0))
```

So, let’s normalize

```
mean = np.mean(x_train, axis=0)
std = np.std(x_train, axis=0)

x_train -= mean
x_train /= (2 * std)

x_test -= mean
x_test /= (2 * std)
```

Use mean and std of training set also for normalizing the test

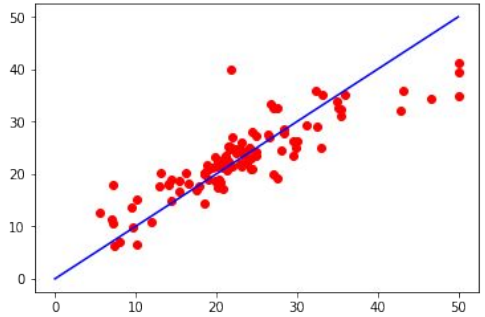
Define, train, and use model

This time input is 13-dimensional

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(13,)))

model.compile(optimizer='sgd', loss='mse')

hist = model.fit(x_train, y_train, epochs=10, batch_size=64)

preds = model.predict(x_test)

print(np.sum(np.round(preds[:,0]) == y_test) / len(y_test))
```

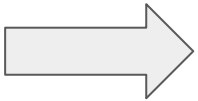
Plot test set values y_test against predictions (and straight line for reference) by:

```
import matplotlib.pyplot as plt
plt.plot(y_test, preds, 'ro')
plt.plot([0,50], [0,50], 'b')
```



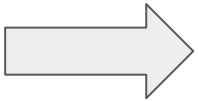
Inspect model

```
model.summary()
```



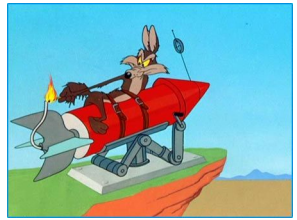
```
Layer (type)                 Output Shape              Param #
=================================================================
dense_4 (Dense)              (None, 1)                 14
=================================================================
Total params: 14
Trainable params: 14
Non-trainable params: 0
_________________________________________________________________
```

```
model.get_weights()
```



```
[array([[ 0.05344158],
       [ 0.63540411],
       [ 0.37158501],
       [ 0.39288247],
       [-0.03653026],
       [-0.19312069],
       [-0.01441616],
       [-0.58739501],
       [ 0.28077316],
       [-0.06558847],
       [ 0.60122478],
       [-0.32361585],
       [ 0.17224699]], dtype=float32), array([ 0.], dtype=float32)]
```

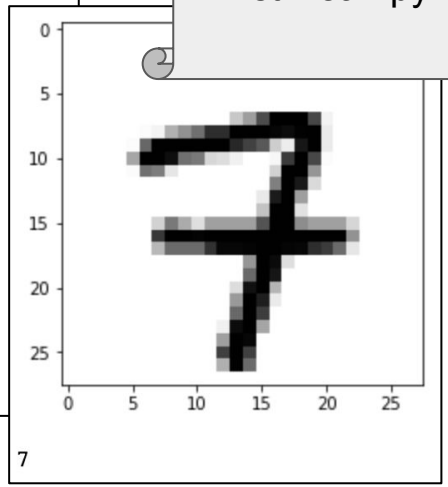
Linear regression with image data



Let's create a model that uses linear regression to recognize handwritten digits. We will use the publicly available NMIST data set of labeled images to do this.

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()

exampleindex = 324
digitimage = x_train[exampleindex]
digitlabel = y_train[exampleindex]

plt.imshow(digitimage, cmap=plt.cm.binary)
plt.show()
print(digitlabel)
```

Example is in file nmistlinear.ipynb

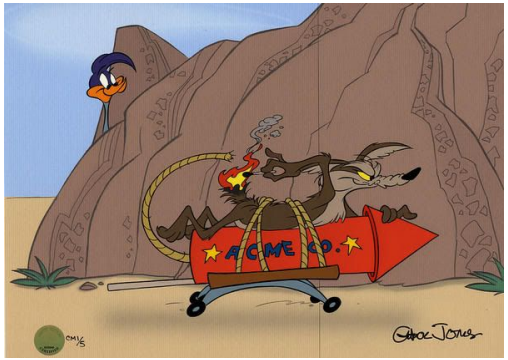


Normalize image data and shape it into vector

Re-arrange 28 by 28 matrix into 28*28 vector

Convert type into float for scaling

```
x_train = x_train.reshape(-1, 28 * 28).astype('float32') / 255.0
x_test = x_test.reshape(-1, 28 * 28).astype('float32') / 255.0
```

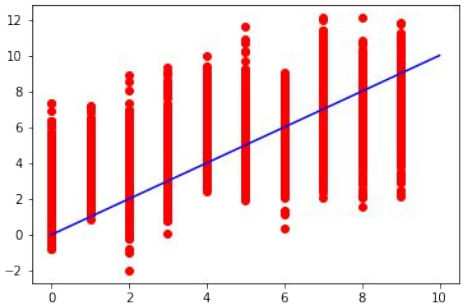


Scale into 0...1

Define, train, and use model

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(28 * 28,)))

model.compile(optimizer='sgd', loss='mse', metrics=['acc'])

hist = model.fit(x_train, y_train, epochs=10, batch_size=64)

preds = model.predict(x_test)

print(np.sum(np.round(preds[:,0]) == y_test) / len(y_test))
```

We get 0.202 and this plot:



Or was it a fail? How do we know?

Compare to theoretical maximum. Depending on how sure one is about the quality of the samples and labeling this would be ~ 100% (in this case).

Compare to a baseline, for example human operator. Fraction of correctly predicted would be ~ 99,9%.

Compare to random guessing. That would give 10%.

So we are better than random guessing but quite far from baseline. How do we know where we are and what can we do?

Installing needed software

What you will need:

- Python (preferably >= 3.5)
- Numpy (tensor manipulation library)
- Matplotlib (to create graphs, diagrams etc)
- Tensorflow (the engine on which machine learning algorithms work. Tf also provides access to your hardware (CPU/GPU) for efficient computation.)
- Keras (the user-friendly framework we are using to create ANNs)
- Jupyter (exercises are to be returned as jupyter notebooks and some materials are available in that format, too)
- (and some more libraries as we move forward)

You might want to use a python environment management system, for example Anaconda, but it is up to you to decide on that. Be careful, esp. with python2 / python3 issue (or feature? or disaster?).

To get started, look here: https://www.tensorflow.org/install/ and here: https://keras.io/#installation

Exercise: grading

grading.npz contains two numpy arrays: first of shape (2000,3) that contains points from all parts of a course, and second, of shape (2000, 1) that contains grades. Study the properties of the data (for example: mean values, minimums and maximums). Train a **linear** model to do the grading. What accuracy do you reach? Does training with more epochs help? Does batch_size have an effect? Do to weights you get make sense?

Hint: to read .npz file you can, for example, do this:

```
import numpy as np
with np.load("grading.npz") as data:
    x = data[ 'x' ]
    y = data[ 'y' ]
```

Reading list for exam 22.1

session01.pdf

“Deep Learning with Python” by Francois Chollet: chapters 1, 2.2, 2.3.1-2.3.5 (available at https://www.manning.com/books/deep-learning-with-python)