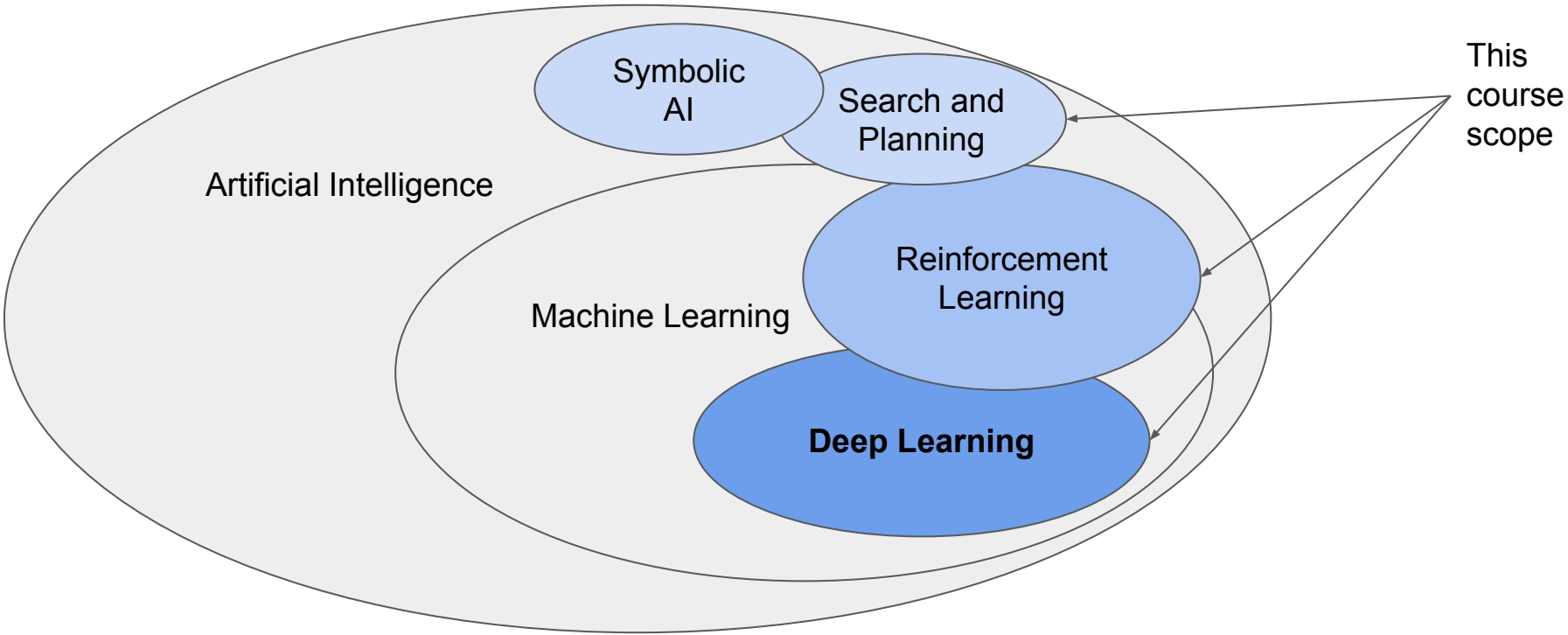
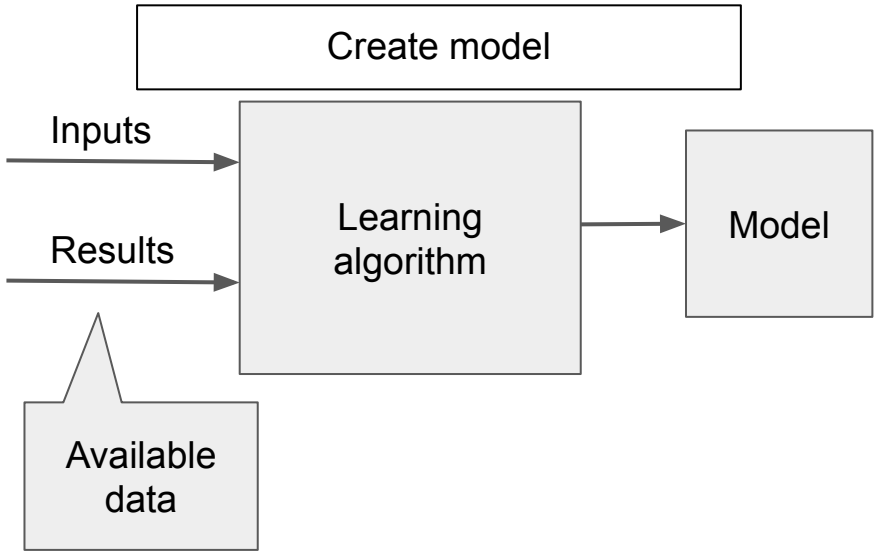
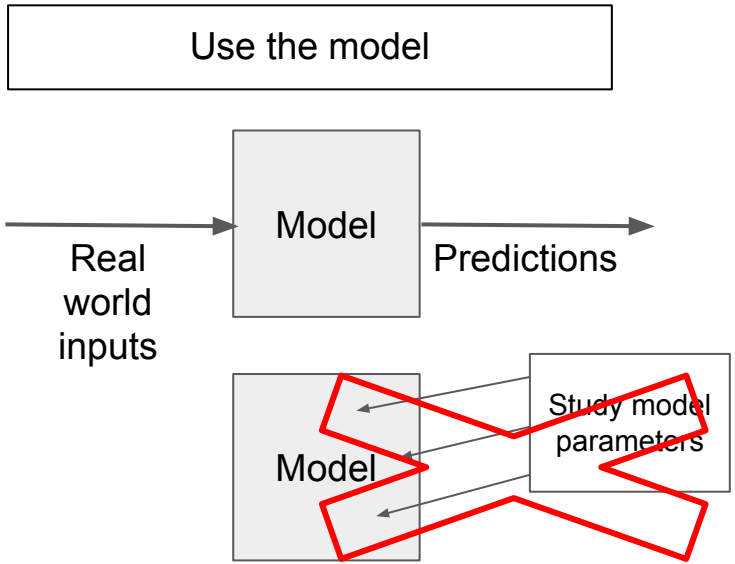
Machine Learning in Artificial Intelligence (AI) landscape



Model - the “algorithm”

Model is an entity that is created from data and allows us to make predictions. Models we talk about are parametric - their behaviour depends on parameters whose values are fixed during model development. Model can be very simple, like $y = w_0 + w_1 x$ (linear regression), or very complicated like a deep net with 152 layers and 130 million parameters. (Note: diagrams below are specific to supervised learning).

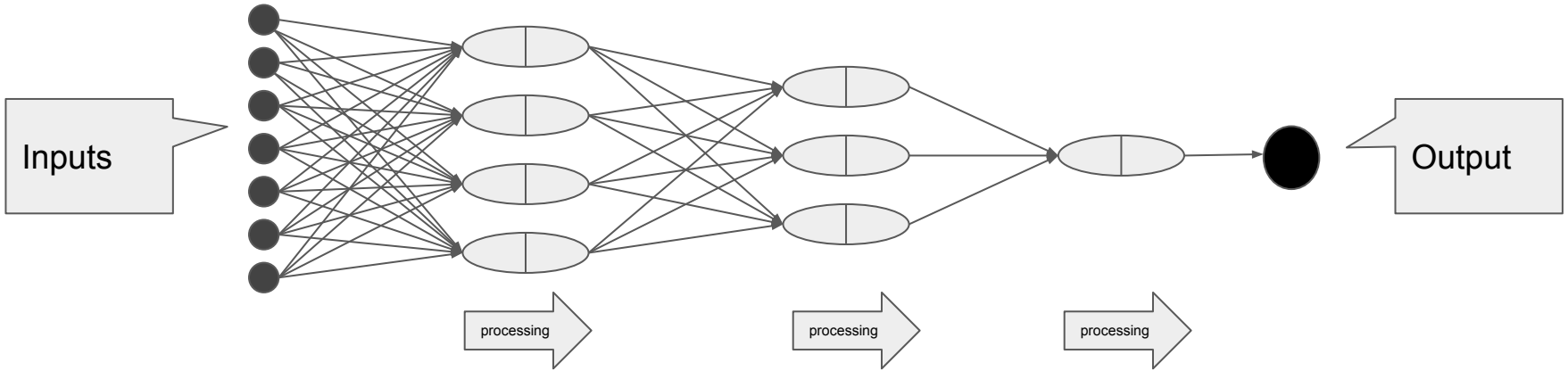




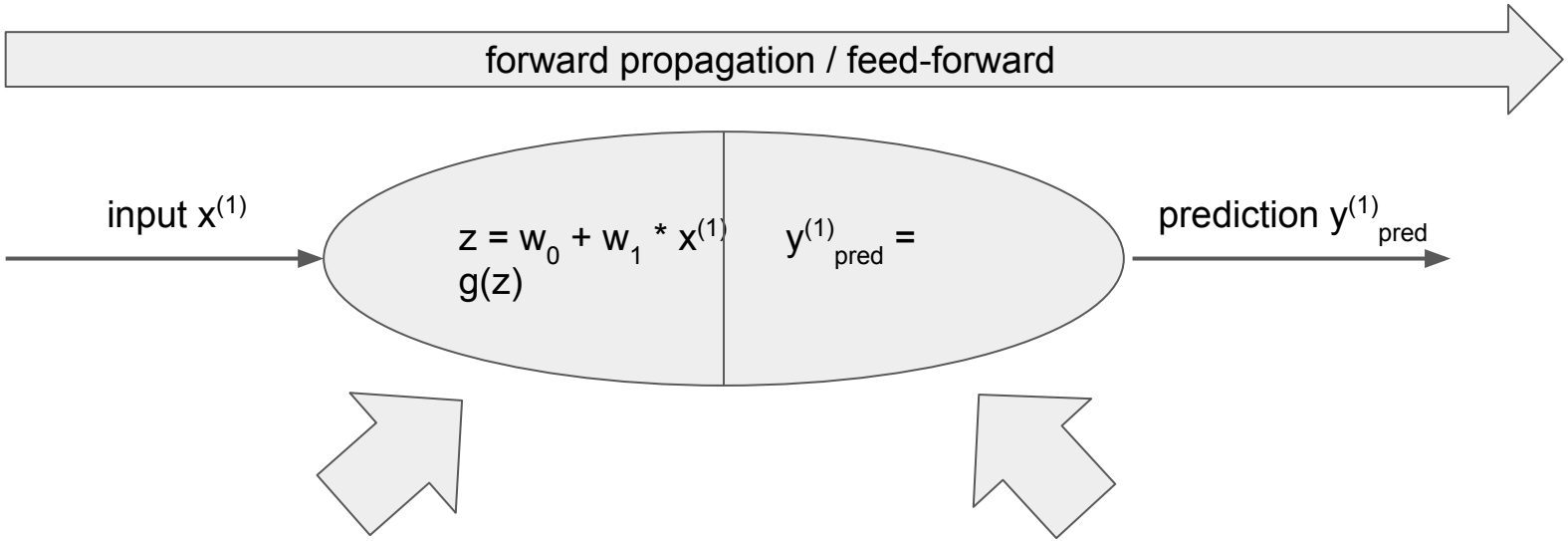
Model in deep learning

Instead of creating the model by trying to find a parametric function that would represent the input/output relationship, we'll create an artificial neural network (ANN) from artificial neurons (ANs). The model is still parametric - the individual ANs have parameters, or weights, that need to be fixed during training. Motivation for this is that we hope the ANN to learn how inputs and outputs are related by finding suitable weights.

The network is organised as a sequence of layers through which data flows when predictions are made.



A single artificial neuron with single input feature



Linear part of the AN. Computes a linear combination of **weights** (the parameters w_0 and w_1 that are to be learned) and the input.

Non-linear part; computes **activation function** *g(z)* whose result is the output of the AN. Several alternatives are commonly used; see next slide.

Some activation functions

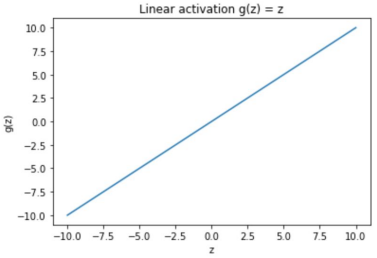
Identity: g(z) = z. If only identity is used in all network layers → completely linear model, no benefit over traditional linear model. Not an often used activation function.

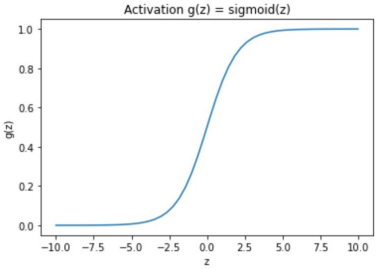
Sigmoid: g(z) = 1 / (1 + e^{-z}). Smooth (and differentiable) with range 0...1. Useful when probabilities are needed as output.

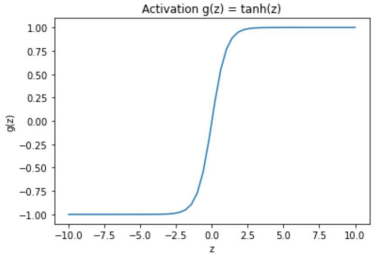
Tanh: g(z) = (1 - e^{-z}) / (1 + e^{-z}). Like sigmoid, but with range -1...1.

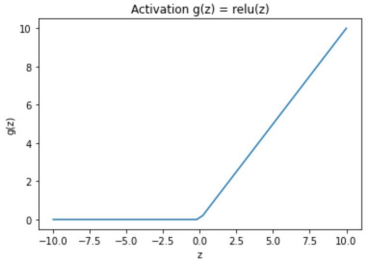
Rectified linear unit (relu): g(z) = max(0, z). Not differentiable at 0, but in practise does not matter. Most popular activation function, especially in convolutional networks (more on those later).

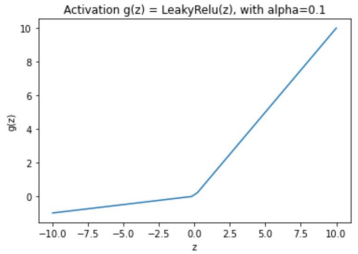
Leaky ReLU: g(z) = z for z >= 0, alpha * z for z < 0, where alpha is usually ~ 0,01



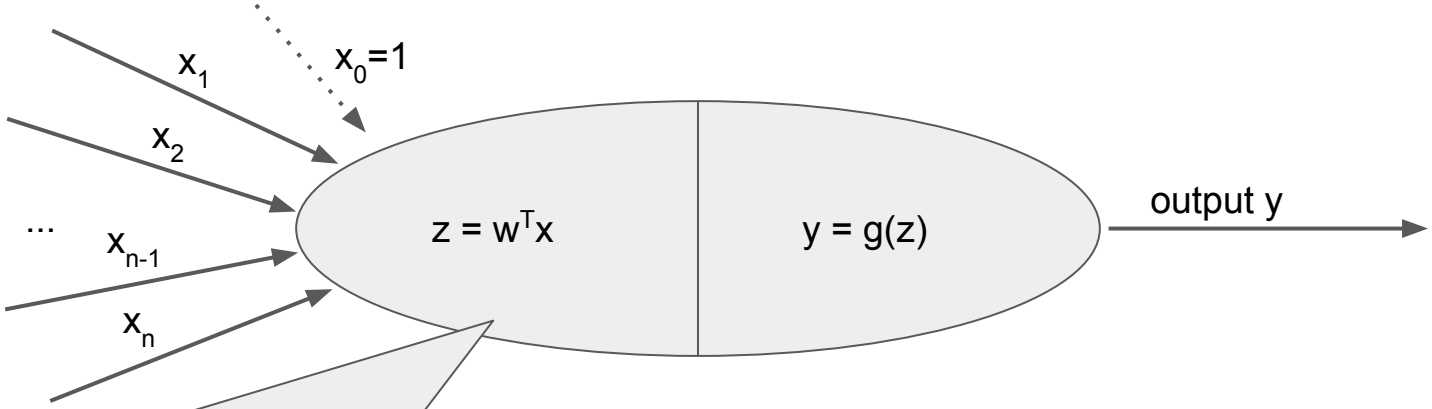








Neuron with vector input

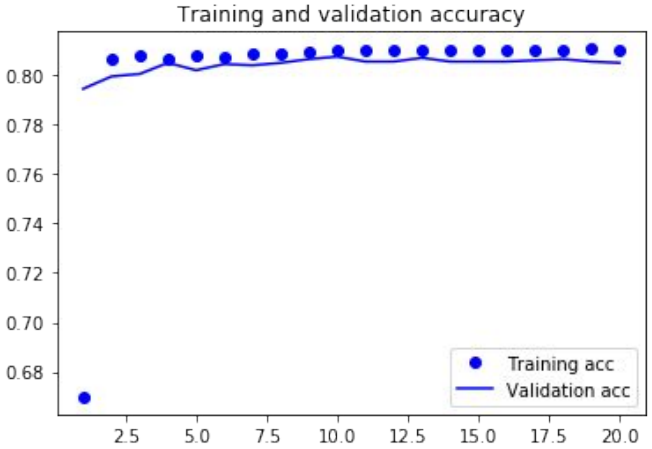


z is computed by multiplying x_i by corresponding weight w_i and summing over all n+1 multiplications: $z = \sum_{i=0,\ldots,n} w_i * x_i$.

(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)`)

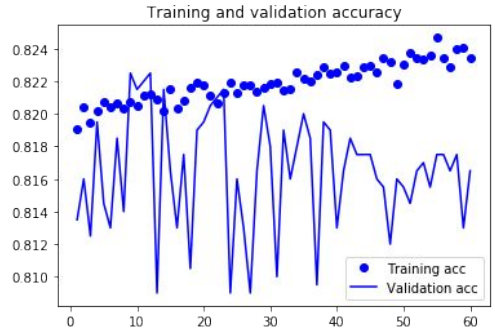
Note that $x_0=1$ means that w_0 is the intercept term for the linear part. (Sometimes also notation $z = w^T x + b$, where b is bias, is used.)

Accuracy for training/validation sets

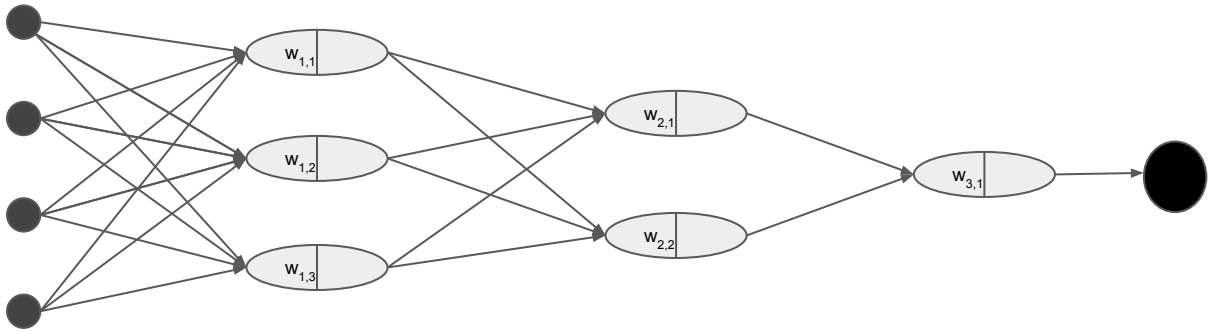


Ok, if ~80% accuracy is enough. If more should be achieved, we are **underfitting** ie. model capacity is not enough to capture input/output association.

Overfitting: training set accuracy increases, validation set accuracy decreases or stays the same.



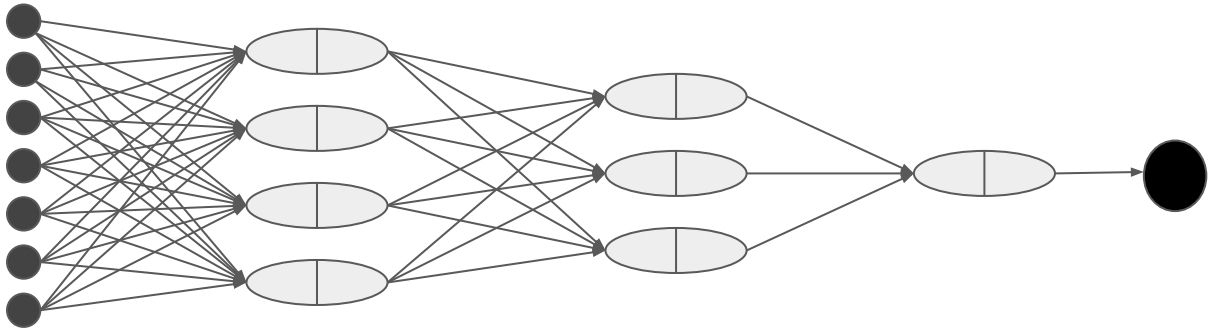
Dense network forward pass



Suppose the input x=(0,1,2,3), weights $w_{1,1}$=(0,1,0,1,0), $w_{1,2}$=(0,1,2,1,2), $w_{1,3}$=(0,1,1,1,1), $w_{2,1}$=(0,-2,1,-2), $w_{2,2}$=(0,1,-1,0), $w_{3,1}$=(-1,-3,2) and activation function in layers 1 and 2: g(z)=relu(z) and in layer 3: g(z)=sigmoid(z).

Now $y_{1,1}$=2, $y_{1,2}$=10, $y_{1,3}$=6 → $y_{2,1}$=relu(-6)=0, $y_{2,2}$=relu(-8)=0 → $y_{3,1}$=sigmoid(-1)=0.3

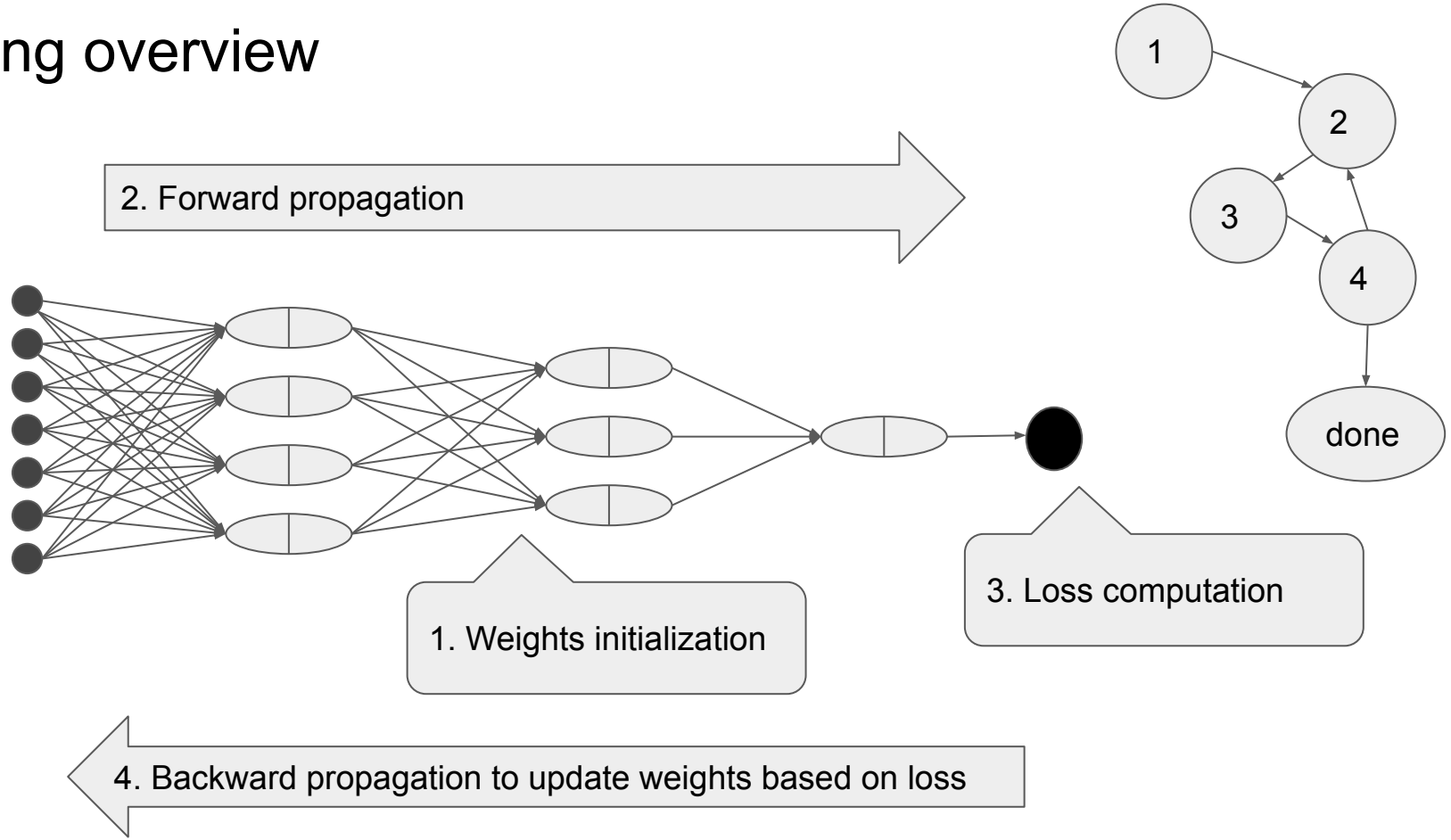
Dense network number of weights



Dense, or completely connected, network; each output is connected to all next layer inputs.

- # of weights in 1st hidden layer: (input dimension (7) + 1) * # of neurons (4) = 32
- # of weights in 2nd hidden layer: (previous layer # of neurons (4) + 1) * # of neurons (3) = 15
- # of weights in output layer: (previous layer # of neurons (3) + 1) * # of neurons (1) = 4

Training overview



Backpropagation and gradient

Use the **gradient** (~ multi-dimensional derivative) of the cost function to decide corrections to weights. The components of gradient are multiplied with **learning rate** (which is a controllable parameter) and subtracted from current weight values.

Calculation of correction terms proceeds from the output towards the input, using chain rule of derivatives in the process to “allocate” error correction to each weight in each layer. This will lead to navigating towards a minimum of the loss function. Thus the name **gradient descent**.

(Those interested in the math of this, please take a look at for example: https://theclevermachine.wordpress.com/2014/09/06/derivation-error-backpropagation-gradient-descent-for-neural-networks/)

Managing overfitting - regularization

Like with shallow models, penalize the network for too large weights; forces more generalized learning. A good way to reduce overfitting.

Modification of cost function (an example, other alternatives do exist):

$J_{reg}(w) = J(w) + \lambda \cdot \| w \|_2$ (where $\| w \|_2$ is the length of vector w)

Modified cost function will be more reluctant to increase weights because increased weights increase cost function value.

In Keras, add regularization parameter to layer specification. This means that one can decide where (in which layers) and how strong regularization is employed.

Managing overfitting - dropout layers

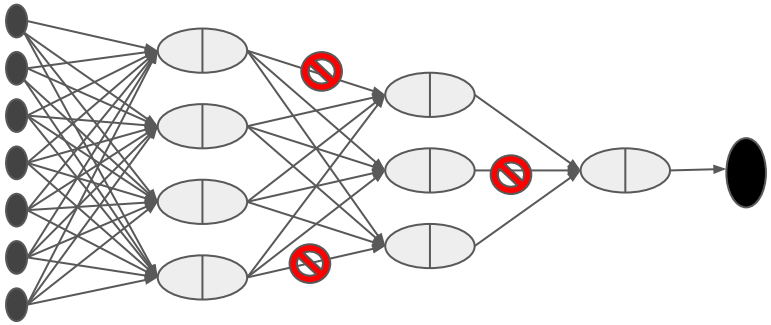
Drop out connections with a given probability → create “holes” in the network.

Forces network to learn in more “wholesome” way. Using dropout can also be interpreted as creating a random ensemble of models.

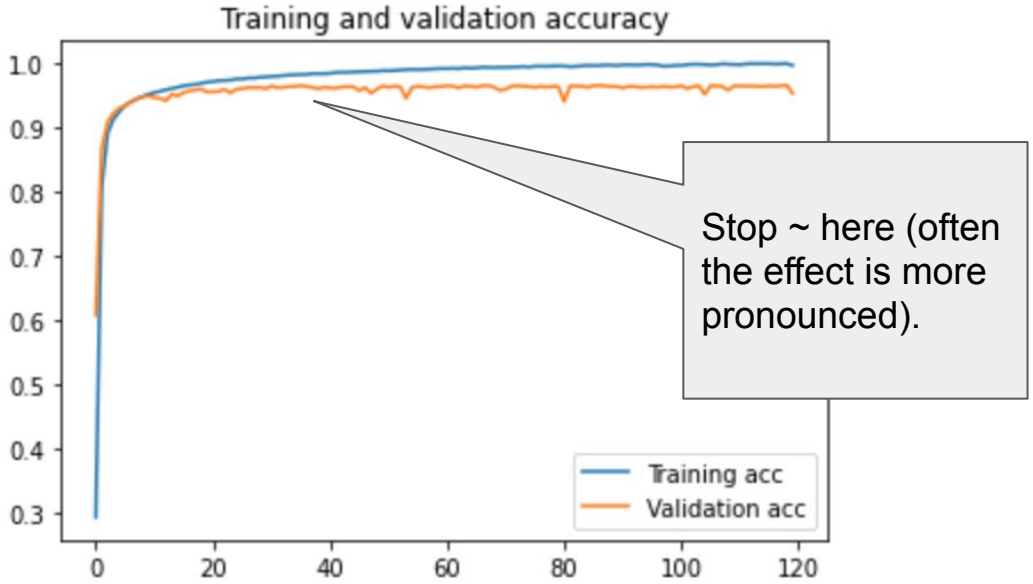
In practise introduce a dropout layer to model with a parameter specifying the probability of dropping out a connection.

Note1: weights must be rebalanced when some neurons are shut down; Keras does this automatically.

Note2: dropout only takes place during training.



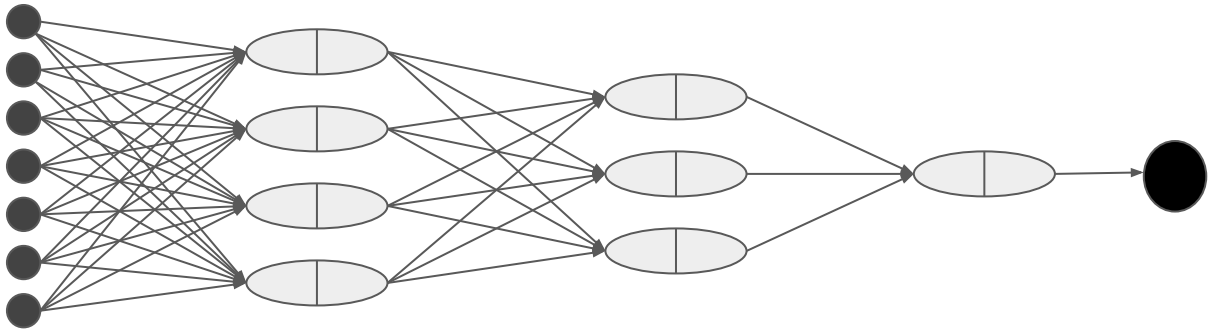
Managing overfitting - early stopping



To implement early stopping:

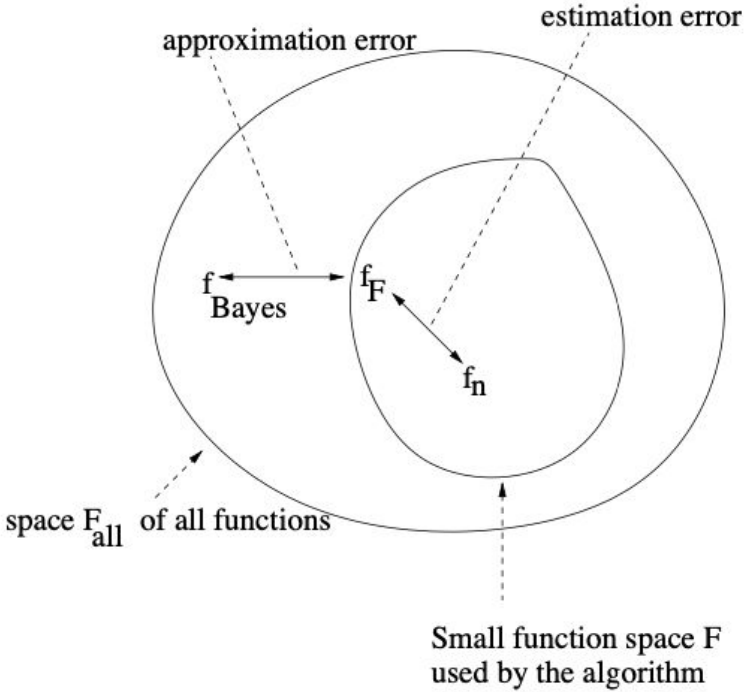
- Train for large number of epochs, look at the training/validation accuracy diagram and pick a point (# of epochs) where validation accuracy doesn't improve
- Train as long as validation accuracy improves over some number of epochs (remember, in Keras subsequent .fit() calls are cumulative).

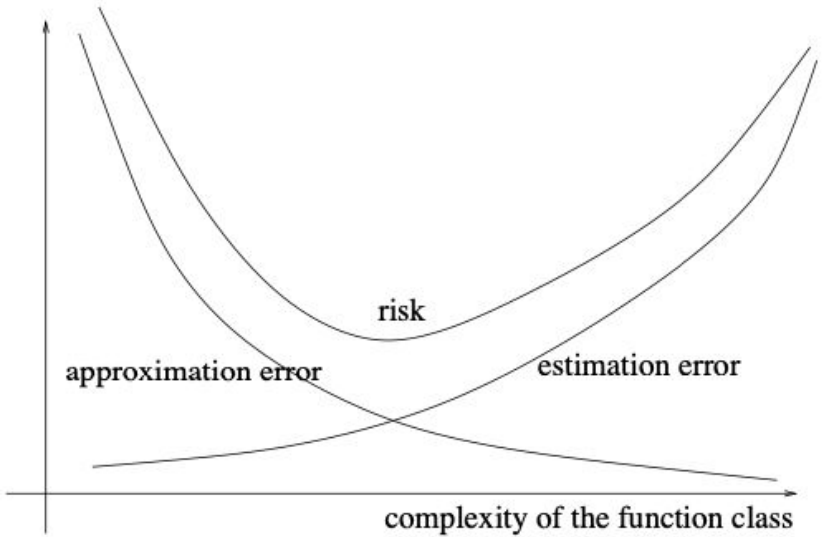
Batch normalization



We usually normalize the input data (center it at 0, and scale to standard deviation 1) before feeding it to the network. But what happens after that? Data coming out from hidden layers is not guaranteed to normalized any more. **Batch normalization** layer maintains moving average and variance per batch, and uses that to adaptively normalize the data. In practice batch normalization is needed with deeper networks architectures.

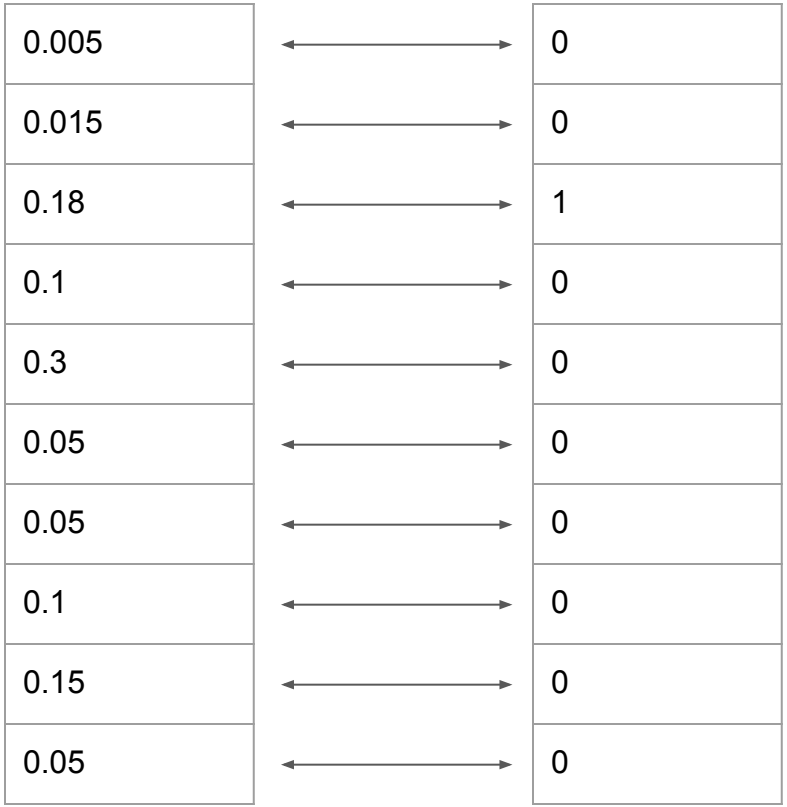
Estimation and approximation errors





From “Statistical Learning Theory: Models, Concepts, and Results” by Luxburg and Schölkopf

Why one-hot encoding



Predictions on the left-hand side are compared with true values on the right-hand side.

Cost should be high when the predictions are different from true values and low when predictions are close to true values.

How to measure cost? Categorical crossentropy.

Image classification and hierarchy

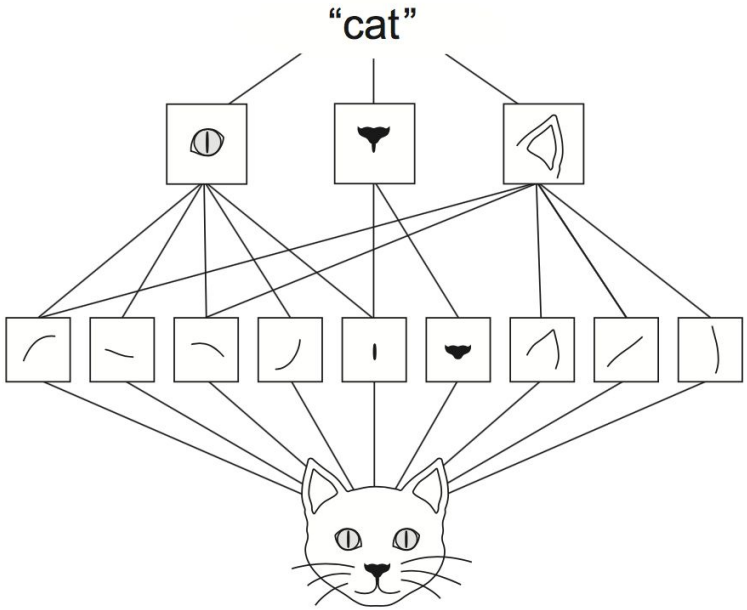


Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat."

From Chollet, chapter 5.1.1

2D convolution

Element-wise multiplication and sum:
1*0+1*3+1*6+0*0+0*4+0*7+(-1)*5+(-1)*1+(-1)*1

0	0	5	2	1	4
3	4	1	1	1	3
6	7	1	1	1	4
8	9	2	1	5	2
2	0	0	0	2	1
6	2	2	2	2	1

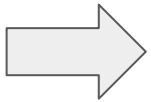
6x6x1 image

*

1	0	-1
1	0	-1
1	0	-1

3x3 **filter**, or **kernel**

Filter size is sometimes called *receptive field*



2	7	4	-7
13	17	-3	-6
13	14	-5	-5
12	8	-5	-1

(6-3+1) x (6-3+1) x 1 result

What is happening in 2D convolution

9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0
9	9	9	9	0	0	0	0

lighter pixels

darker pixels

*

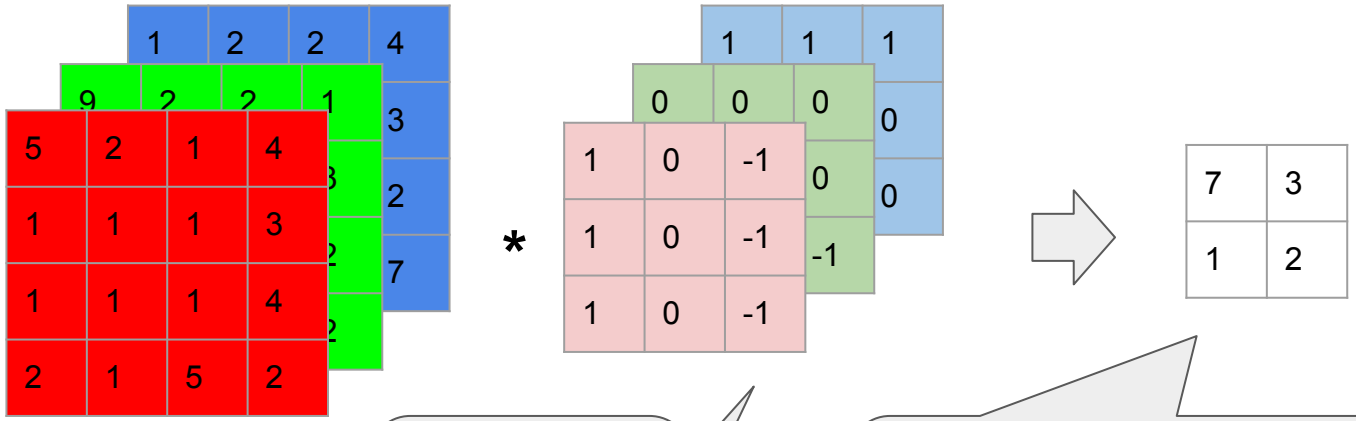
1	0	-1
1	0	-1
1	0	-1

0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0
0	0	27	27	0	0

This particular convolution filter/kernel seems to detect vertical edges.

RGB image convolution

Think of RGB image as 3-dimensional box, where channels are in depth direction:

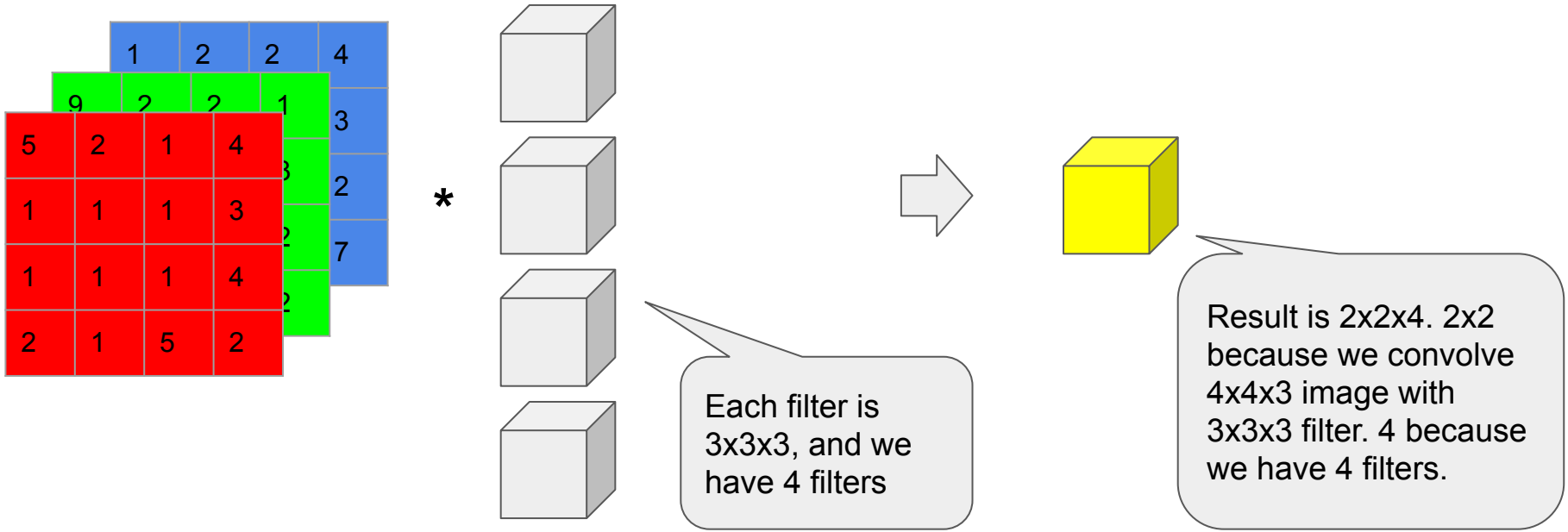


Filter must have the same depth as the sample.

Upper left corner 7 is the result of doing convolution in upper left corner of red channel with light red filter, green with light green filter, blue with light blue filter, and summing all.

Convolution with multiple filters

Usually more than one filter is used in convolution. Each filter is **applied separately**, and **results are stacked in depth dimension**:



Where do the convolution filter values come?

They are **learnable parameters** (or **weights**)!

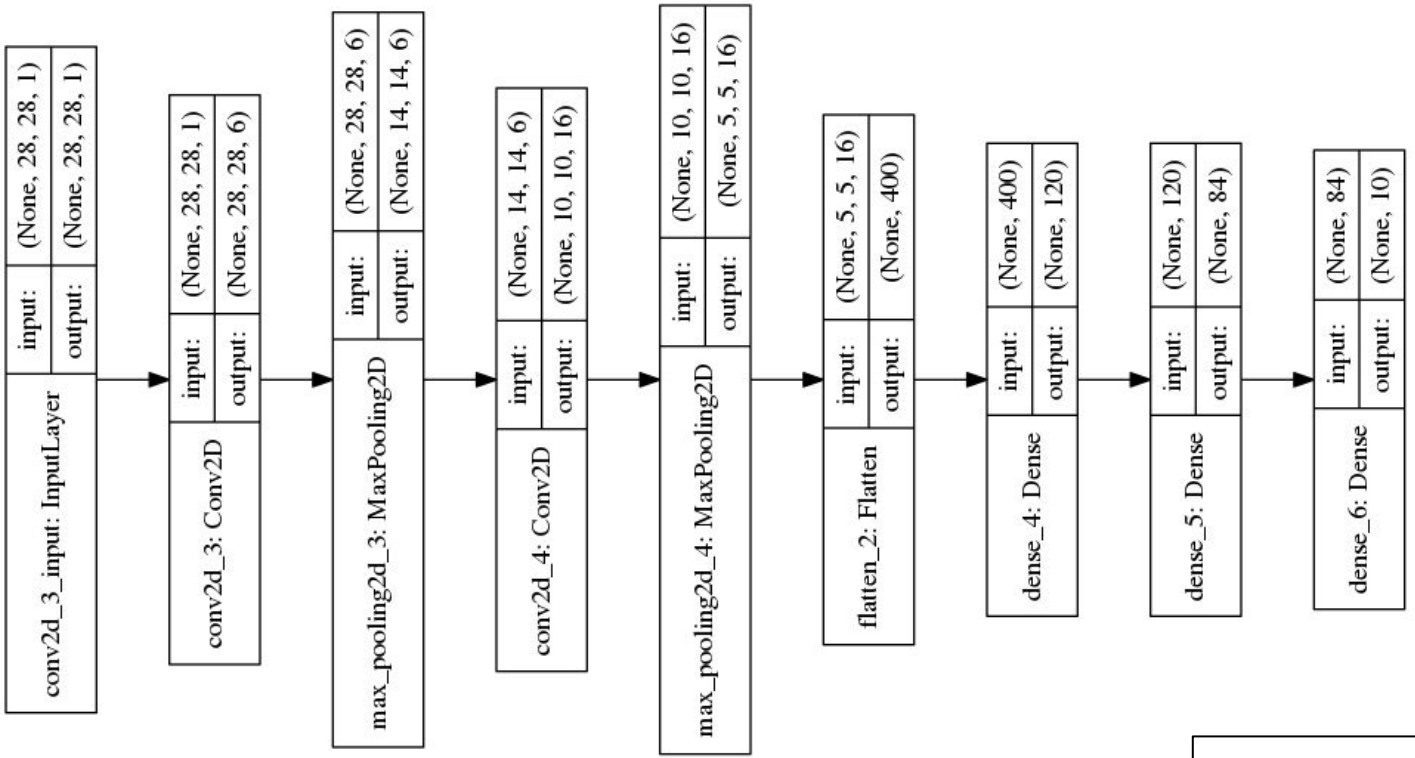
We are not designing the filters beforehand, like in signal processing, but **learn the values in filters during training**. This gives the network ability to **learn whatever filters needed to minimize the loss function**.

Filter parameters get learned once, and are then used in all spatial positions (height & width) of the image - parameter sharing.

A typical convolutional network architecture network that is tasked to do classification is divided into two parts:

- convolutional part where **an internal representation** of the image is found
- classification part which outputs probability vector that contains probabilities for all categories (this is quite like the network we have created before)

Example network: LeNet-5 (1998) (simplified)



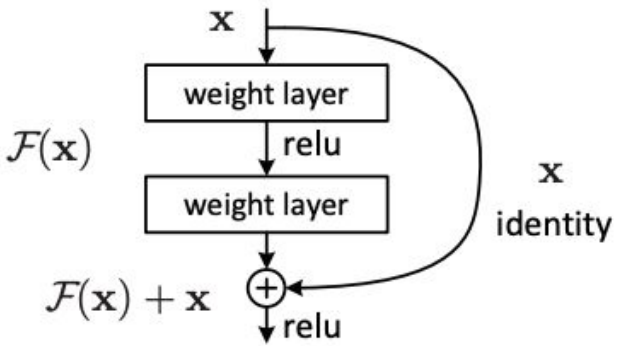
http://yann.lecun.com/exdb/lenet/

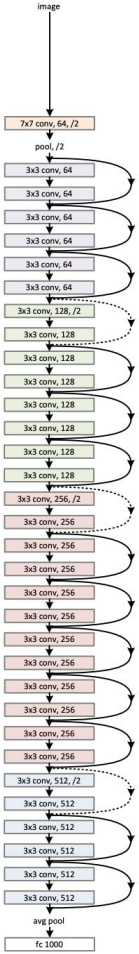
Residual connections

In **ResNet** the network is very deep (up to 152 layers) - gives ability to learn very wide family of functions, but vanishing gradients are a real problem.

Solution: skip connection - add an identity connection that “short-circuits” the conv layer. This way derivative information “leaks” to the earlier layers in the network in back propagation.

But how to implement this?





Data augmentation

Too few samples to train the model? Image classifier models need a respectable amount of data to train them with.

Don’t worry (or do) - but let’s create fake data! Well not completely fake, but for image data, for example, data from real images by random modifications:

- Rotation
- Zooming
- Flipping
- (and more, see https://keras.io/preprocessing/image/ and Chollet 5.2.5)

Transfer learning - using pre-trained convnet

For an image classification task use a pre-trained convolutional network as the basis:

- Assumption is that the convolutional part of a pre-trained network has learned general enough lower-level features when it was trained for the classification task
- The classification (dense) part of the network is specialised in the original classification task, so ignore that and train a new classifier in its place