# TX00DH43-3001
# Introduction to Deep Learning

peter.hjort@metropolia.fi

# Training / validation / test data

To analyze and understand model training progress it makes sense to divide available data into three sets:

- The model is trained (the values for its weights, or parameters, determined) using **training data set**.
- **Validation data set** is used for tuning the model (choosing # of layers, # of neurons in each layer, activation and cost functions, learning rate etc). These are called **hyperparameters**.
- **Test set** is used for checking the performance after training and validation.

(In the examples data, and in this course in general, is usually divided into training/validation sets only)

# Training / validation / test data

Before splitting, **make sure the data is in random order**! (Do a shuffle if not)

How should the data be split?

- If you have small amount of samples ( < 100.000's) standard split is
  - 70% training, 30% validation, or
  - 60% training, 20% validation, 20% test
- For large sample set (in 1.000.000's)
  - 98% / 1% / 1%

These are not hard rules. Next time we talk about K-fold validation which can help in using samples efficiently.

# Logistic regression

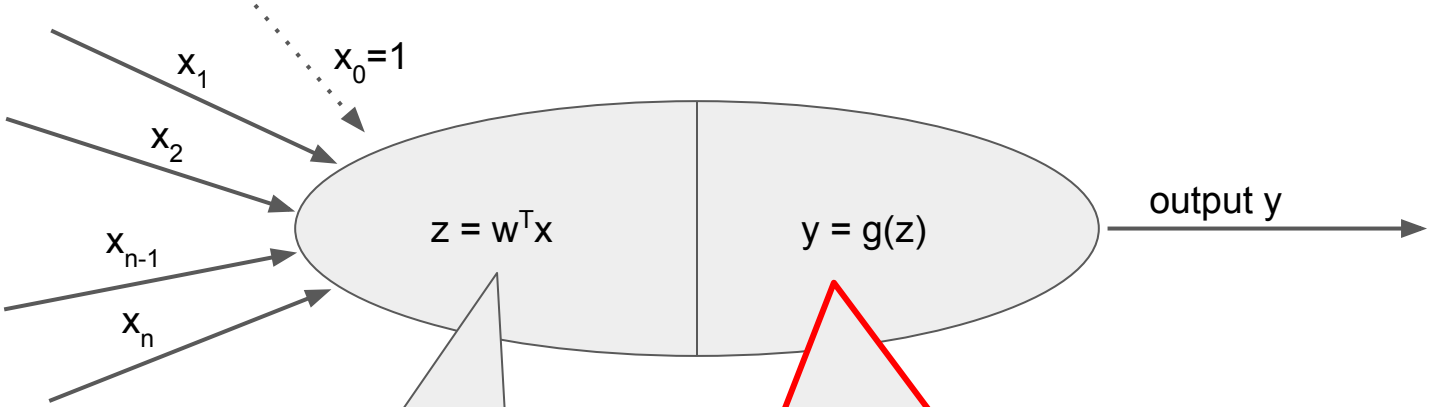

Instead of predicting a number, as in linear regression, we want to predict whether the input belongs to a category or not.

- Is this a picture of a cat?
- Is the course passed?
- Should this person be given credit?

In practice logistic regression gives a probability that input belongs to the given category (or class).

Training samples for logistic regression are pairs: $(x^{(i)}, y^{(i)})$ where $y^{(i)} = 1$ if $x^{(i)}$ belongs to the class, and $y^{(i)} = 0$ if it does not.

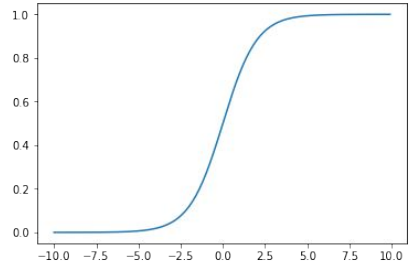
# How to predict values 0 ≤ y ≤1




$z = w^T x = \sum_{i=0,\ldots,n} w_i * x_i$.

(* = normal multiplication)

Use a (non-linear) **activation function**
$g(z) = \text{sigmoid}(z) = 1 / (1 + e^{-z})$



# But how do we measure loss in logistic regression?

Short answer: use `binary_crossentropy`

Mid-size answer: Mean squared error would be the default choice. Its use, however, leads to non-convex loss function → gradient descent does not work well. Instead, measure how similar the distribution of predictions is with the distribution of true values. Minimize log of the **cross entropy**:

- y * log(y_pred) - (1 - y) * log(1 - y_pred)

Longer answer: https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/. (A nice explanation for multiclass case which shows how the expression above is derived).

# Binary classification example: credit card defaults

Data set that contains person attributes and whether they have defaulted cc payments: https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

```
data = np.loadtxt("ccdefaults.csv", delimiter=';')
np.random.shuffle(data)

n_training = 28000

x_train = data[:n_training,0:24]
y_train = data[:n_training,24]

X_val = data[n_training:,0:24]
y_val = data[n_training:,24]

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

Shuffle data to get **same distribution in training and validation sets**.

Select n_training samples to training set. First 24 columns contain input data. Last column is binary default/not info ie. the result.

Rest of the data goes to validation set. Normalize both training and validation sets **with training set statistics**.

ccdefaults.ipynb

# Model, compilation and training

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(1, input_shape=(24,), activation='sigmoid'))

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

sigmoid activation, loss `binary_crossentropy` as discussed earlier, and define metrics to collect during training

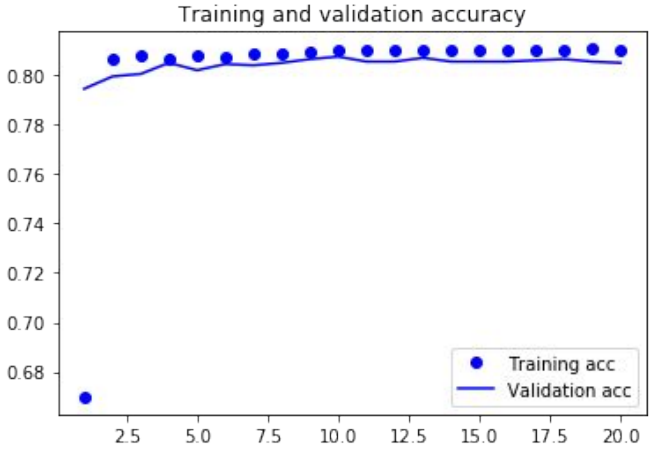
```
hist = model.fit(x_train, y_train, epochs=20, batch_size=64, validation_data=(x_val, y_val))
```

Compute metrics for validation data, too

```
acc = hist.history['acc']
val_acc = hist.history['val_acc']
loss = hist.history['loss']
val_loss = hist.history['val_loss']
epochs = range(1, len(acc) + 1)
```

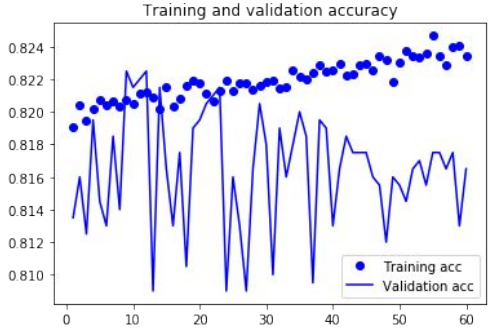
`.fit()` returns data collected during training for future use

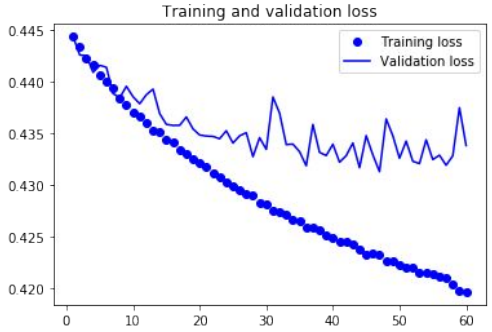
# Accuracy for training/validation sets



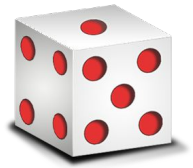

Ok, if ~80% accuracy is enough. If more should be achieved, we are **underfitting** ie. model capacity is not enough to capture input/output association.

**Overfitting**: training set accuracy increases, validation set accuracy decreases or stays the same. Loss diagram points to same diagnosis.



# How about classifying into > 2 classes?



Example: we want to learn to recognize handwritten digits → 10 classes. (This is the traditional NMIST example again).

Compute predicted probabilities y_pred = $y\_pred_i$ , i = 0,...,9 for all 10 digits.

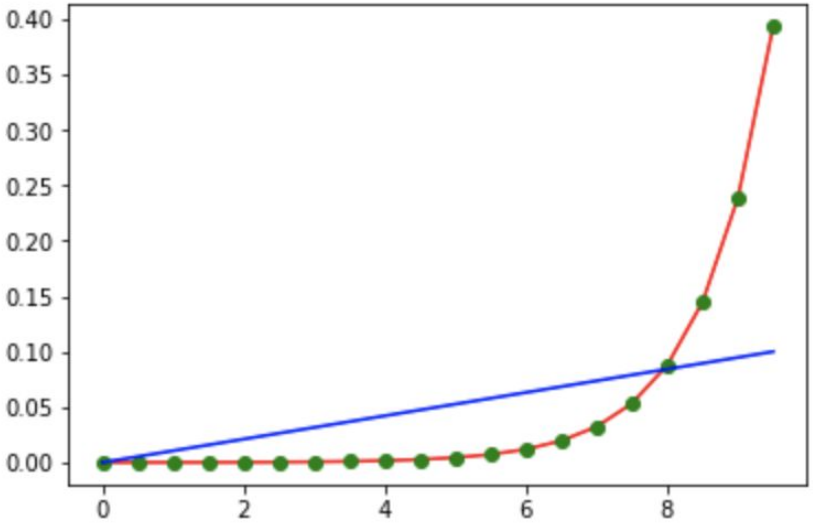
To do this, we use function **`softmax`** as the activation function in the neuron: y = g(z) = softmax(z). This computes a vector of probabilities that sum up to 1. The final result is obtained by selecting the index of the largest probability. (Check numpy **argmax** for this).

For multiclass classification we use **`categorical_crossentropy`** as the loss function.

# Softmax (normalized exponential function)

Given a vector $z = [z_1, z_2, \dots, z_n]$ , where $z_i \in R$, softmax maps it to vector

$s = [s_1, s_2, \dots, s_n]$, where $s_i \in R$ and $s_i = \exp(z_i) / \Sigma_{i=1\dots n}\exp(z_i)$




Softmax and linear scaling for values in range 0,10.
Green dots represent softmax shifted: softmax(x + d).

( → softmax is shift-invariant)

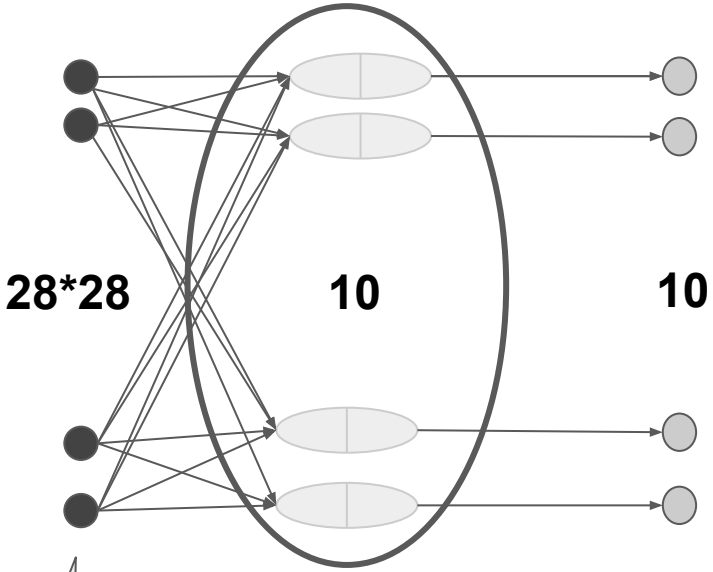
# Preprocessing: one-hot encoding

To prepare training samples into the needed format, we use one-hot encoding.
Encode label as a vector with 1 at index of label id, 0 otherwise:

```
[4 5 6 4 5 8 0 2 4 7]
```


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

# One-layer classification model for MNIST




To calculate the number of parameters, think about softmax with 10 category outputs as a layer of 10 neurons. Each has a weight for each of the 28*28 inputs + bias → 785 weights. So the whole model has 7850 weights.

input layer

softmax output layer

10-dimensional output vector y_pred = [ $y_1$ $y_2$ … $y_{10}$] where $y_i$ is the predicted probability that input x = i and $\sum y_i = 1$.

# One-layer classification model for MNIST

nmistlinearcategorical.ipynb

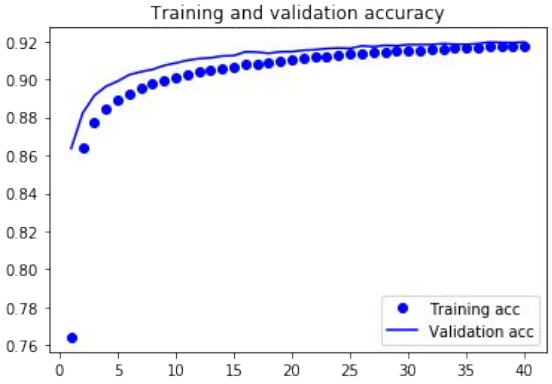
```
model = keras.models.Sequential()
model.add(keras.layers.Dense(10, input_shape=(28 * 28,), activation='softmax'))
```

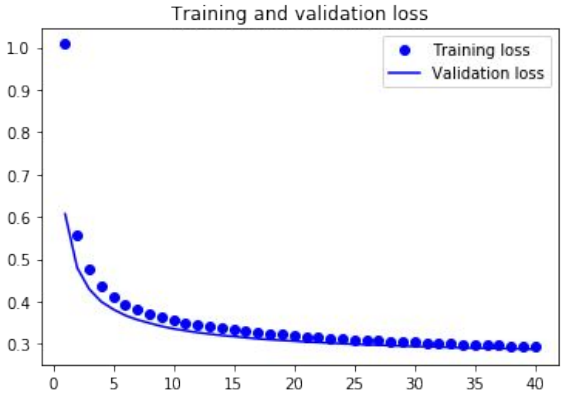
```
model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['acc'])

hist = model.fit(x_train, y_train, epochs=40, batch_size=64,
validation_data=(x_test,y_test))
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 10)                7850
=================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
_________________________________________________________________
```

# One-layer classification model for MNIST



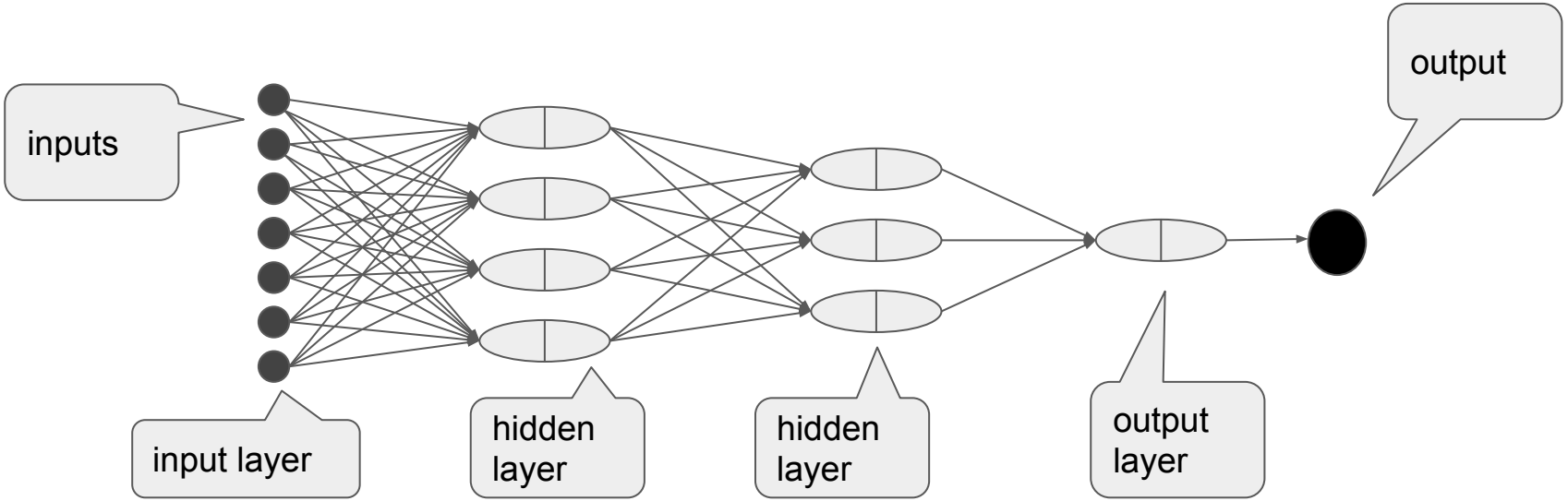




Why is validation accuracy higher than training accuracy?
https://keras.io/getting-started/faq/#why-is-the-training-loss-much-higher-than-the-testing-loss

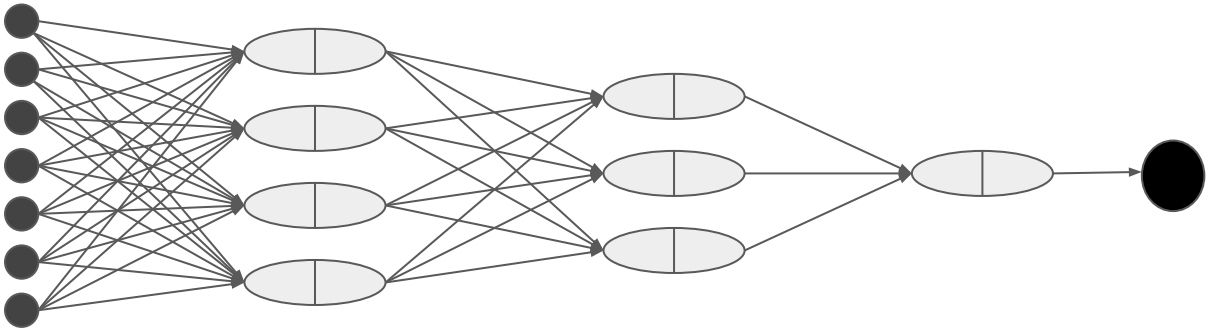
# How about using more neurons?

More neurons → more “intelligence”? We would get more weights in the network, so learning more complicated patterns would become possible.

How to arrange multiple neurons? Let’s take a look at the standard **sequential** (one path from input to output) network architecture.



# Dense network



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

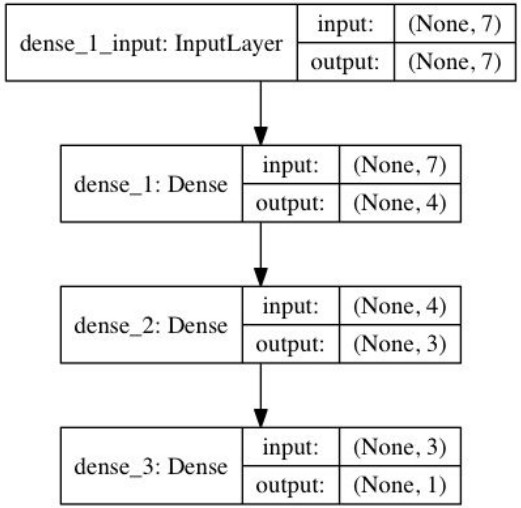
```
import keras

model = keras.models.Sequential()
model.add(keras.layers.Dense(4, input_shape=(7,)))
model.add(keras.layers.Dense(3))
model.add(keras.layers.Dense(1))

model.summary()

from keras.utils import plot_model
plot_model(model, show_shapes=True, to_file='model.png')
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 4)                 32
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 15
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 4
=================================================================
Total params: 51
Trainable params: 51
Non-trainable params: 0
_________________________________________________________________
```



# NMIST classification with a deeper network

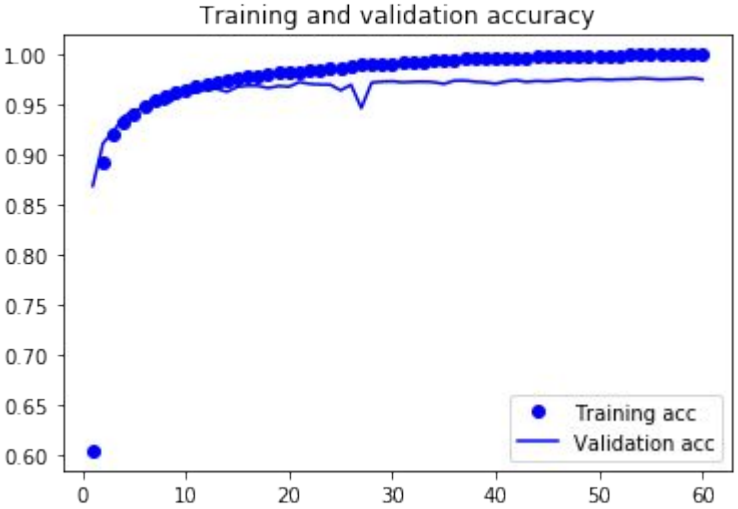
Let's see if we can improve NMIST classification with a larger network. We'll also use **`relu`** activation function **g(z) = max(0,z)** for the hidden layers.

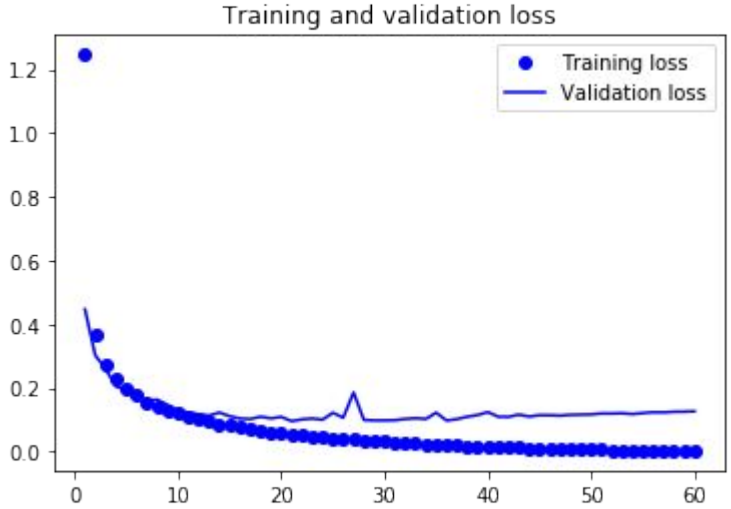
```
model = keras.models.Sequential()
model.add(keras.layers.Dense(100, input_shape=(28 * 28,), activation='relu'))
model.add(keras.layers.Dense(50, activation='relu'))
model.add(keras.layers.Dense(50, activation='relu'))
model.add(keras.layers.Dense(25, activation='relu'))
model.add(keras.layers.Dense(25, activation='relu'))
model.add(keras.layers.Dense(10, activation='softmax'))

model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
dense_5 (Dense)              (None, 100)               78500
_________________________________________________________________
dense_6 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_7 (Dense)              (None, 50)                2550
_________________________________________________________________
dense_8 (Dense)              (None, 25)                1275
_________________________________________________________________
dense_9 (Dense)              (None, 25)                650
_________________________________________________________________
dense_10 (Dense)             (None, 10)                260
```

# NMIST classification with a deep(er) network




… or is there a bit of overfitting?

# Some activation functions

**Identity**: g(z) = z. If only identity is used → completely linear model. Not used often.

**Sigmoid**: g(z) = 1 / (1 + $e^{-z}$). Smooth (and differentiable) with range 0...1. Used to be popular.

**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.

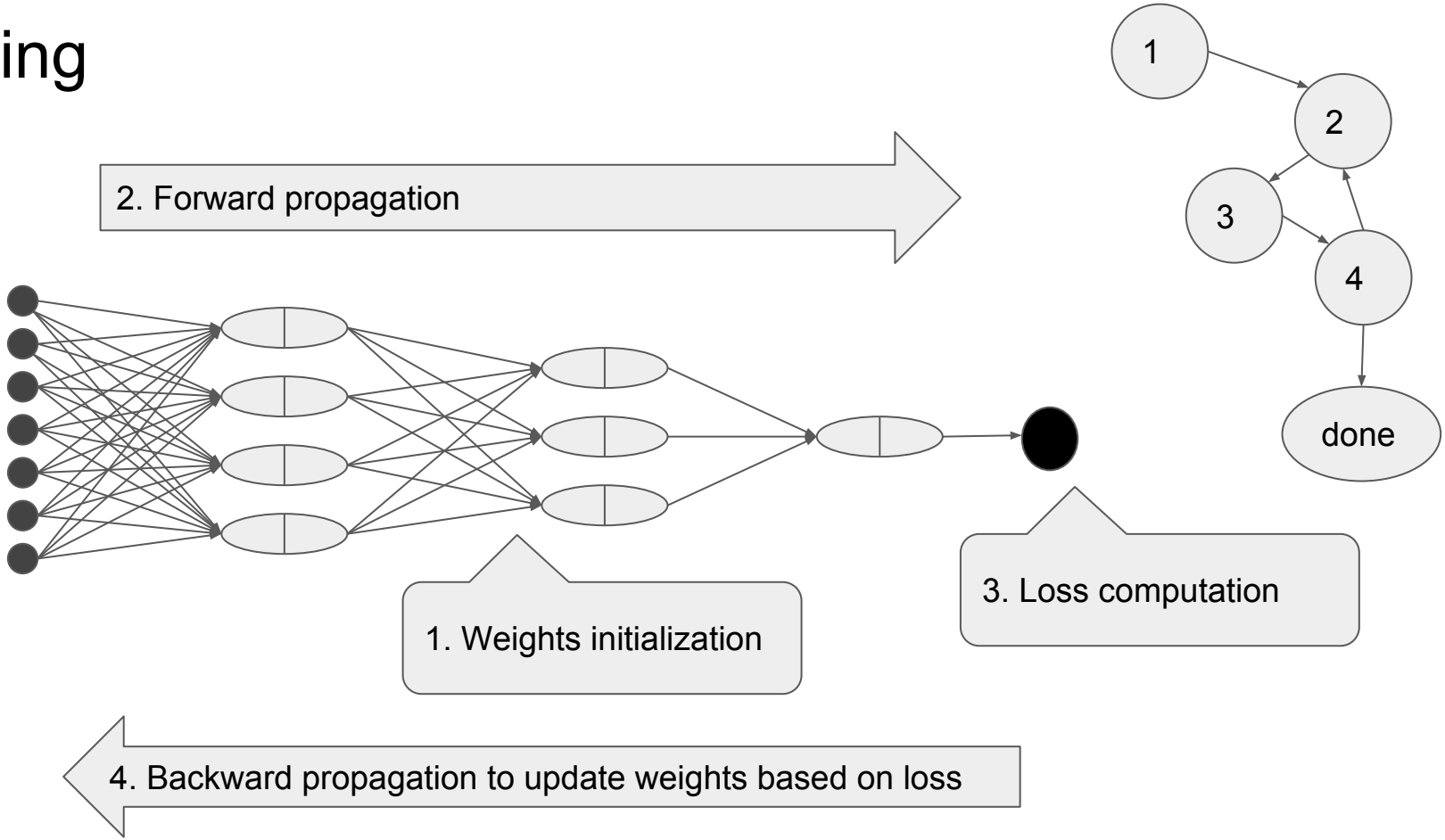
**Leaky ReLU**: g(z) = z for z >= 0, alpha * z for z < 0, where alpha is usually ~ 0,01

# How does training work?

Overview of training algorithm:

1. Initialize network (random, small values for weights, 0s for bias terms)
2. Make a prediction y_pred for an input (forward propagation)
3. Compute loss function value (by comparing y_pred with ground truth y)
4. Change weights in the network based on loss (backward propagation)
5. Go back to step 2 or stop

# Training


1
2
3
4
done
2. Forward propagation
1. Weights initialization
3. Loss computation
4. Backward propagation to update weights based on loss

# Batching

If we perform the forward propagation - loss value computation - backpropagation loop **for each sample**, training the network will become computationally expensive. If, on the other hand, we forward propagate **all training samples** and only then compute loss value and do backpropagation the learning is quite slow.

Strike a balance: divide training samples into mini-batches and forward propagate all samples in a batch, compute loss, and then backpropage.

# Some practicalities

Please note that in jupyter you can run cells separately (good thing).

What happens when you run the cell where you call `model.fit()` multiple times?

- The weights are not re-initialized on successive `.fit()` calls; you are continuing training from the learned weights

Easy way: run model definition & compilation cell to reset.

Be aware of this type of side effects elsewhere, too.

# Exercise: train and use NMIST

Take a look at either the linear softmax or the deep network NMIST example and train it (with or without modifications). Use it for predicting output for at least 20 images of digits you have drawn yourself. Hints:

- Preprocess the image to lighten the background but make sure black is quite black
- Use `keras.preprocessing.image load_img()` for reading an image file (with a given resolution)
- Take a look at pixel values, plot the individual images and see if you need to manipulate them in any way before trying to make predictions

# Exercise: Fashion-MNIST dataset

The traditional MNIST dataset is considered to be too easy to be used as a case for evaluating model performance. A new dataset with more complex images has been made available and is getting popular. See https://github.com/zalandoresearch/fashion-mnist for details about the data set.

Download fashion-mnist and train a **sequential** and **dense** model for it. Make sure to study the the data, plot some images from it etc. to get familiar with the data set. Compare the accuracy you achieve vs. the accuracy achieved in standard mnist example. Analyze model behaviour by plotting loss/accuracy diagrams.

Note: deep doesn’t mean 100’s of layers. Start with small network.

# Reading list for exam 29.1

session02.pdf

Chollet: 3.1, 3.2, 3.4, 3.5, 3.6 (skip 3.6.4) (details of how text data is processed can be skipped, we'll be back to that later)