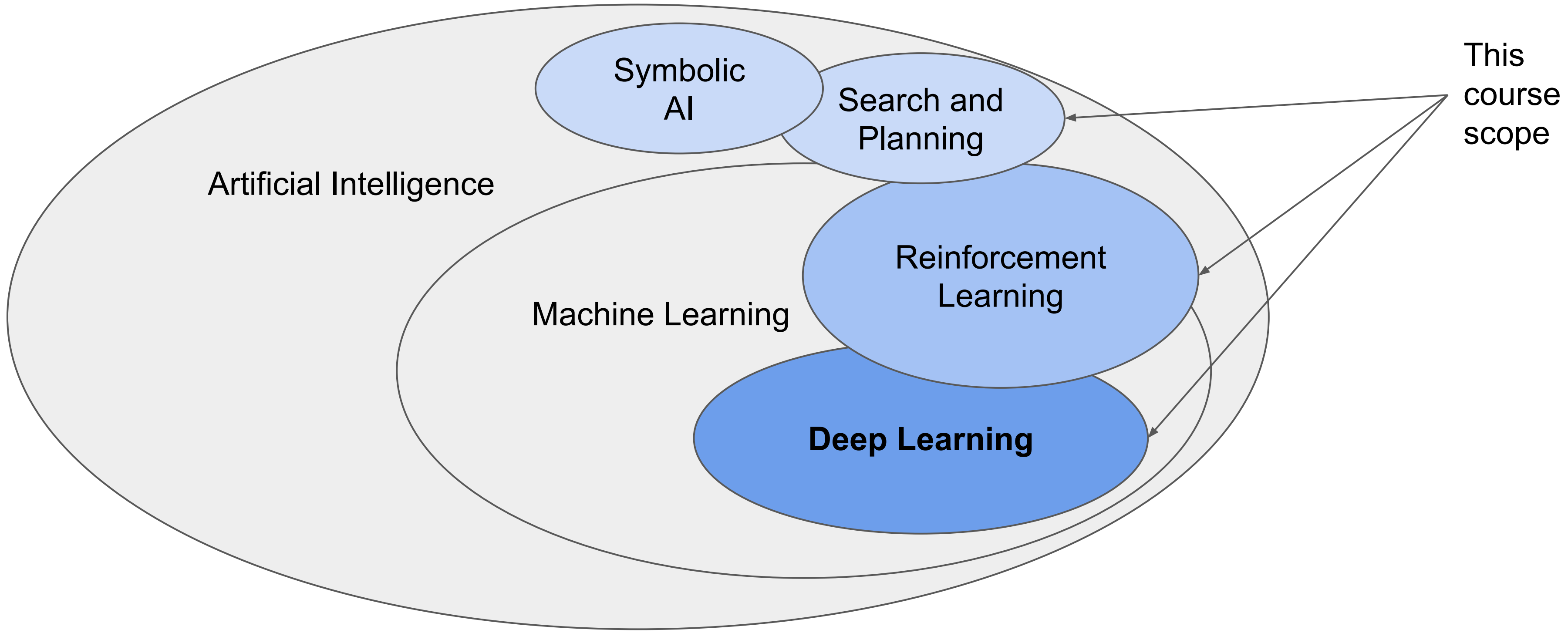
IT00DP82-3007 Artificial Intelligence and Machine Learning

peter.hjort@metropolia.fi

Machine Learning in Artificial Intelligence (AI) landscape



Supervised learning - learn association between input and output

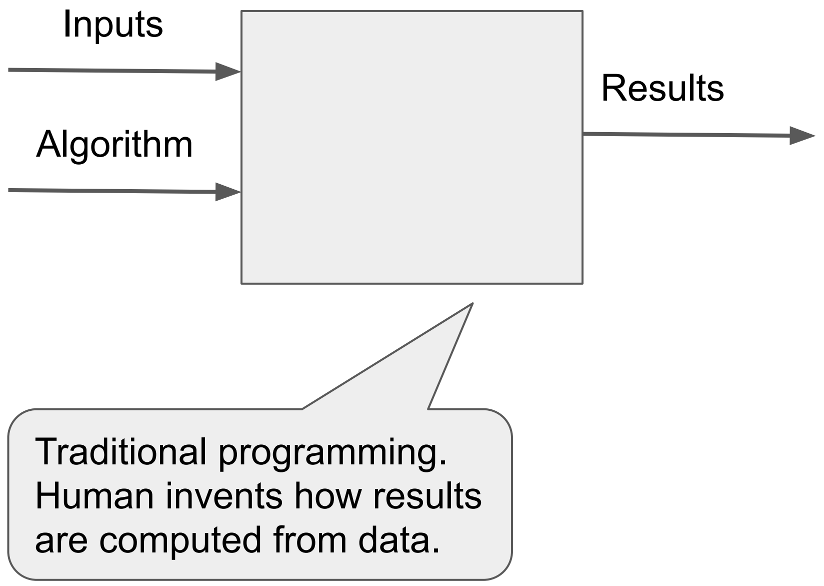
Starting point is that some data, examples of real associations, is available.

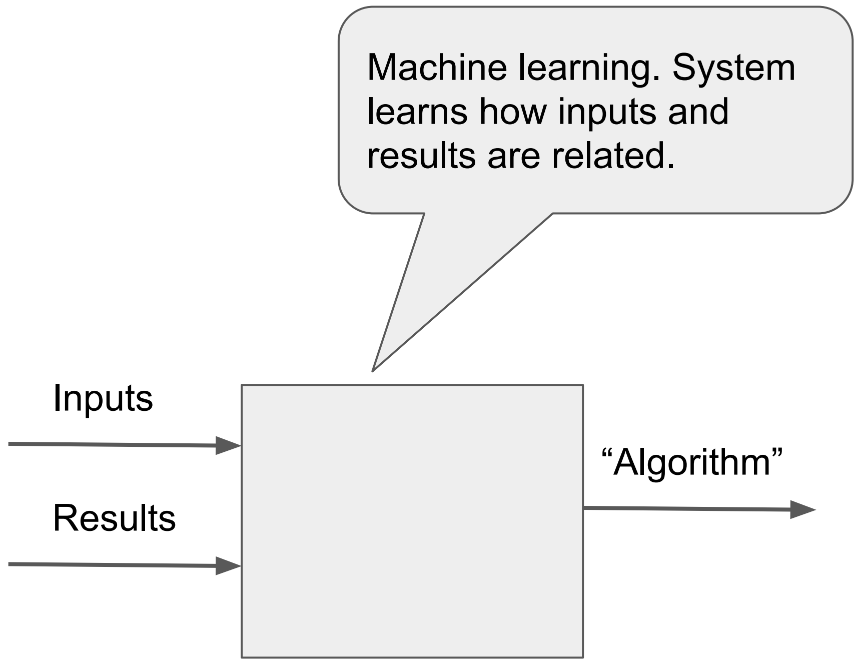
For example, there might be observations of the weight and height of a set of humans. We could (in this case) take either one to be the dependent variable, and the other one the independent variable.

- **Dependent variable**: the output of interest. Value of dependent variable is assumed to depend on the values of independent variable(s)
- **Independent variable(s)**: the input(s).

Target is to learn to predict the value of dependent variable, when values of independent variables are given.

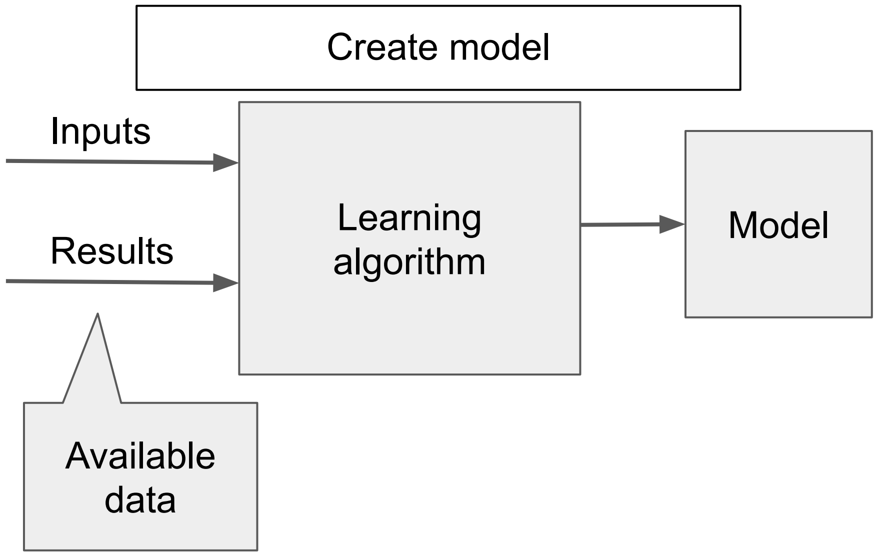
Programming vs model creation, supervised learning case

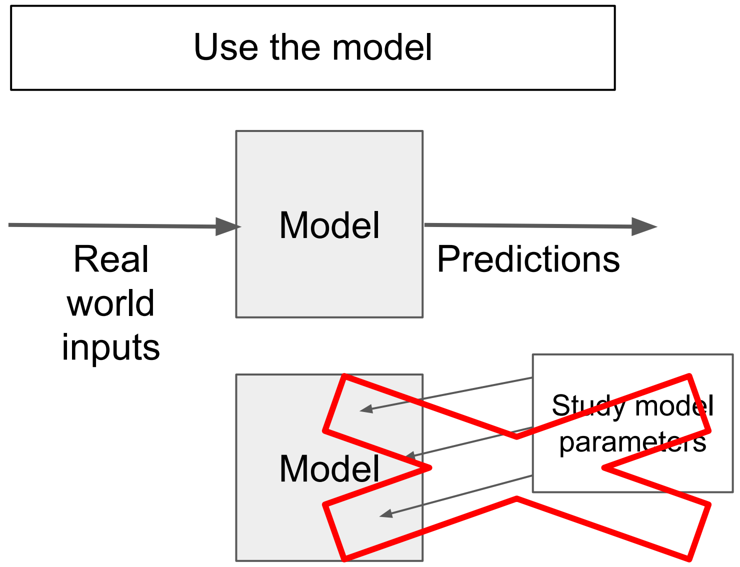

Inputs
Algorithm
Results
Traditional programming. Human invents how results are computed from data.


Machine learning. System learns how inputs and results are related.
Inputs
Results
“Algorithm”

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 $y = w_0 + w_1x$ (linear regression), or very complicated like a deep net with 152 layers and 130 million parameters. (Note: diagrams below are specific to supervised learning).

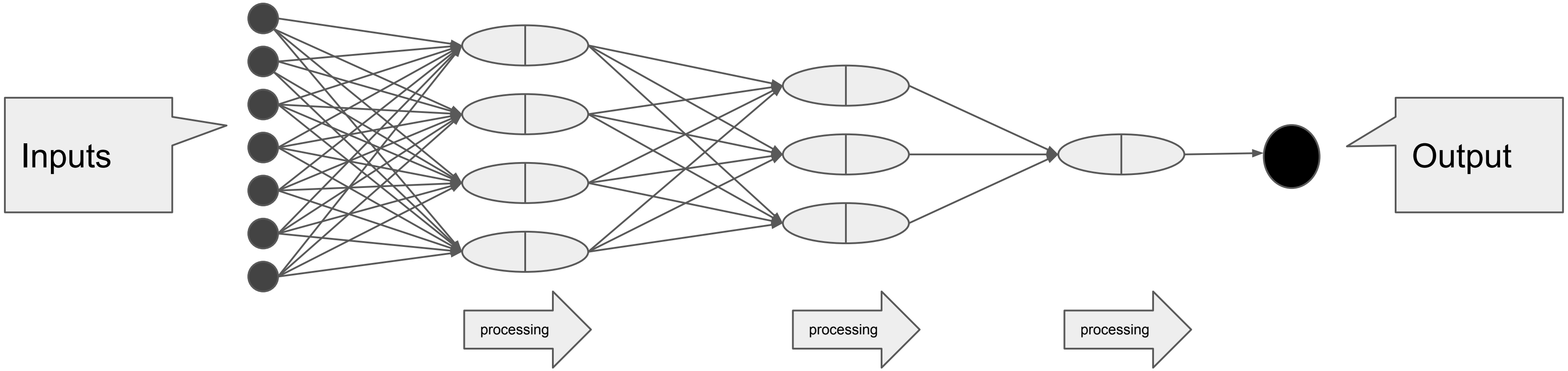




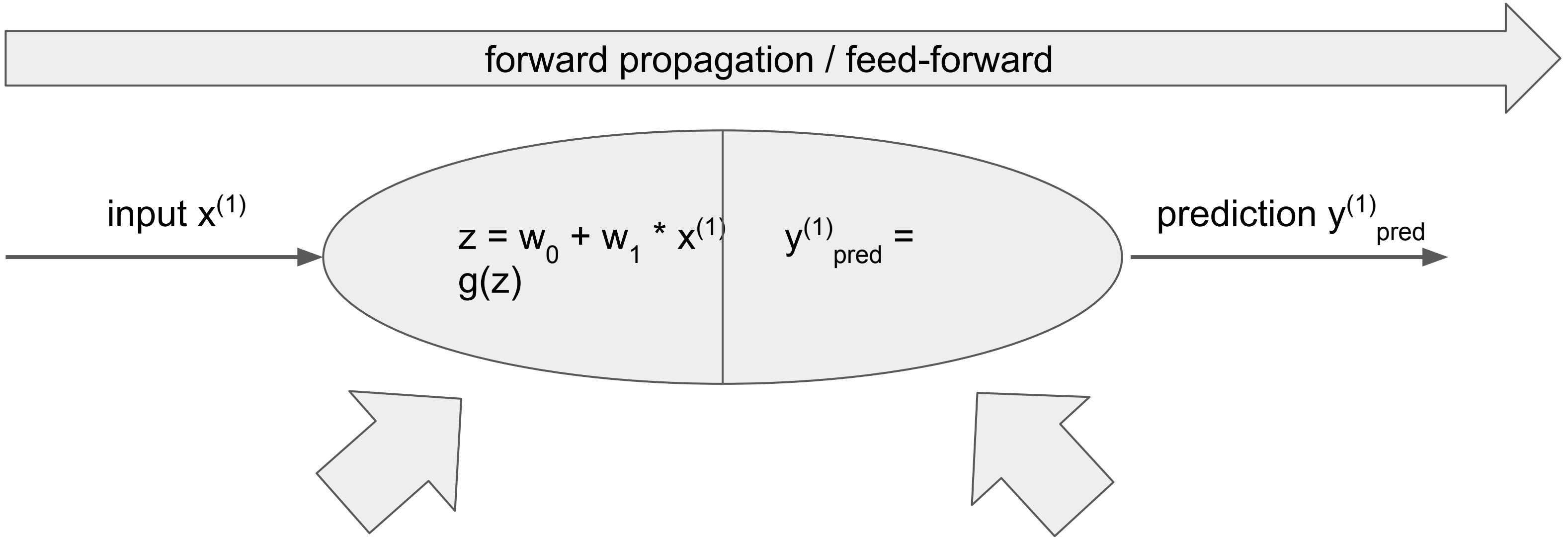
Model in deep learning

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 set of layers through which data flows when predictions are made.



A single artificial neuron with single input feature



forward propagation / feed-forward
input $x^{(1)}$
$z = w_0 + w_1 * x^{(1)}$
$y^{(1)}_{pred}$ = g(z)
prediction $y^{(1)}_{pred}$

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

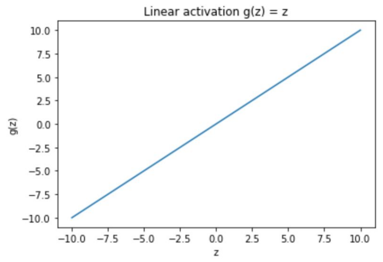
Identity: g(z) = z. If only identity is used in all network layers → completely linear model, no benefit over traditional linear model. Not an often used activation function.

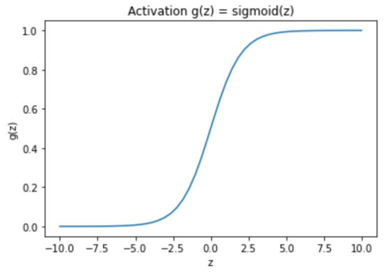
Sigmoid: g(z) = 1 / (1 + e^{-z}). Smooth (and differentiable) with range 0...1. Useful when probabilities are needed as output.

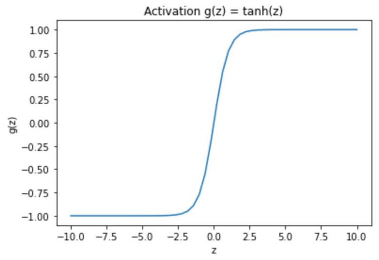
Tanh: g(z) = (1 - e^{-z}) / (1 + e^{-z}). Like sigmoid, but with range -1...1.

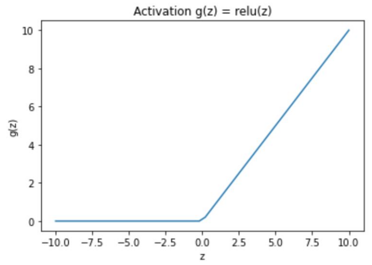
Rectified linear unit (relu): g(z) = max(0, z). Not differentiable at 0, but in practise does not matter. Most popular activation function, especially in convolutional networks (more on those later).

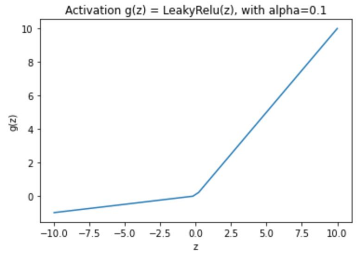
Leaky ReLU: g(z) = z for z >= 0, alpha * z for z < 0, where alpha is usually ~ 0,01











Some deep network frameworks

There are many frameworks that allow one to define a deep network, train it, and use it for making predictions. The frameworks differ in the their abstraction level, runtime support, support for distribution etc.

High-level framework

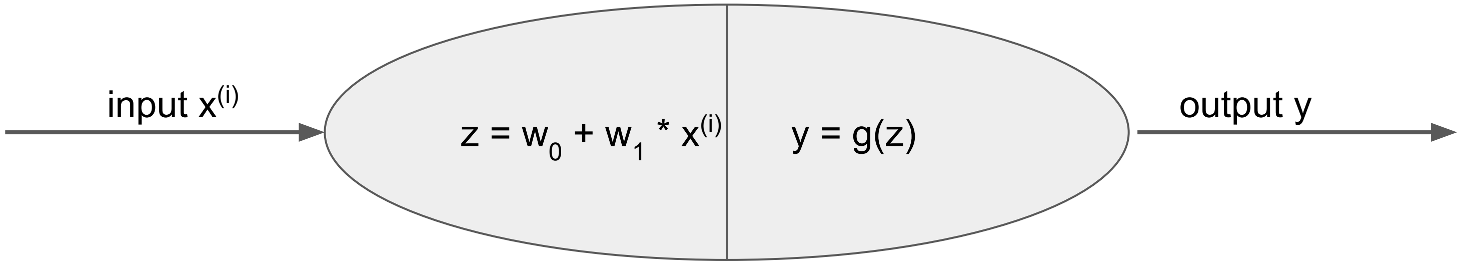
- Keras - allows one to define a network using simple high-level syntax, train the network, and make predictions. Latest version of Keras supports three lower-layer frameworks that contain needed functionality for autodifferentiation, CPU/GPU/TPU support etc. See https://keras.io/keras_3/

Lower-level frameworks. All these can be used to define the model, but one will need to do that on a level (much) lower than Keras

- Pytorch - used quite a bit in academic world. See https://pytorch.org/
- Tensorflow - Google framework whose components can be found in Android devices, too. See https://www.tensorflow.org/
- JAX - latest lower-level framework. Might be replacing Tensorflow. Supports distribution well. https://jax.readthedocs.io/en/latest/index.html

In this course we will use Keras. Keras is automatically available in colab environment.

Single neuron network in Keras



Build a **model** that will have a sequence of **layers**.

```
model = keras.models.Sequential()

model.add(keras.layers.Input(shape=(1,)))

model.add(keras.layers.Dense(1))
```

Define the input layer, ie. the shape of input data.

Add a layer to the model.

Layer is fully connected (Dense), output dimension is 1 (there is one neuron in the layer), and input dimension is 1. No activation function defined: by default identity function $g(z) = z$ is used.

The model needs to be trained, or fitted with data

We now have in place the structure of the ANN, in this case a network with one neuron only.

To find best values for the parameters in the model (in this case w_0 and w_1), we need:

- Cost, or loss, function - for a linear regression case like this Mean Squared Error (mse) familiar from traditional linear regression is fine.
- Way to find the values of w_0 and w_1 that minimise the loss function - the optimisation algorithm. Gradient descent, again familiar from sklearn linear regression implementation, is fine but there are other that may be better suited to specific situations.

Specify loss and optimization

```
model = keras.models.Sequential()

model.add(keras.layers.Input(shape=(1,)))

model.add(keras.layers.Dense(1))

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

Specify optimizer; the way in which suitable model weights are searched for. `sgd` stands for **stochastic gradient descent** - a basic optimizer.

Specify function for computing the loss ie. what will be optimized. `mse` stands for mean squared error (that has been used before in linear regression). Loss function choice is task-specific - different loss function for regression vs. classification etc.

Train, or fit, the model

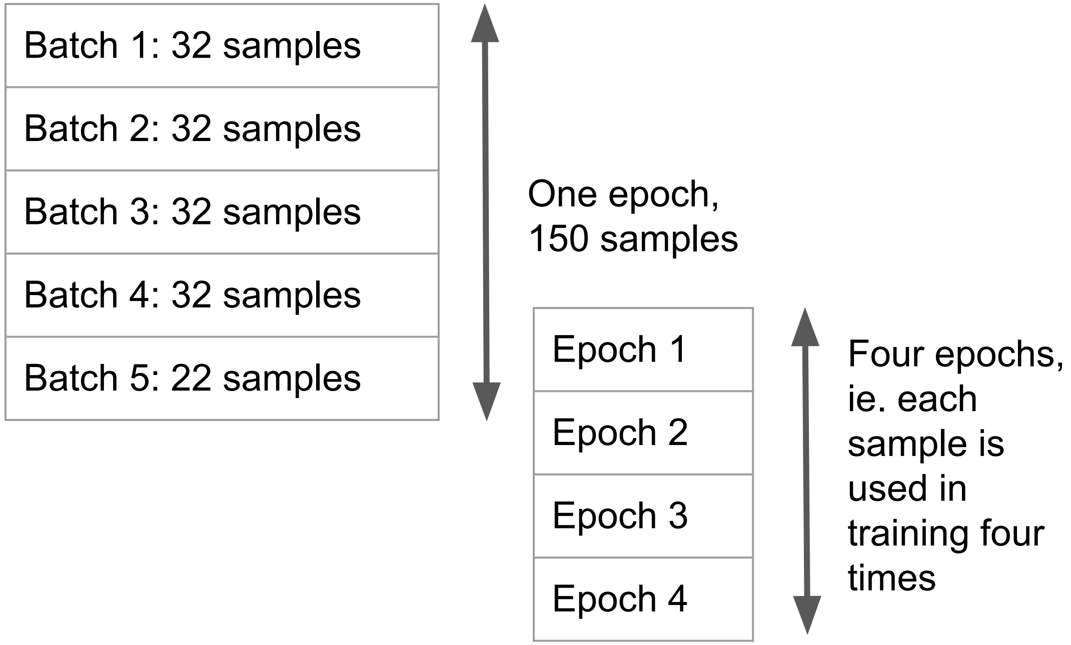
```
model = keras.models.Sequential()

model.add(keras.layers.Input(shape=(1,)))

model.add(keras.layers.Dense(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. `fit()` function returns an object that can be used for plotting training and validation values etc.

Go through the whole training material 10 times in batches of 16 (x,y) pairs.

Epochs and batches

Let’s assume the training material consists of 150 samples. If we use batch size of 32 samples, we’ll get 5 batches:

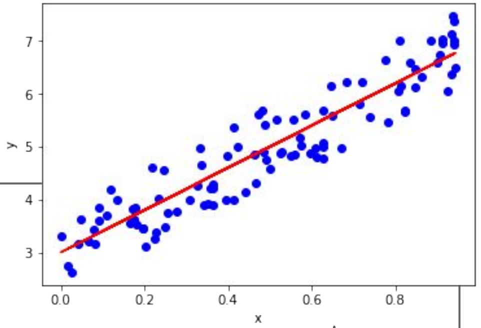


How to choose batch and epoch sizes?

- Larger batch size usually helps to fight overfitting (more later), and improved training performance.
- Batch size should be small enough to fit in GPU/TPU fast memory
- Use as many epochs as needed for the training accuracy to reach acceptable level but observe validation accuracy, too.

Use the model

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

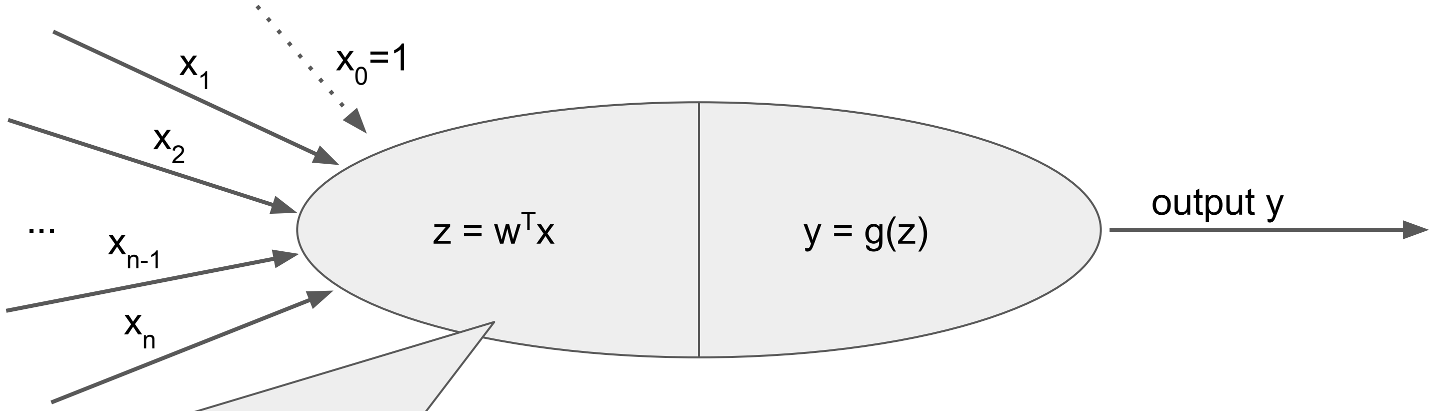


After the model is trained:

- Make a prediction with `predict()` method
- 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 for 40 epochs we get w_0 = 3.012 and w_1 = 3.986. (Values used for creating the data were 3 and 4).

Neuron with vector input

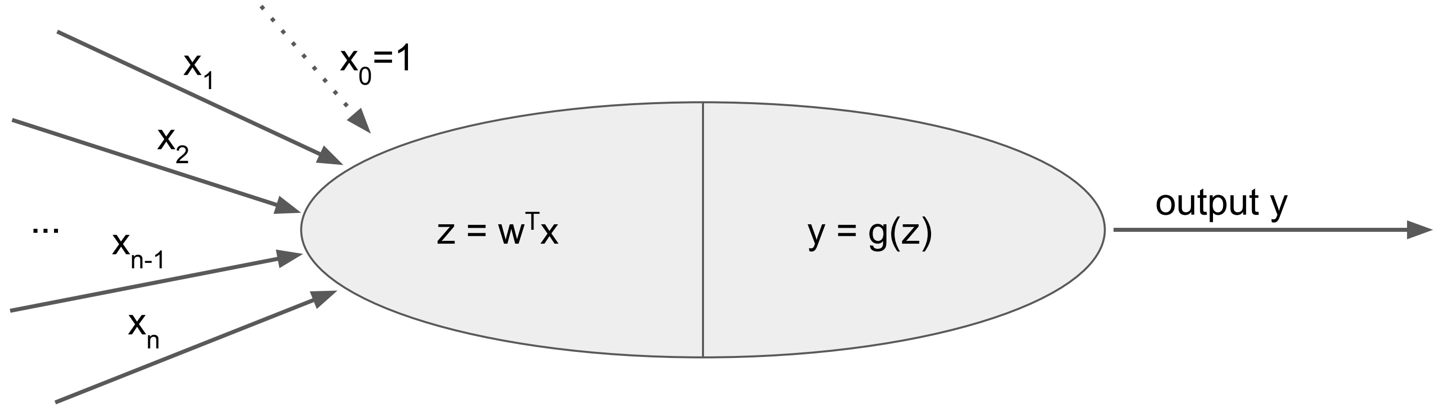


z is computed by multiplying x_i by corresponding weight w_i and summing over all n+1 multiplications: $z = \sum_{i=0,\dots,n} w_i * x_i$.

(With w and x interpreted as vectors, this is their dot, or inner, product marked as $w \cdot x$, or w^Tx, 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 $z = w^Tx + b$, where b is bias, is used.)

Neuron with vector input



For example: if n=3, no bias term ($w_0=0$), input is x=(1,2,0), and weights w=(2,1,3) we have

- $z = w_1x_1+w_2x_2+w_3x_3 = 2*1+1*2+3*0 = 4$

If g(z) = sigmoid(z), then y≈0.98. For g(z)=relu(z), y=4.

Linear regression case: Boston housing dataset

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

```
from keras.datasets import boston_housing

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

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

x_train[78] → [1.71710000e-01 2.50000000e+01 5.13000000e+000.00000000e+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 standardisation

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 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 inverse of standard deviation is a popular strategy.

With sklearn `StandardScaler` this is quite straightforward (and can be done with numpy, too - see examples notebook).

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(x_train)

x_train_ss = scaler.transform(x_train)
x_test_ss = scaler.transform(x_test)
```

Define and train the model

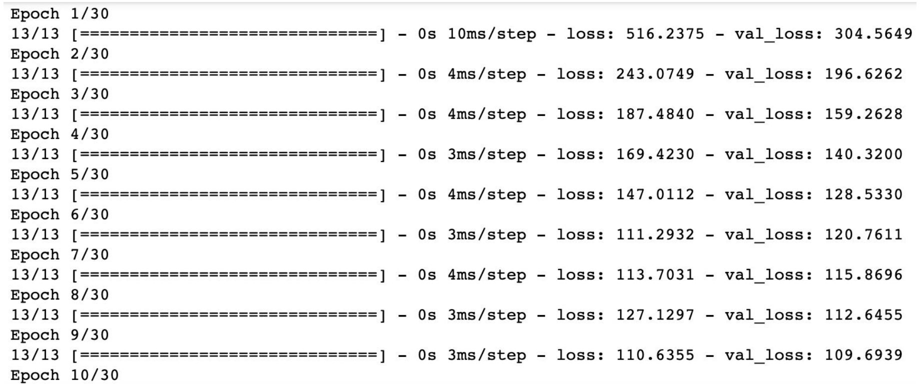
Single neuron model, outputs one number. This time input is 13-dimensional. Relu activation function - outputs only non-negative values.

```
model1 = keras.Sequential()
model1.add(layers.Input(shape=(13,))
model1.add(layers.Dense(1, activation='relu'))

model1.compile(loss='mse', optimizer='sgd')

n_epochs = 30
hist1 = model1.fit(x_train_ss, y_train,
                   epochs=n_epochs, batch_size = 32,
                   validation_data=(x_test_ss, y_test))
```

Specify optimizer and loss function.

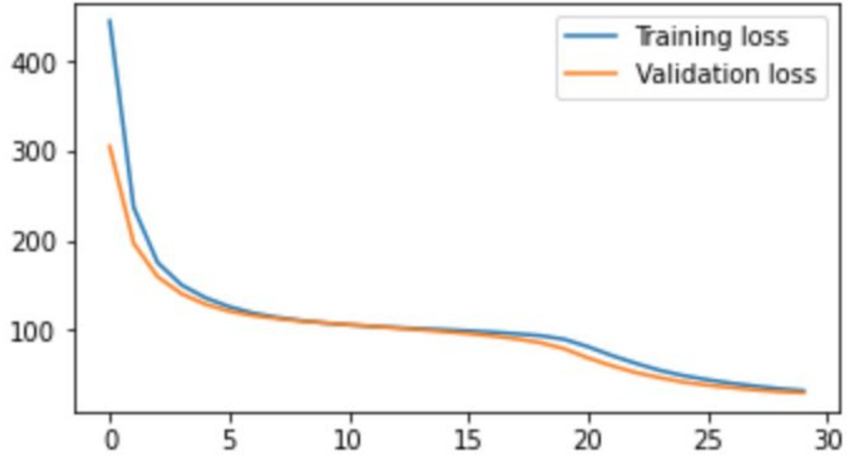
Training data consists of inputs `x_train_ss` and corresponding outputs `y_train`. At the end of each epoch **a validation step is run with validation data** `x_test_ss`, `y_test`. Loss function values for each epoch are stored in variable `hist1`.

Training



```
Epoch 1/30
13/13 [==============================] - 0s 10ms/step - loss: 516.2375 - val_loss: 304.5649
Epoch 2/30
13/13 [==============================] - 0s 4ms/step - loss: 243.0749 - val_loss: 196.6262
Epoch 3/30
13/13 [==============================] - 0s 4ms/step - loss: 187.4840 - val_loss: 159.2628
Epoch 4/30
13/13 [==============================] - 0s 3ms/step - loss: 169.4230 - val_loss: 140.3200
Epoch 5/30
13/13 [==============================] - 0s 4ms/step - loss: 147.0112 - val_loss: 128.5330
Epoch 6/30
13/13 [==============================] - 0s 3ms/step - loss: 111.2932 - val_loss: 120.7611
Epoch 7/30
13/13 [==============================] - 0s 4ms/step - loss: 113.7031 - val_loss: 115.8696
Epoch 8/30
13/13 [==============================] - 0s 3ms/step - loss: 127.1297 - val_loss: 112.6455
Epoch 9/30
13/13 [==============================] - 0s 3ms/step - loss: 110.6355 - val_loss: 109.6939
Epoch 10/30
```

To study the progress per epoch, it is useful to plot the training / validation losses (and possibly other metrics).



Unless `verbose=0` parameter is given to `fit()`, there will be output detailing the average training loss per epoch and validation loss computed after epoch.

Inspect model

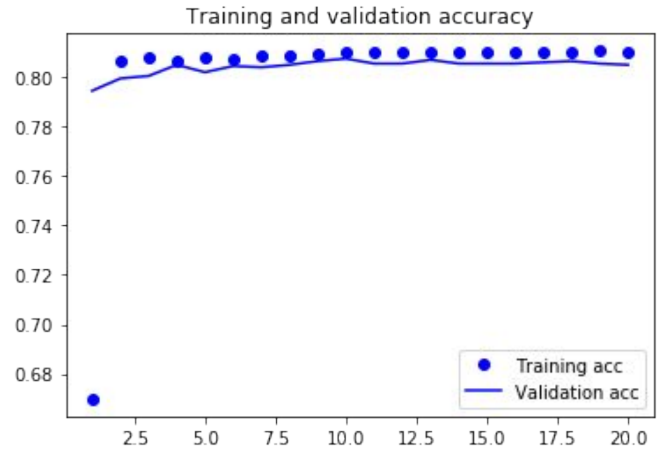
Model structure and parameter counts can by studied: `model1.summary()`

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

After training (the `.fit()` call) model for example model weights are available with: `model1.get_weights()`
This is useful only in simple cases.

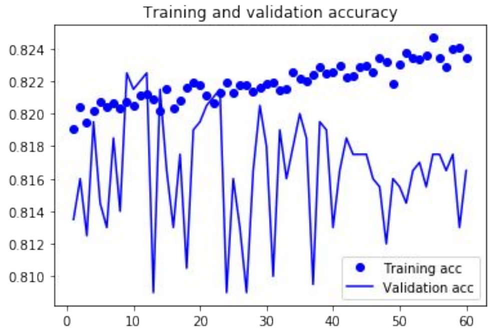
```
[array([[ 0.05344158],
       [ 0.63540411],
       [ 0.37158501],
       [ 0.39288247],
       [-0.03653026],
       [-0.19312069],
       [-0.01441616],
       [-0.58739501],
       [ 0.28077316],
       [-0.06558847],
       [ 0.60122478],
       [-0.32361585],
       [ 0.17224699]], dtype=float32),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

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.

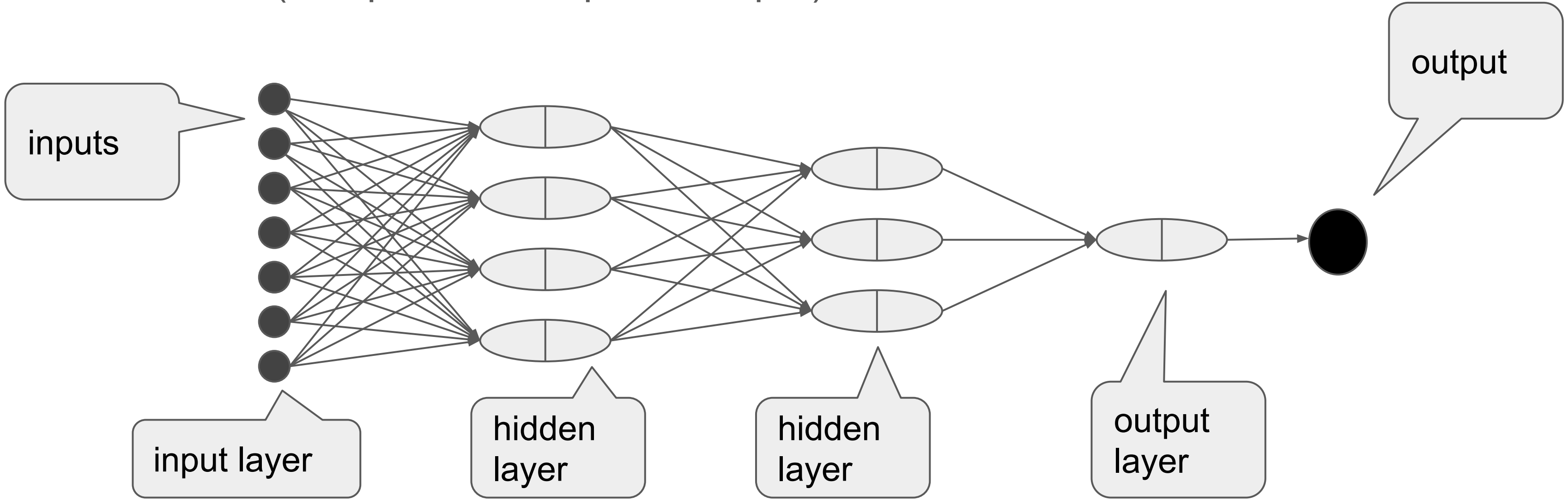


How about using more neurons?

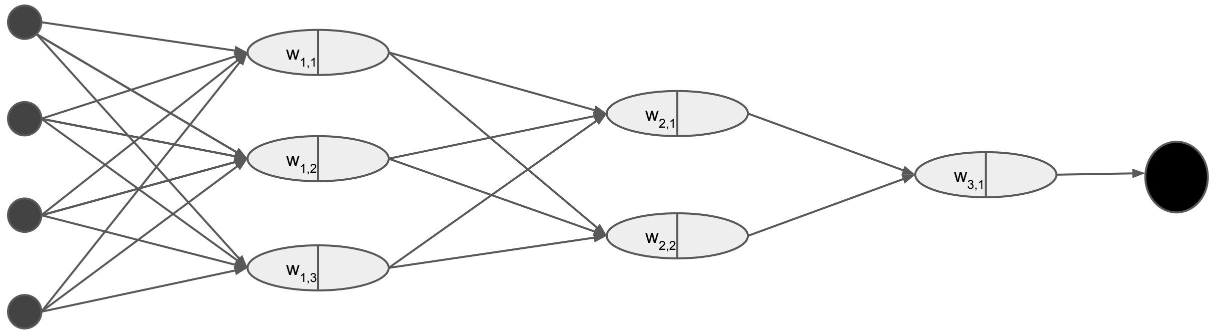
More neurons → better predictions because of more “intelligence”? In general, yes.

The more weights in the network, the more complicated patterns the network would be able to learn.

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



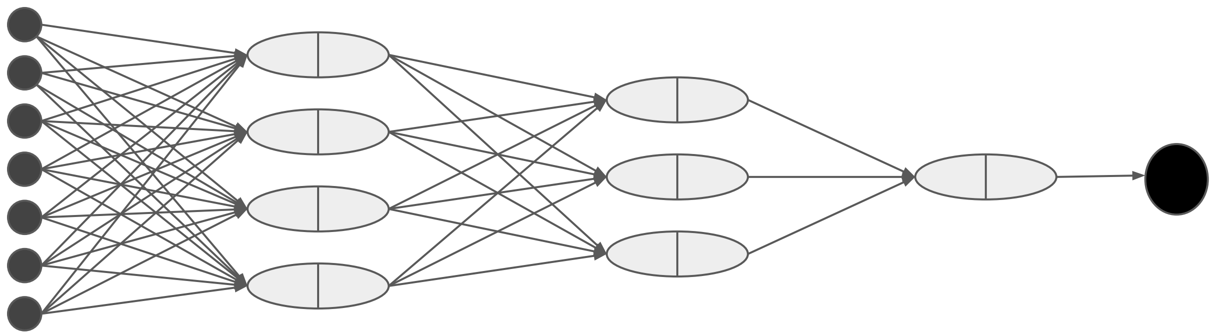
Dense network forward pass



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

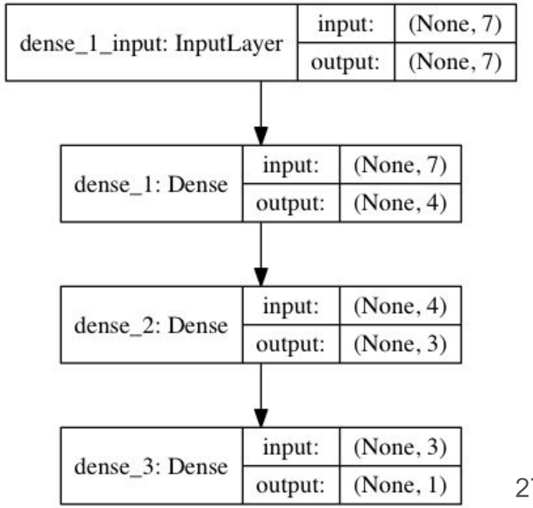


Dense, or completely connected, network; each output is connected to all next layer inputs.

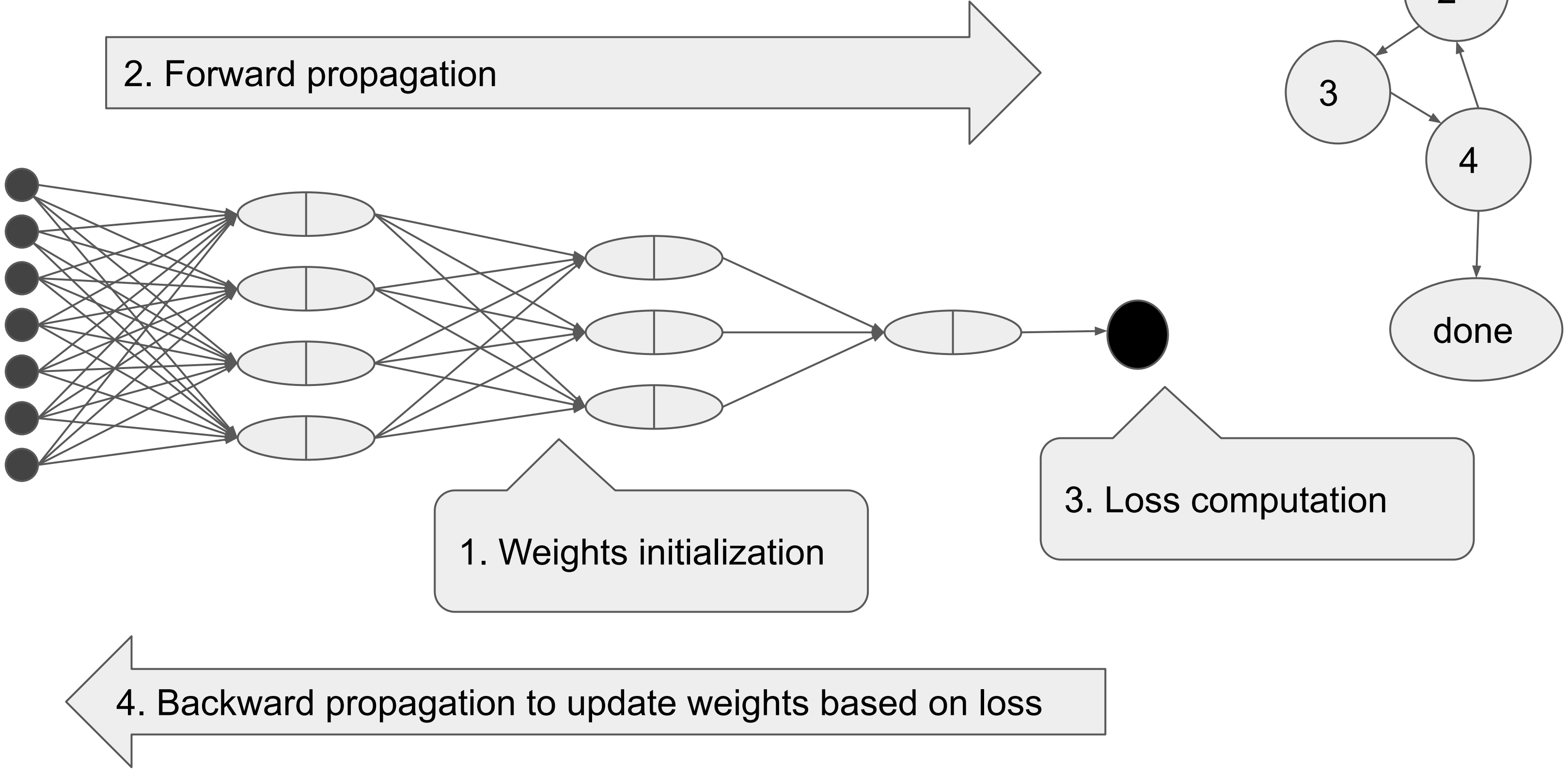
- # of weights in 1st hidden layer: (input dimension (7) + 1) * # of neurons (4) = 32
- # of weights in 2nd hidden layer: (previous layer # of neurons (4) + 1) * # of neurons (3) = 15
- # of weights in output layer: (previous layer # of neurons (3) + 1) * # of neurons (1) = 4

```
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
_________________________________________________________________
```



Training overview


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

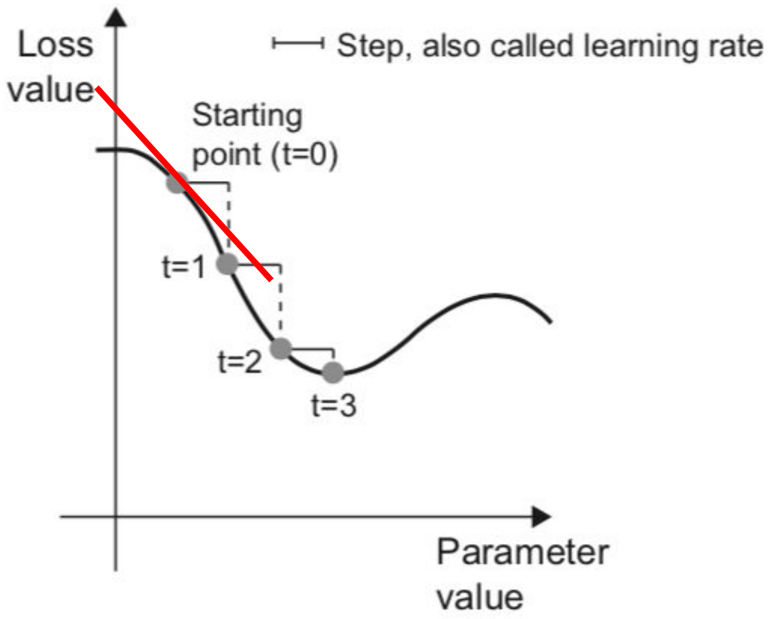
Backpropagation and gradient

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/)

Gradient descent in 1-dimensional case



At parameter point w, compute derivative for loss function wrt weight w: dL/dw = slope of tangent for L.

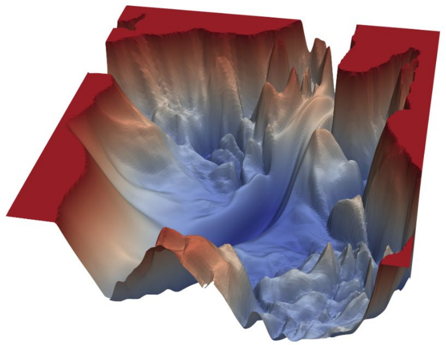
Next value for $w_{t+1} = w_t - \alpha_t \cdot dL/dw$ (α_t is the decaying learning rate at time step t)

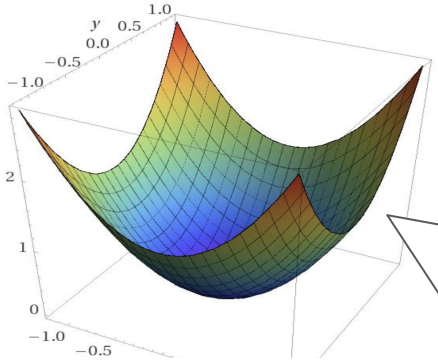
Slope of tangent < 0 → move towards higher w

Slope of tangent > 0 → move towards lower w

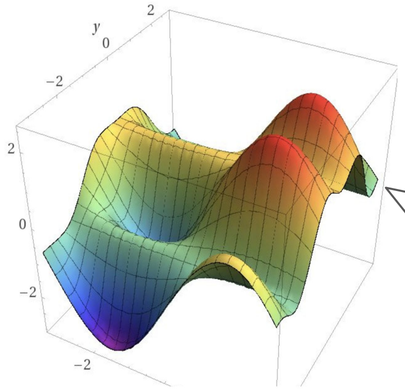
α_t decay schedule is either power series or exponential (see Murphy section 8.4.3)

Cost function shape and optimisation





Convex cost function, generated for example by the mean sum of squares function J we used. Finding minimum is easier.



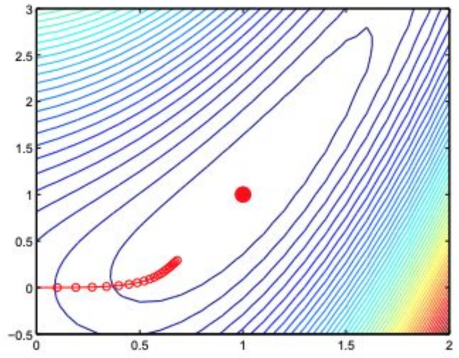
Cost function surface in general case. Finding the minimum is hard

For large networks with large number of weights, the dimension of the space over which optimisation is done is high-dimensional (a convnet might have 130 million parameters…).

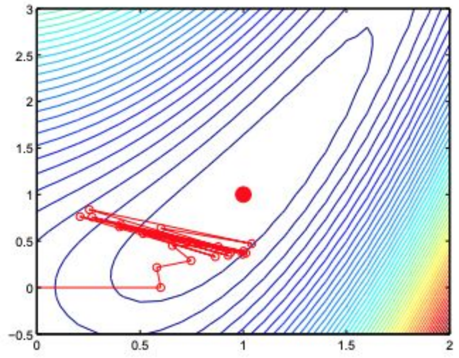
The shape of the cost function is not convex, but very irregular. Finding **THE** minima is not possible but in practice it seems that good enough local minima can be found.

From https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/ , illustration top right is from https://www.cs.umd.edu/~tomg/projects/landscapes/

2-dimensional case and effect of learning rate



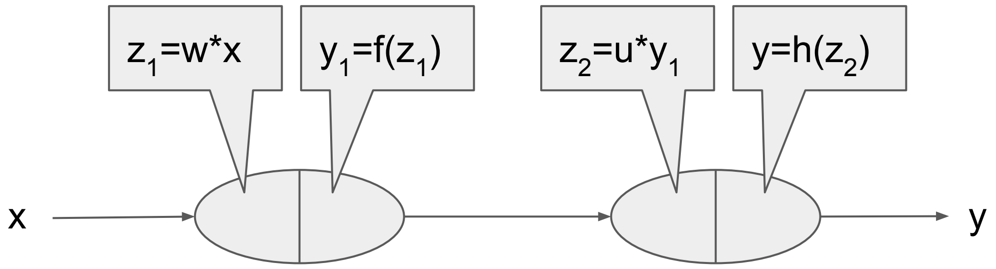
(a)



(b)

Gradient descent with constant learning rate, left small, right large. With small learning rate will find minimum but slowly, large learning rate can result in oscillation and no convergence. Image from Murphy, p. 268.

Backpropagation for scalar case



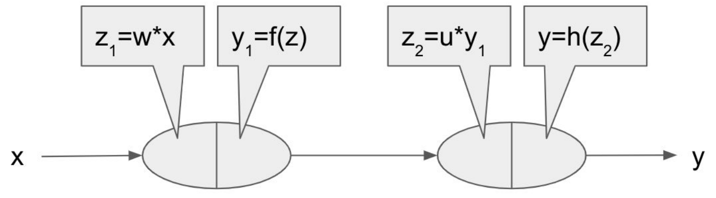
Forward propagation: compute value for function

$y = h(u * y_1) = h(u * f(w * x))$

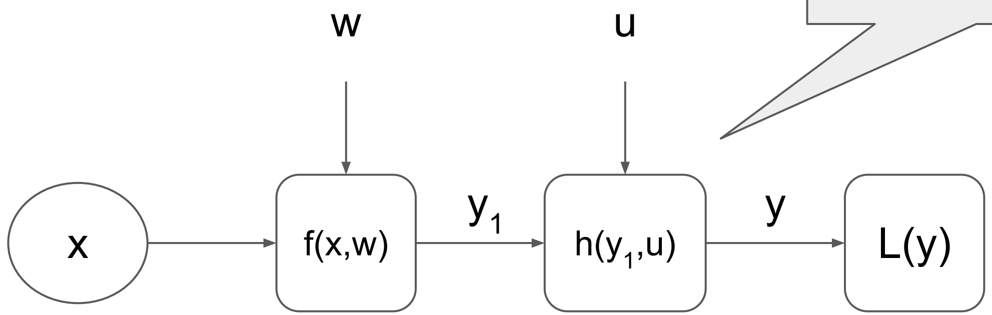
where w and u are weights for layers 1 and 2, (no bias term) and f and h activation functions for layers 1 and 2

Loss function L = L(y)
(x and known true value y_{true} are considered fixed from cost function optimisation point of view).

Computational graph for scalar case

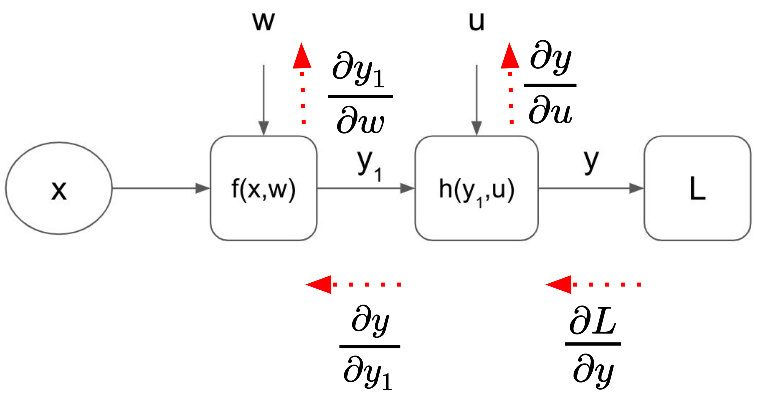


$$y = h(u * y_1) = h(u * f(w * x))$$



In both *f()* and *h()* inputs and weights are treated as parameters.

Backpropagation for scalar case


w
u
$\frac{\partial y_1}{\partial w}$
$\frac{\partial y}{\partial u}$
x
f(x,w)
y_1
h(y_1,u)
y
L
$\frac{\partial y}{\partial y_1}$
$\frac{\partial L}{\partial y}$

Derivative of L wrt last layer parameter u:

$$\frac{\partial L}{\partial u} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial u}$$

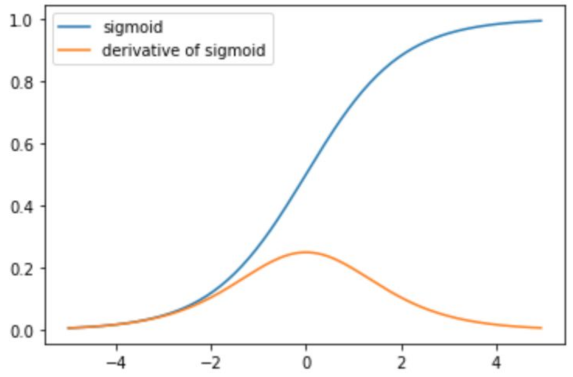
Derivative of L wrt first layer parameter w:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial y_1}\frac{\partial y_1}{\partial w} = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial w}$$

These derivatives are computed by Tensorflow/Pytorch/JAX numerically when training a Keras model. There are different strategies for this, see for example Murphy chapter 13.3

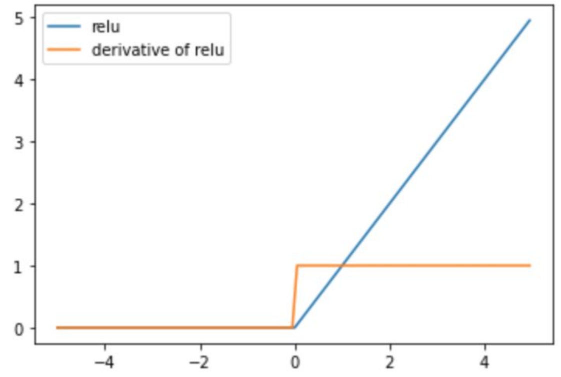
Vanishing gradients

In some networks, especially deep ones, the gradient signal approaches zero when it progresses towards the input layers. If this happens, there is no signal to direct the changes to weights - **vanishing gradients** problem. Sigmoid activation function is prone to this →



One way to avoid this is to use relu activation →

Relu, on the other hand, has zero value and derivative with negative values which in case of large negative weights leads to **dead relu** problem. Leaky relu is one way to rectify this.



Batching and training

Batching - do a forward pass for all samples in a batch and then compute the mean loss function value. Propagate the “error signal” in backwards direction based on the mean loss, only once per batch. Why does this make sense?

- In forward propagation batching makes it possible to compute the outputs for each of the neurons in a layer in parallel - the inputs are constant and the weights will not change until backpropagation is done at the end of the forward pass.
- Only one backpropagation pass is needed per batch.
- There are other factors related to overfitting and learning rate selection, more later.

How to detect overfitting - cross-validation

To diagnose the overfitting problem, a technique called cross-validation is often used: the labeled data is divided into *disjoint* but *statistically similar* sets:

- **Training set**: this data is used for finding the parameter values that minimize the cost function.
- **Validation set**: this set is used for fine-tuning hyperparameters (more later) and in general comparing different models to select the one performing well on previously unseen data.
- **Test set**: the data not used in the development of the model that is used for assessing the final performance of the system.

Need for separate validation / test data sets comes from the fact that the use of validation set can “leak” information towards the model - a separate set is needed for final verification. Note: terminology is a bit confusing here: often validation set is called test set. In frameworks such as sklearn only training and validation (which is called test) sets are considered.

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

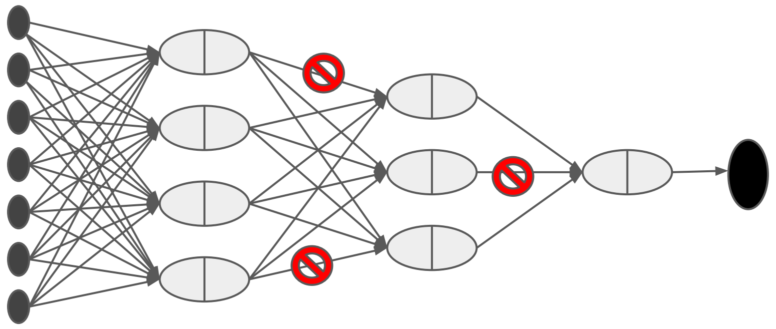
Drop out connections with a given probability → create “holes” in the network.

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.



Regularization and dropout in Keras

```
model.add(keras.layers.Dense(50, activation='relu',
kernel_regularizer=keras.regularizers.l2(0.01)))
```

L_2 regularizer with $\lambda = 0.01$.

```
model.add(keras.layers.Dense(50,
activation='relu'))
model.add(keras.layers.Dropout(0.3))
model.add(keras.layers.Dense(25,
activation='relu'))
```

Drop 3/10 of input units, ie. cut out some connections.

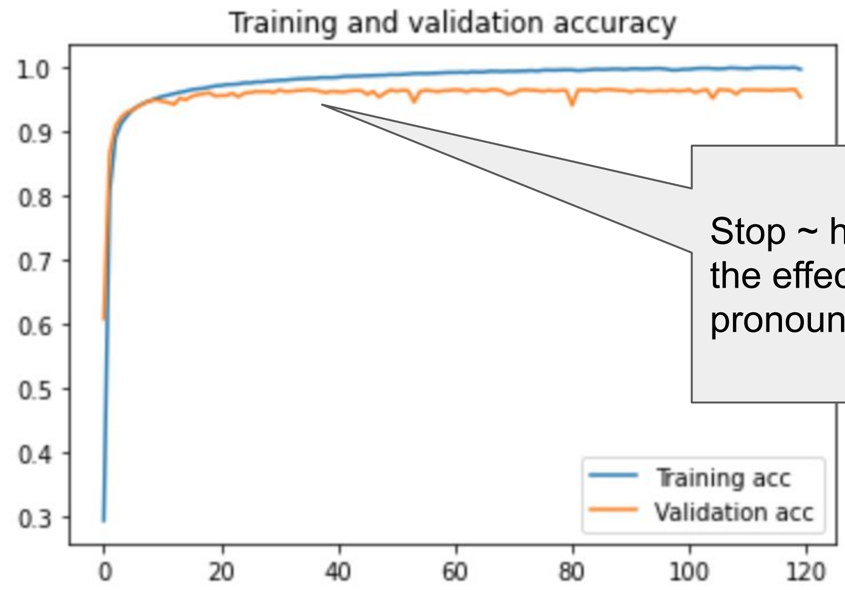
(Managing overfitting - white noise)

Adding some Gaussian (normally distributed random data) to input values provides one additional way of regularisation.

```
x = x + np.random.normal(0, scale=noise_std,
size=x.shape)
```

Note this might not be very good with images but for scalar values a valid way.

Managing overfitting - early stopping

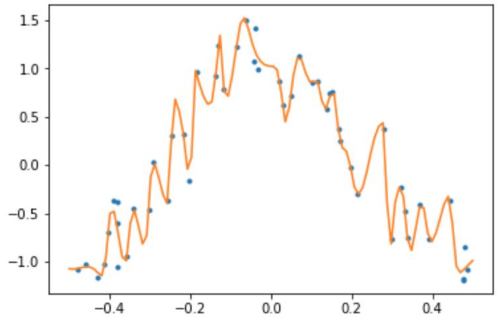


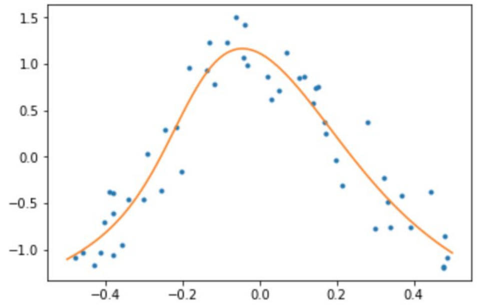
Stop ~ here (often the effect is more pronounced).

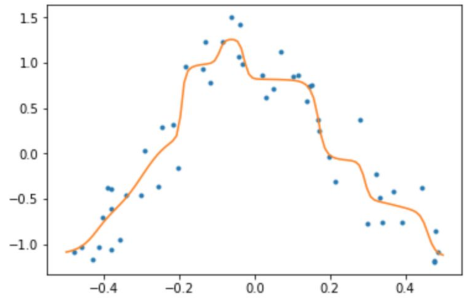
To implement early stopping:

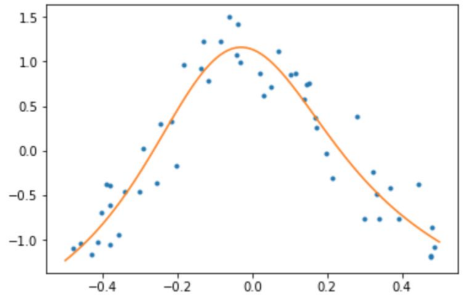
- Train for large number of epochs, look at the training/validation accuracy diagram and pick a point (# of epochs) where validation accuracy doesn't improve
- Train as long as validation accuracy improves over some number of epochs (remember, in Keras subsequent .fit() calls are cumulative).

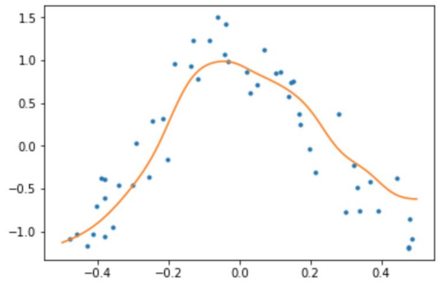
Managing overfitting in function approximation

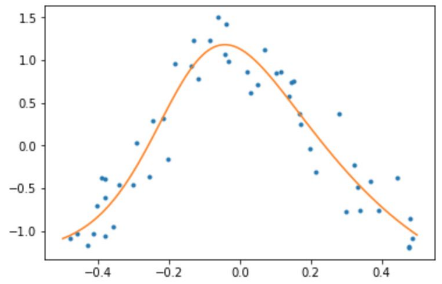

No regularisation


Weight decay (L2 regularization)

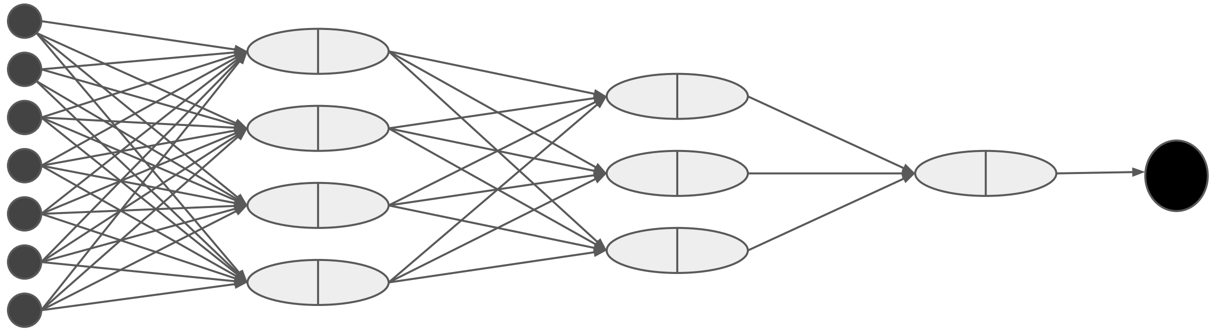

Dropout layer


Early stopping


Random noise


Reduced model
complexity

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.

Batch normalization in Keras

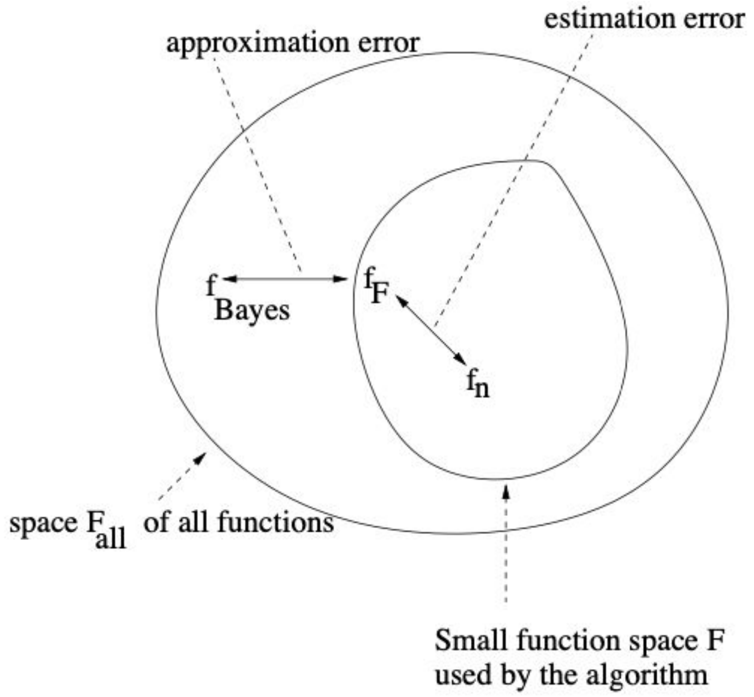
```
model.add(Dense(120, activation='relu'))
model.add(BatchNormalization())
```

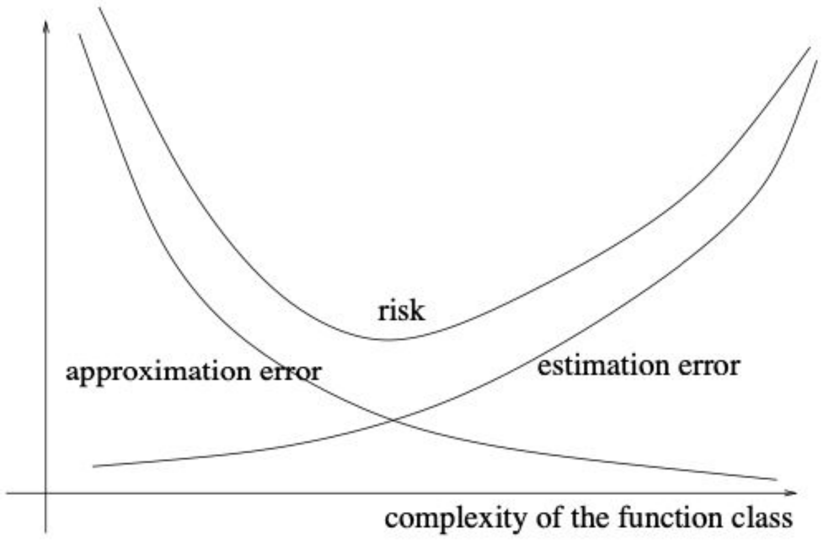
So, normalization takes place after applying nonlinearity. To apply normalization before nonlinearity:

```
model.add(Dense(120))
model.add(BatchNormalization())
model.add(Activation('relu'))
```

Original paper on batch normalization (https://arxiv.org/pdf/1502.03167.pdf) uses the latter approach.

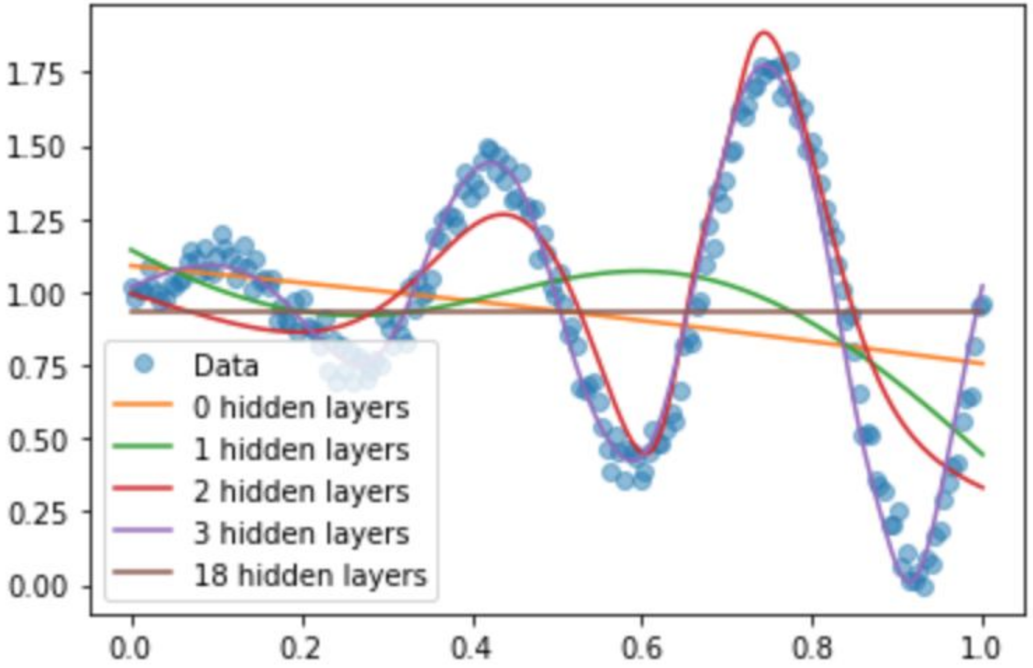
Estimation and approximation errors


approximation error
estimation error
f_{Bayes}
f_F
f_n
space F_{all} of all functions
Small function space F used by the algorithm



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

Estimation and approximation errors example



Small amount of layers (0-2, assuming same width) in the network → smaller function space F:

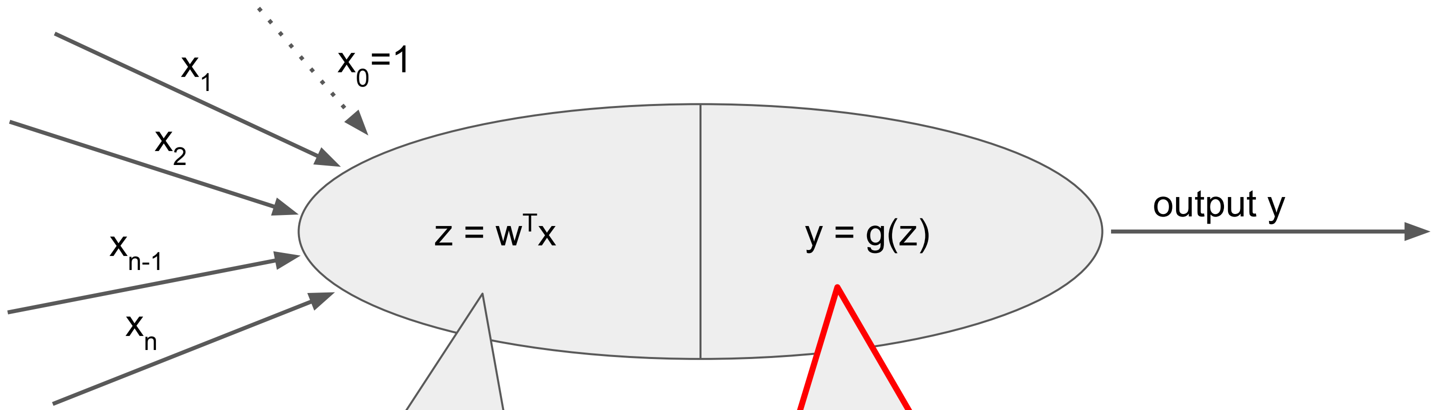
- Estimation error is (probably) small, but approximation error larger

18 layers → large function space F

- Estimation error is large (optimisation algorithm fails to find minimum), but approximation error in theory small

(18 layers case is a bit exaggerated)

How to predict values $0 \leq y \leq 1$ to classify?

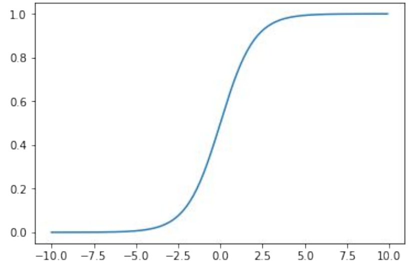


The linear part of AN:

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

(* = normal multiplication)

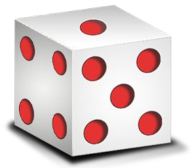
Use a non-linear **activation function** $g(z) = \text{sigmoid}(z) = 1 / (1 + e^{-z})$. It has range from 0 to 1 → can be interpreted as probability.



Binary classification - predictions

Sample	Ground truth	Predictions 1	Cost 1	Predictions 2	Cost 2
	1.0	0.084		0.781	
	1.0	0.556	1.27	0.898	0.21
	0.0	0.531		0.231	

How about classifying into > 2 classes?



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

For each sample, compute predicted probabilities y_pred_i , i = 0,...,9 for all 10 digits.

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

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

Digit image classification

0
1
2
3
4
5
6
7
8
9

Ground truth label values.

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:

```
[3 2 7 9 0 1]
```

```
keras.utils.to_categorical(a)
```

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

Values for each of the positions can be interpreted as probabilities - for the training samples we know the probabilities for sure so there is ‘1’ at the position of the right label and ‘0’ elsewhere.

Digit image classification

0	[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
1	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
2	[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
3	[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
4	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
5	[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
6	[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
7	[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
8	[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
9	[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]

One-hot encoded ground truth label values.

Predicting probabilities for all categories - softmax

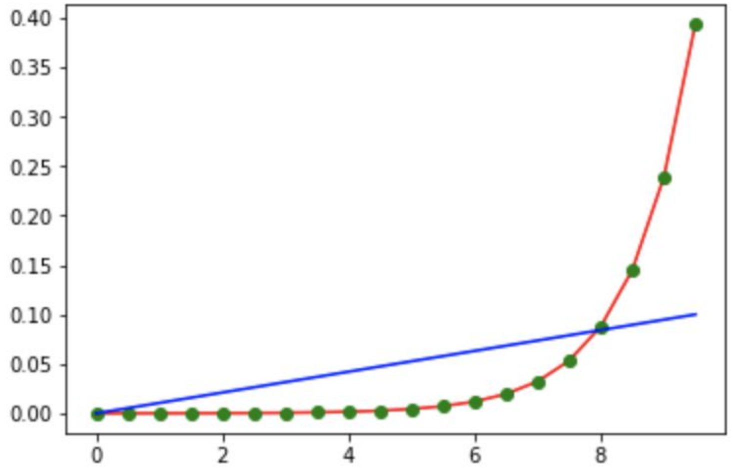
Softmax activation function takes a vector of values (results from the linear part of neuron) and scales them into values in range 0...1 that sum up to one. These values can be treated as the probabilities for each of the vector positions. Definition of softmax is:

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)

Why one-hot encoding

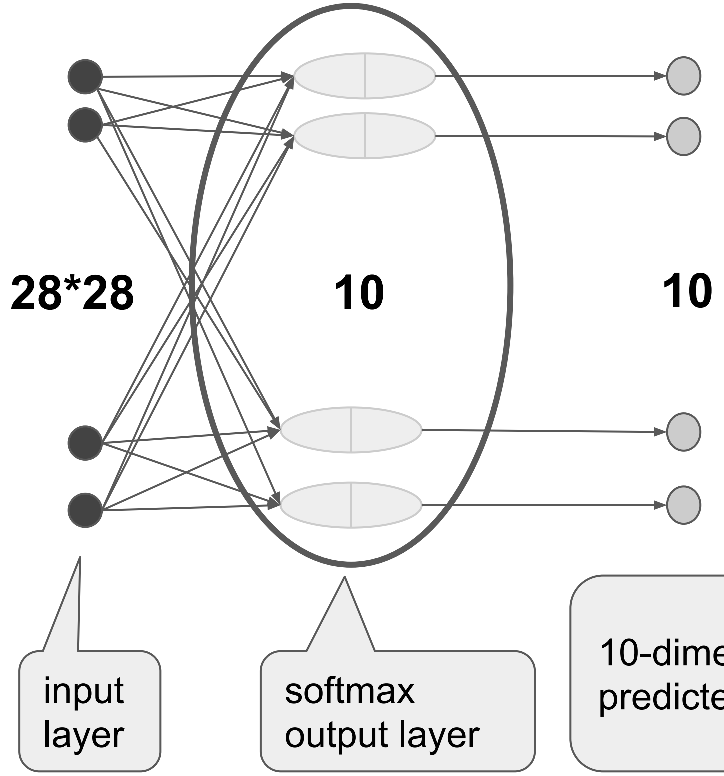
0.005	↔	0
0.015	↔	0
0.18	↔	1
0.1	↔	0
0.3	↔	0
0.05	↔	0
0.05	↔	0
0.1	↔	0
0.15	↔	0
0.05	↔	0

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.

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 10 * 785 = 7850 weights.

10-dimensional output vector y_pred = [y_0 y_1 … y_9] where y_i is the predicted probability that input x = i and $\sum y_i = 1$.

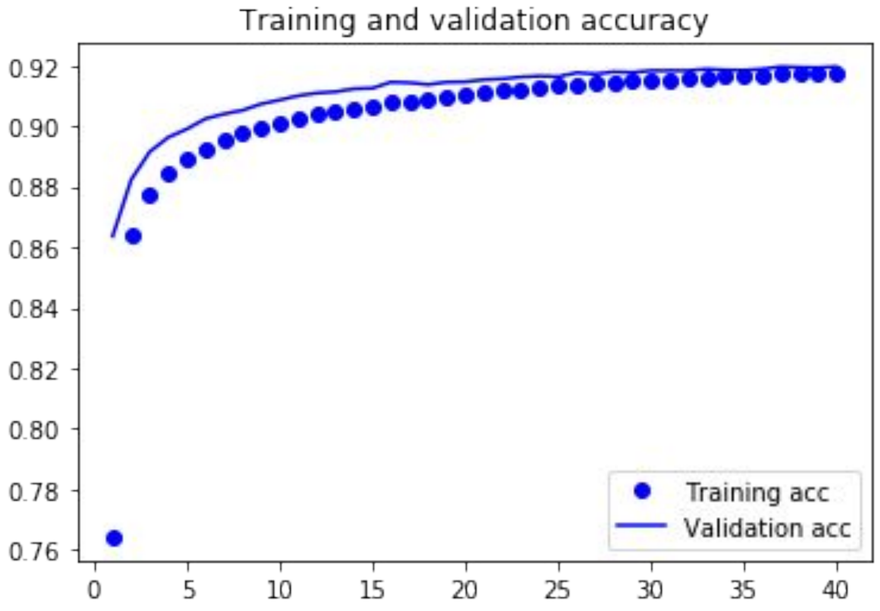
One-layer classification model for MNIST

```
model = keras.models.Sequential()
model.add(keras.layers.Input(shape=(28 * 28)))
model.add(keras.layers.Dense(10, 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

From probabilities to predictions

0.005
0.015
0.18
0.1
0.3
0.05
0.05
0.1
0.15
0.05

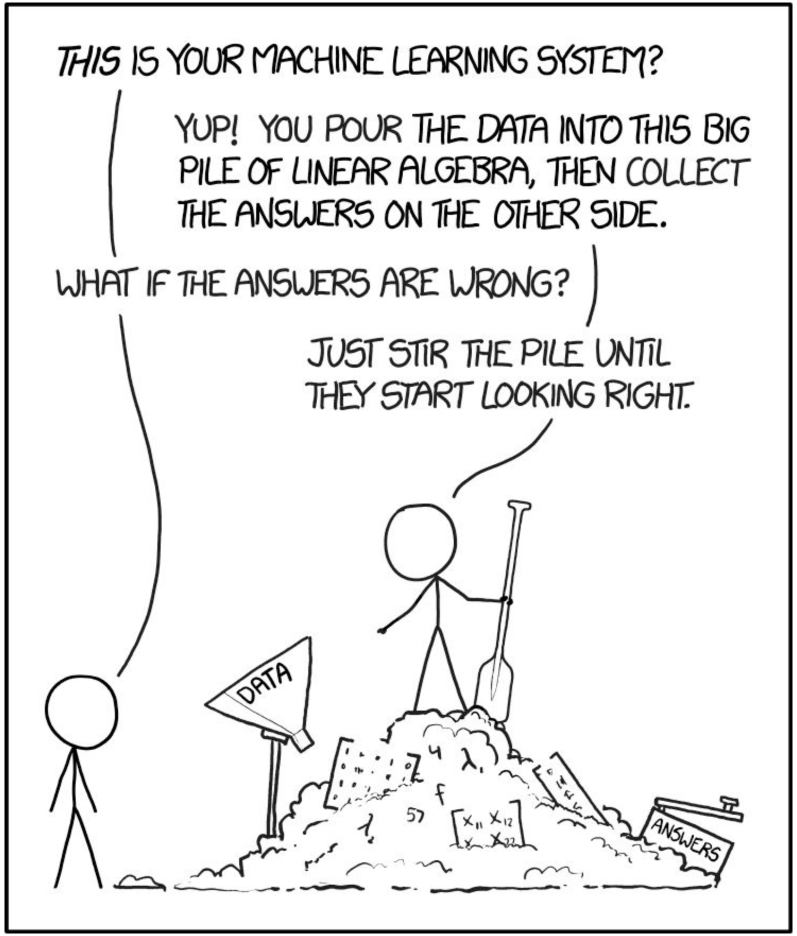
```
np.argmax(predictions)
```

4

Model building is an iterative process

1. Acquire data
2. Study and clean the data
3. Choose model
4. Choose hyperparameters (such as training algorithm, # of epochs, number of layers, number of neutrons per layer, activation functions, loss function, ...)
5. Train (with training data); not happy - goto 1, 2, 3, 4
6. Validate with unseen samples (validation data); not happy - goto 1, 2, 3, 4
7. Verify with test data; not happy - goto 1, 2, 3, 4

“All models are wrong, but some are useful.” -- George Box


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

https://xkcd.com/1838/

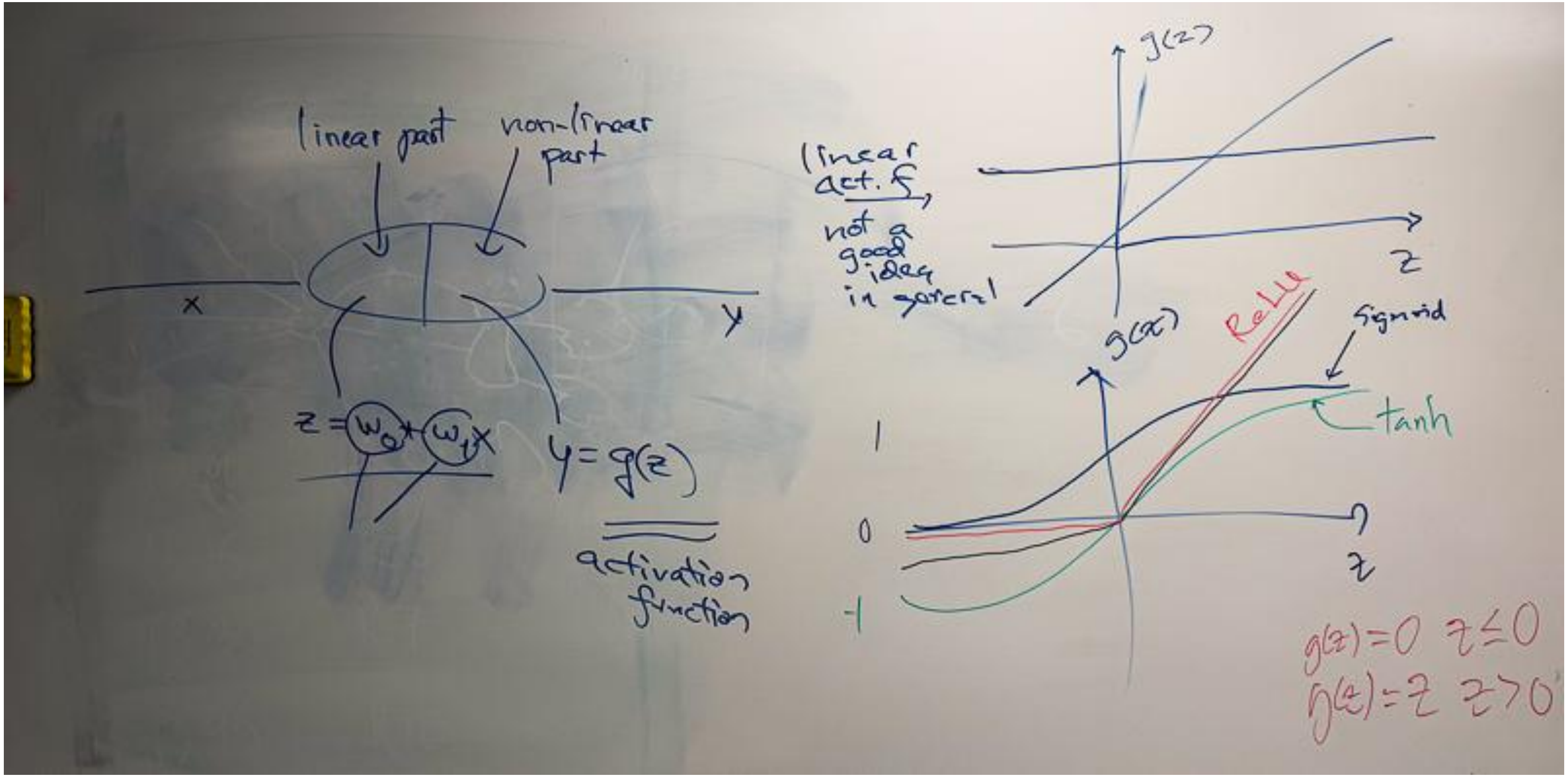
Some recommended books

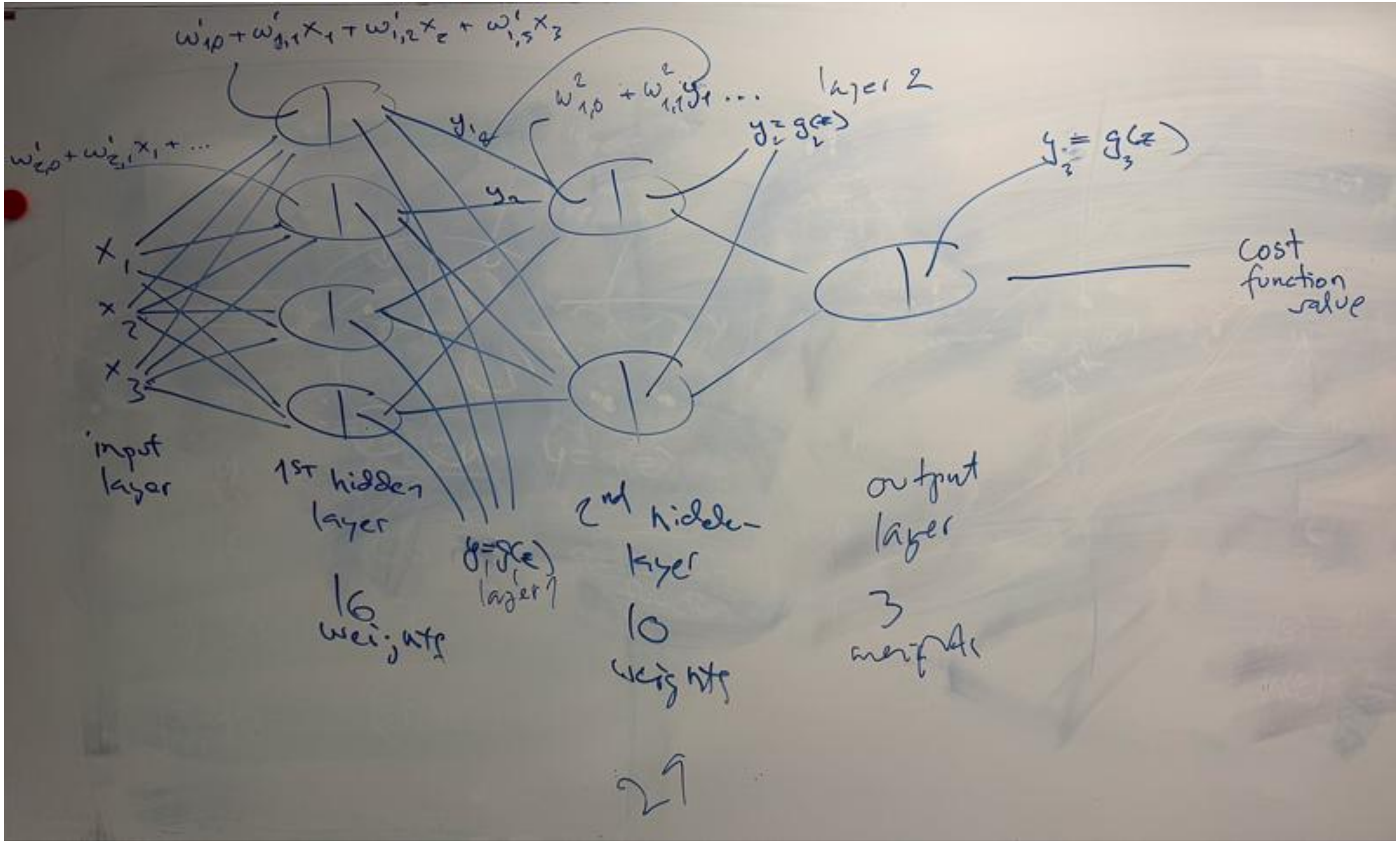
Deep Learning with Python, Second Edition by Francois Chollet. Book by Keras framework author. Nice entry to deep learning literature.

Probabilistic Machine Learning: An Introduction by Kevin Murphy. Very nice, a bit mathematically oriented overview. Also available at https://probml.github.io/pml-book/book1.html

Deep Learning by Ian Goodfellow, Yoshua Bengio, Aaron Courville. Also available at https://www.deeplearningbook.org/ . Overview of deep learning 2016. A bit outdated but good reference to basic + some advanced topics.

Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig. A wide overview of the field, including classical not so often discussed topics in the field.


linear part
non-linear part
x
y
z = w_0 + w_1 x
y = g(z)
activation function
linear act. f
not a good idea in general
g(z)
z
g(x)
ReLU
sigmoid
tanh
1
0
-1
z
g(z) = 0 z ≤ 0
g(z) = z z > 0


$w^1_{1,0} + w^1_{1,1}x_1 + w^1_{1,2}x_2 + w^1_{1,3}x_3$
$w^1_{2,0} + w^1_{2,1}x_1 + \ldots$
$w^2_{1,0} + w^2_{1,1}y_1 \ldots$
layer 2
y_1
y_2
$y_2 = g_2(z)$
$y_3 = g_3(z)$
x_1
x_2
x_3
input layer
1st hidden layer
$y_1 = g_1(z)$
layer 1
2nd hidden layer
output layer
cost function value
16 weights
10 weights
3 weights
29

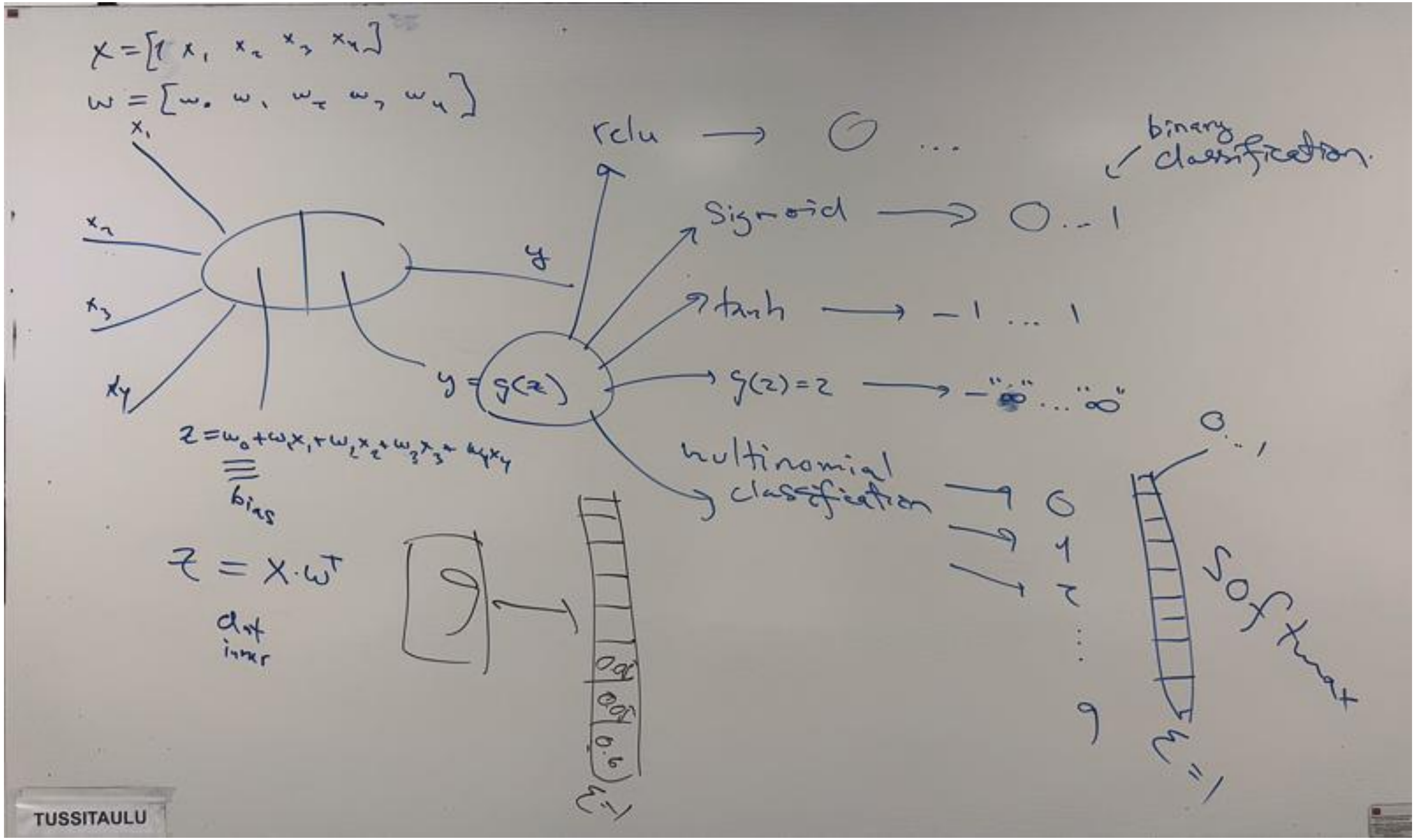
$x = [1 \; x_1 \; x_2 \; x_3 \; x_4]$

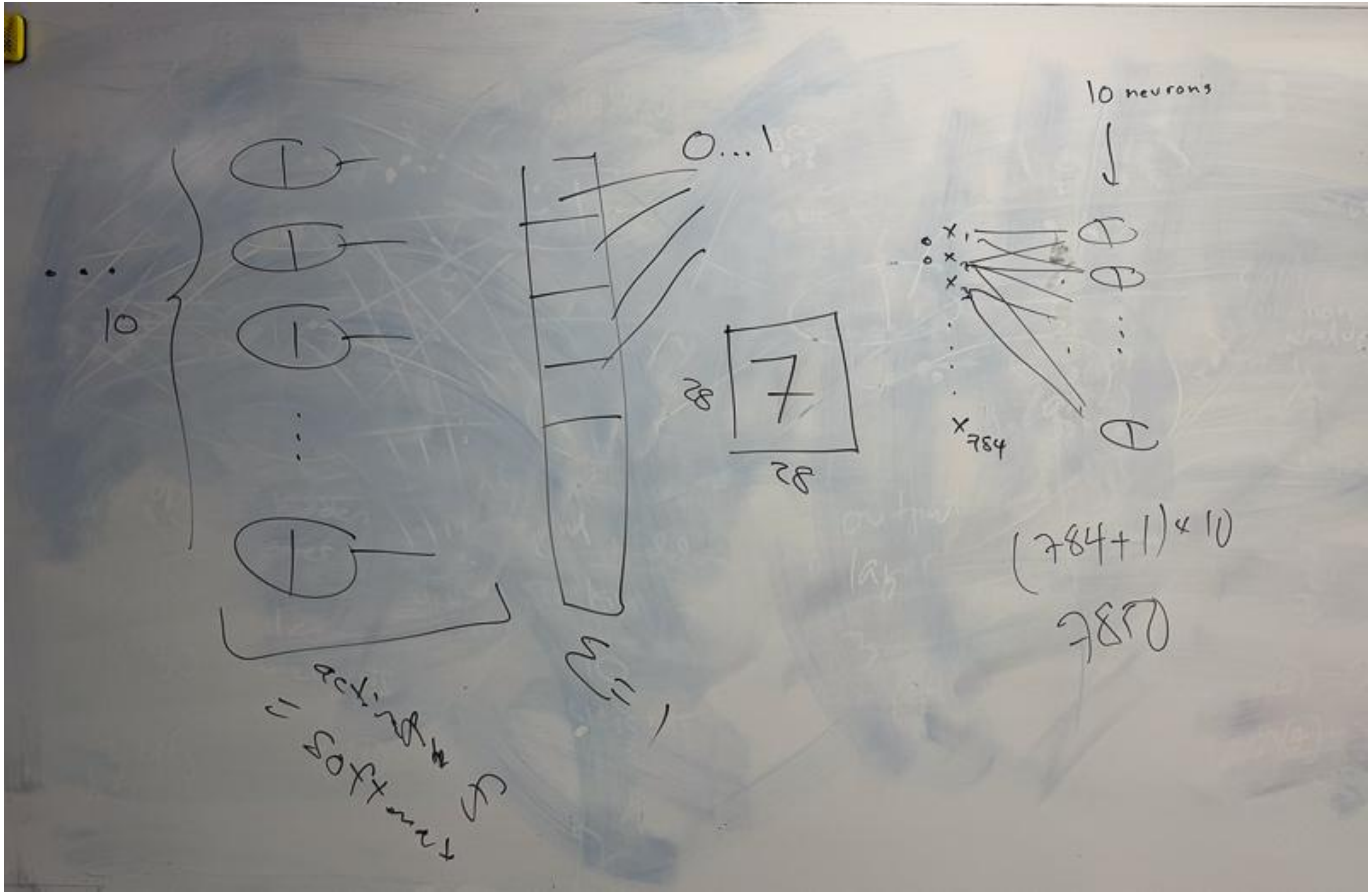
$w = [w_0 \; w_1 \; w_2 \; w_3 \; w_4]$

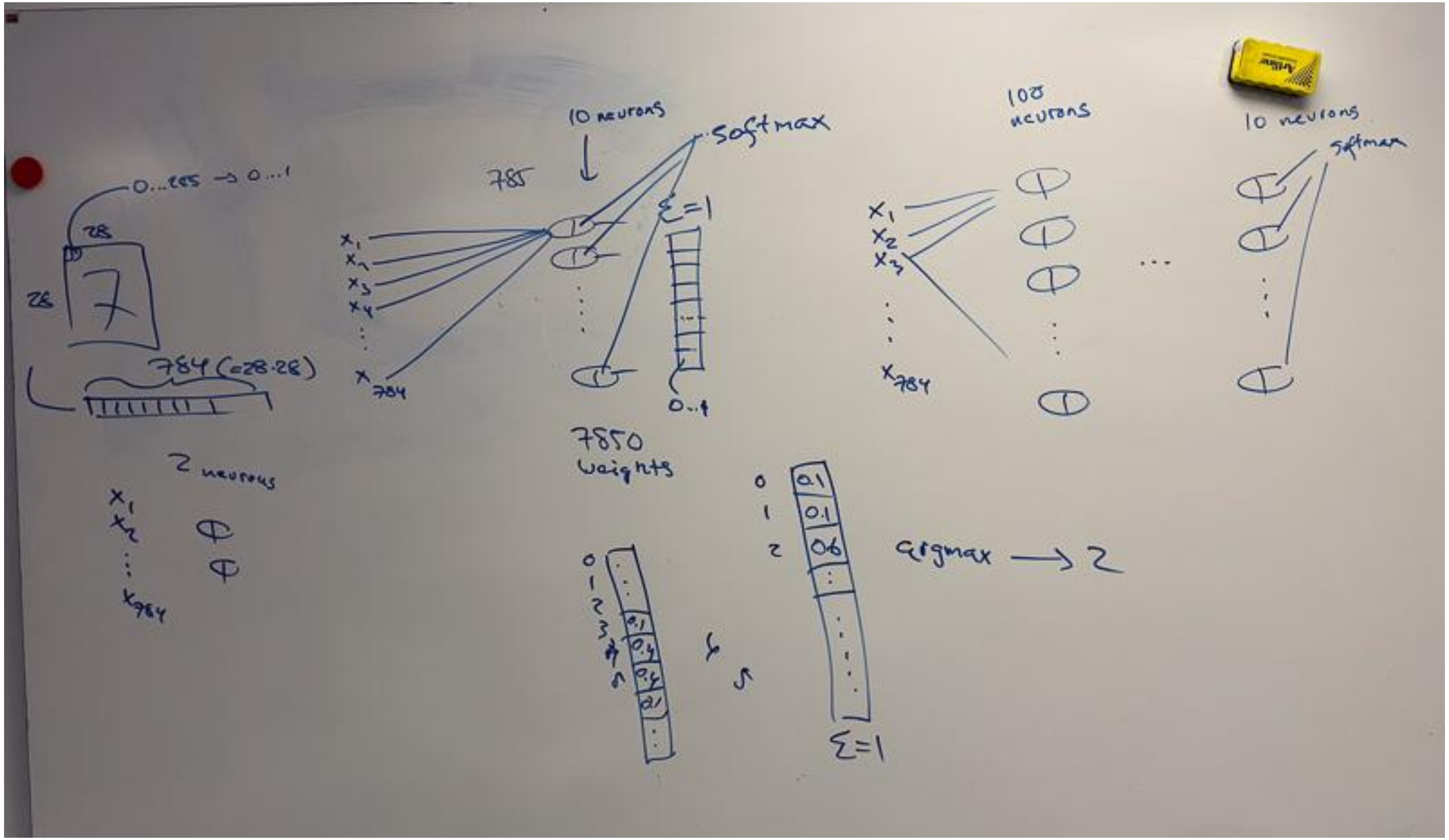
$z = \underbrace{w_0}_{\text{bias}} + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4$

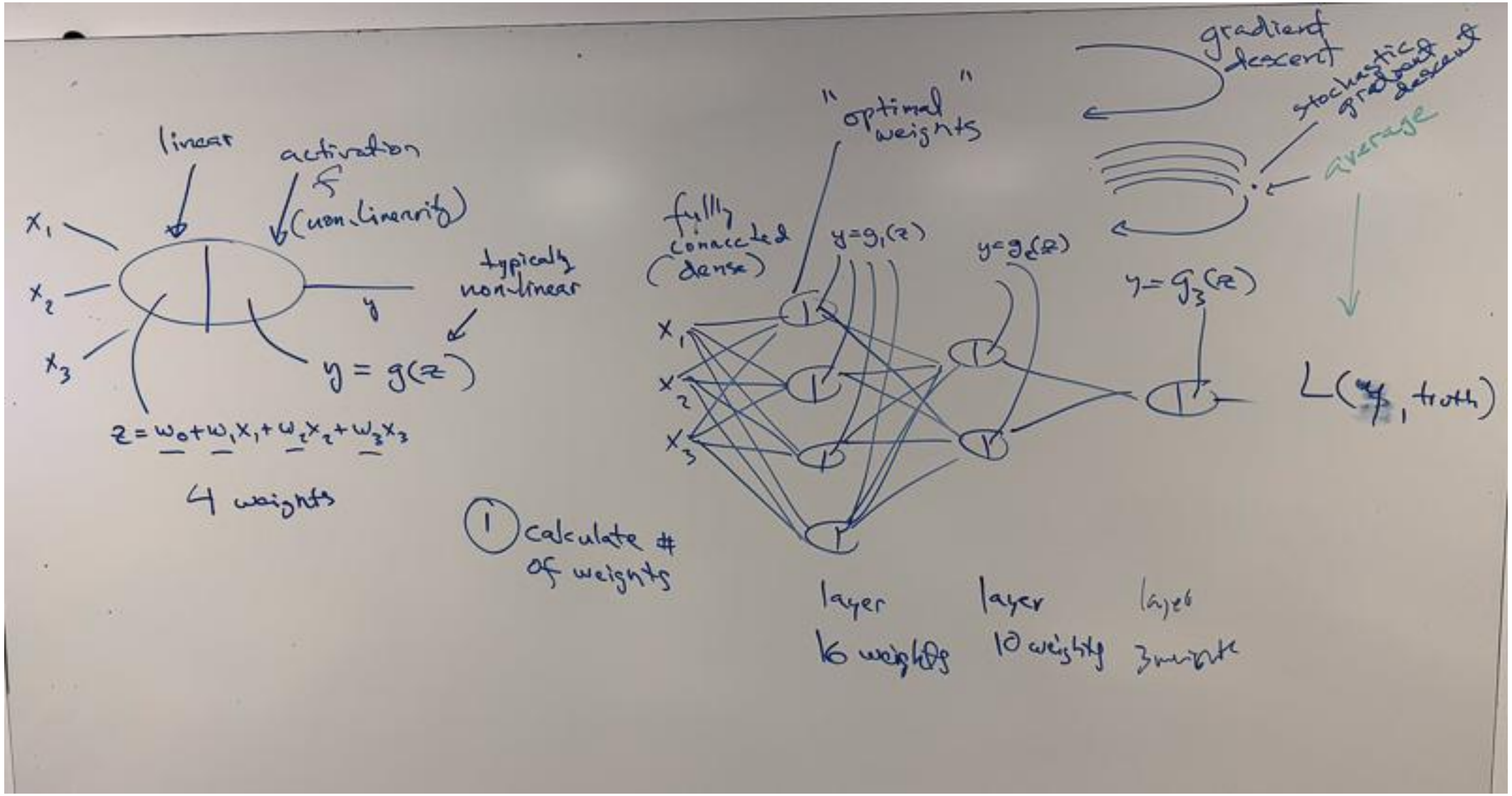
$z = X \cdot w^T$

dot
inner




10 neurons
0...1
10
28
7
28
x1
x2
x3
x784
(784+1)×10
7850
activation: softmax
Σ = 1


0...255 → 0...1
28
28
7
784 (=28·28)
2 neurons
x1
x2
x784
x1
x2
x3
x4
x784
785
10 neurons
softmax
Σ=1
0..1
7850 weights
0
1
2
3
4
5
0.1
0.4
0.4
0.1
4
5
0
1
2
0.1
0.1
0.6
Σ=1
argmax → 2
100 neurons
x1
x2
x3
x784
10 neurons
softmax


linear
activation
(non-linearity)
x_1
x_2
x_3
y
typically non-linear
y = g(z)
z = w0 + w1x1 + w2x2 + w3x3
4 weights
1 calculate # of weights
fully connected (dense)
"optimal" weights
y = g1(z)
y = g2(z)
y = g3(z)
layer 16 weights
layer 10 weights
layer 3 weights
gradient descent
stochastic gradient descent
average
L(y, truth)