POLITECNICO
MILANO 1863

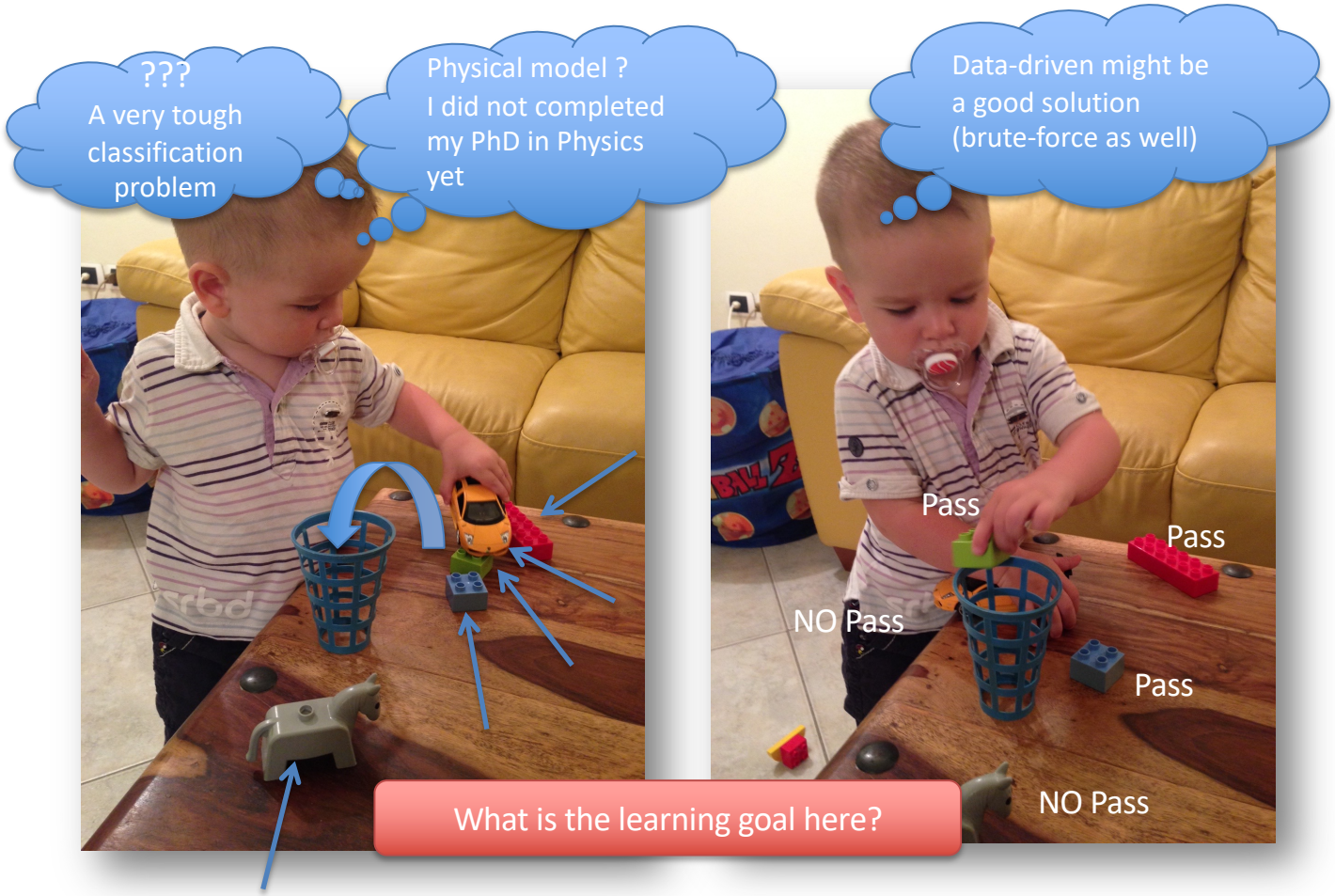
Hardware Architectures for Embedded and Edge AI

Prof Manuel Roveri – manuel.roveri@polimi.it

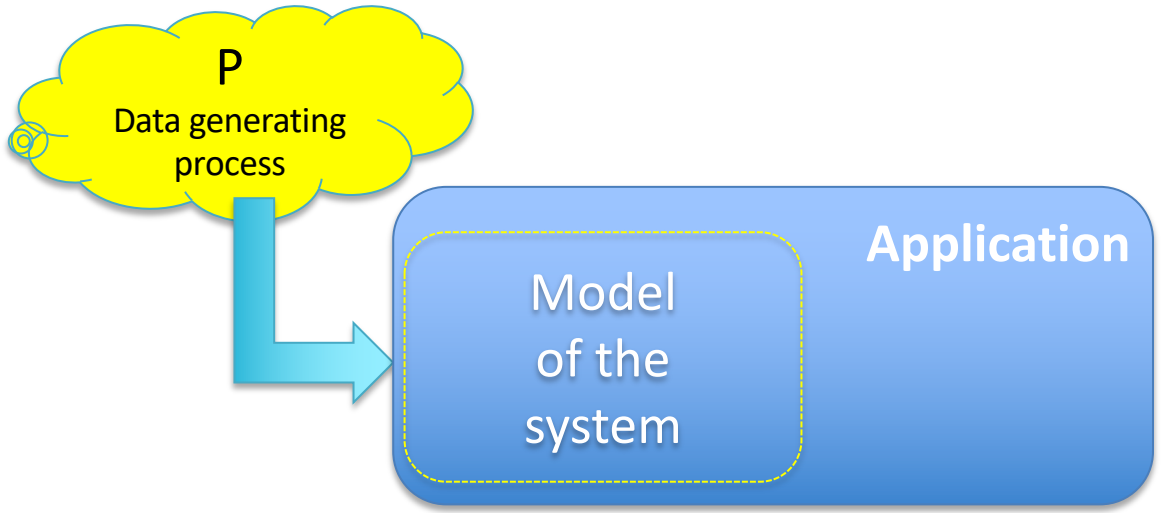
Lecture 4 – Machine Learning for Embedded and Edge AI

The basics of learning

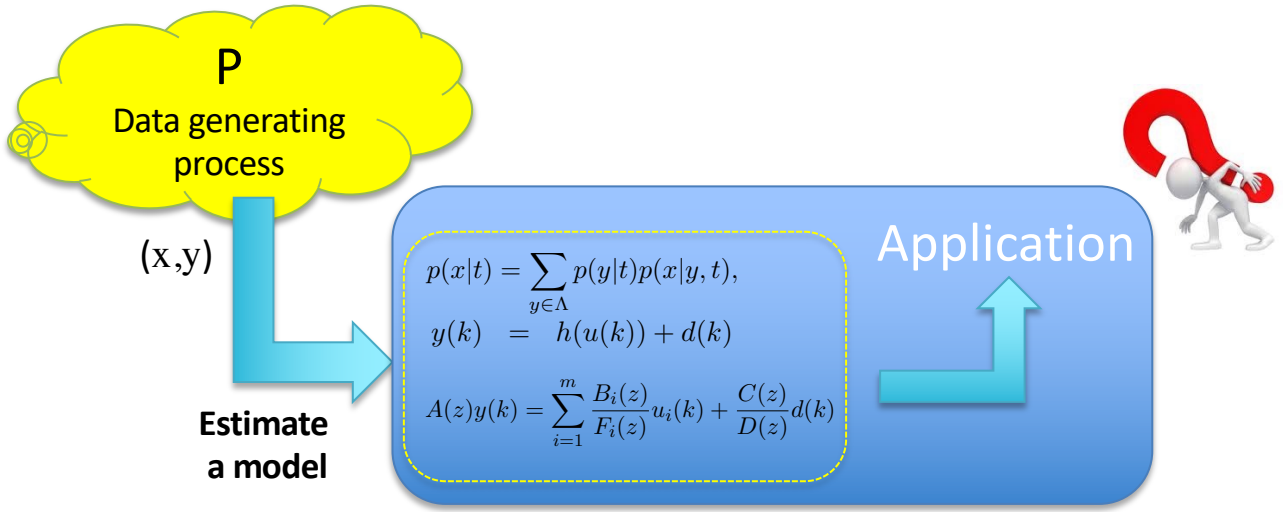
A "toy" example



Data-processing and applications

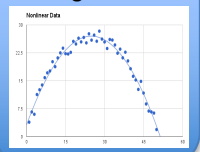


Learning the system model

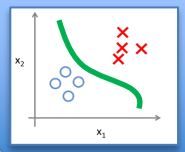


Supervised Learning: Statistical framework

Regression



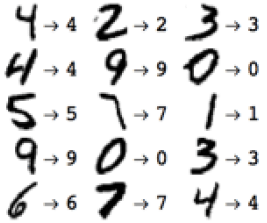
Classification

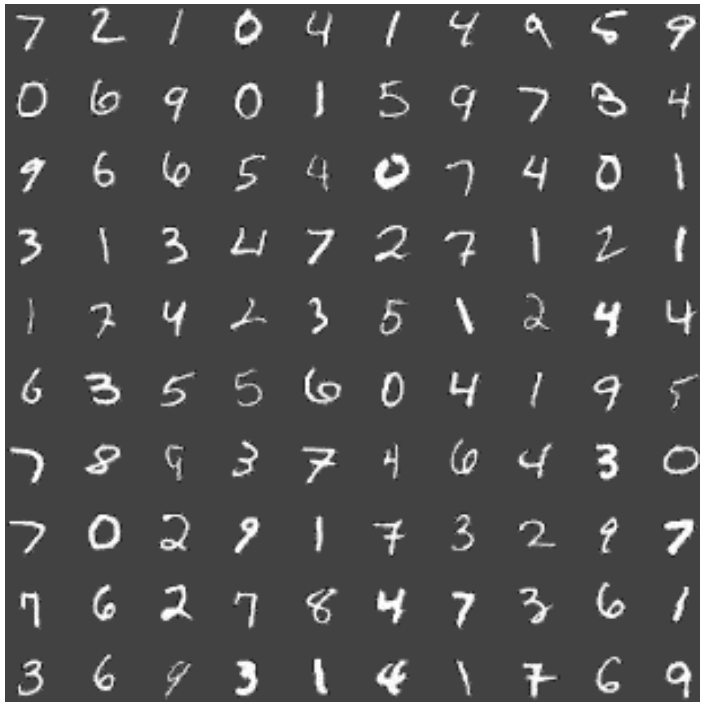


Statistical Learning: the approach

The training set

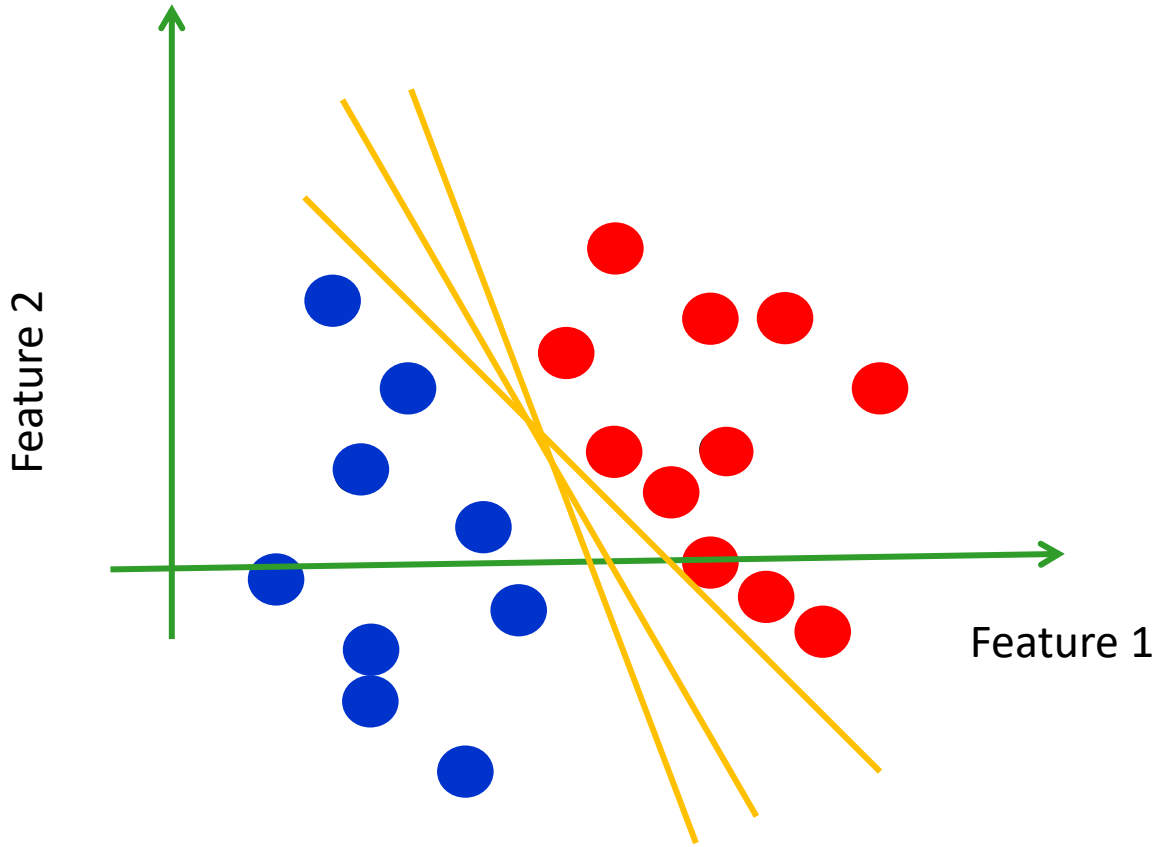
$$Z_N = \{(x_1, y_1), \ldots, (x_N, y_N)\}$$





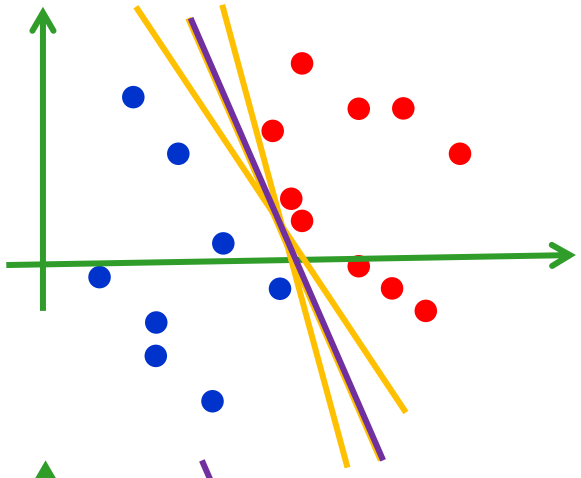
Designing a classifier

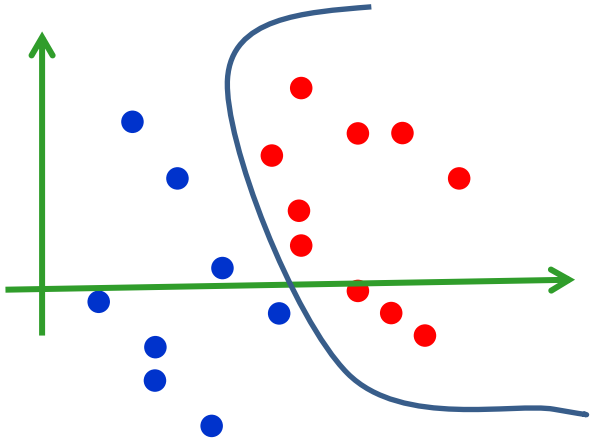
Consider a bidimensional problem

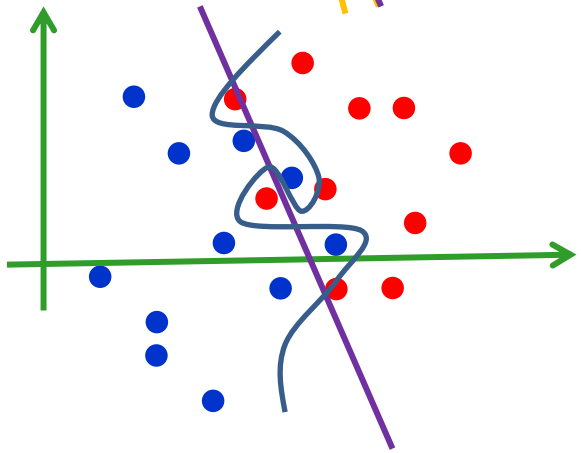


Designing a classifier requires identification of the function separating the labeled points

Some issues we need to focus on

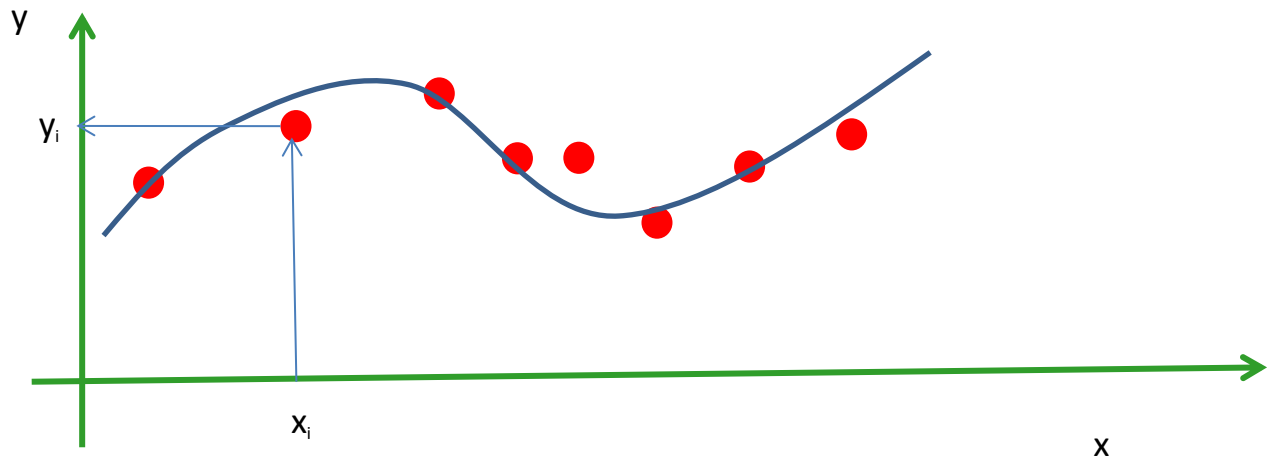






- Linear vs. non linear
- Many points versus the available points
- Several tecniques are available to design the classifier (KNN, feedforward NN, SVM…)

Non linear regression



Given a set of *n* noise affected couples (x_i,y_i) we wish to reconstruct the unknown function

Non-linear regression: statistical framework

The time invariant process generating the data

$$y = g(x) + \eta,$$

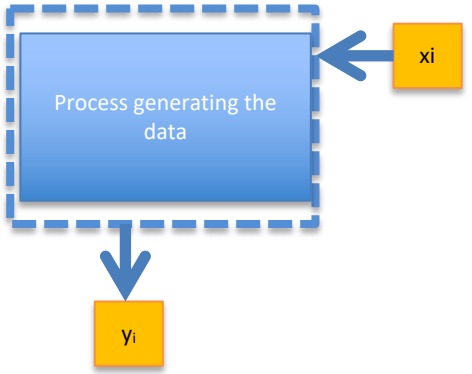
provides, given input x_i output instance

$$y_i = g(x_i) + \eta_i$$

We collect a set of couples (training set)

$$Z_N = \{(x_1, y_1), \ldots, (x_N, y_N)\}$$

And wish to model unknown $g(x)$ with parameterized family of models $f(\theta, x)$



The goal of **learning** is to build the simplest approximating model able to explain past data Z_N and future instances provided by the data generating process.

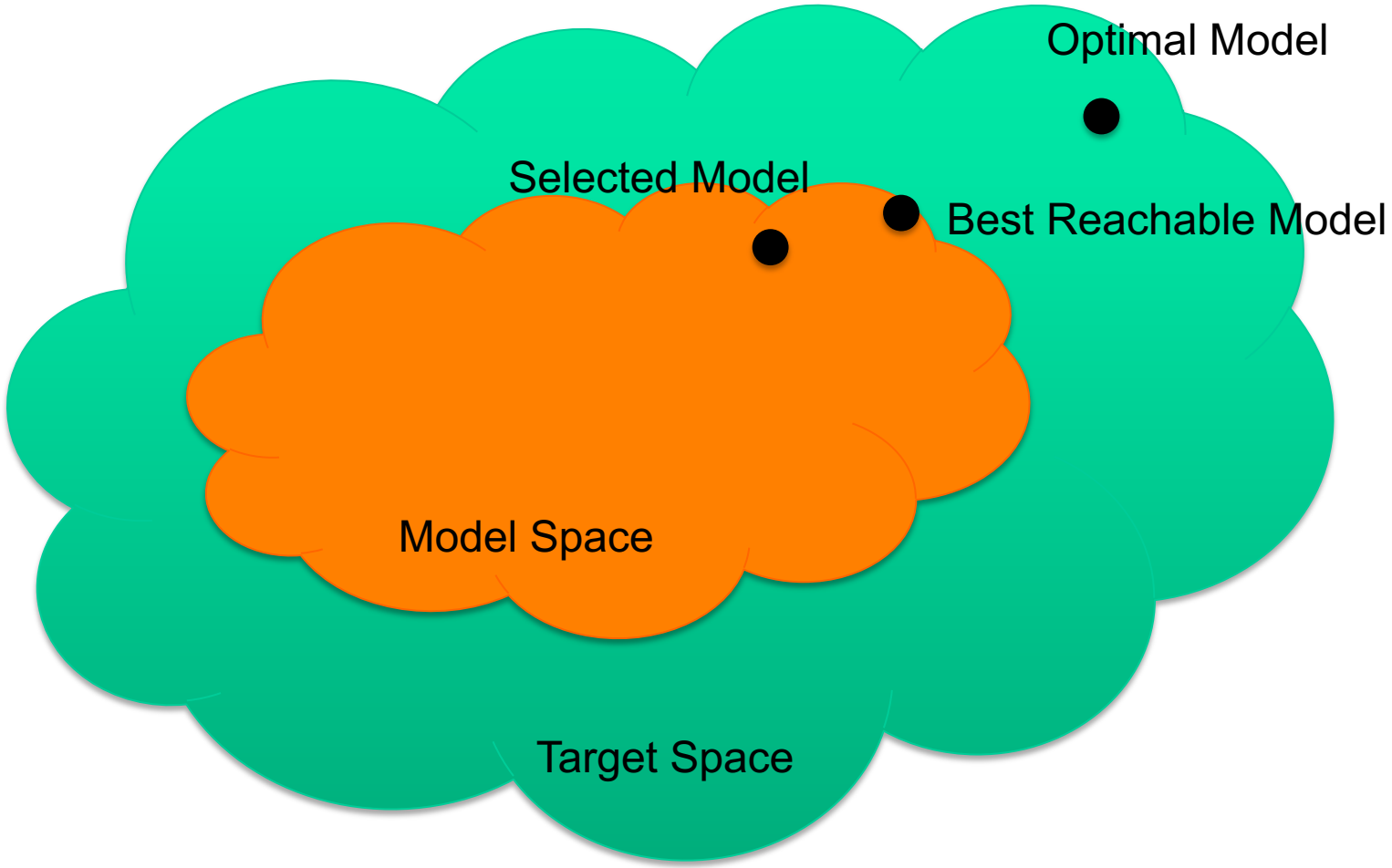
Inherent, approximation and estimation risks

$$\bar{V}(\hat{\theta}) = \left(\bar{V}(\hat{\theta}) - \bar{V}(\theta^o)\right) + \left(\bar{V}(\theta^o) - V_I\right) + V_I.$$

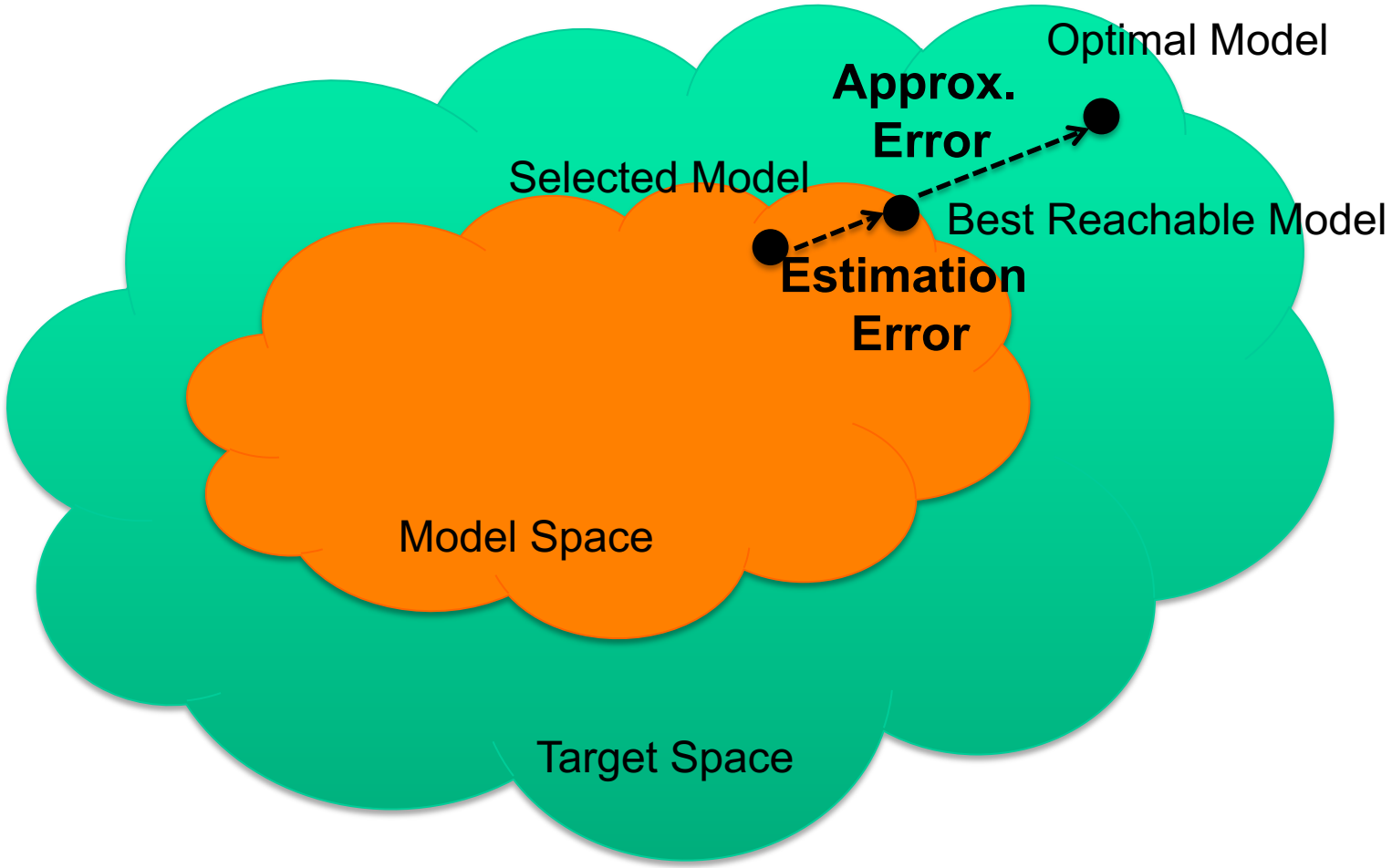
estimation risk — approximation risk — inherent risk

- The inherent risk depends only on the structure of the learning problem and, for this reason, can be reduced only by improving the problem itself
- The approximation risk depends on how close the model family (also named hypothesis space) is to the process generating the data
- The estimation risk depends on the ability of the learning algorithm to select a parameter vector $\hat{\theta}$ close to θ^o

Approximation and estimation risks

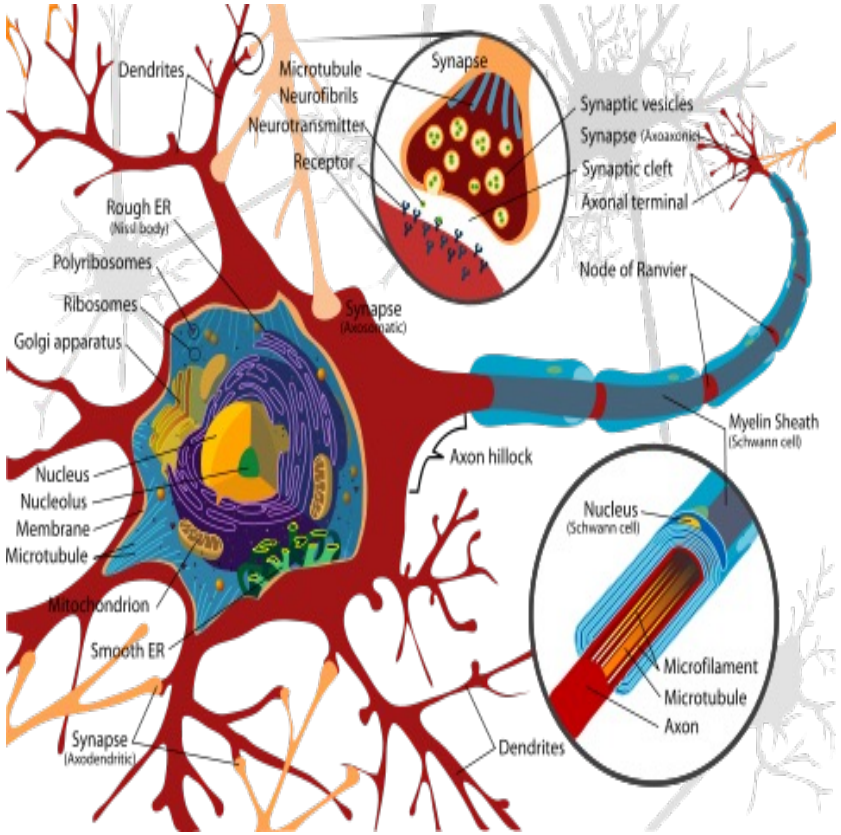


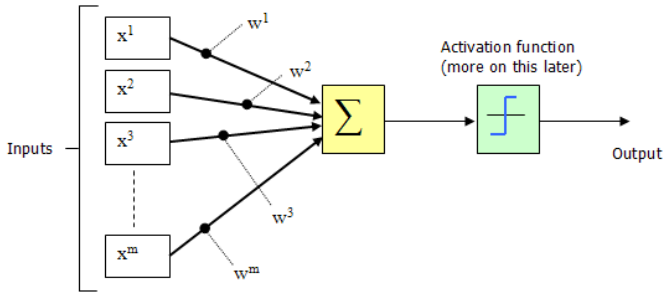
Approximation and estimation risks



What about Neural Networks?

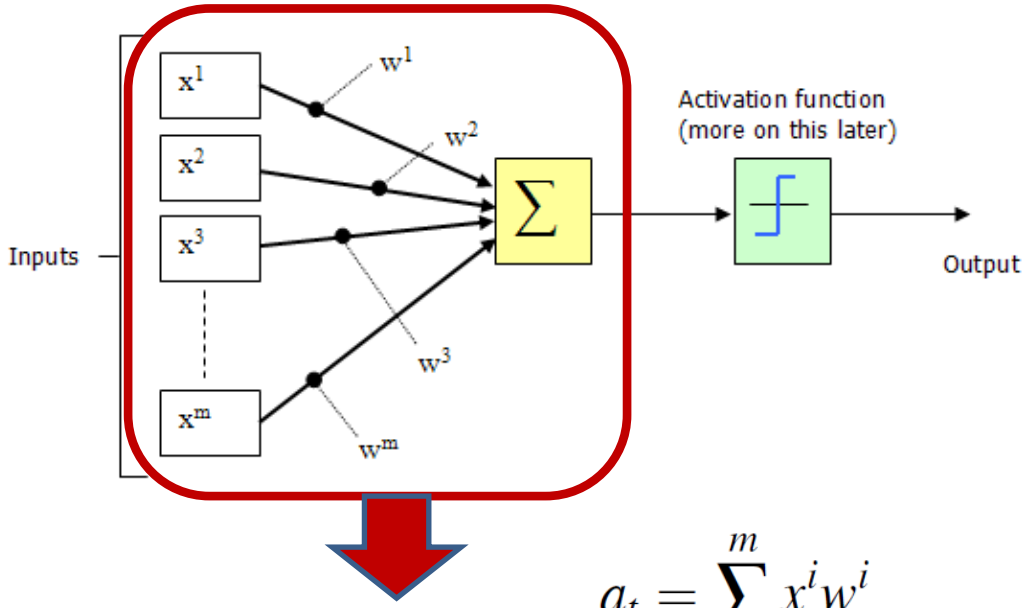
Modelling space and time





The model encompasses the concept of space, time and status of the neuron

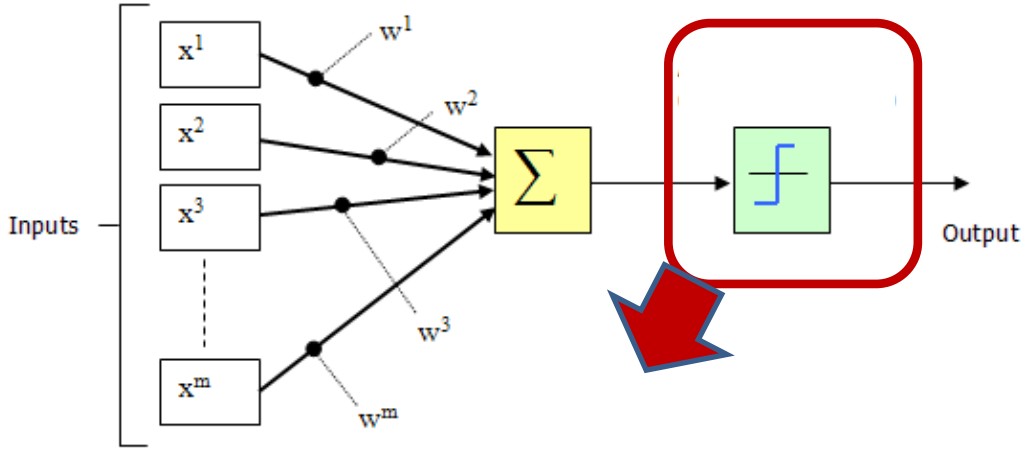
Neural computation



$$a_t = \sum_{i=1}^{m} x^i w^i$$

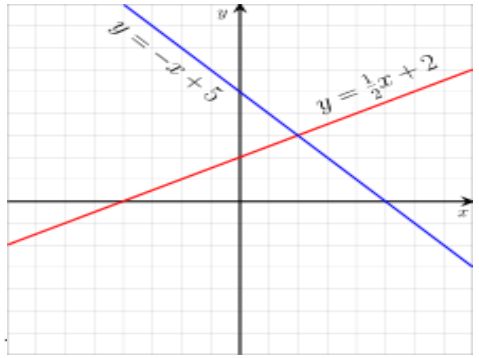
The scalar product:
evaluate the affinity of values

Neural computation

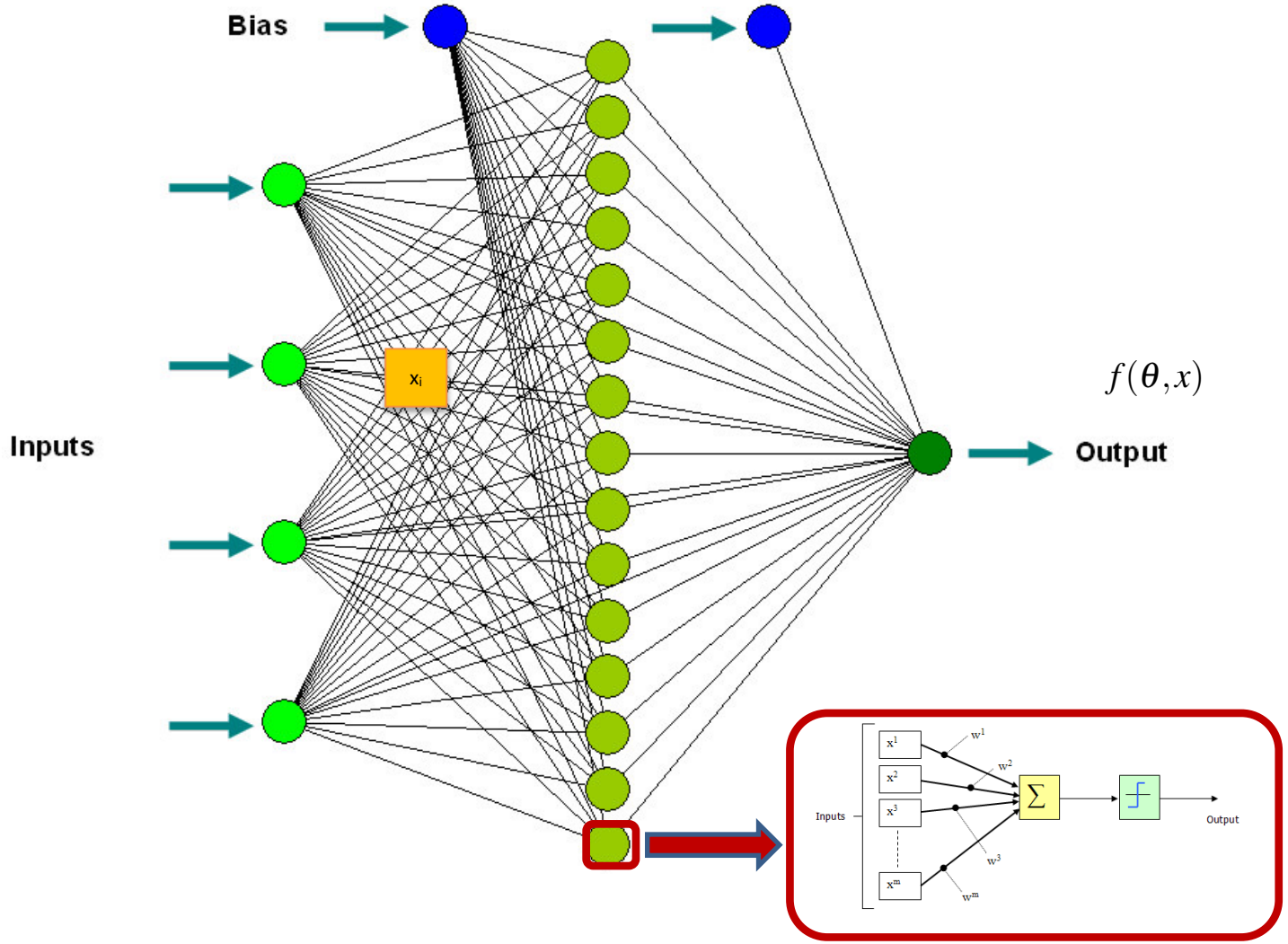


Activation function

- Heaviside
- Sigmoidal
- Linear



Multi-layer Neural Networks

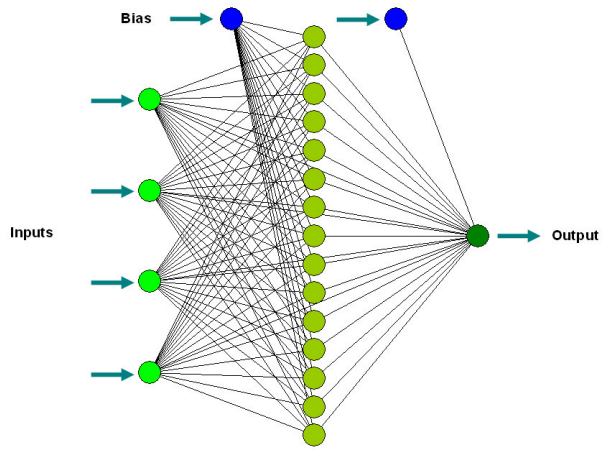


Why Neural Networks?

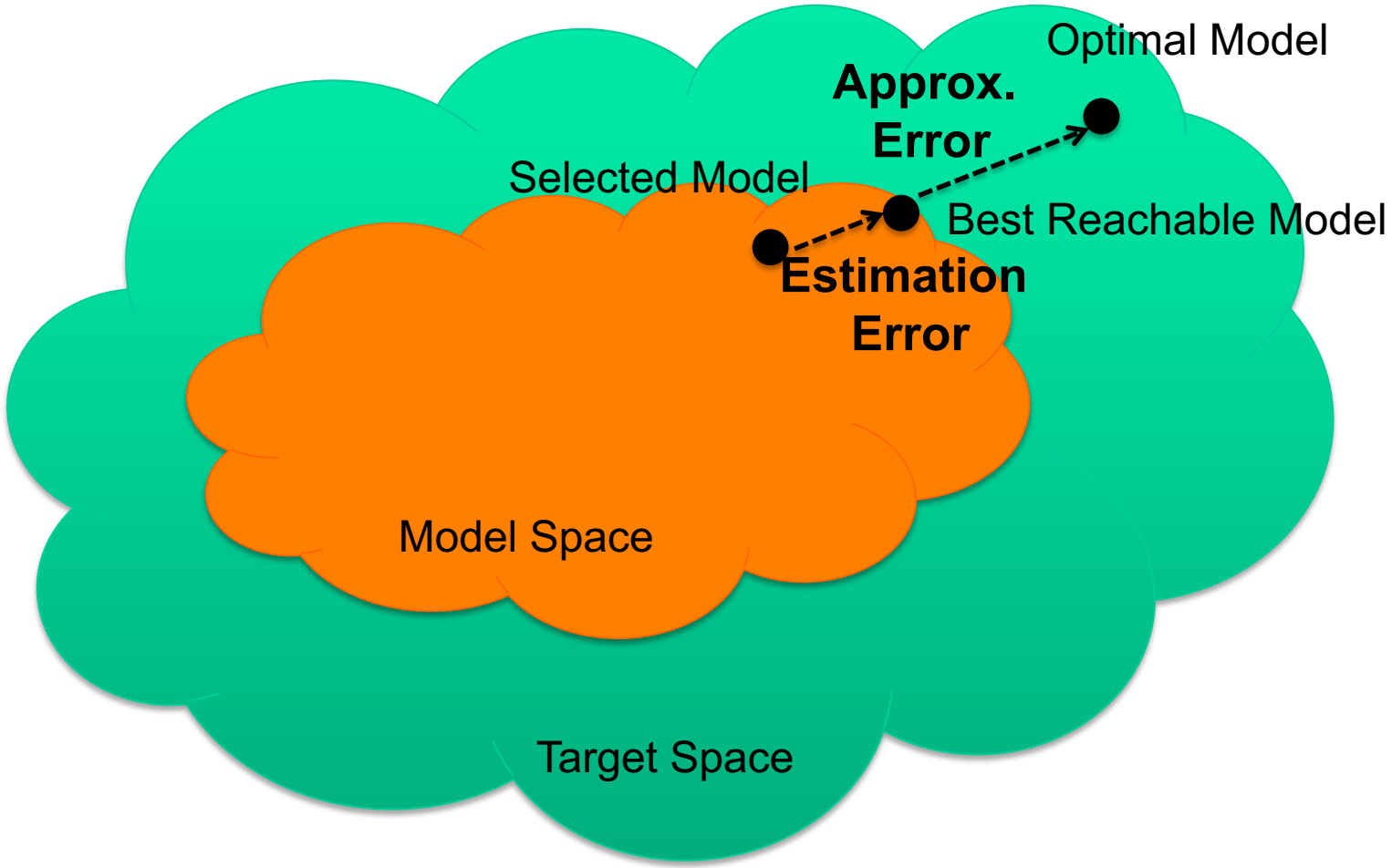
Universal approximation theorem

A feedforward network with a single hidden layer containing a finite number of neurons approximates any continuous function defined on compact subsets

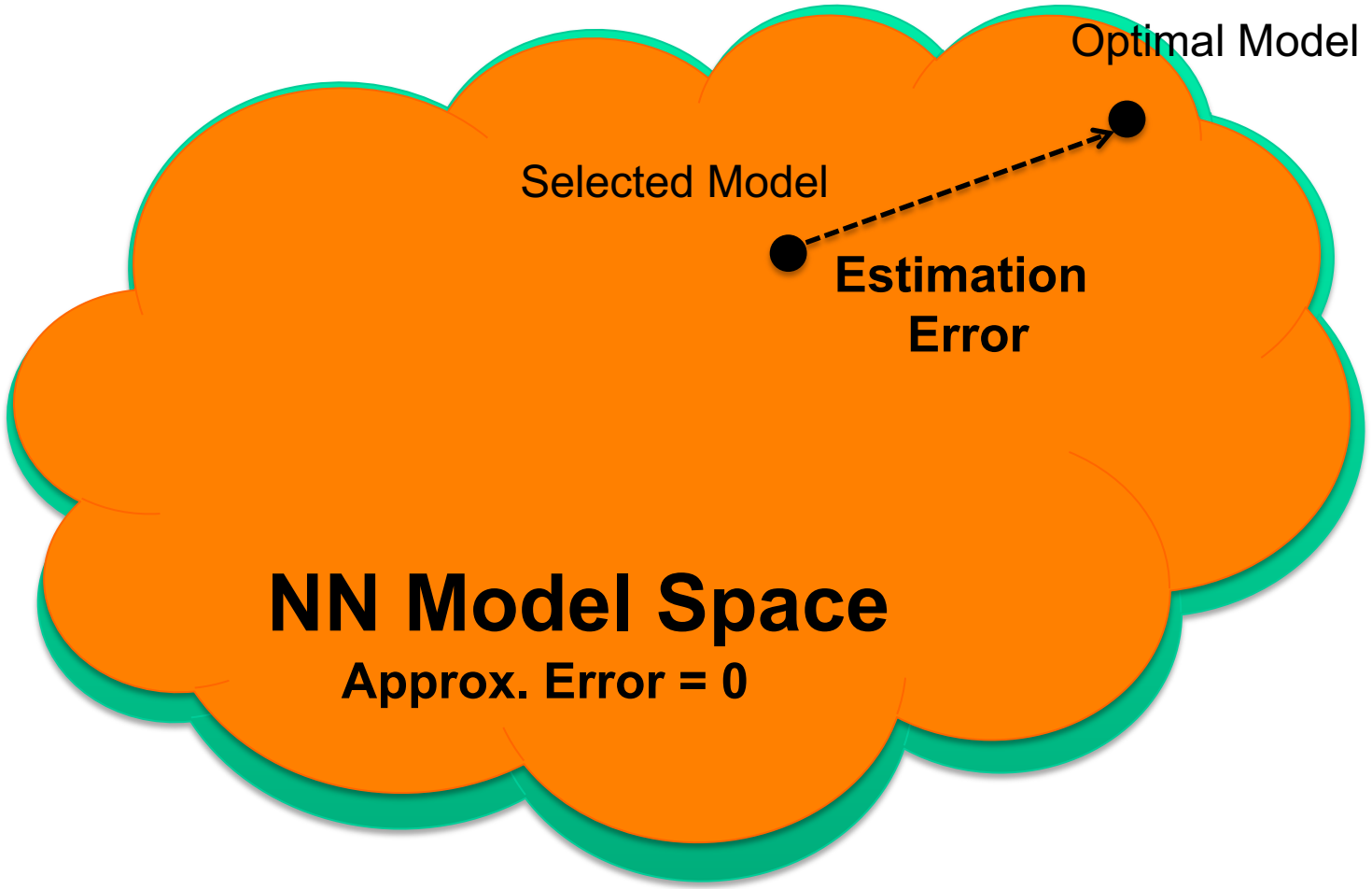
K. Hornik, "Approximation Capabilities of Multilayer Feedforward Networks", *Neural Networks*, No.4 Vol. 2, 251–257, 1991



Approximation and estimation risks

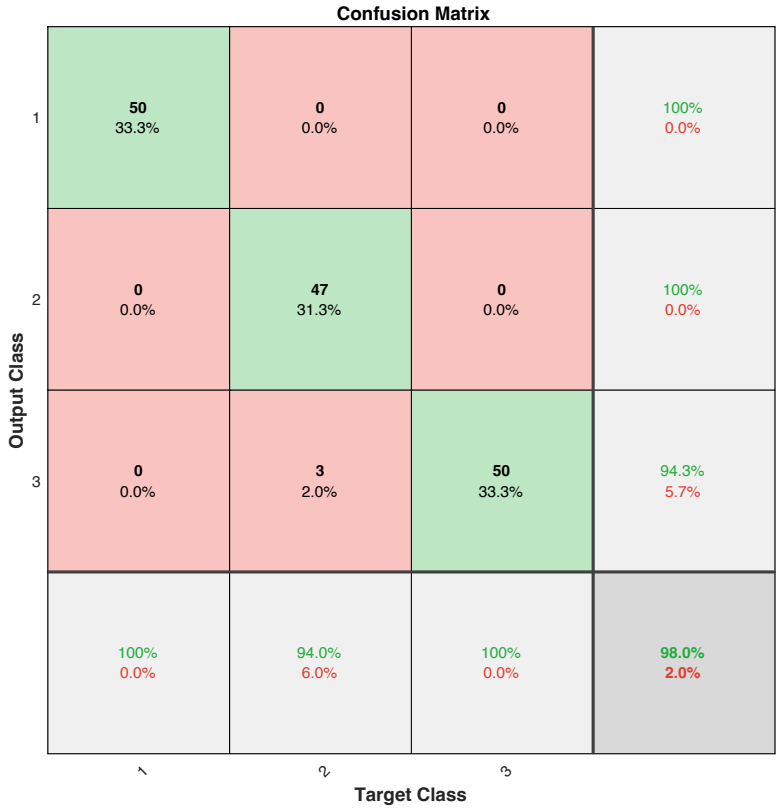


Approximation and estimation risks

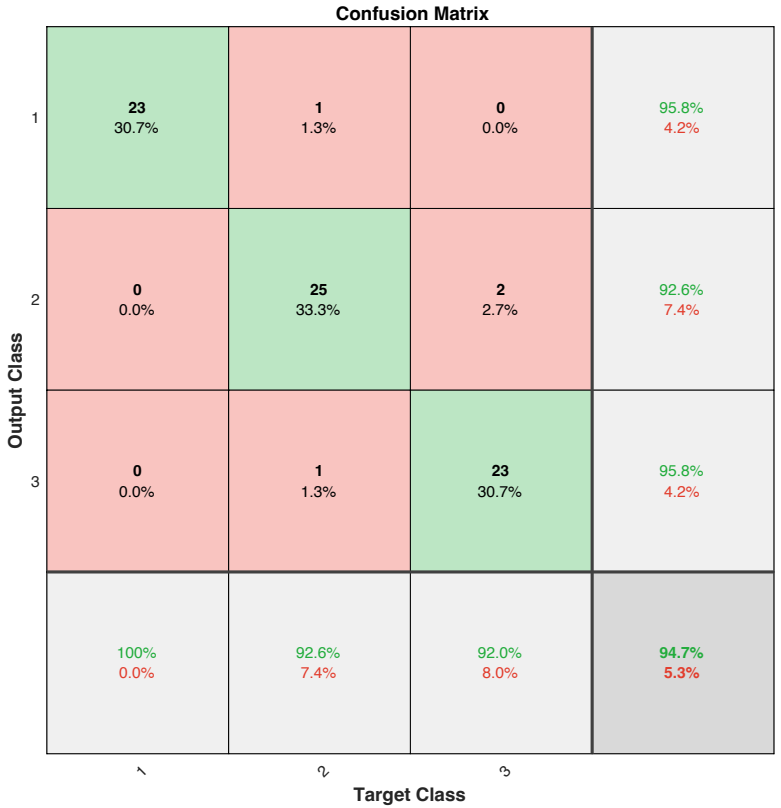


How “good” is my good ML solution?

Two examples: how good is my good solution?



Apparent Error Rate



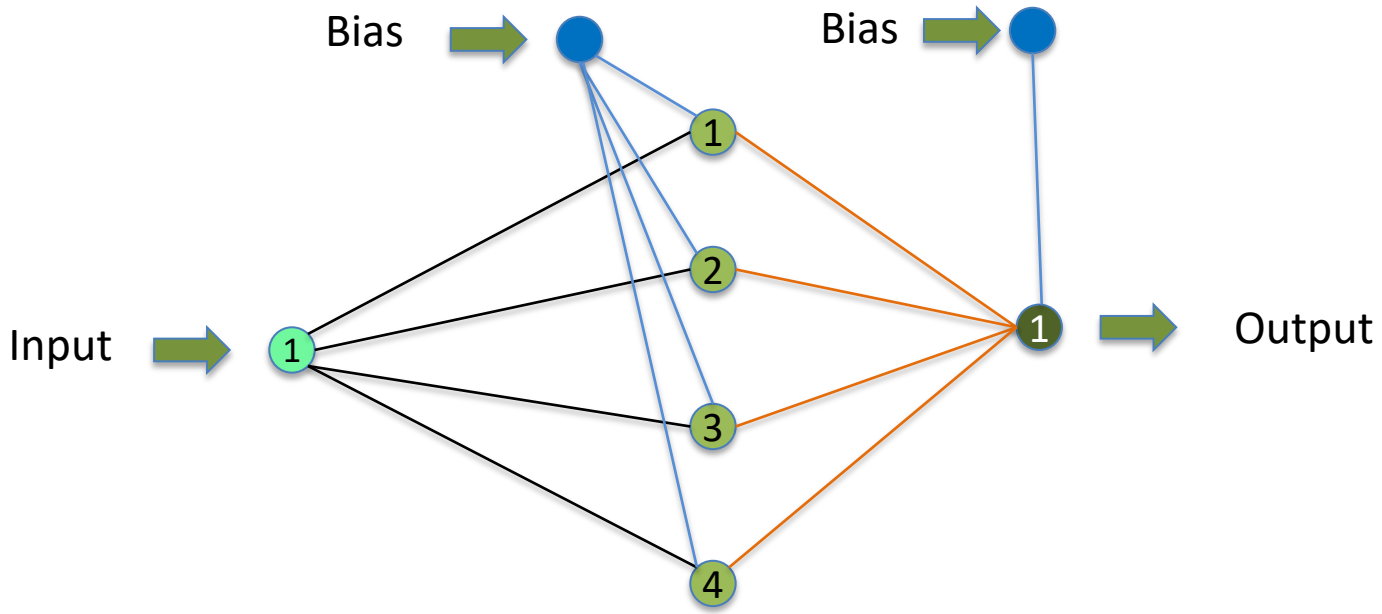
Sample Partitioning

Assessing the performance

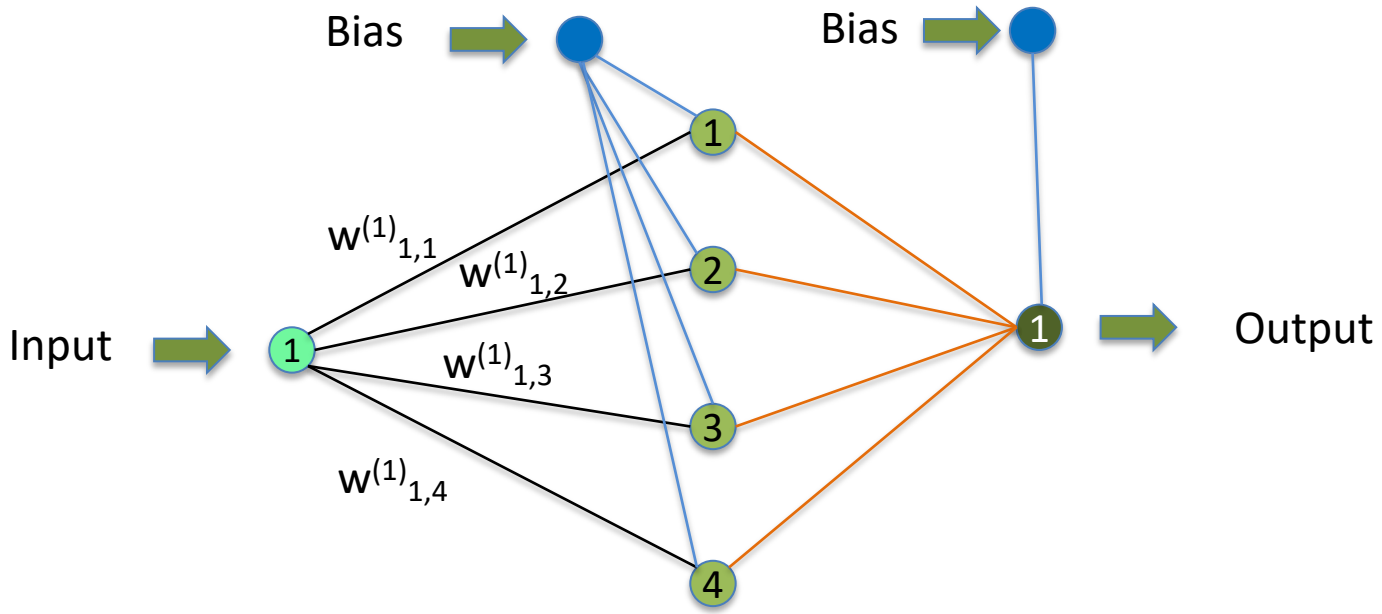
- *Apparent Error Rate* (AER),or *resubstitution*: The whole set Z_N is used both to infer the model and to estimate its error
- *Sample Partitioning* (SP): S_D and S_E are obtained by randomly splitting Z_N in two disjoint subsets. S_D is used to estimate the model and S_E to estimate its accuracy.
- *Leaving-One-Out* (LOO): S_E contains one pattern in Z_N, and S_D contains the remaining n − 1 patterns. The procedure is iterated n times by holding out each pattern in Z_N, and the resulting n estimates are averaged.
- *w-fold Crossvalidation* (wCV): Z_N is randomly split into w disjoint subsets of equal size. For each subset the remaining w − 1 subsets are merged to form S_D and the reserved subset is used as S_E . The w estimates are averaged.

Memory demand of NNs

Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output

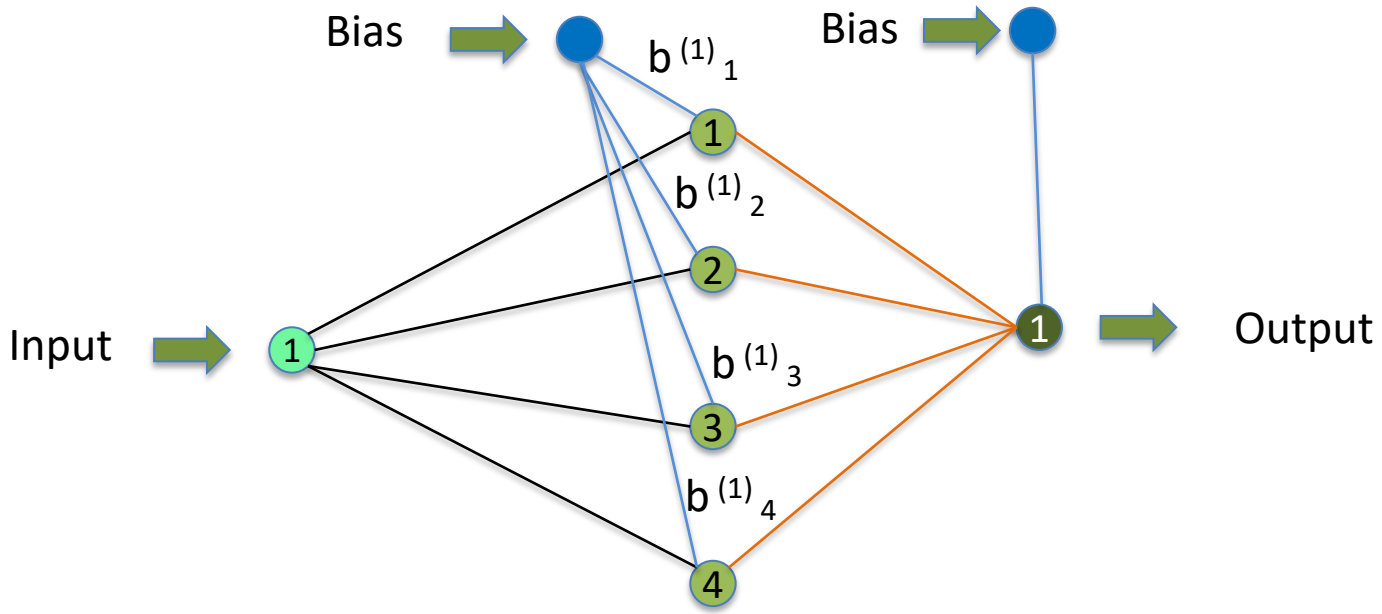


Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output



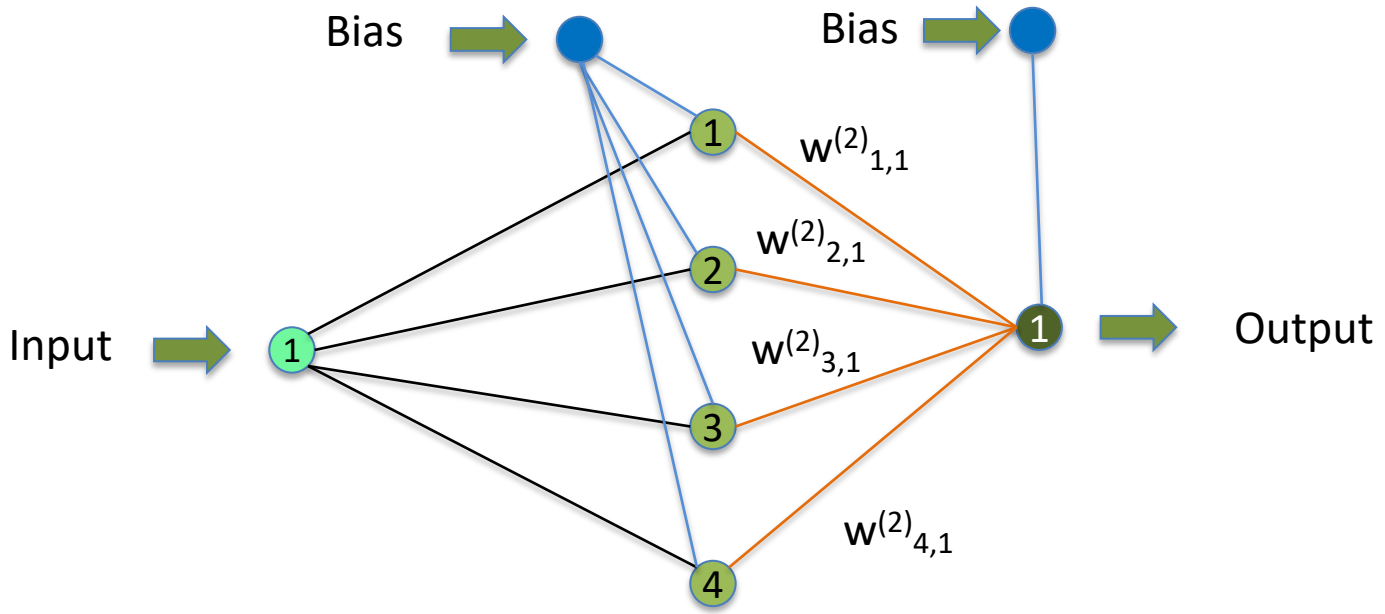
$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, w^{(1)}_{1,3}, w^{(1)}_{1,4}]$

Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output



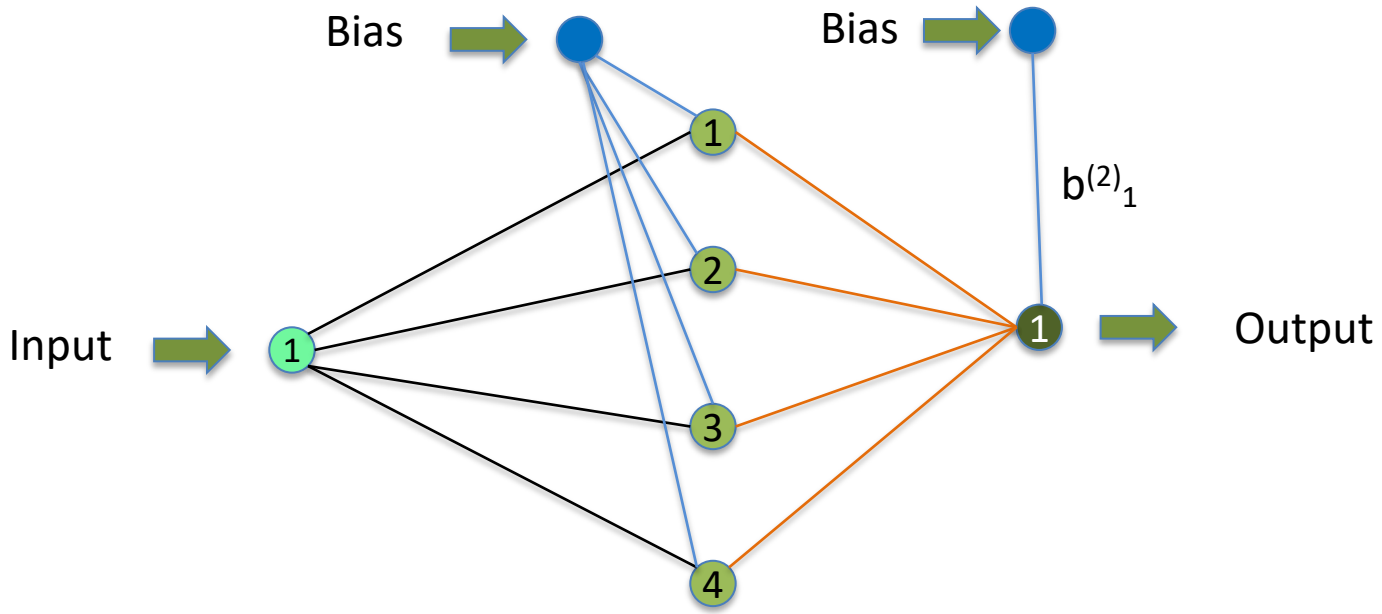
$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, w^{(1)}_{1,3}, w^{(1)}_{1,4}] + [b^{(1)}_{1}, b^{(1)}_{2}, b^{(1)}_{3}, b^{(1)}_{4}]$

Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output



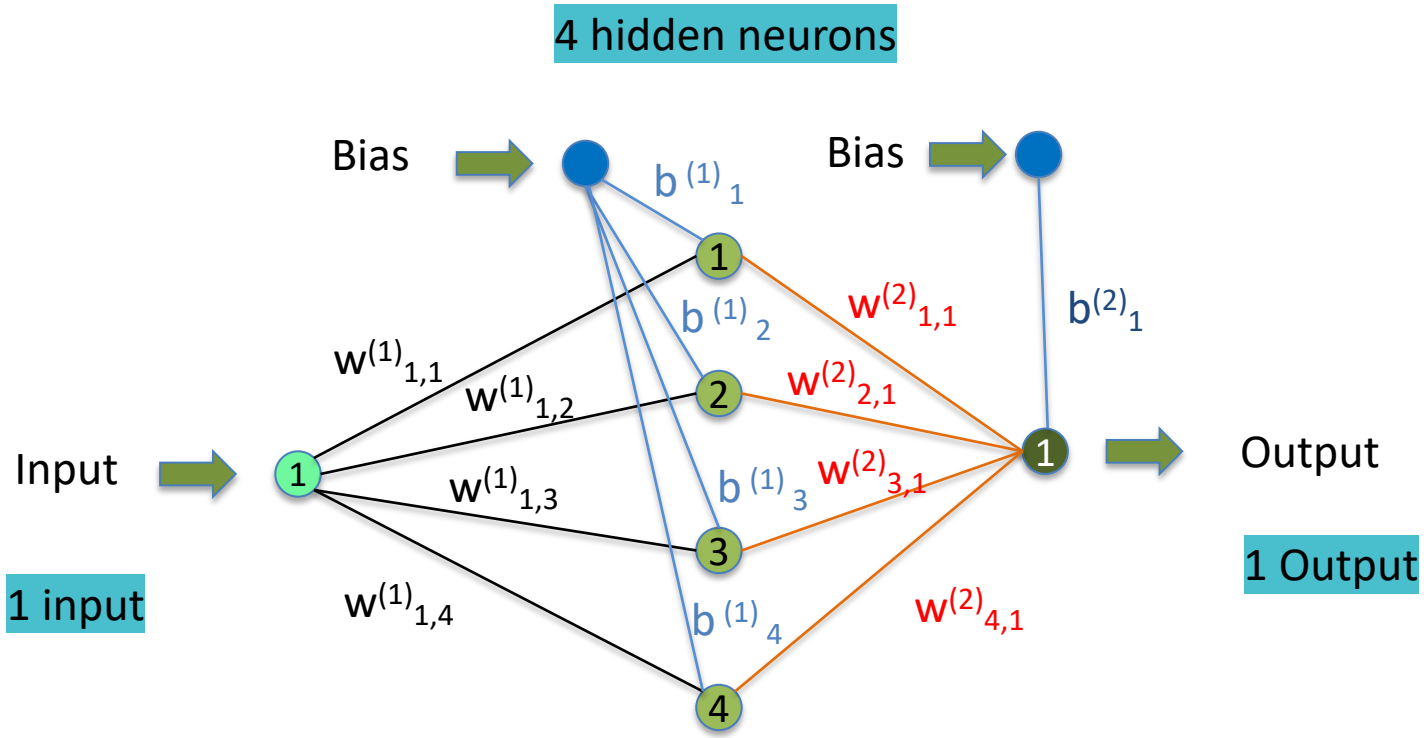
$$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, w^{(1)}_{1,3}, w^{(1)}_{1,4}] + [b^{(1)}_{1,1}, b^{(1)}_{1,2}, b^{(1)}_{1,3}, b^{(1)}_{1,4}] + [w^{(2)}_{1,1}, w^{(2)}_{2,1}, w^{(2)}_{3,1}, w^{(2)}_{4,1}]$$

Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output



$$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, w^{(1)}_{1,3}, w^{(1)}_{1,4}] + [b^{(1)}_{1,1}, b^{(1)}_{1,2}, b^{(1)}_{1,3}, b^{(1)}_{1,4}] + [w^{(2)}_{1,1}, w^{(2)}_{2,1}, w^{(2)}_{3,1}, w^{(2)}_{4,1}] + [b^{(2)}_1]$$

Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output

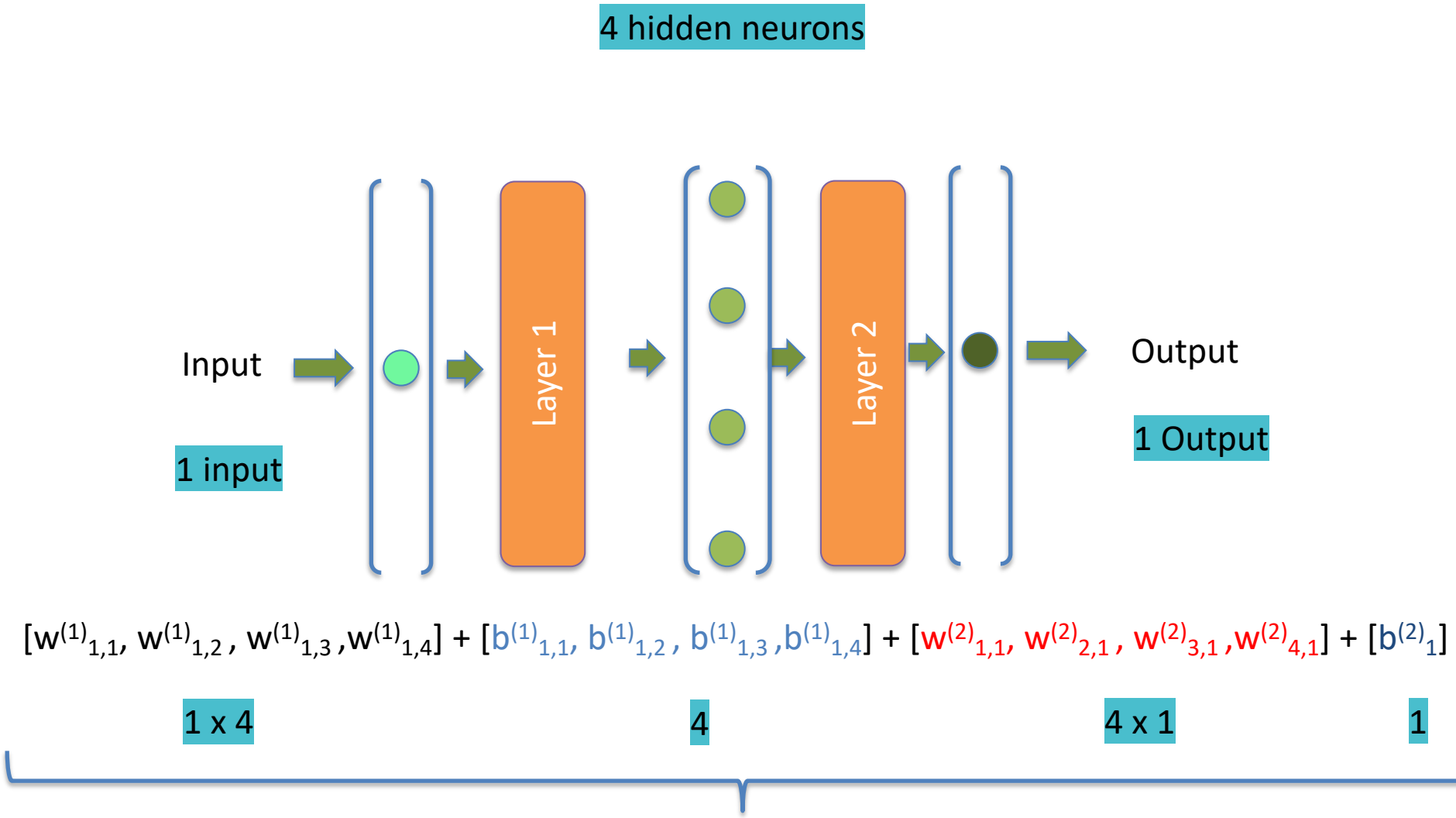


$$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, w^{(1)}_{1,3}, w^{(1)}_{1,4}] + [b^{(1)}_{1,1}, b^{(1)}_{1,2}, b^{(1)}_{1,3}, b^{(1)}_{1,4}] + [w^{(2)}_{1,1}, w^{(2)}_{2,1}, w^{(2)}_{3,1}, w^{(2)}_{4,1}] + [b^{(2)}_1]$$

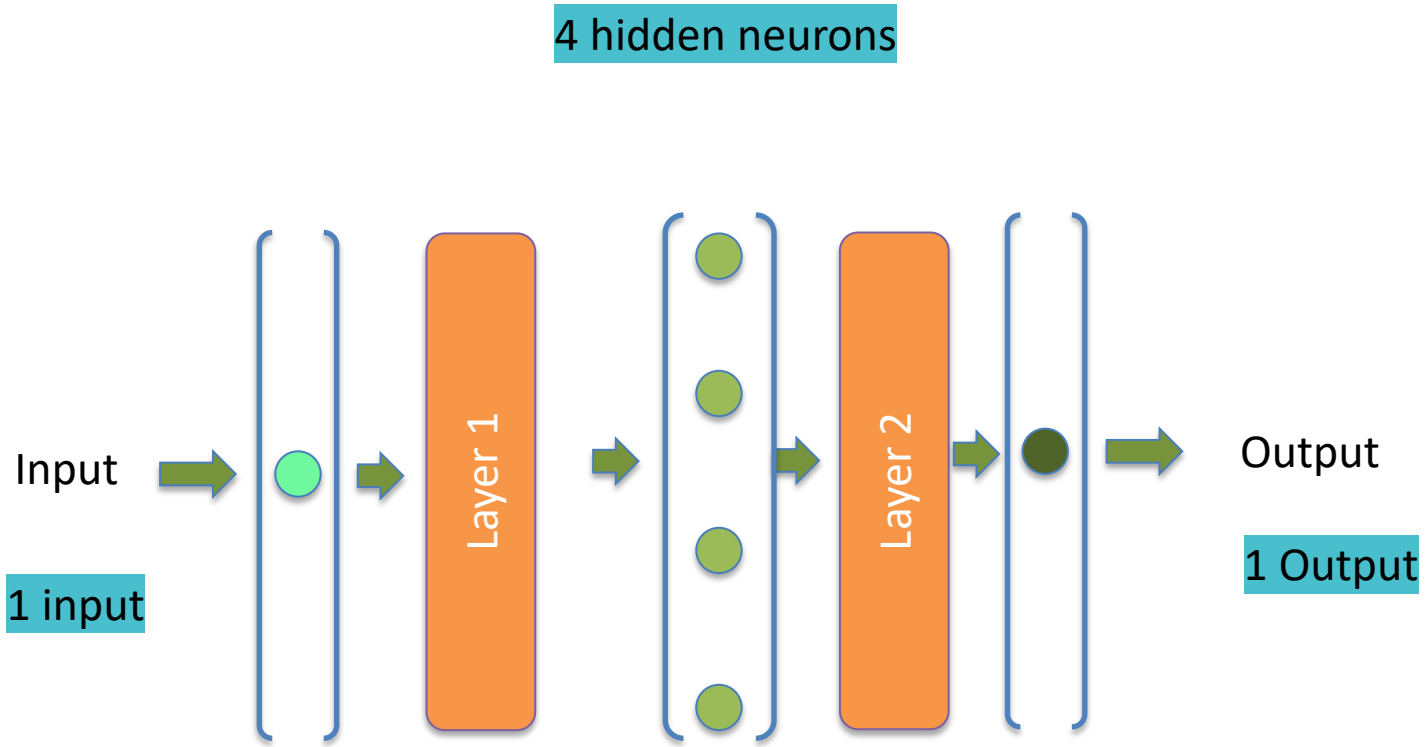
1 x 4 | 4 | 4 x 1 | 1

Total amount of weights = 13

Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output



Let's start with a simple 1-hidden layer FFNN: 1 input, 4 hidden, 1 output



$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, w^{(1)}_{1,3}, w^{(1)}_{1,4}] + [b^{(1)}_{1,1}, b^{(1)}_{1,2}, b^{(1)}_{1,3}, b^{(1)}_{1,4}] + [w^{(2)}_{1,1}, w^{(2)}_{2,1}, w^{(2)}_{3,1}, w^{(2)}_{4,1}] + [b^{(2)}_{1}]$

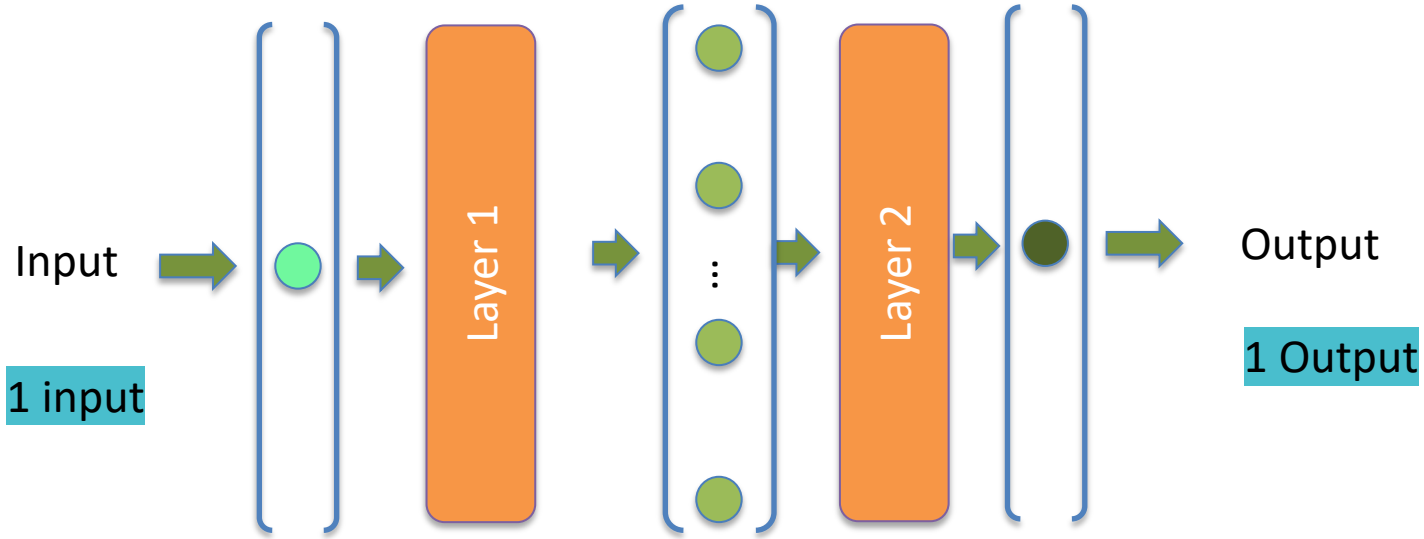
1 x 4 — 4 — 4 x 1 — 1

Weights first layer = 8

Weights second layer = 5

Let's start with a simple 1-hidden layer FFNN: 1 input, 16 hidden, 1 output

16 hidden neurons

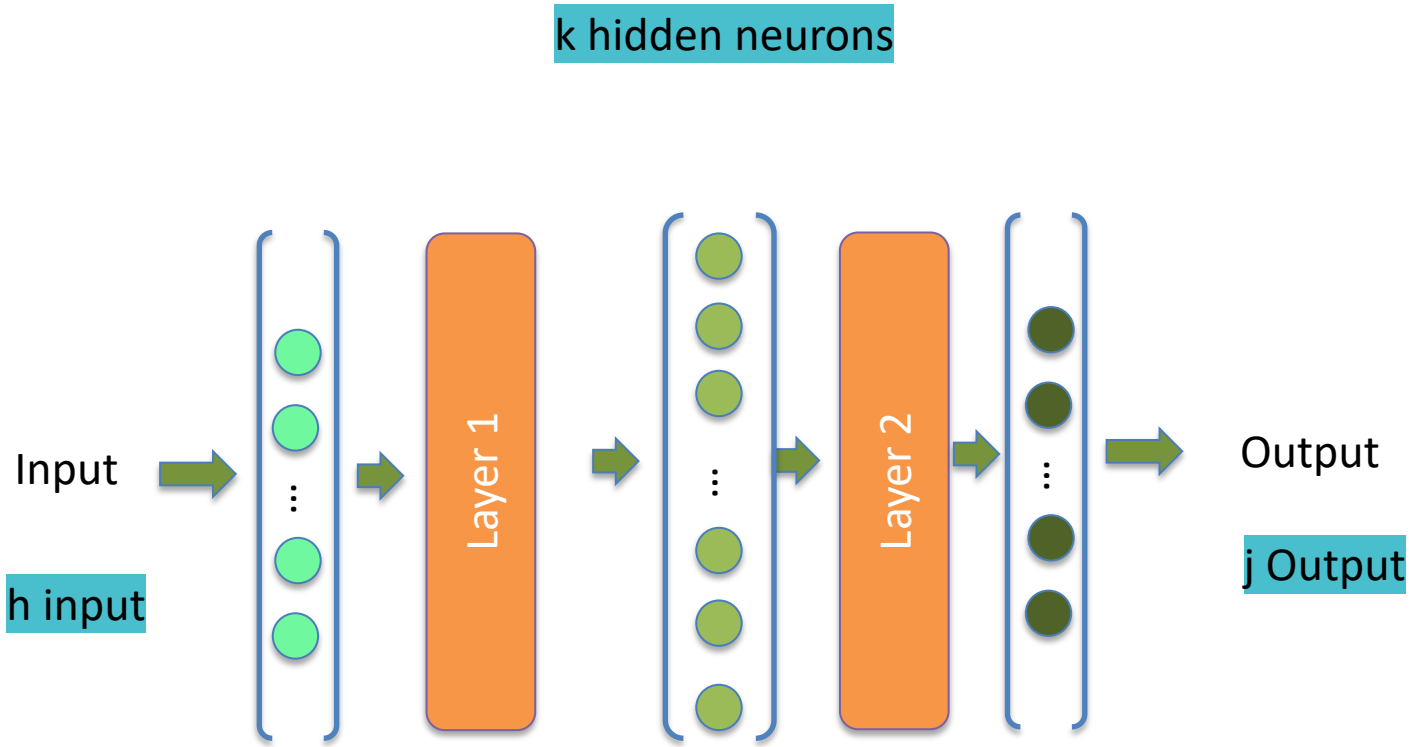


$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, \dots, w^{(1)}_{1,16}] + [b^{(1)}_{1,1}, b^{(1)}_{1,2}, \dots, b^{(1)}_{1,16}] + [w^{(2)}_{1,1}, w^{(2)}_{2,1}, \dots, w^{(2)}_{16,1}] + [b^{(2)}_{1}]$

1 x 16 — 16 — Weights first layer = 32

16 x 1 — 1 — Weights second layer = 17 → Total amount of weights = 49

Generalizing the memory demand in multi-input/multi-output NNs



$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, \ldots, w^{(1)}_{h,k-1}, w^{(1)}_{h,k}] + [b^{(1)}_{1,1}, \ldots, b^{(1)}_{1,k}] + [w^{(2)}_{1,1}, w^{(2)}_{1,2}, \ldots, w^{(2)}_{k,j-1}, w^{(2)}_{k,j}] + [b^{(2)}_{1,\ldots}, b^{(2)}_{j}]$

h x k

k

k x j

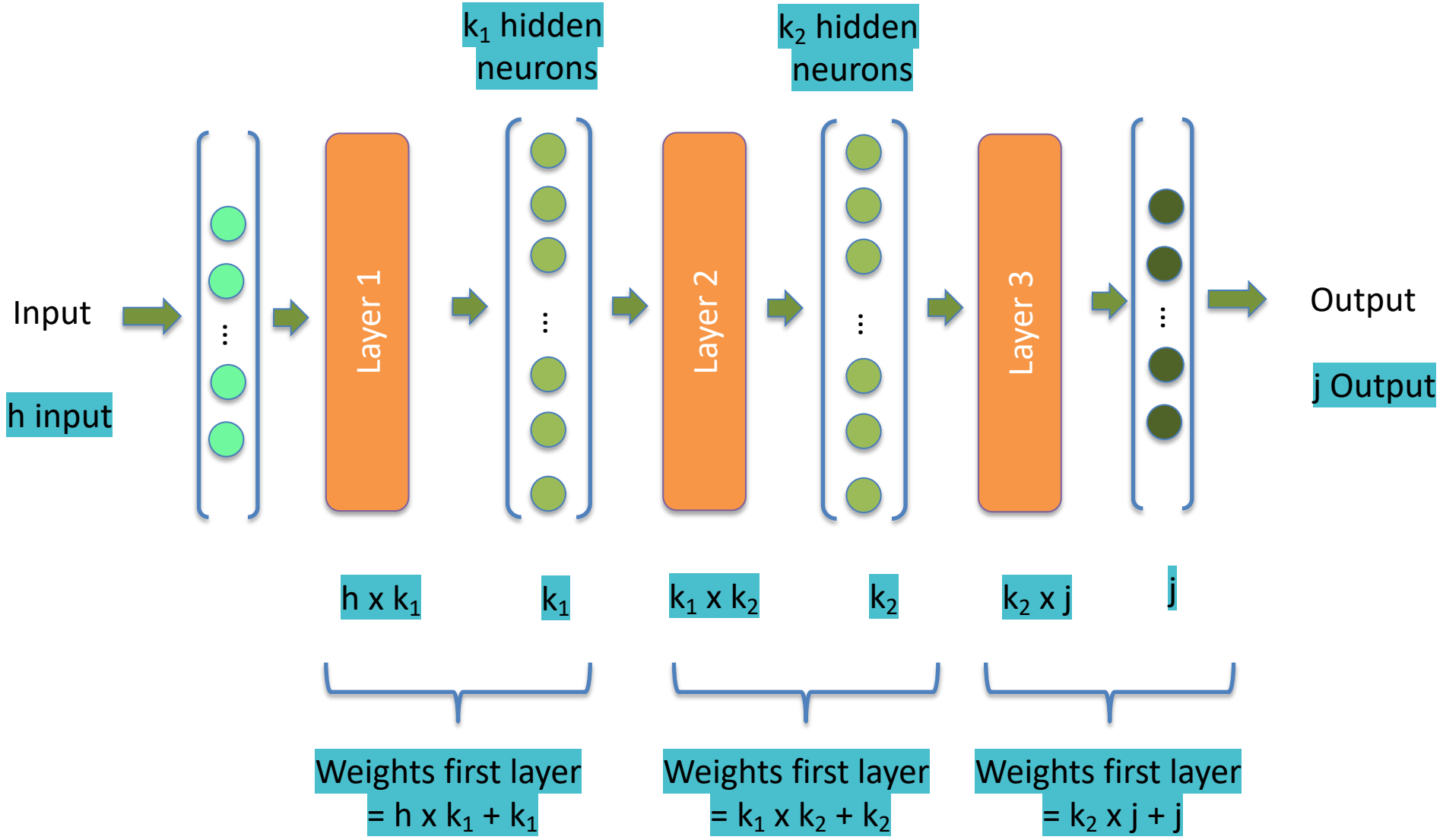
j

Weights first layer = h x k + k

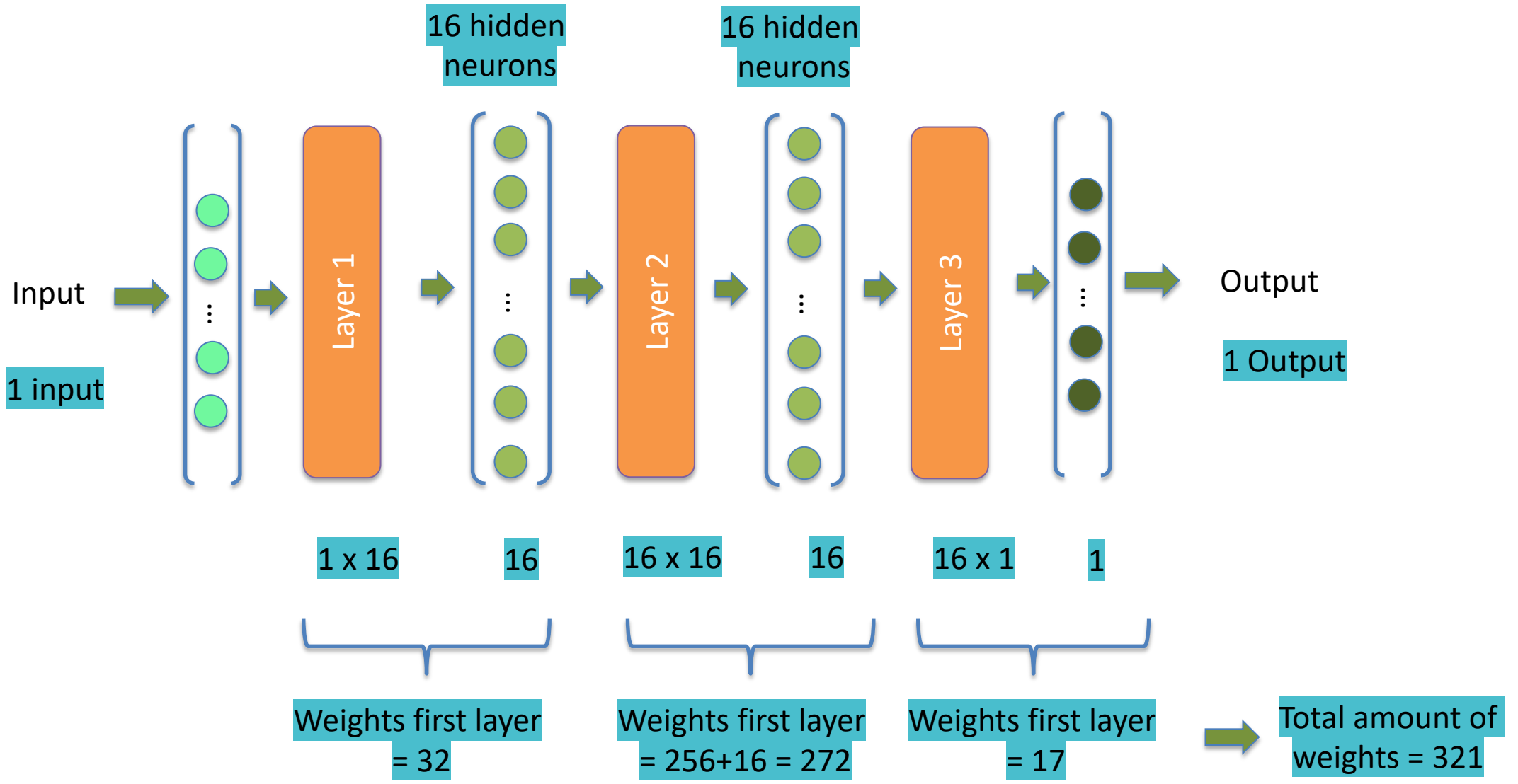
Weights second layer = k x j + j

What about multiple hidden layers? Let's see with 2 hidden layers

What about multiple hidden layers? Let's see a FFNN with 2 layers



An example of a 1-16-16-1 NN: the memory demand



An example with Tensorflow

An example of neural network training in tensorflow

This is a tutorial for building a simple tensorflow model. We will need:

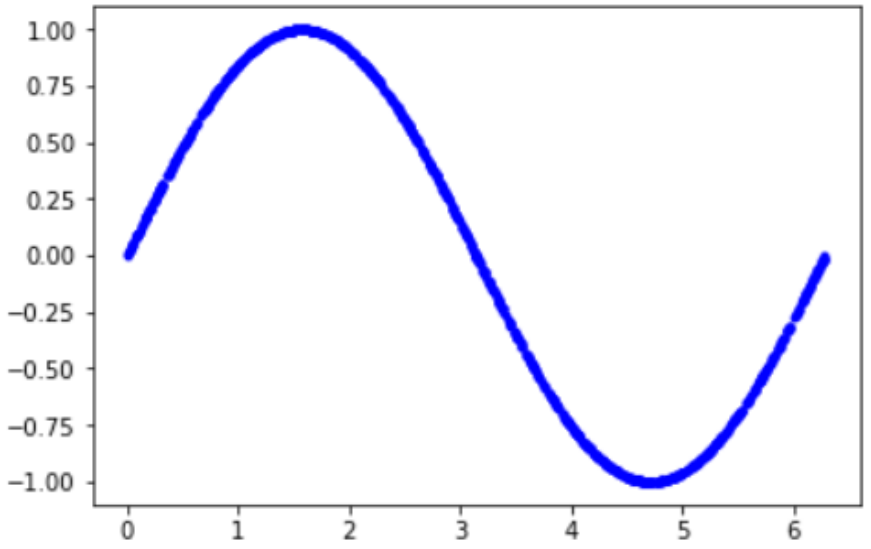
- A task to solve
- Some (generated) data
- The libraries that we need imported in your environment

```
# TensorFlow is an open source machine learning library
!pip install tensorflow==2.0
import tensorflow as tf
# Numpy is a math library
import numpy as np
# Matplotlib is a graphing library
import matplotlib.pyplot as plt
# math is Python's math library
import math
```

Task and data

- Goal: To train a network to model data generated by a sine function.
- This will result in a model that can take a value, *x*, and predict its sine, *y*.
- Since *y* is a continous dependent variable, this is a **Regression** task
- For this example, we're using some code to generate a dataset.

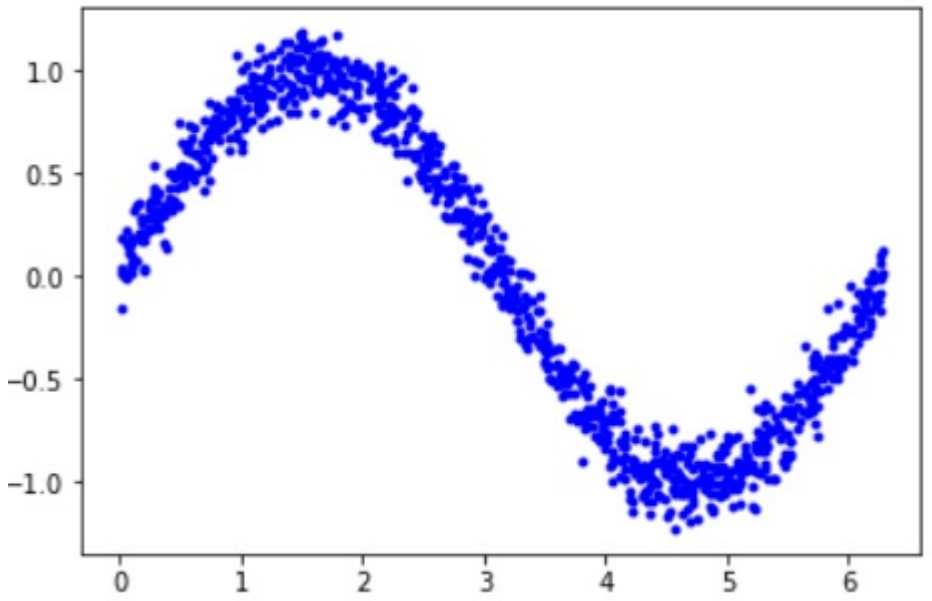
```
# We'll generate this many sample datapoints
SAMPLES = 1000

# Set a "seed" value, so we get the same random numbers each time we run this
# notebook. Any number can be used here.
SEED = 1337
np.random.seed(SEED)
tf.random.set_seed(SEED)

# Generate a uniformly distributed set of random numbers in the range from
# 0 to 2π, which covers a complete sine wave oscillation
x_values = np.random.uniform(low=0, high=2*math.pi, size=SAMPLES)

# Shuffle the values to guarantee they're not in order
np.random.shuffle(x_values)

# Calculate the corresponding sine values
y_values = np.sin(x_values)
```



Add some noise

- Since it was generated directly by the sine function, our data fits a nice, smooth curve.
- Machine learning models are good at extracting meaning from messy, real world data.
- To demonstrate this, we can add some noise to our data.

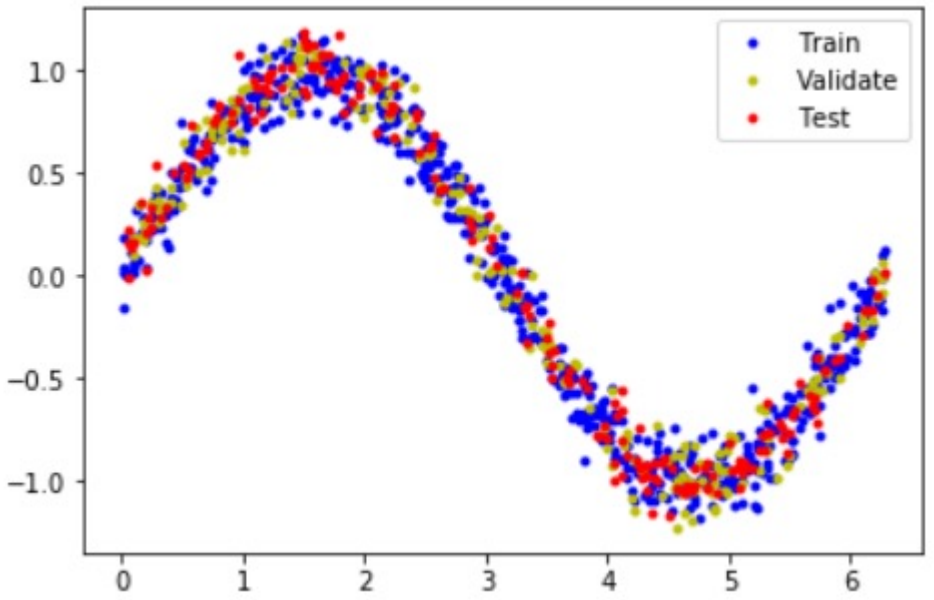
```
# Add a small random number to each y value
y_values += 0.1 * np.random.randn(*y_values.shape)
```



Train – Validation – Test split

- To evaluate the accuracy of the trained model, we'll need to compare its predictions to real data.
- This evaluation happens during training (validation) and after training (testing)
- It's important in both cases that we use data not already used to train the model.
- We'll reserve 20% of our data for validation, and another 20% for testing.

```
# We'll use 60% of our data for training and 20% for testing. The remaining 20%
# will be used for validation. Calculate the indices of each section.
TRAIN_SPLIT =  int(0.6 * SAMPLES)
TEST_SPLIT = int(0.2 * SAMPLES + TRAIN_SPLIT)

# Use np.split to chop our data into three parts.
# The second argument to np.split is an array of indices where the data will be
# split. We provide two indices, so the data will be divided into three chunks.
x_train, x_validate, x_test = np.split(x_values, [TRAIN_SPLIT, TEST_SPLIT])
y_train, y_validate, y_test = np.split(y_values, [TRAIN_SPLIT, TEST_SPLIT])

# Double check that our splits add up correctly
assert (x_train.size + x_validate.size + x_test.size) ==  SAMPLES
```



Define the network

- Define an empty model with **`Sequential()`**
- For each layer in the network, we call **`model.add(layer)`**
- To create a Dense layer we call **`layers.Dense()`**
- The parameters of this function are :
  - The number of neurons
  - The activation (if present)
  - The input shape, if it's the first layer of the network (the others are implied)

```
# We'll use Keras to create a simple model architecture
from tensorflow.keras import layers
model_1 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons". The
# neurons decide whether to activate based on the 'relu' activation function.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Final layer is a single neuron, since we want to output a single value
model_1.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for regression
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Print a summary of the model's architecture
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 49
Trainable params: 49
Non-trainable params: 0
_________________________________________________________________
```

Define the network - 2

- After all the layer have been added, we need to compile the model with **`model.compile()`**
- The parameters are:
  - The optimizer used to train
  - The loss function
  - The metrics used in evaluating the network
- We can print a summary of the model with **`model.summary()`**

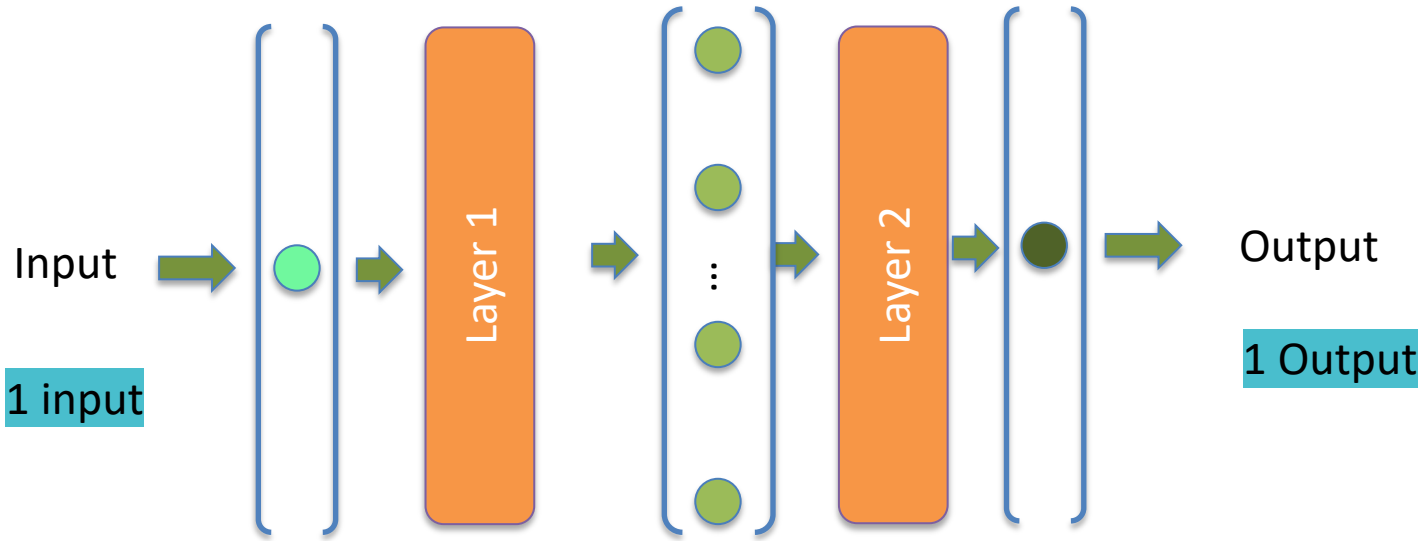
```
# We'll use Keras to create a simple model architecture
from tensorflow.keras import layers
model_1 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons". The
# neurons decide whether to activate based on the 'relu' activation function.
model_1.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# Final layer is a single neuron, since we want to output a single value
model_1.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for regression
model_1.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Print a summary of the model's architecture
model_1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 49
Trainable params: 49
Non-trainable params: 0
_________________________________________________________________
```

A recap of the memory demand of a 1-16-1 NN

16 hidden neurons

Already seen ..



$[w^{(1)}_{1,1}, w^{(1)}_{1,2}, \dots, w^{(1)}_{1,16}] + [b^{(1)}_{1,1}, b^{(1)}_{1,2}, \dots, b^{(1)}_{1,16}] + [w^{(2)}_{1,1}, w^{(2)}_{2,1}, \dots, w^{(2)}_{16,1}] + [b^{(2)}_{1}]$

1 x 16 — 16 — 16 x 1 — 1

Weights first layer = 32

Weights second layer = 17 → Total amount of weights = 49

Training the network

- The training starts by calling **`model.fit()`**
- We need to specify:
  - The training data **`x_train`**
  - The labels **`y_train`**
  - The number of epochs
  - The batch size
  - The validation data

```
# Train the model on our training data while validating on our validation set
history_1 = model_1.fit(x_train, y_train, epochs=1000, batch_size=16,
                        validation_data=(x_validate, y_validate))
```

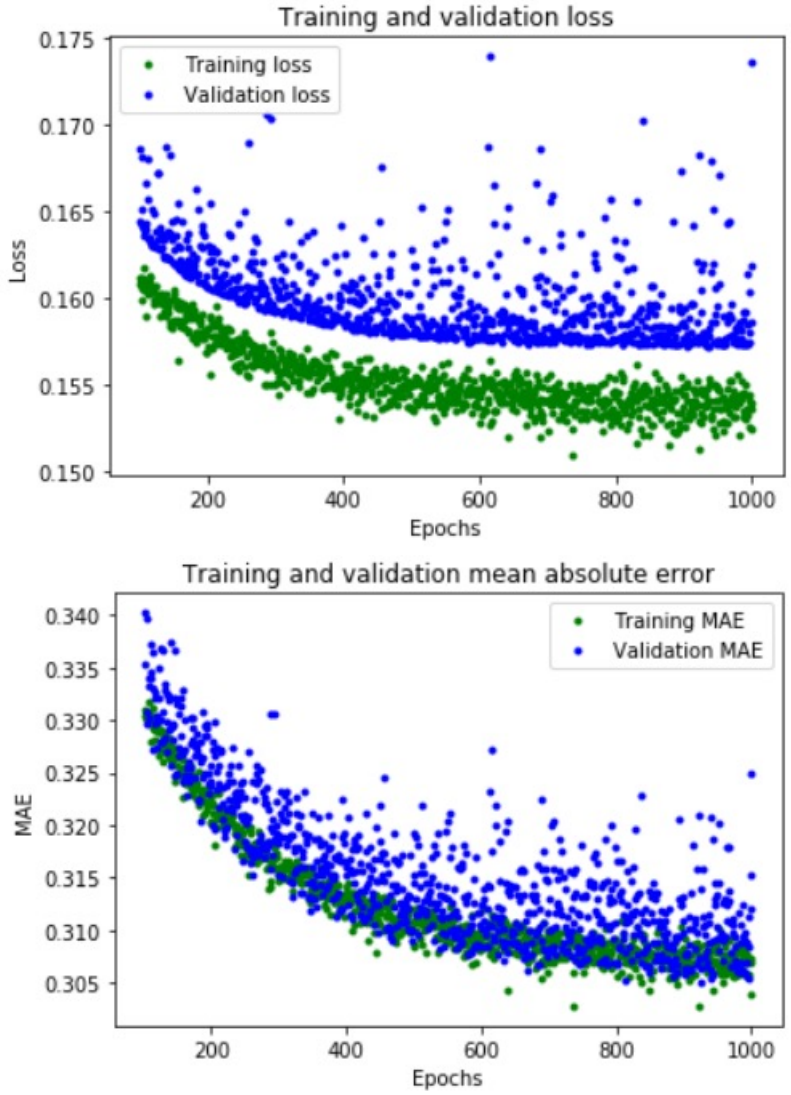
```
Train on 600 samples, validate on 200 samples
Epoch 1/600
600/600 [==============================] - 1s 1ms/sample - loss: 0.6993 - mae: 0.7257 - val_loss: 0.4758 - val_mae: 0.6040
Epoch 2/600
600/600 [==============================] - 0s 153us/sample - loss: 0.4000 - mae: 0.5489 - val_loss: 0.3766 - val_mae: 0.5306
...Epoch 599/600
600/600 [==============================] - 0s 150us/sample - loss: 0.0116 - mae: 0.0860 - val_loss: 0.0104 - val_mae: 0.0804
Epoch 600/600
600/600 [==============================] - 0s 150us/sample - loss: 0.0115 - mae: 0.0859 - val_loss: 0.0104 - val_mae: 0.0806
```

Evaluating the model – training and validation loss and MAE

We can retrieve the training and validation loss and MAE from **`history`**, the output of the function **`fit()`**

```
# Draw a graph of the loss, which is the distance between
# the predicted and actual values during training and validation.
loss = history_2.history['loss']
val_loss = history_2.history['val_loss']
```

```
# Draw a graph of mean absolute error, which is another way of
# measuring the amount of error in the prediction.
mae = history_2.history['mae']
val_mae = history_2.history['val_mae']
```



The model working in practice (on the training set)

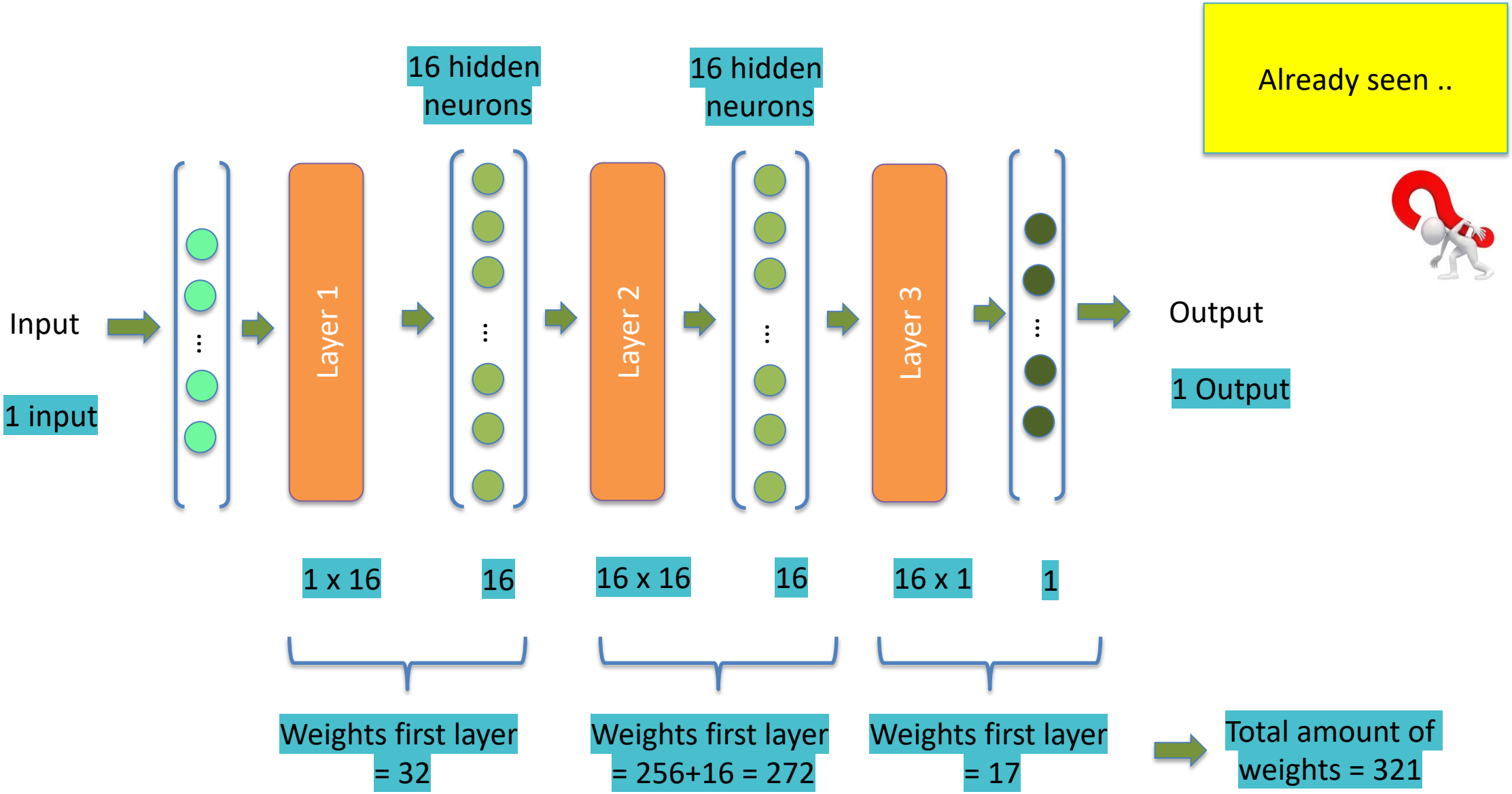
Define a more complex network

- We can define a second, more complex model to understand if we can improve the results
- Add a second dense layer

```
model_2 = tf.keras.Sequential()

# First layer takes a scalar input and feeds it through 16 "neurons". The
# neurons decide whether to activate based on the 'relu' activation function.
model_2.add(layers.Dense(16, activation='relu', input_shape=(1,)))

# The new second layer may help the network learn more complex representations
model_2.add(layers.Dense(16, activation='relu'))

# Final layer is a single neuron, since we want to output a single value
model_2.add(layers.Dense(1))

# Compile the model using a standard optimizer and loss function for regression
model_2.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

# Show a summary of the model
model_2.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 16)                32
_________________________________________________________________
dense_3 (Dense)              (None, 16)                272
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
_________________________________________________________________
```

A recap of the memory demand of a 1-16-16-1 NN

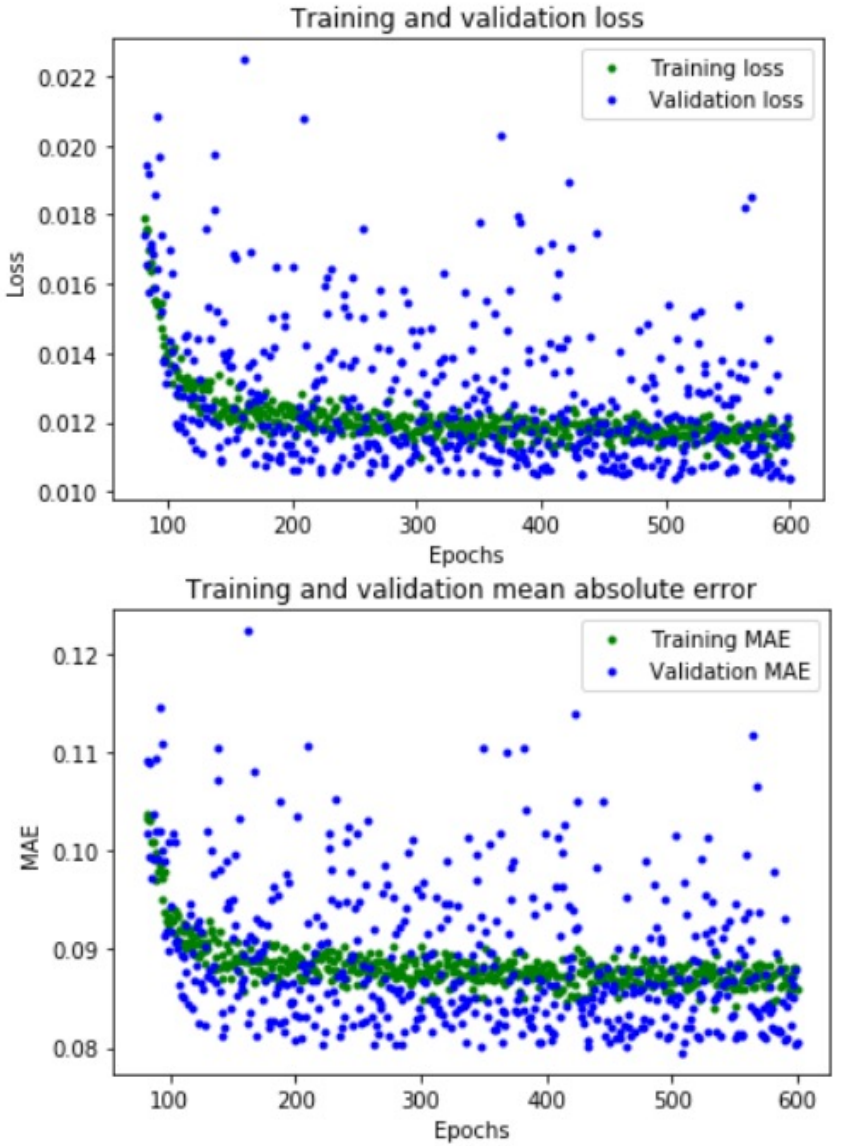


Evaluating the model – training and validation loss and MAE

As before, we retrieve the training and validation loss and MAE from **`history`**, the output of the function **`fit()`**

```
# Draw a graph of the loss, which is the distance between
# the predicted and actual values during training and validation.
loss = history_2.history['loss']
val_loss = history_2.history['val_loss']
```

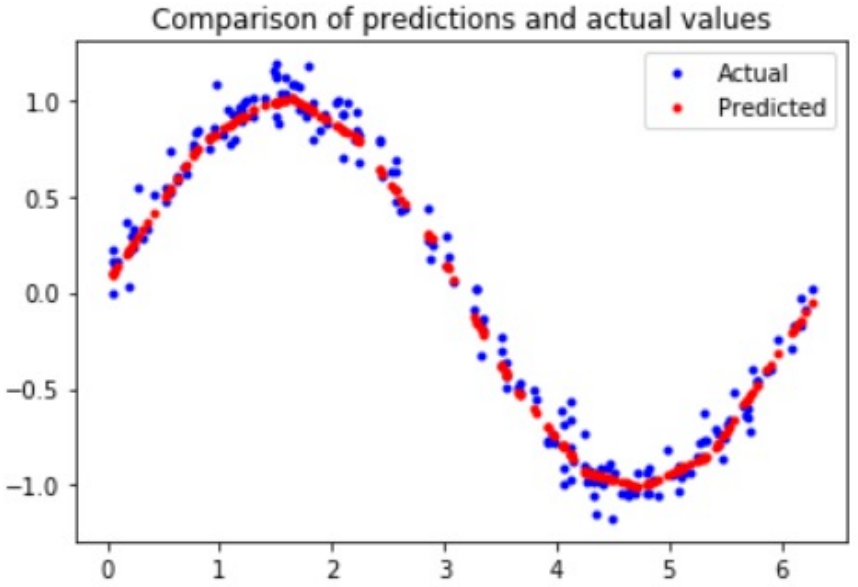
```
# Draw a graph of mean absolute error, which is another way of
# measuring the amount of error in the prediction.
mae = history_2.history['mae']
val_mae = history_2.history['val_mae']
```



Test the model on the test set

- Finally, we use the function **`model.predict()`** to obtain the prediction of the trained model on the test set **`x_test`**
- The results improved a lot from the single layer model!

```
# Calculate and print the loss on our test dataset
loss = model_2.evaluate(x_test, y_test)

# Make predictions based on our test dataset
predictions = model_2.predict(x_test)
```



ML is not just Neural Networks

Several families of machine learning algorithms

- Decision trees/Random forests
- K-Nearest Neighbors
- Support Vector Machines
- …

We will not explore them but
it's important to know they exist