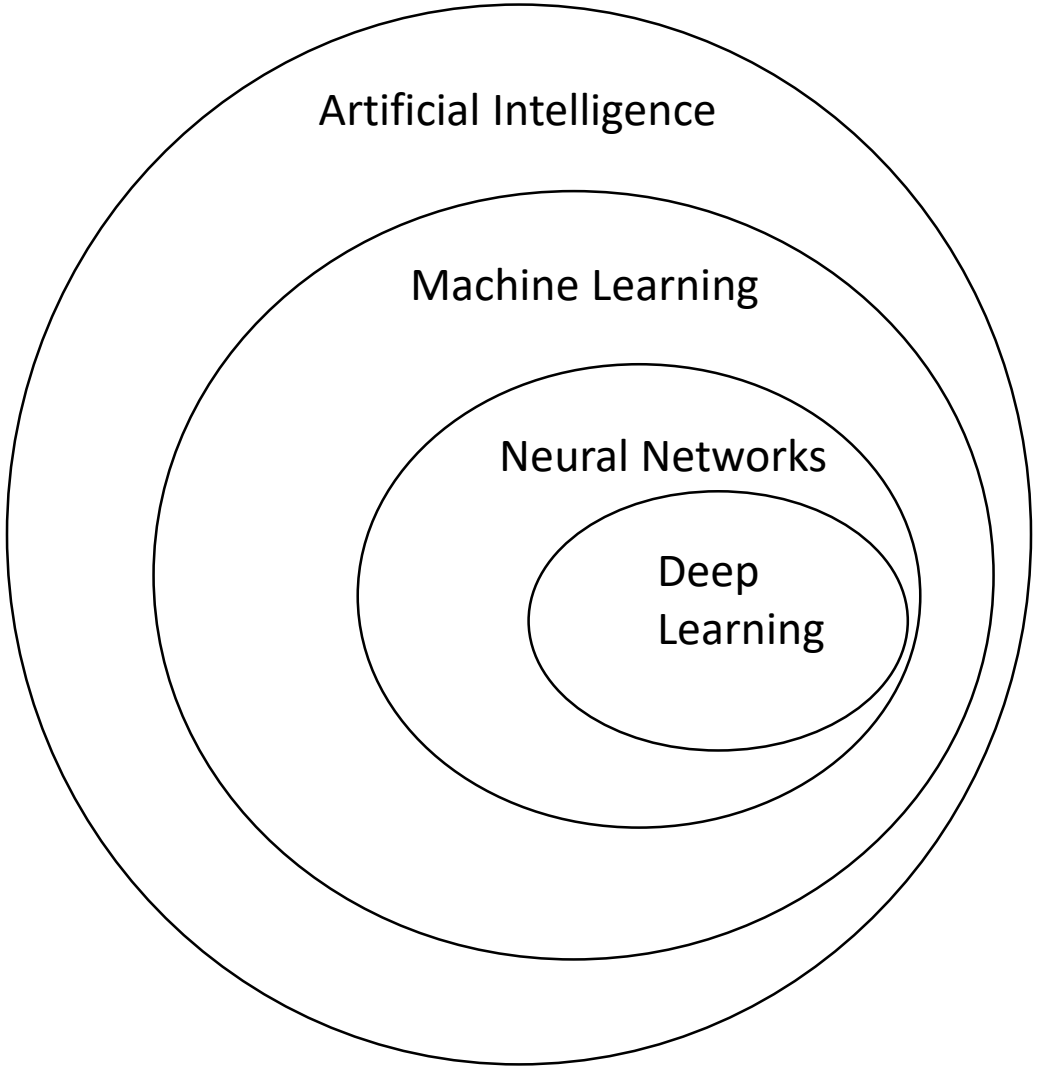
Neural Networks and related fields

- Artificial Intelligence
  - designing computer systems capable of tasks thought to require human intelligence
- Machine Learning
  - designing computer systems capable of improved performance on learning from data, without the need of explicit programming
- Neural Networks
  - family of machine-learning algorithms providing input-output mappings by using successive computational layers
- Deep Learning
  - employs NNs with a large number of layers to attack complex learning tasks

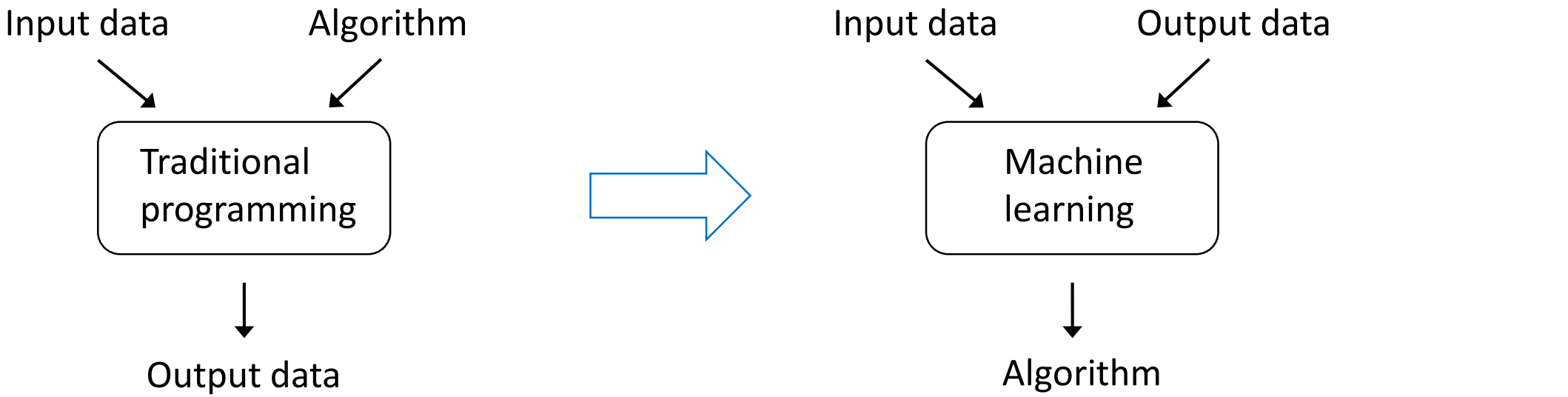


Machine learning in health technology

- Analyzing unstructured patient data
- Informing diagnosis and treatment decisions
- Optimizing patient selection in clinical trial matching
- Reducing drug discovery times
- Improving communication between patients and healthcare providers
- Providing virtual assistance to patients

…

Machine learning approach to problem solving

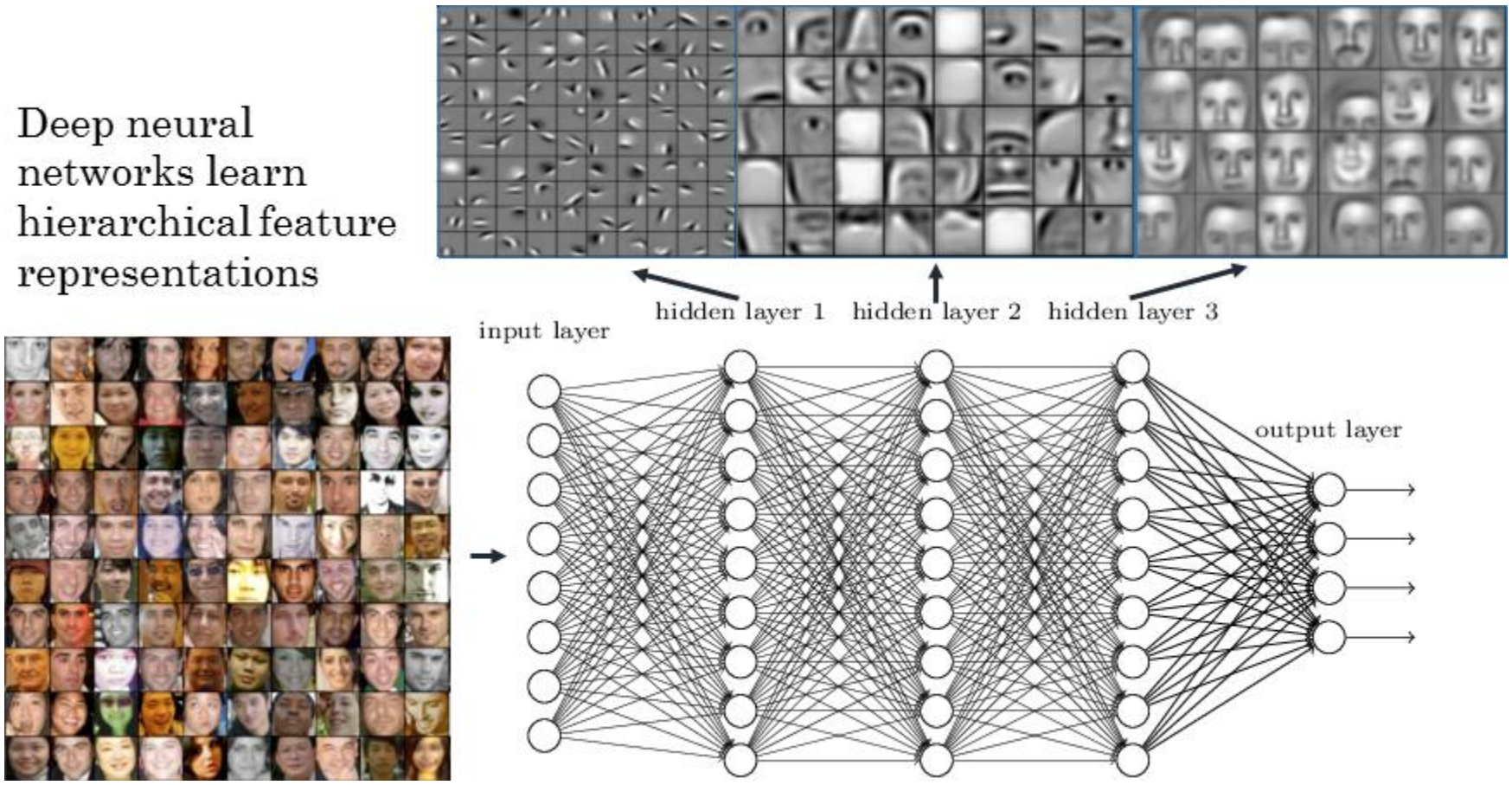


The algorithm needed to solve the problem (figuring out the correct output for a particular input) has to be expressed as an algorithm, and supplied to the computer system by a human expert.

In machine learning, the computer system is trained with several examples of data. The system learns the necessary algorithm relating inputs to outputs without explicit programming. After training, the algorithm can be used with previously unseen data.

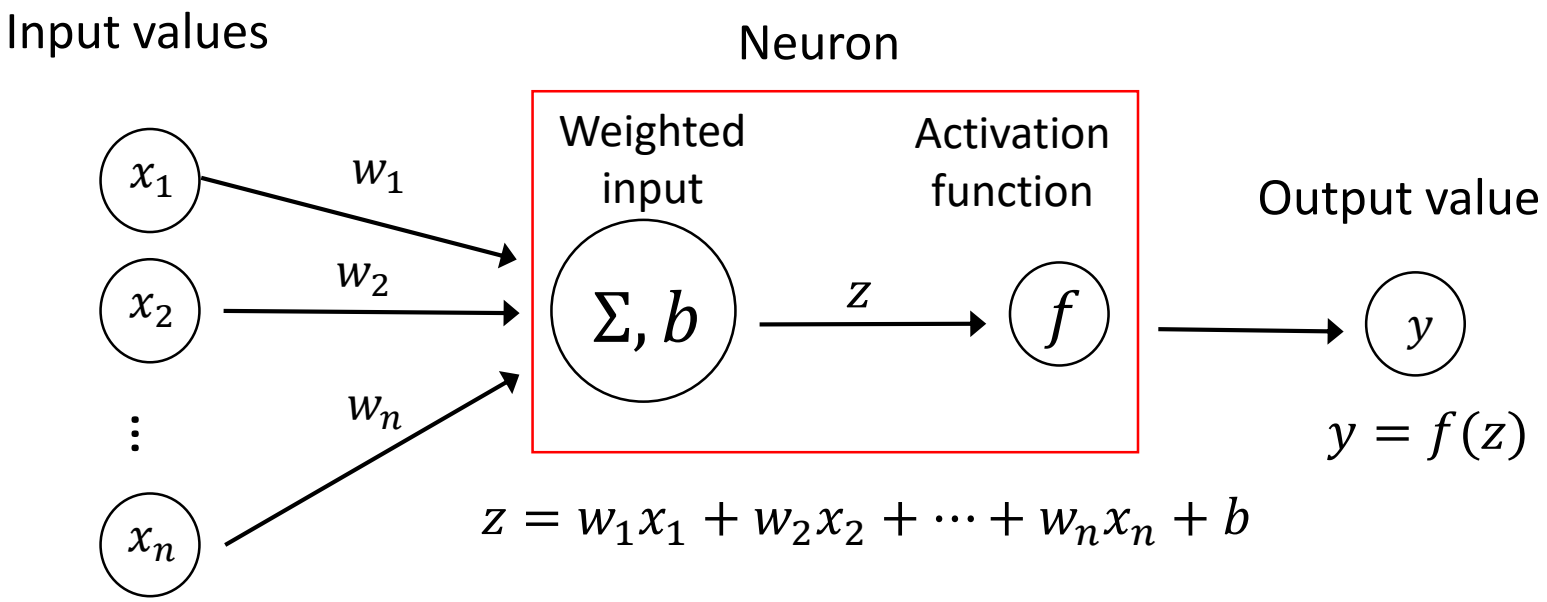
Feature engineering by deep learning

Deep neural networks enable sophisticated machine learning with successive layers of representations.

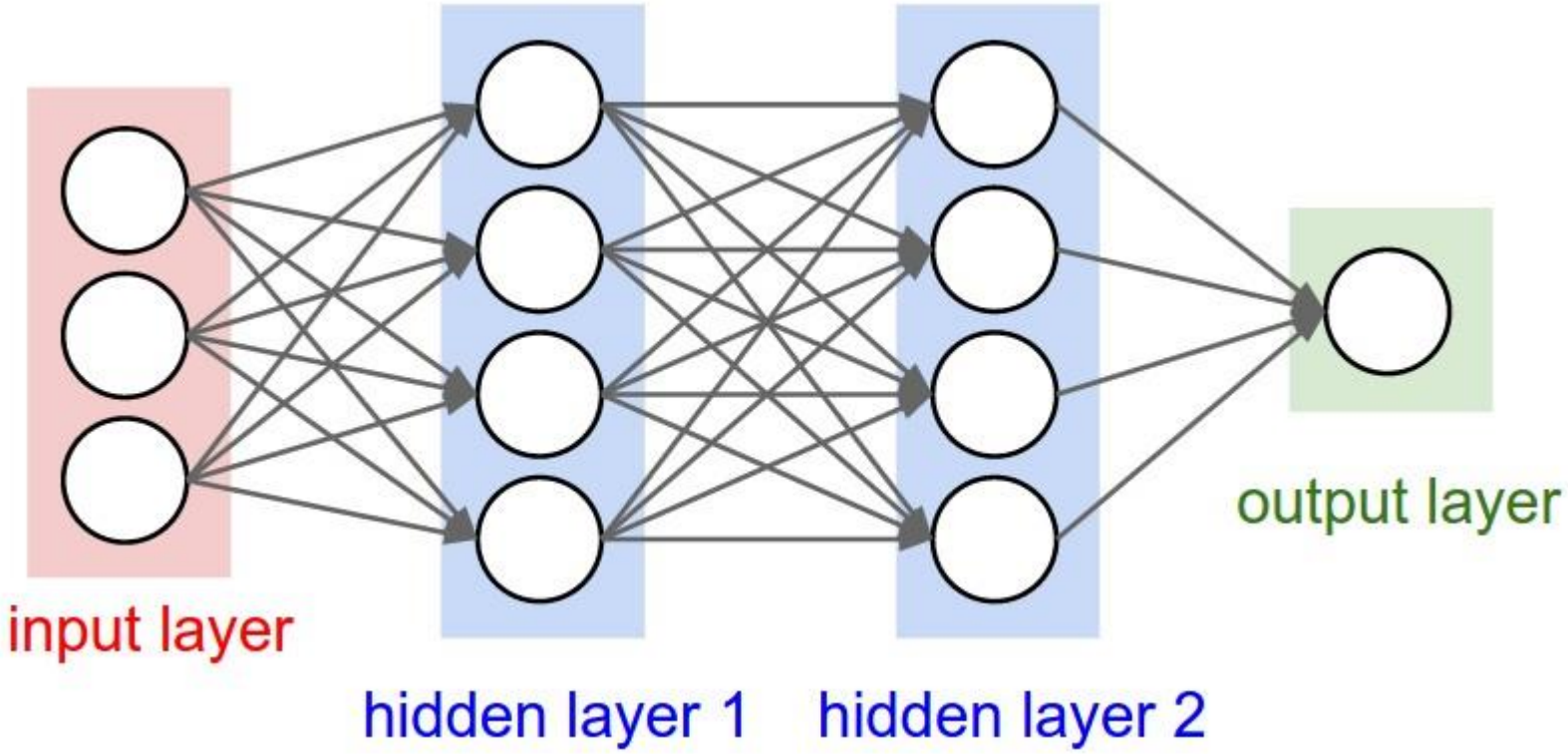


Basic unit of neural networks

Neural networks provide a very general way of representing (almost arbitrarily complex) algorithms relating input data to output data. They contain collections of similar units, neurons, performing simple mathematical operations, and arranged in connected layers. Each neuron takes as an input a collection of output values (real numbers) from neurons in the preceding layer. First, it computes a weighted sum of these input values, adding a possible bias value. It then feeds this intermediate value to a nonlinear activation function, which produces the final output value for the neuron.



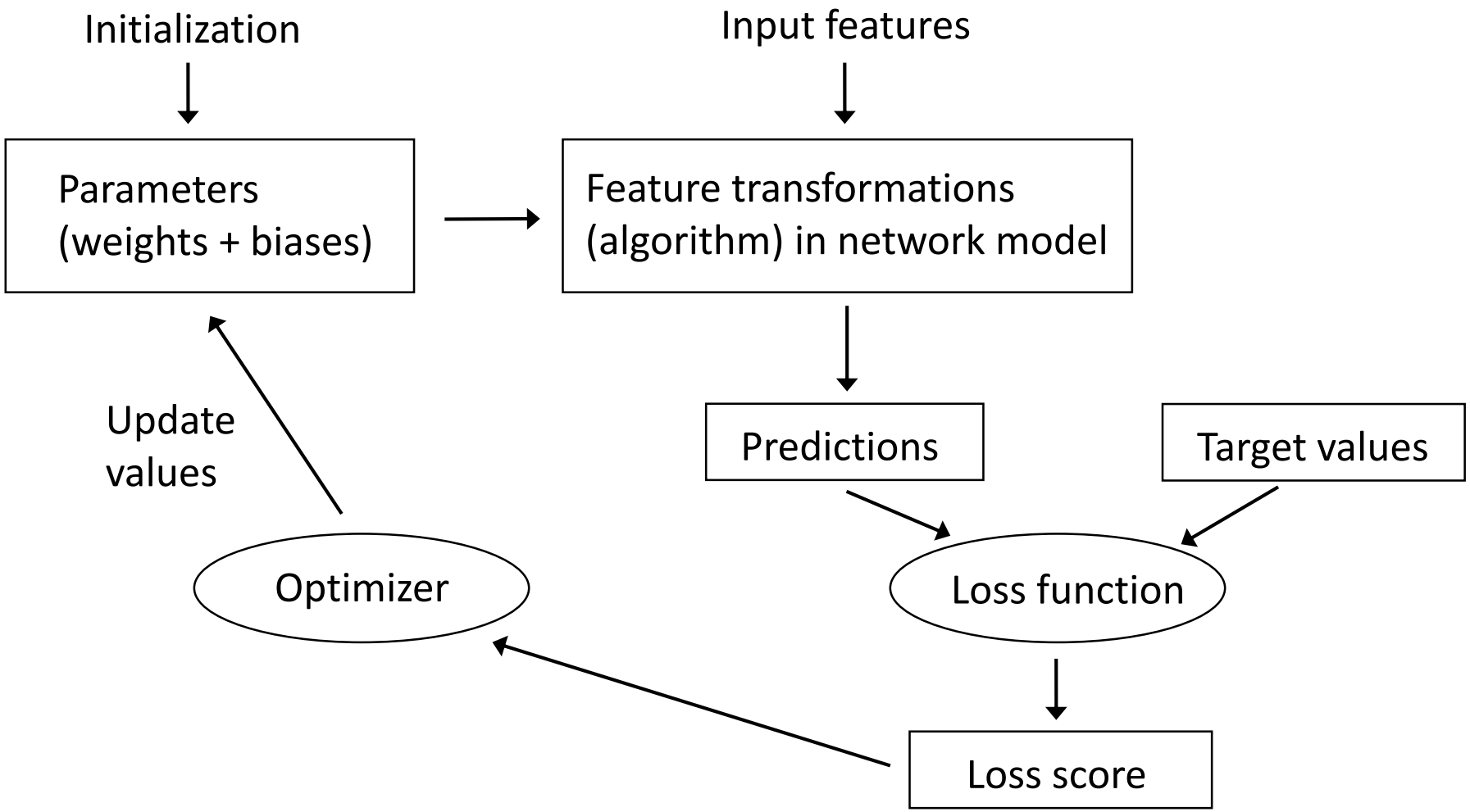
Structure of a simple neural network



Question: How many free parameters (weights and biases) does this network contain?

Each hidden-layer neuron has its own individual set of weights and biases. However, all neurons in a given layer share the same activation function. The number of hidden layers (the depth of the network) and the number of neurons in any single layer are hyperparameters selected by the programmer.

Problem solving with neural networks



Example: classifying handwritten digits

- Python deep learning library Keras

- Training input data: 60000 grayscale images
  - 28 x 28 = 784 pixels
  - Numpy array of shape $(60000, 28, 28)$
  - integers of type $\mathrm{uint8}$ in range 0 … 255

- Training target data: 60000 digit labels
  - Numpy array of integers ranging from 0 to 9

- Test data: additional 10000 images + labels
  - not used in training
  - needed for assessing the ability of the model to classify previously unseen data

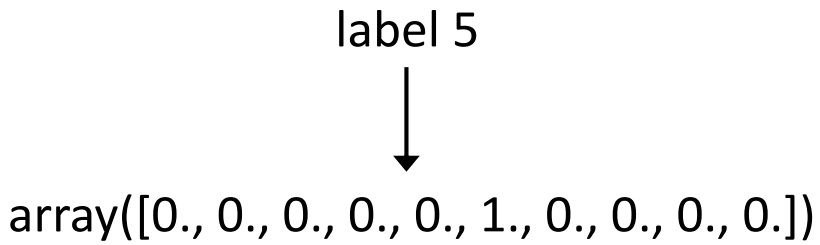
```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images,
                test_labels) = mnist.load_data()
```

Data preprocessing

- Image data is converted from integer arrays of shape $(60000, 28, 28)$ to $\mathrm{float32}$ arrays of shape $(60000, 784)$ and data values are scaled (by dividing with 255) to range 0 … 1

- Data labels are converted from integer values to *one-hot* encoded arrays of ten elements (one for each digit): the target-digit element has value 1, others 0

```
train_images = train_images.reshape((60000, 784))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 784))
test_images = test_images.astype('float32') / 255

from tensorflow.keras.utils import to_categorical

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```



Network architecture

- Chosen network model: input layer of 784 units, a single hidden layer of 512 units, and output layer of 10 units (note that the sample size doesn't have to be specified here)
- Each neuron in a *dense* layer is connected to every neuron output of the preceding layer
- After specifying the numbers of units, Keras defines the necessary parameters (weights and biases) automatically, and initializes them properly

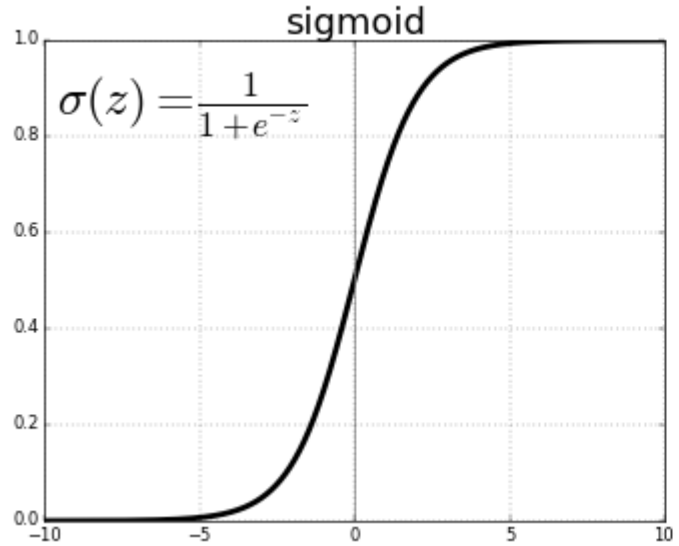
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

network = Sequential()
network.add(Dense(512, activation='relu', input_shape=(784,)))
network.add(Dense(10, activation='softmax'))
```

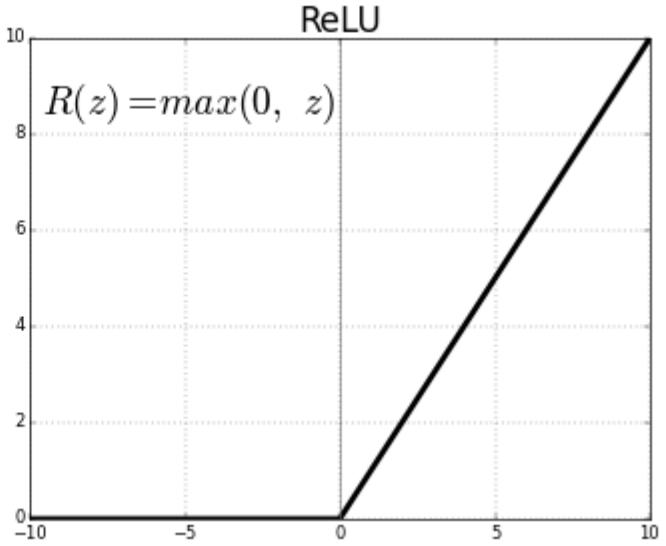
Choosing appropriate values for hyperparameters like the number of hidden layers and the unit numbers in these layers is an important (and difficult) task in building a successful model!

Activation functions

- Network layers (except for the input layer) typically contain a nonlinear activation function; without it, the units would only compute linear functions of their inputs, which would be too restrictive. The added nonlinearity improves the model's capability of expressing complex dependencies, extending its hypothesis space.

- Two often encountered choices are the *sigmoid* function (outputs probability-like values in range 0 … 1), and the recently very popular *rectified linear unit* or ReLU function.





Softmax activation

- In classification problems, the desirable final output in many situations consists of an array of numbers between 0 and 1, interpreted as probabilities to belong in a particular class.

- Such an activation function should take in any n numbers $z_1, z_2, \dots, z_n$, and output n numbers $y_1, y_2, \dots, y_n$ so that each one of them is in the range from 0 to 1, with their total sum equal to 1. This can be achieved with the *softmax* function

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \text{with} \quad y_i = \frac{e^{z_i}}{e^{z_1} + e^{z_2} + \cdots + e^{z_n}}$$

Network compilation

During training, the network output values are compared to the true target values by selecting a loss function, and computing its value for the training set. The loss score of the network is a measure of how well (or poorly) the output values match the target values.

The role of the optimizer is to iteratively update the parameters (weights and biases) of the network so as to reduce the loss score associated with the training set. Different optimizers have slightly different strategies to achieve this (RMSProp optimizer is often a good choice).

Finally, in the compilation step the choice of metrics dictates how much information appears on the screen during the training of the network.

```
network.compile(optimizer='rmsprop', loss='categorical_crossentropy',
                metrics=['accuracy'])
```

Network training and testing

Network is now ready for training. Typically, the loss function is not computed with the entire training set (here 60000 images), but only a small subset of it (*batch*) at a time. A single traversal through the training set (*epoch*) then results in several updates of the weight parameters.

```
network.fit(train_images, train_labels, epochs=5, batch_size=128)
```

After training, the network is tested with previously unseen images (test data). If the test accuracy is much lower than the accuracy with the training set, the model suffers from *overfitting*.

```
test_loss, test_acc = network.evaluate(test_images, test_labels)
print ('test_acc:', test_acc)
```