Data Analysis & Machine Learning: Lecture 09

Dr. Conor D. Rankine

Learning Objectives

At the end of this lecture, you'll be able to answer questions like:

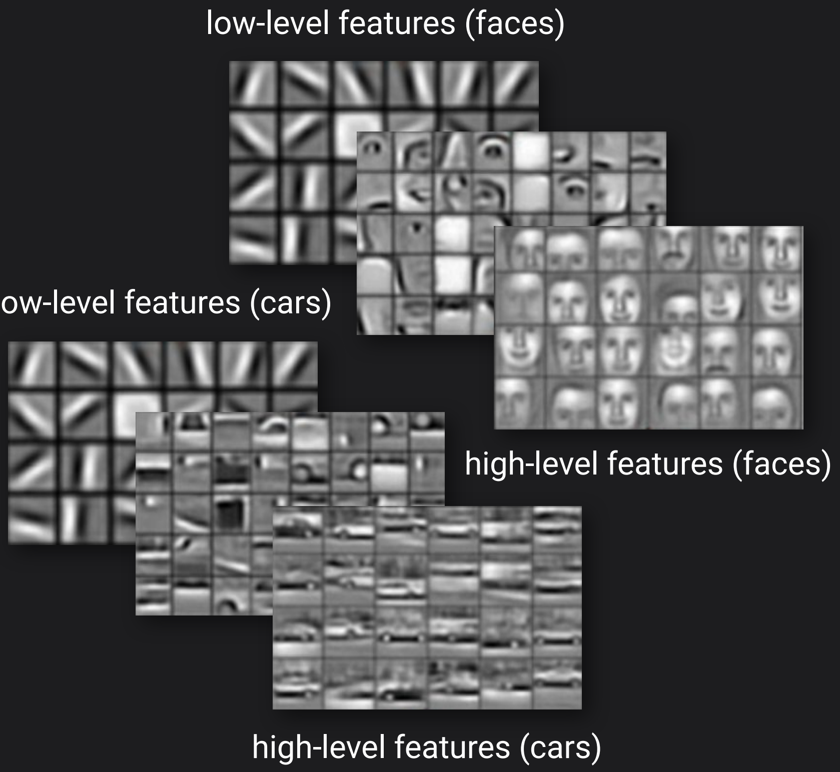
- What is a perceptron, and what are its key components?
- What is a multilayer perceptron or artificial neural network?
- How are artificial neural networks trained using gradient descent?
- What is backpropagation, and how does it work?
- How are regularisation techniques like dropout and early stopping used in training artificial neural networks?

Deep Learning

Deep learning is a sub-field of machine learning that is concerned with **many-layered** (or **deep**) **artificial neural networks**:

- **shallow learning**: model parameters are learned directly from input features, ***X***;
- **deep learning**: (most) model parameters are learned from output of other layers.

Can we learn the relevant features directly from our data without hand-coding them?



Deep Learning

Why is everyone interested in deep learning today?

- **big data:**
    - bigger datasets than ever before;
    - better data collection and storage;
- **hardware:**
    - availability of GPUs and cloud compute;
- **software:**
    - new deep neural network models for structured data (*e.g.* image, sequence);
    - ‘plug-and-play’ Python toolboxes (*e.g.* TensorFlow, PyTorch)

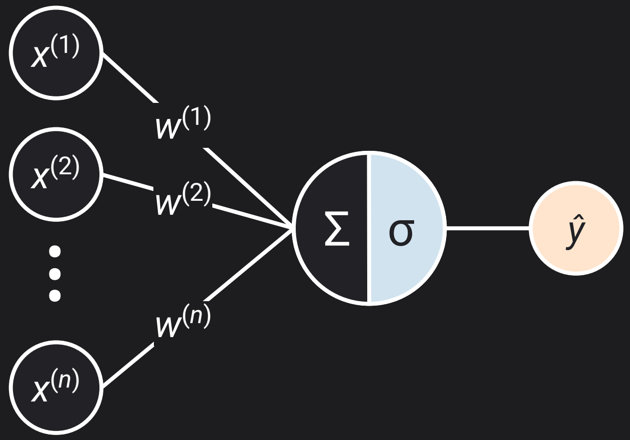
The Perceptron

The **perceptron** is the simplest kind of **artificial neural network**.

$$\hat{y} = \sigma\left(\sum_{i=1}^{n} x^{(i)} w^{(i)}\right)$$

Components:

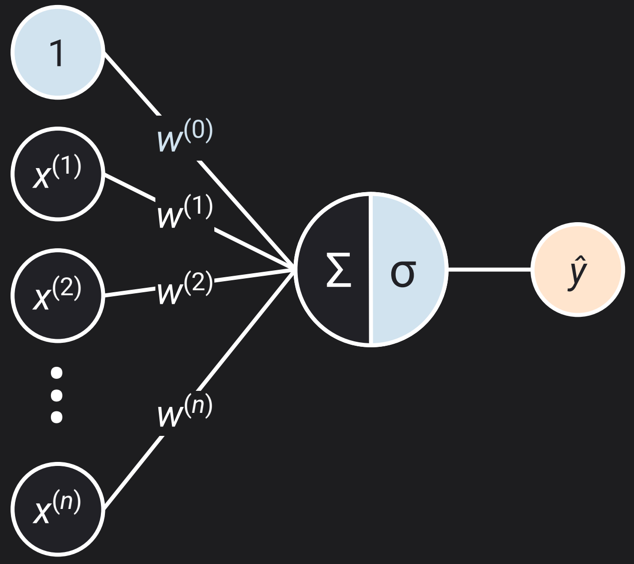
- inputs, $\{x_i\}$
- weights, $\{w_i\}$
- summation function, Σ
- activation function, σ



The Perceptron

The **bias**, $w^{(0)}$, is an additional parameter that complements the weights, $\{w_i\}$.

$$\hat{y} = \sigma\left(w^{(0)} + \sum_{i=1}^{n} x^{(i)} w^{(i)}\right)$$

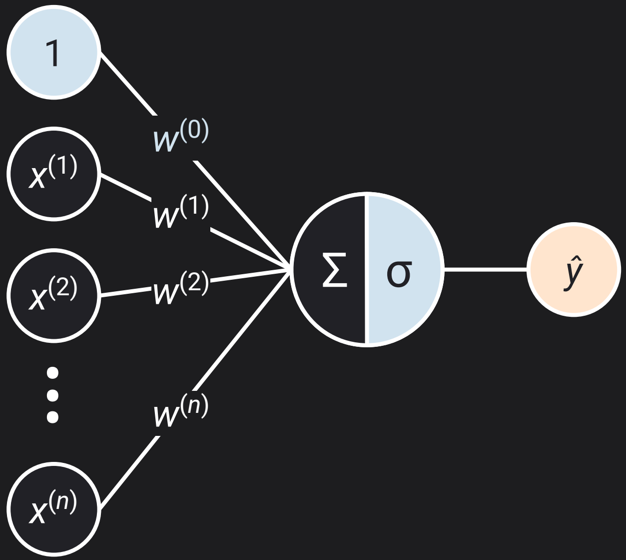


The Perceptron

The mathematics is more compact in a vector representation:

$$\hat{y} = \sigma\left(w^{(0)} + \sum_{i=1}^{n} x^{(i)} w^{(i)}\right) = \sigma\left(w^{(0)} + \mathbf{X}^{\mathrm{T}}\mathbf{W}\right)$$

$$\mathbf{X} = \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(n)} \end{bmatrix} \qquad \mathbf{W} = \begin{bmatrix} w^{(1)} \\ \vdots \\ w^{(n)} \end{bmatrix}$$



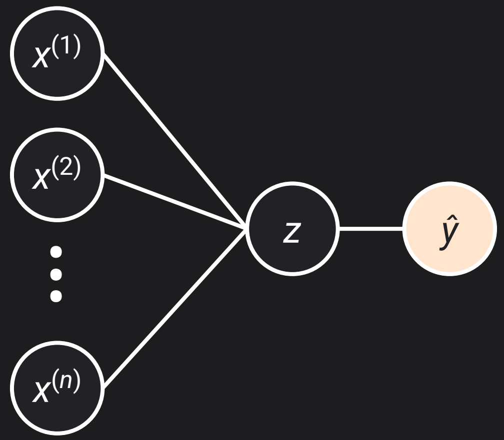
The Perceptron

Let's simplify things further by using z to denote:

- the multiplication of the inputs, $\{x_i\}$, with the weights, $\{w_i\}$;
- their subsequent summation.

$$z = w^{(0)} + \sum_{i=1}^{n} x^{(i)} w^{(i)} = w^{(0)} + \mathbf{X}^{\mathrm{T}}\mathbf{W}$$

$$\hat{y} = \sigma(z)$$

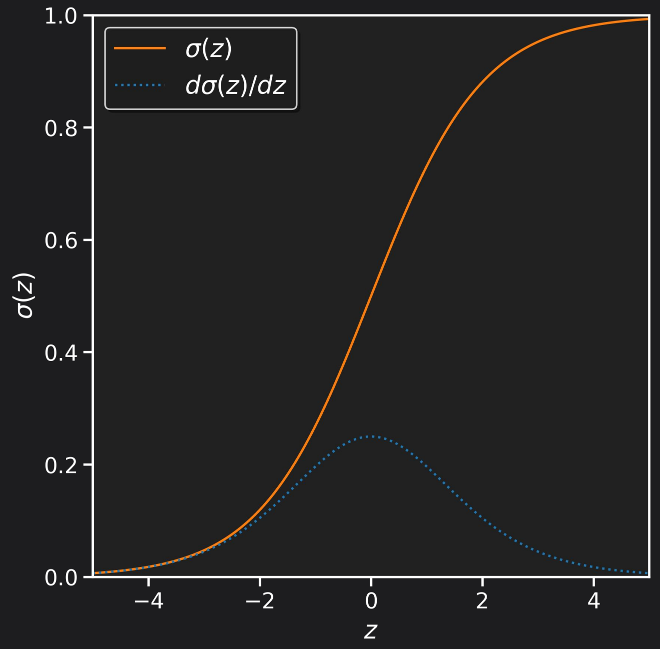


The Perceptron

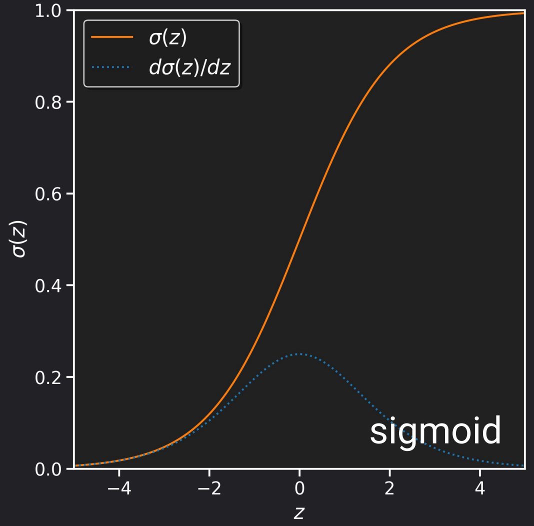
A perceptron with a nonlinear **activation function**, $\sigma(z)$, can separate nonlinearly separable data:

- patterns in data are often nonlinear;
- without a nonlinear activation function, $\hat{y}_i \leftarrow \boldsymbol{X}_i$ is a linear transformation.
- the sigmoid function is a popular nonlinear activation function.

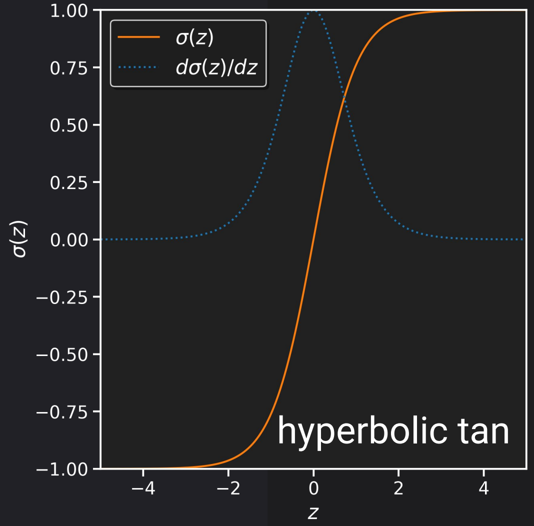
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



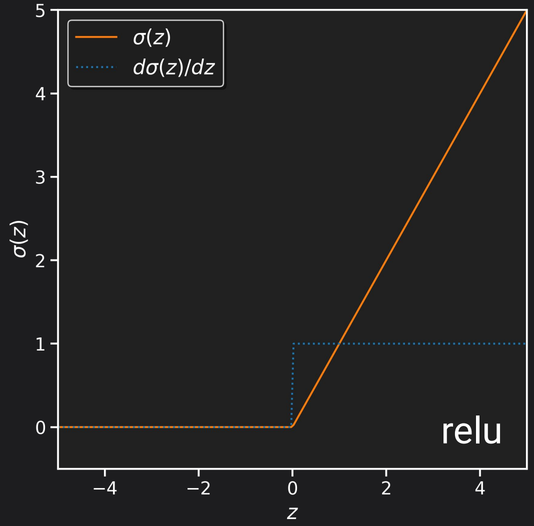
The Perceptron



$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



$$\sigma(z) = \max(0,\ z)$$

The Perceptron

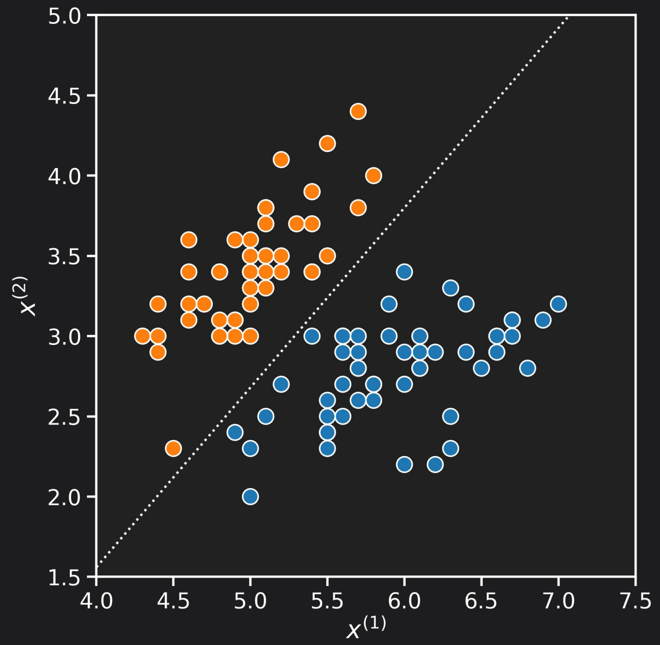
$$\hat{y} = \sigma\left(w^{(0)} + \mathbf{X}^{\mathrm{T}}\mathbf{W}\right)$$

The expression on which σ is applied, *i.e. z*, is the equation of a linear **decision boundary**.

For the iris dataset ($\boldsymbol{X} = \{x^{(0)}, x^{(1)}\}; \hat{y} \in \{0, 1\}$):

$$\hat{y} = \sigma\left(-5 + \begin{bmatrix} x^{(1)} \\ x^{(2)} \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} +1.9 \\ -1.7 \end{bmatrix}\right)$$

$$\hat{y} = \sigma\left(-5 + 1.9x^{(1)} - 1.7x^{(2)}\right)$$



The Perceptron

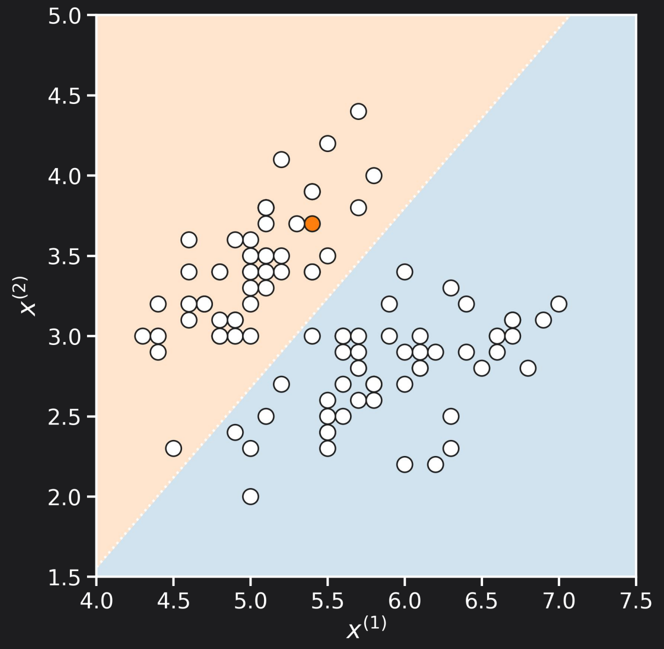
Let's consider an example of a Setosa iris:

- $x^{(1)} = 5.4$
- $x^{(2)} = 3.7$

$$\hat{y} = \sigma\left(-5 + 1.9x^{(1)} - 1.7x^{(2)}\right)$$

$$\hat{y} = \sigma(-5 + (1.9 \times 5.4) - (1.7 \times 3.7))$$

$$\hat{y} = \sigma(-1.0) = 0.27$$



The Perceptron

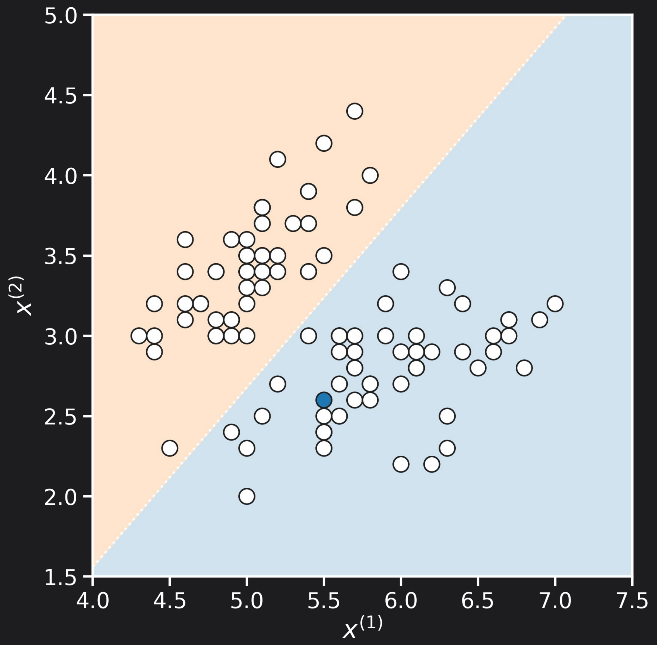
Let's consider an example of a Versicolor iris:

- $x^{(1)} = 5.5$
- $x^{(2)} = 2.6$

$$\hat{y} = \sigma\left(-5 + 1.9x^{(1)} - 1.7x^{(2)}\right)$$

$$\hat{y} = \sigma(-5 + (1.9 \times 5.5) - (1.7 \times 2.6))$$

$$\hat{y} = \sigma(1.0) = 0.73$$



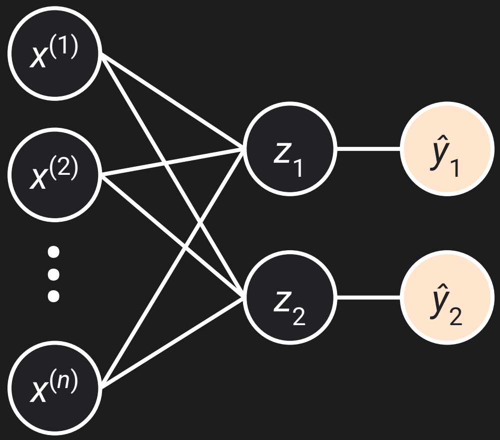
The Multi-Output Perceptron

A perceptron is not limited to a single output.

A **multi-output perceptron** extends the concept to multiclass classification problems.

$$\hat{y}_1 = \sigma(z_1) \qquad z_1 = w^{(0,1)} + \sum_{j=1}^{n} x^{(j)} w^{(j,1)}$$

$$\hat{y}_2 = \sigma(z_2) \qquad z_2 = w^{(0,2)} + \sum_{j=1}^{n} x^{(j)} w^{(j,2)}$$

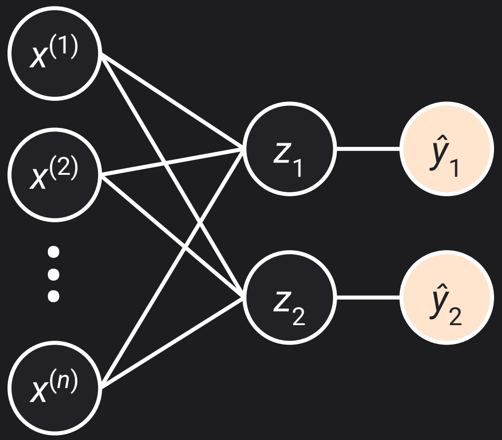


The Multi-Output Perceptron

Each output, z_i:

- shares the same inputs, $\{x^{(j)}\}$;
- has its own set of weights, $\{w^{(j,i)}\}$;
- has its own bias, $w^{(0,i)}$.

$$z_i = w^{(0,i)} + \sum_{j=1}^{n} x^{(j)} w^{(j,i)}$$

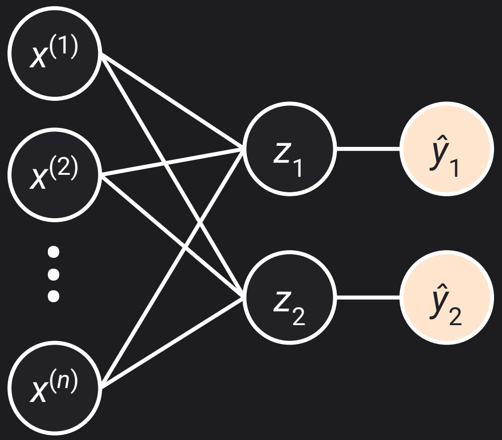


The Multi-Output Perceptron

The mathematics can be better and more compactly represented with vectors/matrices:

$$\mathbf{Z} = \mathbf{W}^{(0)} + \mathbf{X}^{\mathrm{T}}\mathbf{W}$$

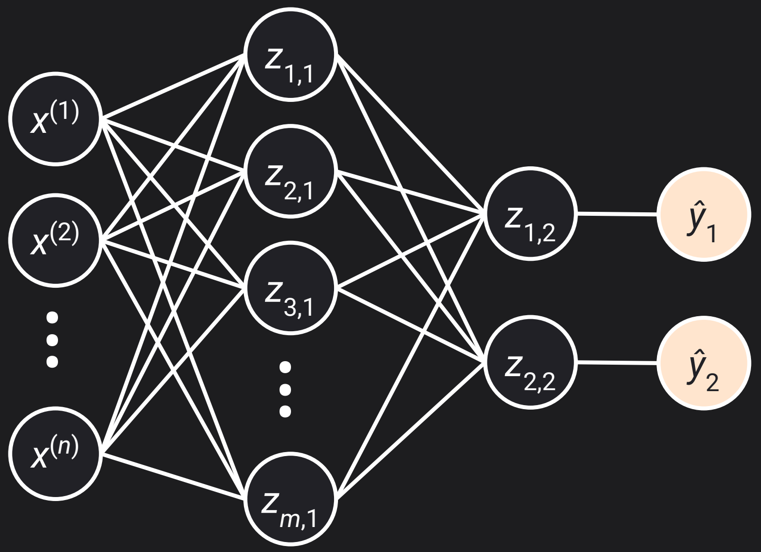
$$\mathbf{X} = \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(n)} \end{bmatrix} \quad \mathbf{W}^{(0)} = \begin{bmatrix} w^{(0,1)} \\ w^{(0,2)} \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} w^{(1,1)} & w^{(1,2)} \\ \vdots & \vdots \\ w^{(n,1)} & w^{(n,2)} \end{bmatrix}$$



The Multilayer Perceptron

A **multilayer perceptron** (**MLP**) stacks layers of multi-output perceptrons:

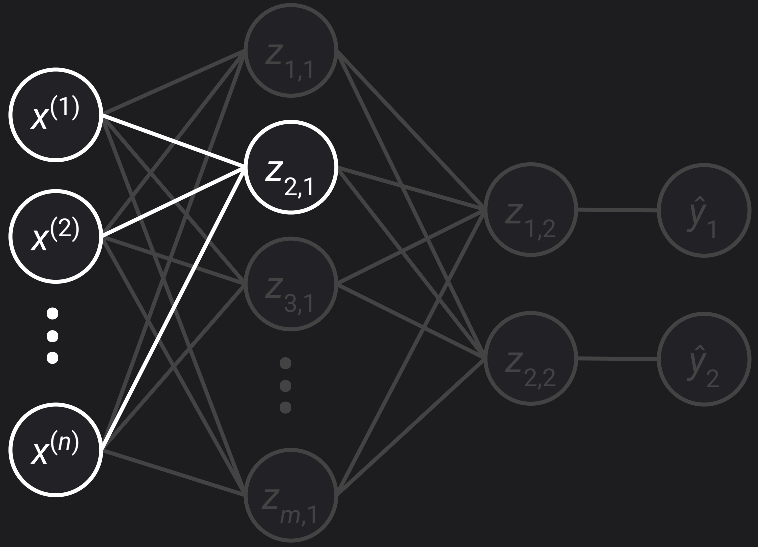
- one or more **hidden layers** are sandwiched between an i**nput layer** and an **output layer**;
- a hidden layer, *k*, takes the output of the previous (k–1th) layer as input;
- all layers are **dense** or **fully connected**, *i.e.* every neuron in the k^{th} layer is connected to every neuron in the k–1th layer.



The Multilayer Perceptron

Let's consider how the activation of the second neuron in the first hidden layer, $z_{2,1}$, is computed:

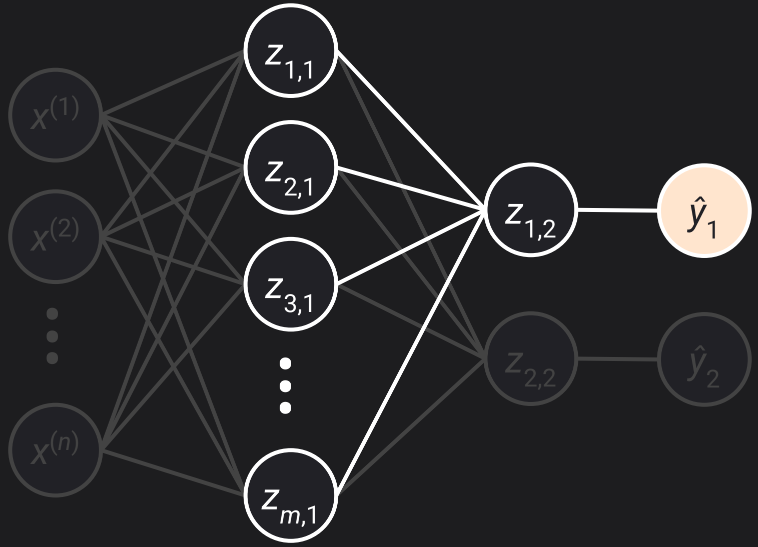
$$z_{2,1} = w_1^{(0,2)} + \sum_{j=1}^{n} x^{(j)} w_1^{(j,2)}$$



The Multilayer Perceptron

Let's consider how the activation of the first neuron in the output layer, $z_{1,2}$, is computed:

$$z_{1,2} = w_2^{(0,1)} + \sum_{j=1}^{m} \sigma(z_{j,1}) w_2^{(j,1)}$$



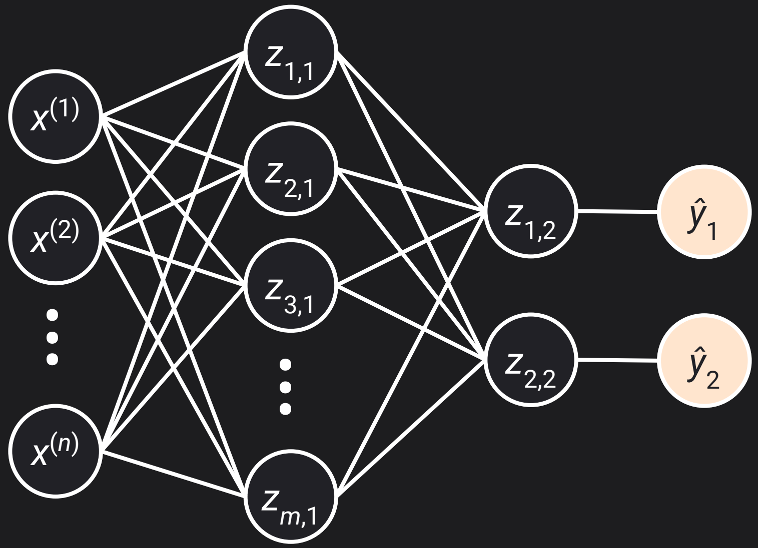
The Multilayer Perceptron

A general expression for the activation of the *i*th neuron in the *k*th layer looks like:

$$z_{i,k} = w_k^{(0,i)} + \sum_{j=1}^{n_{k-1}} \sigma(z_{j,k-1}) w_k^{(j,i)}$$

In the vector representation, this expression simplifies to look like:

$$\mathbf{Z}_k = \mathbf{W}_k^{(0)} + \sigma\left(\mathbf{X}_{k-1}^{\mathrm{T}}\right)\mathbf{W}_k$$



Training Multilayer Perceptrons

Remember:

- the **loss** is defined for a single example, *i*, as function of the predicted, $\hat{y}_i$, and actual, y_i, class labels or regression targets:

$$\mathcal{L}(\hat{y}_i, y_i) = \mathcal{L}(f(\mathbf{X}_i, \mathbf{W}), y_i)$$

- the **cost**, or **objective**, **function** is the average of the loss defined over the *N* examples in the dataset:

$$\mathcal{J}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \frac{1}{N} \sum_{i}^{N} \mathcal{L}(\hat{y}_i, y_i)$$

Training Multilayer Perceptrons

A popular loss function for MLP regressors is the (mean) **squared error**:

$$\mathcal{L}(\hat{y}_i, y_i) = (y_i - \hat{y}_i)^2$$

Advantage:

- greater penalty applied to greater errors.

Disadvantage:

- non-normalised; closer to zero is better, but there is no upper bound.

A popular loss function for MLP classifiers is the **categorical cross-entropy**:

$$\mathcal{L}(\hat{y}_i, y_i) = y_i \log_2(\hat{y}_i) + (1 - y_i) \log_2(1 - \hat{y}_i)$$

Advantage:

- extensible to multiclass classification.

Disadvantage:

- assumes classes are mutually exclusive; not applicable for multiclass membership.

Training Multilayer Perceptrons

The purpose of training is to obtain iteratively the optimum set of weights, **W***, that minimise the loss:

$$\mathbf{W}^* = \underbrace{\text{argmin}}_{\mathbf{W}}(\mathcal{J}(\mathbf{X}, \mathbf{y}, \mathbf{W}))$$

$$\mathbf{W} = \{\mathbf{W}_1, \mathbf{W}_2, \ldots, \mathbf{W}_{k-1}\}$$

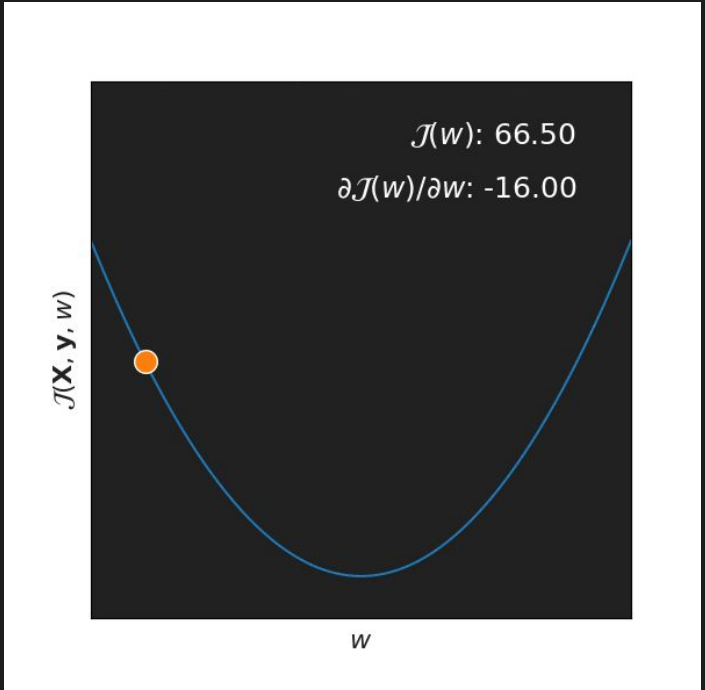
This is a function **optimisation/minimisation** problem.

One way to solve this problem is *via* **gradient descent**.

Gradient Descent

Algorithm:

1) initialise **W** randomly;
2) evaluate the objective;
3) compute the gradient of the objective with respect to **W**;
4) update **W** to minimise the objective;
5) repeat 2), 3), and 4) until convergence.



Gradient Descent

The **learning rate**, η, is a hyperparameter that controls the magnitude, or '*step size*', of the weight update at each cycle.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta\left(\frac{\partial \mathcal{J}(\mathbf{X}, \mathbf{y}, \mathbf{W})}{\partial \mathbf{W}}\right)$$

Options for η:

- **constant**: η is fixed throughout;
- **scheduled**: η is gradually decreased with each successive cycle;
- **adaptive**: η is increased or decreased in response to, *e.g.*, the:
  - magnitude of the gradient(s);
  - magnitude of the weight(s);
  - speed at which learning is occurring.

Gradient Descent

The gradient descent algorithm can benefit from **minibatching** when, *e.g.*:

- it is too time-/compute-intensive to calculate gradients over the entire dataset;
- the entire dataset does not fit in memory.

$$\mathbf{W} \leftarrow \mathbf{W} - \eta\left(\frac{1}{n}\sum_{i}^{n}\frac{\partial\mathcal{J}(\mathbf{X}_i, \mathbf{y}_i, \mathbf{W})}{\partial\mathbf{W}}\right)$$

The size of the batch, *n*, is an additional hyperparameter:

- **the batch contains all examples:** (classic) gradient descent;
- **the batch contains only one example:** stochastic gradient descent;
- **the batch contains some intermediate number of examples**: minibatched gradient descent.

Gradient Descent

The development of **adaptive learning** algorithms is a very active area of research!

Popular Options:

- adaptive moment estimation (**ADAM**);
- adaptive gradient algorithm (**AdaGrad**)
- adaptive delta (**AdaDelta**)
- root-mean-square propagation (**RMSprop**)

Backpropagation

How do we calculate the gradient of the objective with respect to the weights?

Problem:

- there is no directly-accessible expected or target output for a hidden layer;

We can calculate how much the i^{th} neuron in the k^{th} layer contributes to the downstream error through **backpropagation**.

Backpropagation

Let's consider how the gradient of the objective with respect to w_2 is computed using the chain rule:

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_2} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_2}$$

Backpropagation

Let's consider how the gradient of the objective with respect to w_1 is computed using the chain rule:

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_1}$$

$$\frac{\partial \mathcal{J}(\mathbf{W})}{\partial w_1} = \frac{\partial \mathcal{J}(\mathbf{W})}{\partial \hat{y}} \times \left(\frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w_1} \right)$$

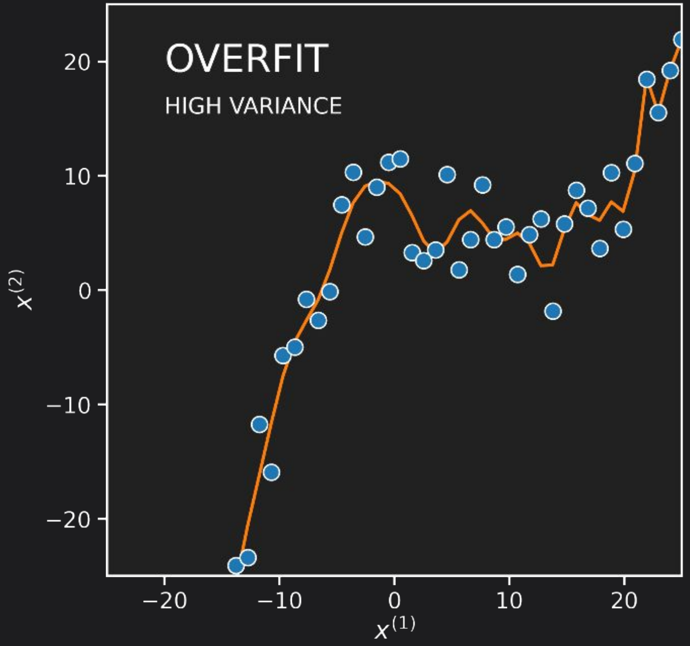
Overfitting in Neural Networks

Deep neural networks can easily have tens of thousands to millions/billions of weights, and so have a high propensity for **overfitting**.

Regularisation is often necessary to control the overfitting in deep neural networks.

Popular Techniques:

- early stopping;
- dropout.

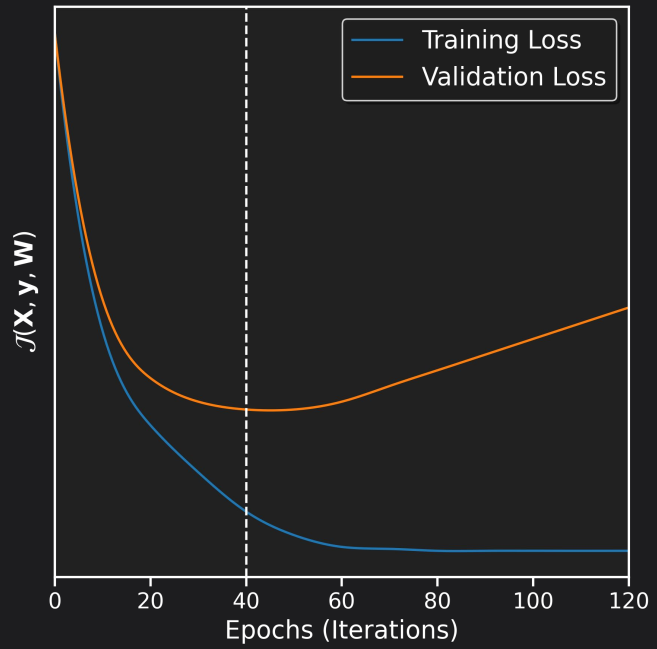


Early Stopping

Early stopping: a regularisation technique that involves monitoring the validation loss and pausing training when it no longer improves.

Purpose:

- to minimise overfitting;
- to limit redundant compute time.

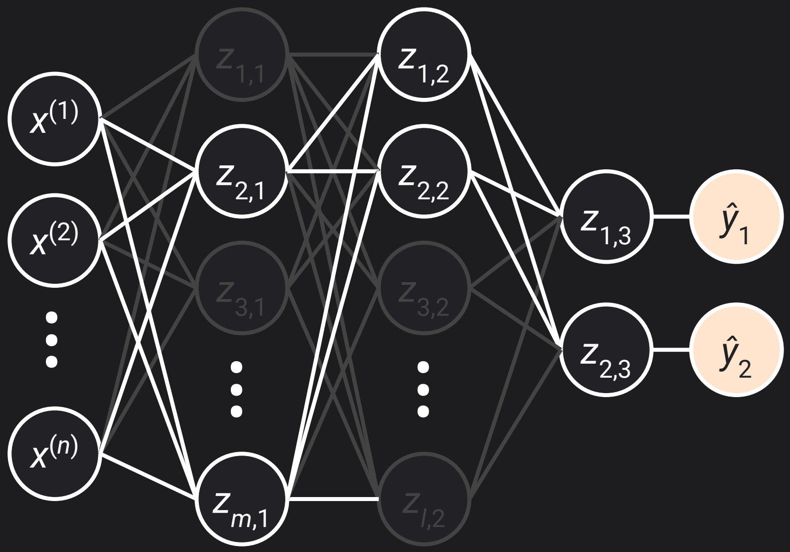


Dropout

Dropout: a regularisation technique that involves '*turning off*' some proportion, ρ, of the neurons in the network at each epoch.

Purpose:

- to minimise overfitting;
- to encourage the model to learn an ensemble of simpler models;
- to encourage the model to distribute the weights in a balanced way.



Multilayer Perceptrons in Scikit-Learn

The `sklearn.neural_network` module has a multilayer perceptron algorithm:

```
model = MLPClassifier(
    hidden_layer_sizes = [100],
    activation = ‘relu’,
    batch_size = ‘auto’,
    learning_rate = ‘constant’,
    learning_rate_init = 0.001,
    max_iter = 200,
    early_stopping = False
)
```