Hands-on Machine Learning

10. Introduction to Neural Networks

History of ANNs

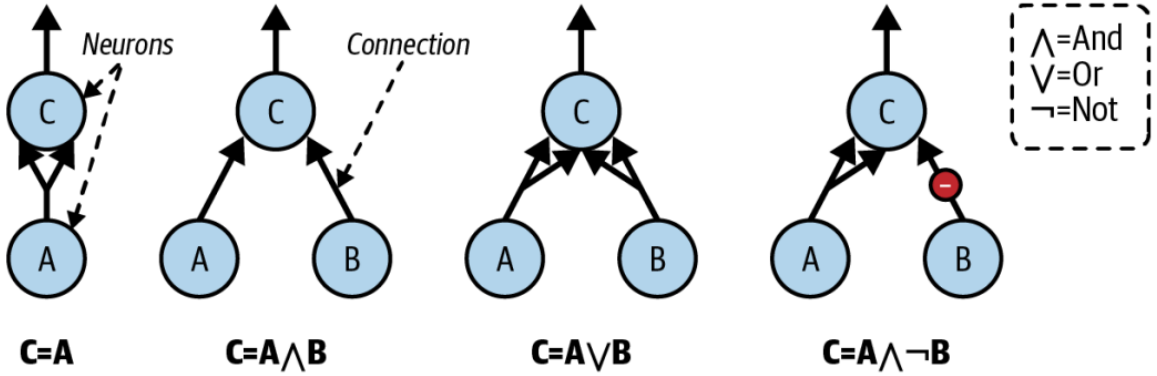
- *Artificial neural networks* (ANNs) are inspired by the networks of biological neurons found in our brains.
- ANNs were introduced in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts, in their landmark paper "A Logical Calculus of Ideas Immanent in Nervous Activity".
- In the early 1980s, new architectures were invented and better training techniques were developed.
- In 1990s other machine learning techniques such as support vector machines had been invented that seemed to offer better results.

New Interest in ANNs

- We are now witnessing another wave of interest in ANNs. Why?
  - Thanks to the internet, there is now a huge quantity of data available to train neural networks.
  - The increase in computing power since the 1990s makes it possible to train large neural networks in a reasonable time.
  - The training algorithms have been improved.
  - Some theoretical limitations of ANNs have turned out to be benign in practice.
  - ANNs entered a virtuous circle of funding and progress.

Logical Computations with Neurons

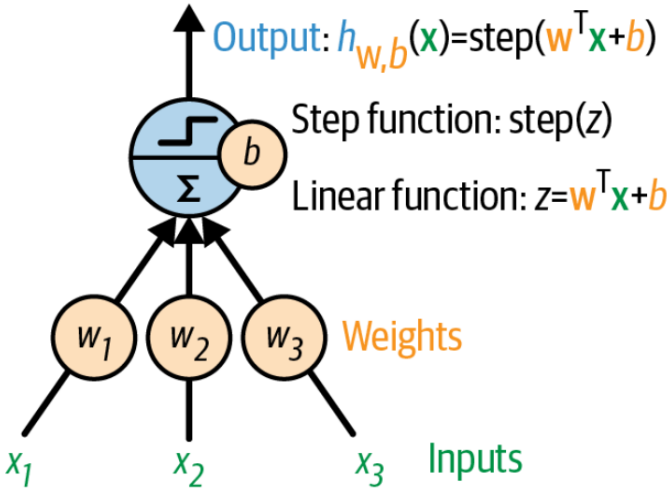
- McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*:
  - It has one or more binary (on/off) inputs and one binary output.
- The artificial neuron activates its output when more than a certain number of its inputs are active.



1.
Perceptron

The Perceptron

- The *perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt.
- It is based on a different artificial neuron called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU).



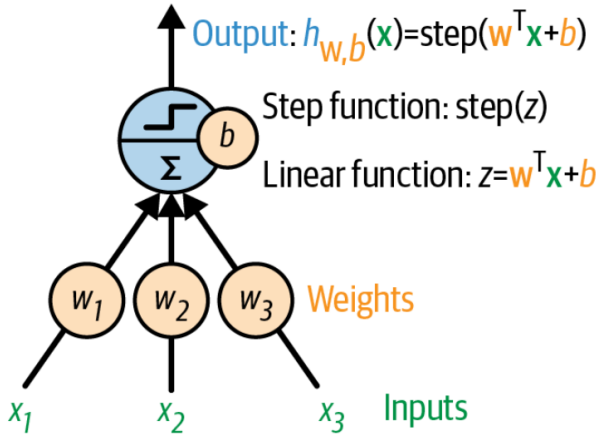
The Perceptron

- The TLU first computes a linear function of its inputs:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b = \mathbf{w}^\intercal \mathbf{x} + b.$$

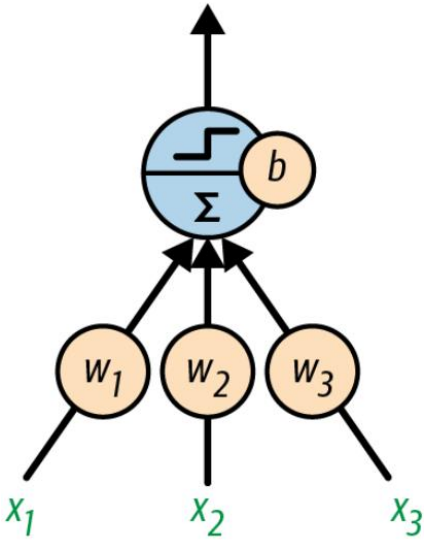
- Then it applies a *step function* to the result: $h_{\mathbf{w}}(\mathbf{x}) = step(z)$
- The model parameters are the input weights $\mathbf{w}$ and the bias term b.
- The most common step function used in perceptrons is *heaviside step function*:

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



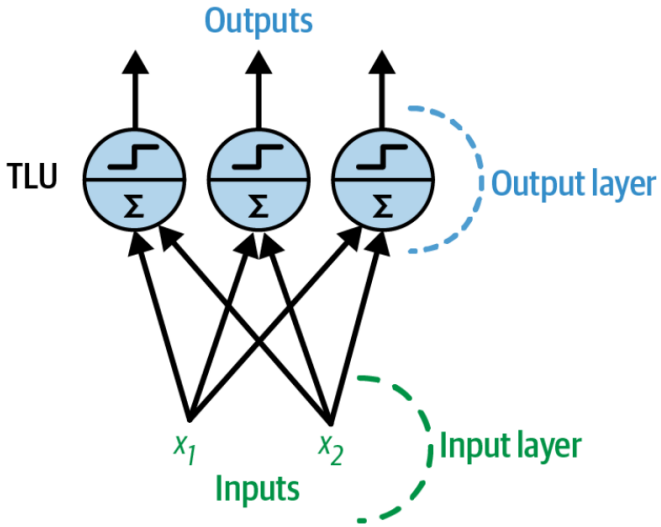
The Perceptron

- A single TLU can be used for simple linear binary classification.
  - It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class.
  - Similar to logistic regression or linear SVM classification.
- Training such a TLU would require finding the right values for w_1, w_2, … and b.

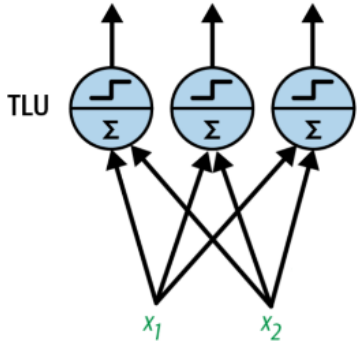


The Perceptron

- A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input.
  - Such a layer is called a *fully connected layer*, or a *dense layer*.
- The inputs constitute the *input layer*.
- Since the layer of TLUs produces the final outputs, it is called the *output layer*.



The Perceptron

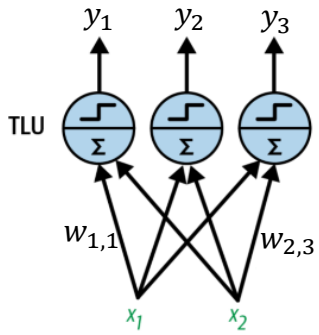


- Computing the outputs of a fully connected layer:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

- $\mathbf{X}$ represents the matrix of input features.
  - It has one row per instance and one column per feature.
- The weight matrix $\mathbf{W}$ contains all the connection weights.
  - It has one row per input and one column per neuron.
- The bias vector $\mathbf{b}$ contains all the bias terms: one per neuron.
- The function ϕ is called the *activation function*.
  - when the artificial neurons are TLUs, it is a step function.

Training a Perceptron



- Perceptrons are trained using *Hebb's rule* that considers the error made by the network when it makes a prediction:
  - Reinforce connections that help reduce the error:
    - Perceptron is fed one training instance at a time, and for each instance it makes a prediction.
    - For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction:

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(y_j - \hat{y}_j\right)x_i$$

  - $w_{i,j}$ is the connection weight between the i-th input and the j-th neuron.
  - x_i is the i-th input value of the current training instance.
  - $\hat{y}_j$, y_j are the output and target output of the j-th output neuron for the current training instance, respectively.
  - η is the learning rate.

Perceptron Convergence Theorem

- The decision boundary of each output neuron is linear, so perceptrons are incapable of learning complex patterns.
- The *perceptron convergence theorem*: if the training instances are linearly separable, this algorithm would converge to a solution.

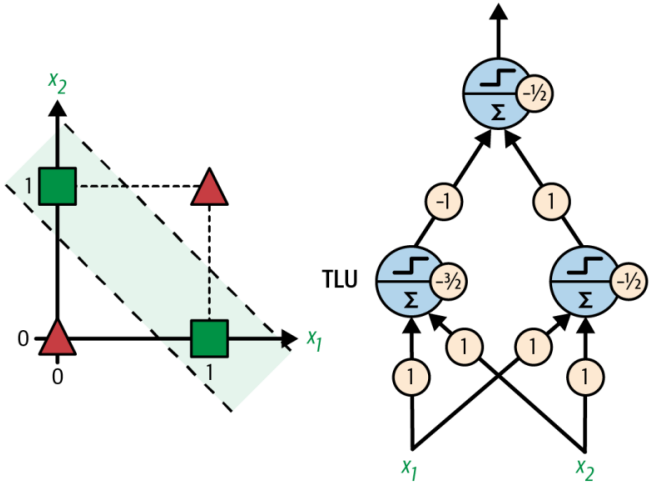
```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0)  # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new)  # predicts True and False for these 2 flowers
```

Perceptron vs. Logistic Regression

- Contrary to logistic regression classifiers, perceptrons do not output a class probability.
  - This is one reason to prefer logistic regression over perceptrons.
- Perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set.
  - The model typically does not generalize as well as logistic regression or a linear SVM classifier.
- Perceptrons may train a bit faster.

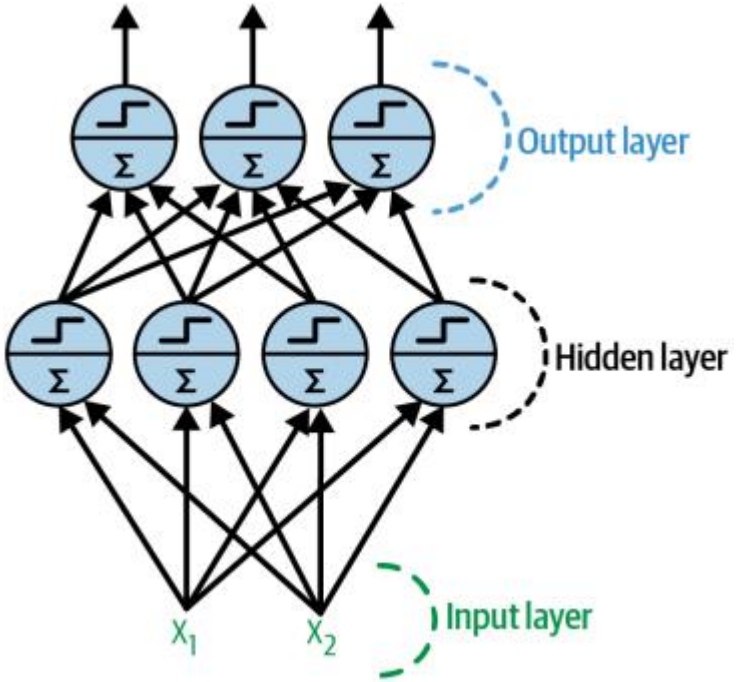
Multilayer Perceptron

- *Perceptrons* are incapable of solving some trivial problems (e.g., the *exclusive OR* (XOR) classification problem).
- Some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons.
  - The resulting ANN is called a *multilayer perceptron* (MLP).



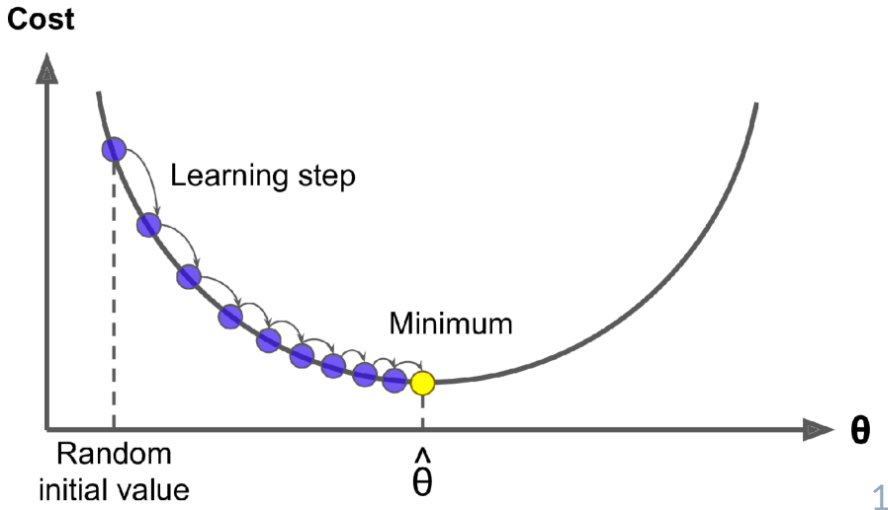
Multilayer Perceptron

- An MLP is composed of one input layer, one or more layers of TLUs called *hidden layers*, and one final layer of TLUs called the *output layer*.
- The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*.
- The signal flows only in one direction, so this architecture is a *feedforward neural network* (FNN).



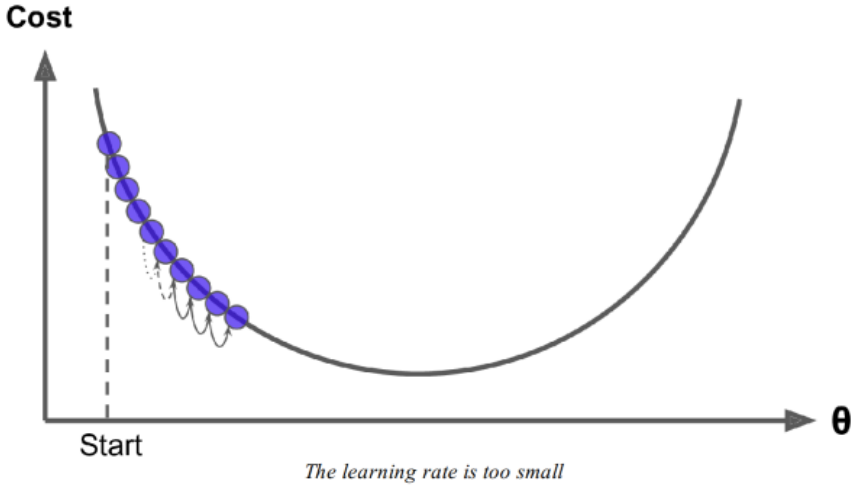
Review of Gradient Descent

- *Gradient Descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.
- We start by filling **θ** with random values (*random initialization*).
- Improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum.

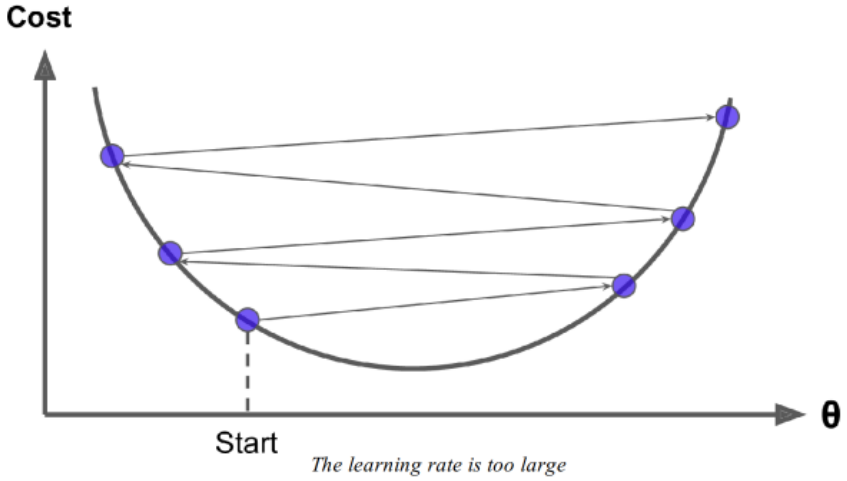


Learning Rate

- *Learning Rate (η):* a hyperparameter of gradient descent that determines the size of the steps.
  - Too small: convergence will take a long time.
  - Too large: the algorithm might diverge.



The learning rate is too small



The learning rate is too large

Batch Gradient Descent

- To implement Gradient Descent, you need to compute the gradient of the cost function with regard to each model parameter θ_j.
- Take partial derivative of the cost function with regard to θ_j:

$$\mathrm{MSE}\left(\mathbf{X}, h_{\boldsymbol{\theta}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

$$\frac{\partial}{\partial\theta_j}\mathrm{MSE}\left(\boldsymbol{\theta}\right) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\mathsf{T}}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

$$\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}\left(\boldsymbol{\theta}\right) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}\mathrm{MSE}\left(\boldsymbol{\theta}\right) \\ \frac{\partial}{\partial\theta_1}\mathrm{MSE}\left(\boldsymbol{\theta}\right) \\ \vdots \\ \frac{\partial}{\partial\theta_n}\mathrm{MSE}\left(\boldsymbol{\theta}\right) \end{pmatrix} = \frac{2}{m}\mathbf{X}^{\mathsf{T}}\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right)$$

Batch Gradient Descent

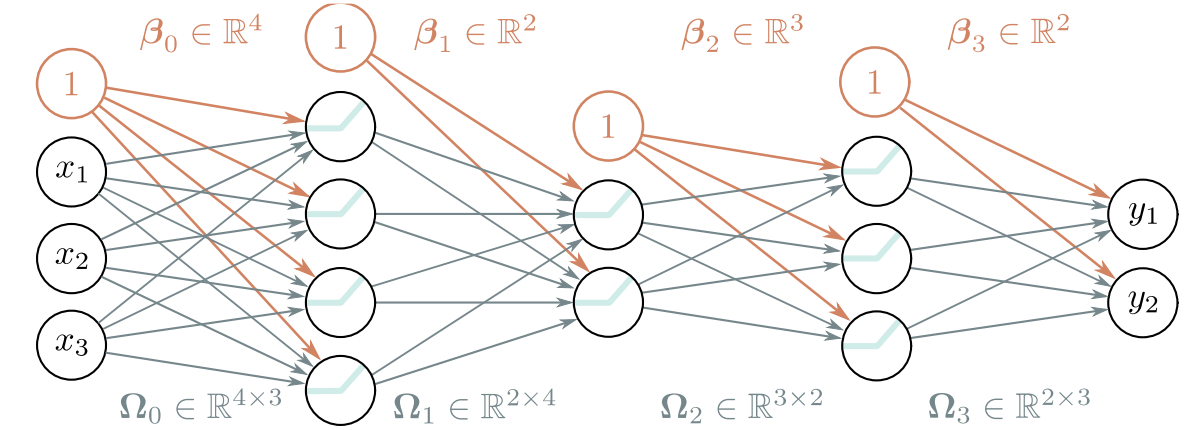
- Gradient vector involves calculations over the full training set $\mathbf{X}$, at each Gradient Descent step: $\nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \mathbf{X}^{\intercal} (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$
  - This is why the algorithm is called *Batch Gradient Descent*.
- Gradient Descent step: $\boldsymbol{\theta}^{(\text{next step})} = \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \operatorname{MSE}(\boldsymbol{\theta})$

```
eta = 0.1  # learning rate
n_epochs = 1000
m = len(X_b)  # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

```
theta
```

```
array([[4.21509616],
       [2.77011339]])
```

Computing Gradients for MLP



$$
\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\
\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_3 + \boldsymbol{\Omega}_3\mathbf{h}_3
\end{aligned}
$$

Computing Gradients for MLP

- A neural network is just an equation:

$$\begin{aligned} y' = \phi_0' + \phi_1' \mathrm{a}\left[\psi_{10} + \psi_{11}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{12}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{13}\mathrm{a}[\theta_{30} + \theta_{31}x]\right] \\ + \phi_2' \mathrm{a}[\psi_{20} + \psi_{21}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{22}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{23}\mathrm{a}[\theta_{30} + \theta_{31}x]] \\ + \phi_3' \mathrm{a}[\psi_{30} + \psi_{31}\mathrm{a}[\theta_{10} + \theta_{11}x] + \psi_{32}\mathrm{a}[\theta_{20} + \theta_{21}x] + \psi_{33}\mathrm{a}[\theta_{30} + \theta_{31}x]] \end{aligned}$$

- But it's a huge equation, and we need to compute derivative:
  - for every parameter
  - for every point in the batch
  - for every iteration of SGD

Backpropagation

- For many years researchers struggled to find a way to train MLPs.
  - In the early 1960s the possibility of using gradient descent to train neural networks was discussed unsuccessfully.
- In 1970, Seppo Linnainmaa introduced *reverse-mode automatic differentiation* (or *reverse-mode autodiff*) technique to compute all the gradients automatically and efficiently.
  - In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network's error.
- These gradients can then be used to perform a gradient descent step.
- The combination of reverse-mode autodiff and gradient descent is now called *backpropagation* (or *backprop*).

Backpropagation

- Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks.
- In 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a groundbreaking paper analyzing how backpropagation allowed neural networks to learn useful internal representations.
- Today, it is by far the most popular training technique for neural networks.

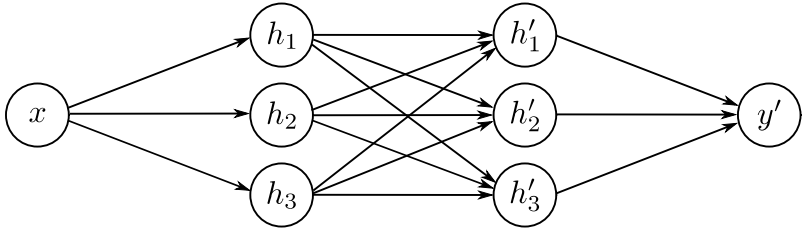
Backpropagation

- It handles one mini-batch at a time, and it goes through the full training set multiple times. Each pass is called an *epoch*.
  - Each mini-batch enters the network through the input layer.
- The *forward pass*:
  - The algorithm computes the output of all the neurons in the first hidden layer, for every instance in the mini-batch.
  - The result is passed on to the next layer, its output is computed and passed to the next layer, … until we get the output of the last layer.
- It is like making predictions, except all intermediate results are stored.
- Next, the algorithm measures the network's output error.

Backpropagation

- The *backward pass*:
  - The algorithm computes how much each output bias and each connection to the output layer contributed to the error by applying the *chain rule.*
  - Then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer.
- This reverse pass efficiently measures the error gradient across all the connection weights and biases.
- Finally, the algorithm performs a gradient descent step to tweak all the connection weights in the network, using the error gradients it just computed.

Backpropagation



- Summary of the algorithm:
  - **forward pass**: makes predictions for a mini-batch
  - measures the error
  - **reverse pass**: goes through each layer in reverse to measure the error contribution from each parameter.
  - **gradient descent step**: tweaks the connection weights and biases to reduce the error.
- It is important to initialize all the hidden layers' connection weights randomly, or else training will fail.
  - If you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and your model will act as if it had only one neuron per layer.

2.
Activation Function

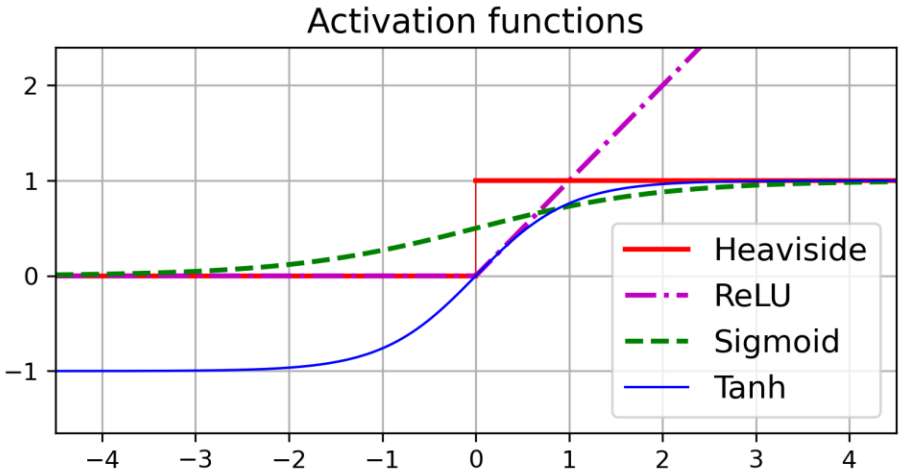
Sigmoid Activation Function

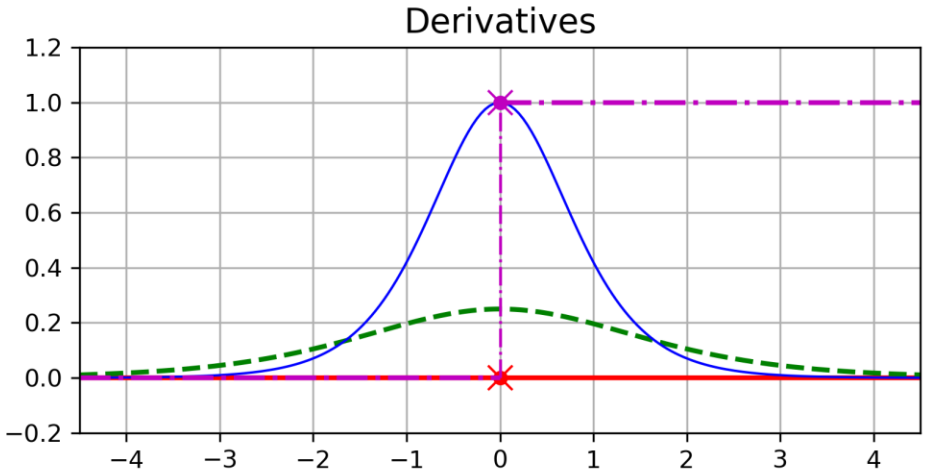
- Rumelhart and his colleagues replaced the step function with the logistic (aka. *sigmoid*) function, $\sigma(z) = \frac{1}{1+e^{-z}}$.
- The step function contains only flat segments, so there is no gradient to work with.
  - gradient descent cannot move on a flat surface
- The sigmoid function has a well-defined nonzero derivative everywhere, allowing gradient descent to make some progress at every step.
- Why do we need activation functions?
  - If you chain several linear transformations, you get a linear transformation.
  - So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer.

Other Activation Functions

- The *hyperbolic tangent* function: $\tanh(z) = 2\sigma(2z) - 1$
  - Similar to the sigmoid function, $\tanh$ is *S*-shaped, continuous, and differentiable, but its output value ranges from –1 to 1.
  - That range tends to make each layer's output more or less centered around 0 at the beginning of training, which helps speed up convergence.
- The *rectified linear unit* function: $ReLU(z) = \max(0, z)$
  - The ReLU function is continuous but not differentiable at $z = 0$ and its derivative is 0 for $z < 0$.
  - In practice it works very well and has the advantage of being fast to compute, so it has become the default.
  - The fact that it does not have a maximum output value helps reduce some issues during gradient descent

Comparing Activation Functions





Regression MLPs

- If you want to predict a single value, you just need a single output neuron: its output is the predicted value.
- For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension.
  - For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons.
- Scikit-Learn includes an `MLPRegressor` class.
  - We use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset.

Regression MLPs

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

housing = fetch_california_housing()
X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full, random_state=42)

mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], random_state=42)
pipeline = make_pipeline(StandardScaler(), mlp_reg)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_valid)
rmse = mean_squared_error(y_valid, y_pred, squared=False)
```

Regression MLPs

- The `MLPRegressor` class does not support activation functions in the output layer, so it's free to output any value it wants.
- To guarantee that the output will always be positive, then you should use the ReLU or the *softplus* activation function in the output layer.
  - Softplus is a smooth variant of ReLU: $softplus(z) = \log(1 + e^z)$
    - It is close to 0 when z is negative, and close to z when z is positive.
- To guarantee that the predictions will always fall within a given range of values, use the sigmoid or the hyperbolic tangent, and scale the targets to the appropriate range.

Regression MLPs

- The MLPRegressor class uses the mean squared error (MSE), which is usually what you want for regression.
- If you have a lot of outliers in the training set, you may prefer to use the mean absolute error (MAE).
- We can also use the *Huber loss*, which is a mix of MSE and MAE.
  - It is quadratic when the error is smaller than a threshold δ (typically 1) but linear when the error is larger than δ.
  - The linear part makes it less sensitive to outliers.
  - The quadratic part allows it to converge faster and be more precise.
- `MLPRegressor` only supports the MSE.

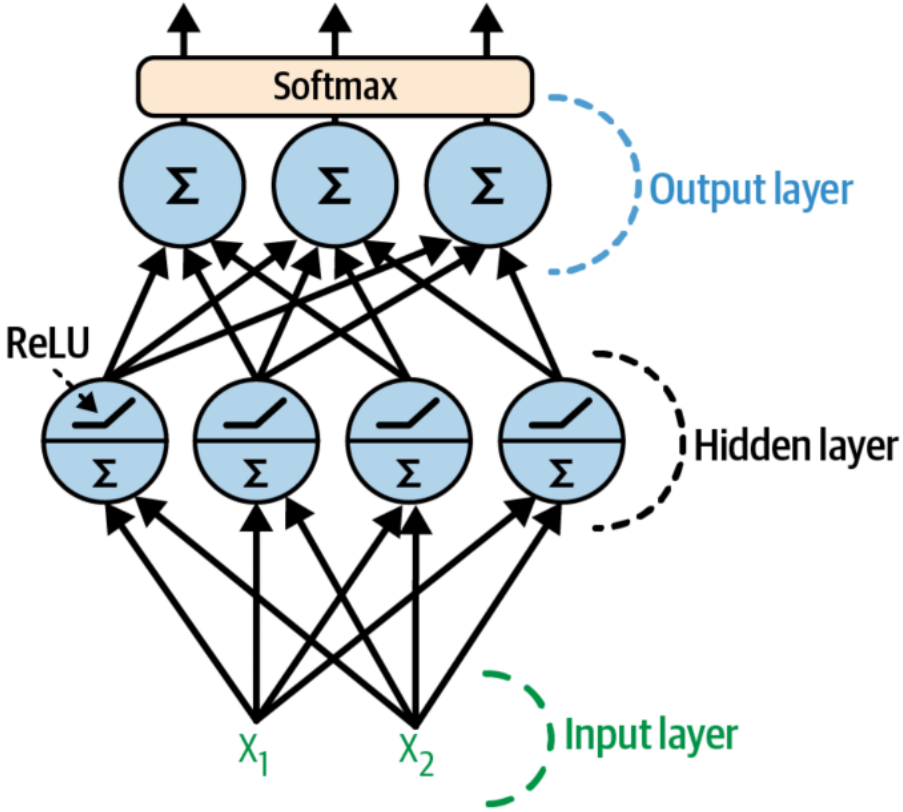
Typical Regression MLP Architecture

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

Classification MLPs

- For a binary classification problem, you need a single output neuron using the sigmoid activation function.
  - The output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class.
- MLPs can handle multilabel binary classification tasks.
  - e.g. an email classification system that predicts whether each email is ham/spam, and simultaneously predicts it is an urgent/nonurgent email.
  - In this case, you would need two output neurons, both using the sigmoid activation function.
- For multiclass classification, you need to have one output neuron per class, and use the softmax activation for the whole output layer.

Classification MLPs



Classification MLPs

- Scikit-Learn has an `MLPClassifier` class, similar to `MLPRegressor` class, except that it minimizes the cross entropy rather than the MSE.

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers	Typically 1 to 5 layers, depending on the task		
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy	X-entropy	X-entropy