STEVENS
INSTITUTE OF TECHNOLOGY
1870

PEP 559
Machine Learning in Quantum Physics

Dr. Chunlei Qu

Spring 2026

- So far, we have understood the building block of neural networks --- the
perceptron
with the following key features:

binary output (0, 1)

single layer (input layer)

Rosenblatt perceptron

- **Single-layer** NN
- The weights are updated based on a **step function**
- The weight update is calculated incrementally after **EACH** training example

$$w_j := w_j + \Delta w_j$$
$$\text{and} \quad b := b + \Delta b$$

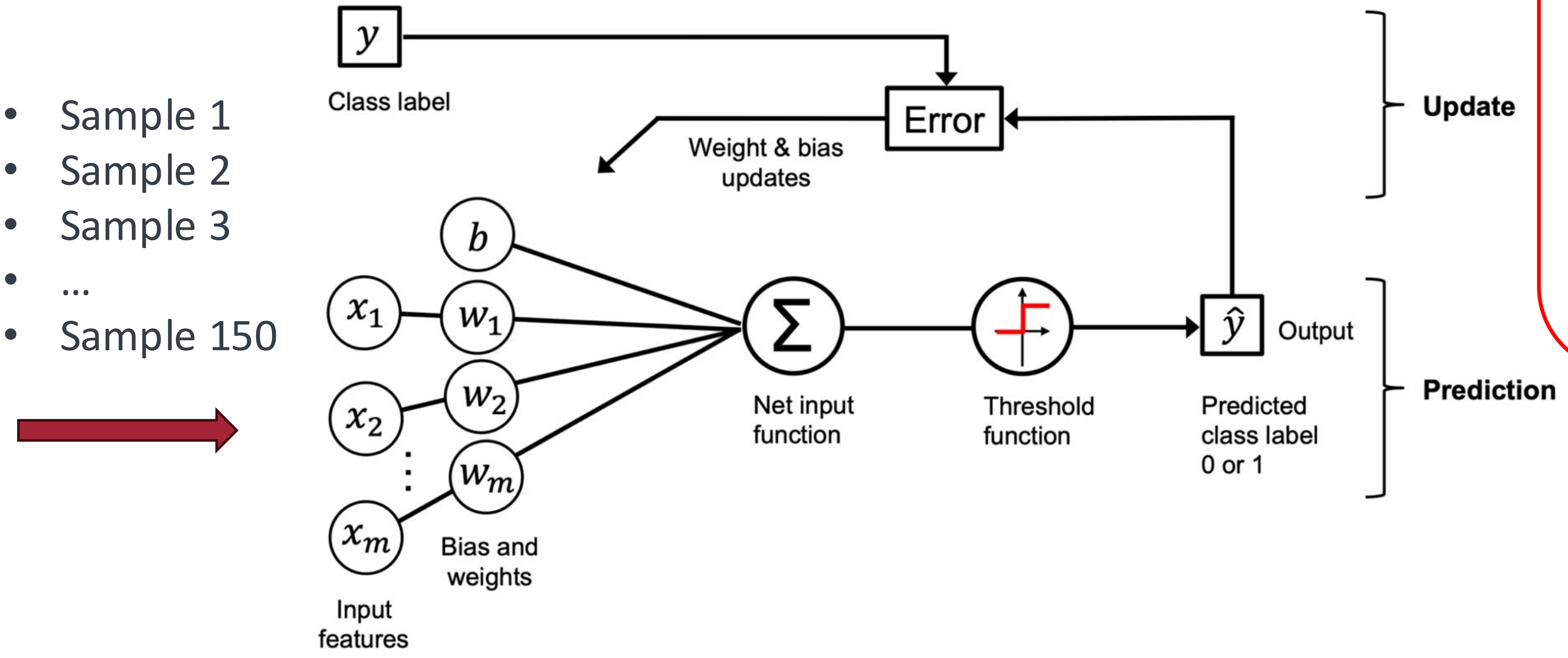
$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$
$$\text{and} \quad \Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

learning rate

actual class label

predicted class label

- Sample 1
- Sample 2
- Sample 3
- …
- Sample 150



j: feature index

i: sample index

For Iris dataset

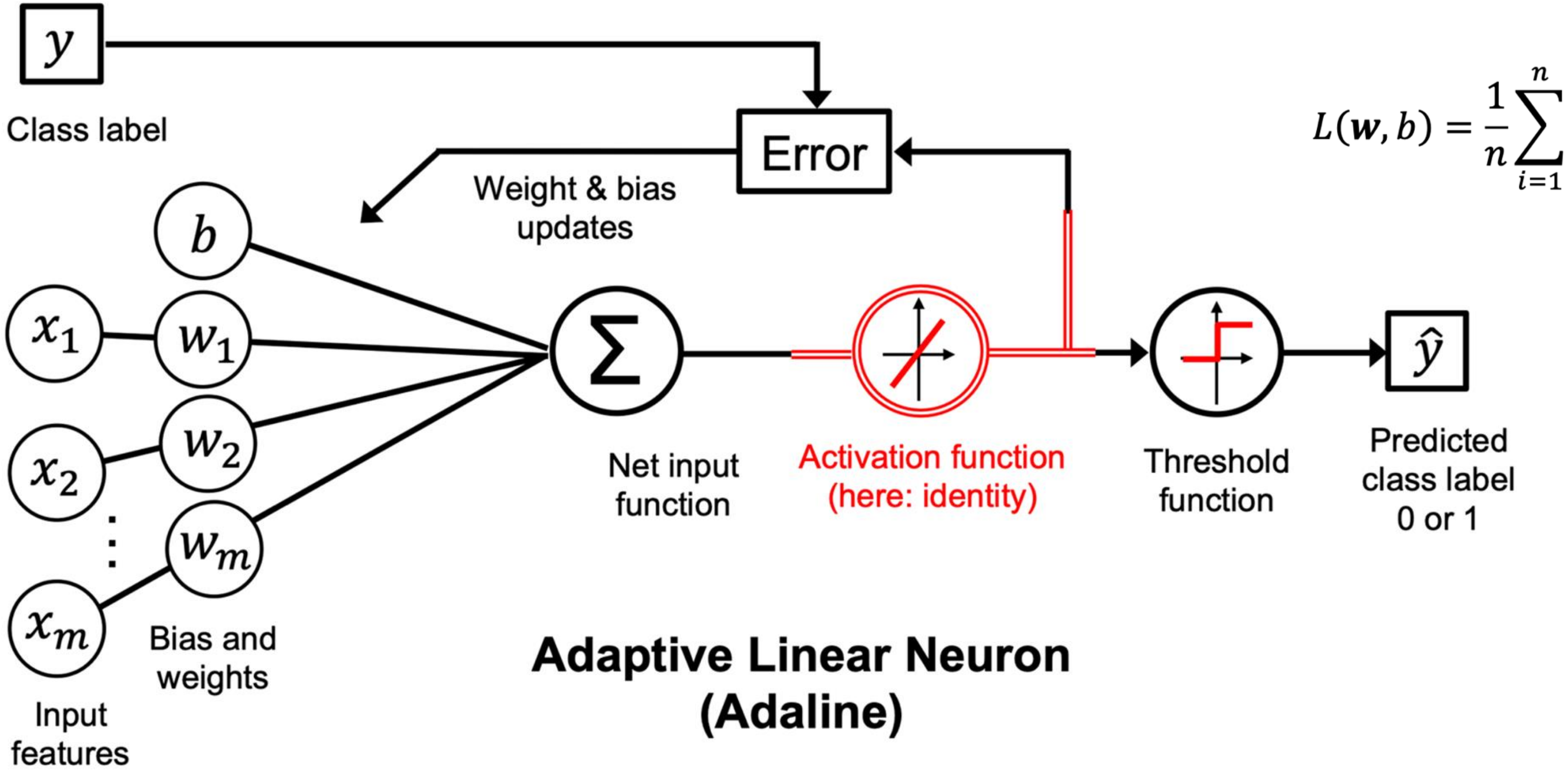
m=4 features

Adaptive linear neuron (Adaline)

- A generalized Rosenblatt's neuron model by Bernard Widrow and Tedd Hoff (**1960**)

Mean squared errors (MSE)

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)^2$$



Adaptive Linear Neuron (Adaline)

Multilayer Perceptron (MLP) Model

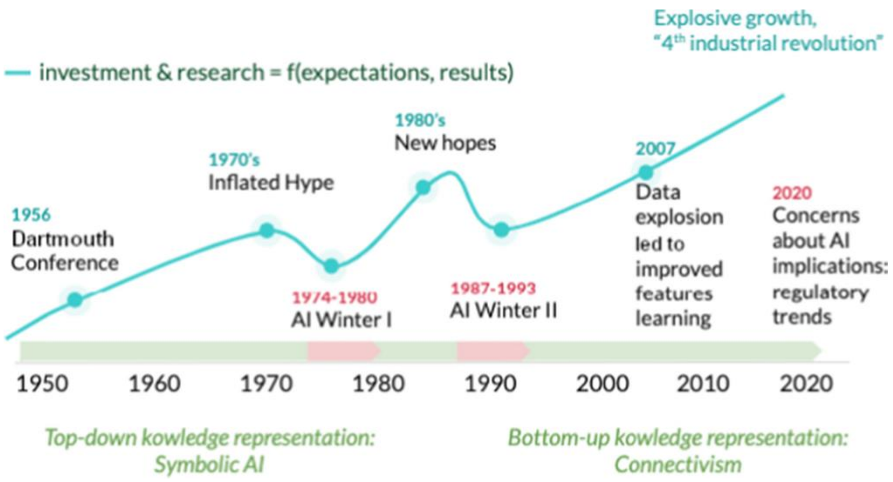
- Rosenblatt’s **perceptron** was first implemented in **1950s**

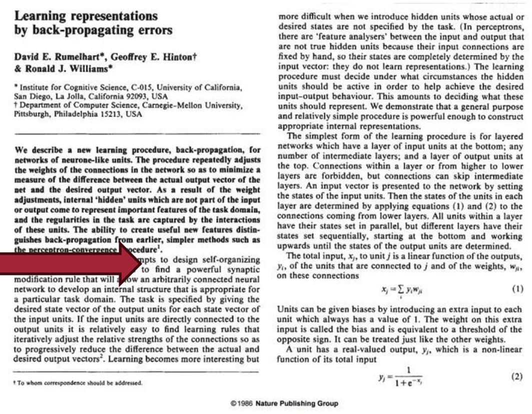
- It was not clear how to **train a multilayer neural network** efficiently

- People started to lose interest in neural networks (1970s to early 1980s): **the AI winter I**

- Interest in NN was rekindled in **1986**

- Market crashed again due to expensive expert system: **the AI winter II**





Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal ‘hidden’ units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

[illegible] mpts to design self-organizing [illegible] to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are ‘feature analysers’ between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \quad (1)$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

$$y_j = \frac{1}{1+e^{-x_j}} \quad (2)$$

† To whom correspondence should be addressed.

©1986 Nature Publishing Group

The backpropagation algorithm was first presented in 1986 by David Rumelhart, Geoffrey Hinton and Ronald Williams. For this, and other accomplishments, Geoffrey Hinton was awarded the Turning award in 2018 (along with Yoshua Bengio and Yann LeCun).

A two-layer MLP

- Input layer (*m* neurons)

$$x_i^{(in)}$$

- Hidden layer (*d* neurons)

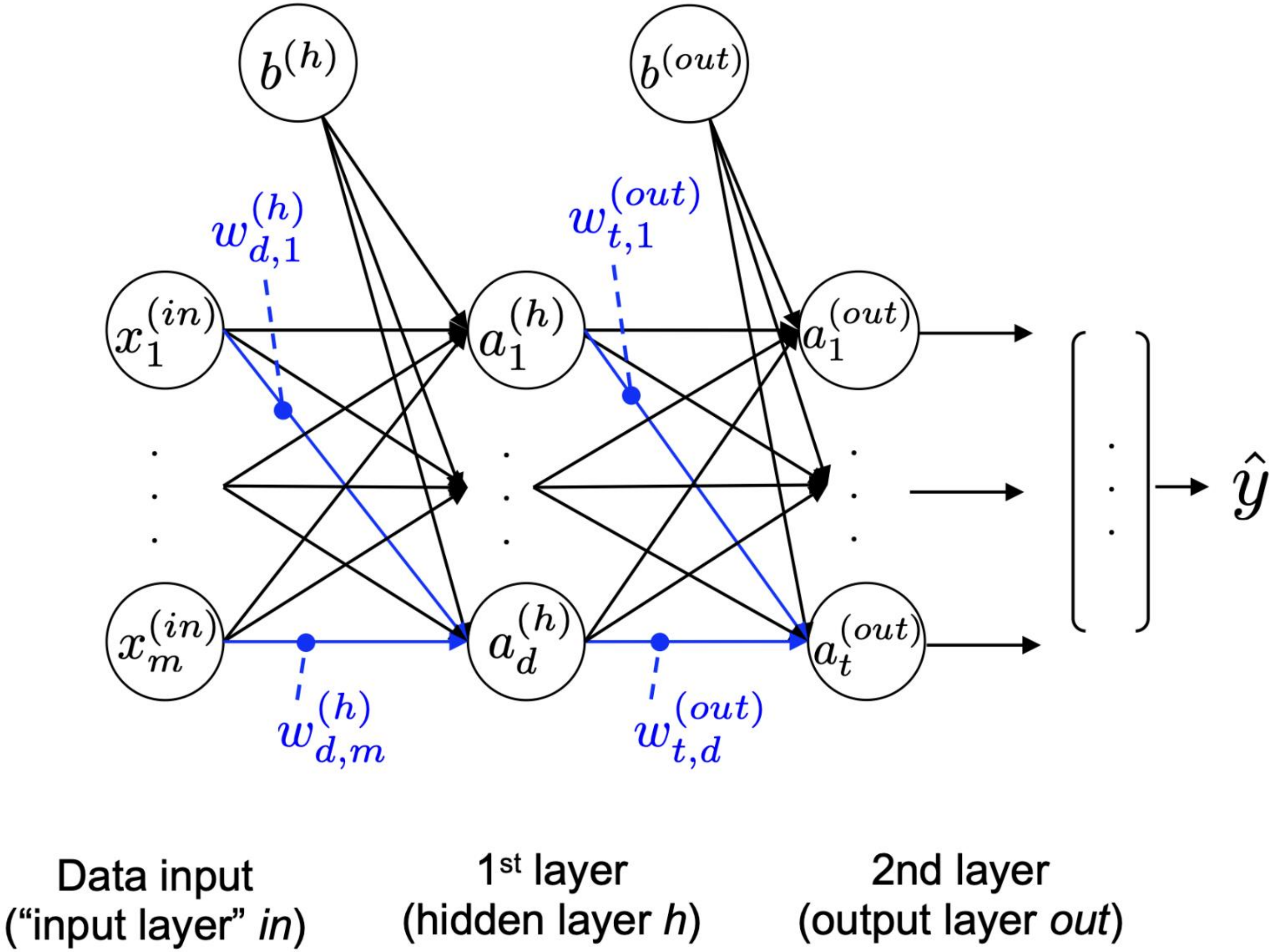
$$a_j^{(h)}$$

- Output layer (*t* neurons)

$$a_k^{(out)}$$

$$w_{j,i}^{(h)}$$

$$w_{k,j}^{(out)}$$



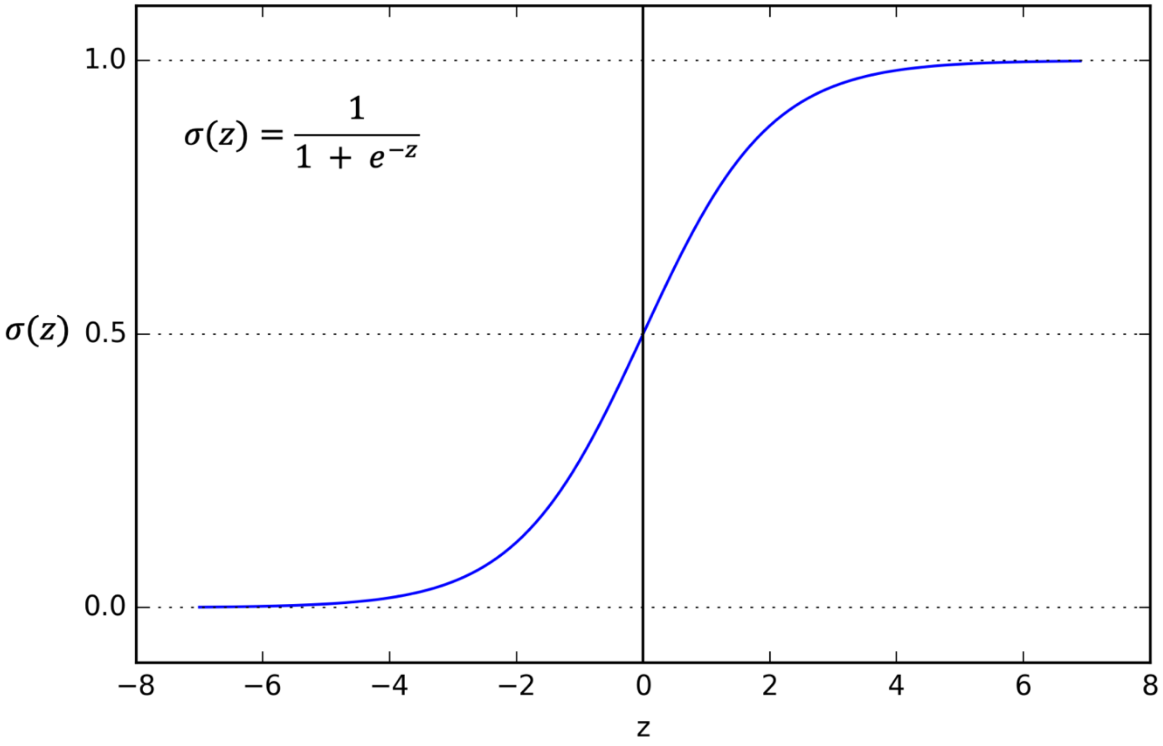
Nonlinear activation

- For hidden layer neuron, the pre-activation (weighted sum) and the post-activation are

$$z_j^{(h)} = \sum_{i=1}^{m} x_i^{(\text{in})} w_{j,i}^{(h)} + b_j^{(h)}$$

$$a_j^{(h)} = \sigma\left(z_j^{(h)}\right)$$

Sigmoid function: mapping input z to a real value between 0 and 1



- For simplicity, we assume the output layer applies the same nonlinear activation function

Vectorization

- Input vector

$$\mathbf{x}^{(\text{in})} = \begin{bmatrix} x_1^{(\text{in})} \\ x_2^{(\text{in})} \\ \vdots \\ x_m^{(\text{in})} \end{bmatrix} \in \mathbb{R}^m$$

- Weight matrix

$$\mathbf{W}^{(h)} = \begin{bmatrix} w_{1,1}^{(h)} & w_{1,2}^{(h)} & \cdots & w_{1,m}^{(h)} \\ w_{2,1}^{(h)} & w_{2,2}^{(h)} & \cdots & w_{2,m}^{(h)} \\ \vdots & \vdots & w_{j,i}^{(h)} & \vdots \\ w_{d,1}^{(h)} & w_{d,2}^{(h)} & \cdots & w_{d,m}^{(h)} \end{bmatrix} \in \mathbb{R}^{d \times m}$$

- Bias vector

$$\mathbf{b}^{(h)} = \begin{bmatrix} b_1^{(h)} \\ b_2^{(h)} \\ \vdots \\ b_d^{(h)} \end{bmatrix} \in \mathbb{R}^d$$

- Pre-activation

$$\mathbf{z}^{(h)} = \begin{bmatrix} z_1^{(h)} \\ z_2^{(h)} \\ \vdots \\ z_d^{(h)} \end{bmatrix} = \mathbf{W}^{(h)} \mathbf{x}^{(\text{in})} + \mathbf{b}^{(h)}$$

- Post-activation

$$\mathbf{a}^{(h)} = \begin{bmatrix} a_1^{(h)} \\ a_2^{(h)} \\ \vdots \\ a_d^{(h)} \end{bmatrix} = \sigma\left(\mathbf{z}^{(h)}\right)$$

From hidden layer to output layer

- Weight matrix

$$\mathbf{W}^{(\text{out})} = \begin{bmatrix} w_{1,1}^{(\text{out})} & w_{1,2}^{(\text{out})} & \cdots & w_{1,d}^{(\text{out})} \\ w_{2,1}^{(\text{out})} & w_{2,2}^{(\text{out})} & \cdots & w_{2,d}^{(\text{out})} \\ \vdots & \vdots & w_{k,j}^{(out)} & \vdots \\ w_{t,1}^{(\text{out})} & w_{t,2}^{(\text{out})} & \cdots & w_{t,d}^{(\text{out})} \end{bmatrix} \in \mathbb{R}^{t\times d}$$

- Bias vector

$$\mathbf{b}^{(\text{out})} = \begin{bmatrix} b_1^{(\text{out})} \\ b_2^{(\text{out})} \\ \vdots \\ b_t^{(\text{out})} \end{bmatrix} \in \mathbb{R}^t$$

$$z_k^{(\text{out})} = \sum_{j=1}^{d} w_{k,j}^{(\text{out})} a_j^{(h)} + b_k^{(\text{out})}, \qquad k = 1, \ldots, t$$

$$a_k^{(\text{out})} = \sigma\left(z_k^{(\text{out})}\right)$$

$$\mathbf{z}^{(\text{out})} = \mathbf{W}^{(\text{out})}\mathbf{a}^{(h)} + \mathbf{b}^{(\text{out})}$$

$$\mathbf{a}^{(\text{out})} = \sigma\left(\mathbf{z}^{(\text{out})}\right)$$

Putting everything together …

- The one-hidden-layer feedforward neural network with identical activation function at each layer

$$\boxed{\mathbf{a}^{(\mathrm{out})} = \sigma\left(\mathbf{W}^{(\mathrm{out})}\,\sigma\left(\mathbf{W}^{(h)}\mathbf{x}^{(\mathrm{in})} + \mathbf{b}^{(h)}\right) + \mathbf{b}^{(\mathrm{out})}\right)}$$

For a batch with ***n*** data examples or samples, we extend the input data vector to input data matrix, with each column corresponding to one example

$$\mathbf{X}^{(\mathrm{in})} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1^{(\mathrm{in})} & \mathbf{x}_2^{(\mathrm{in})} & \cdots & \mathbf{x}_n^{(\mathrm{in})} \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{m\times n}$$

Its elements are denotated as:

$$X_{i,\alpha}^{(\mathrm{in})} : \qquad i = 1,\ldots,m,\ \alpha = 1,\ldots,n$$

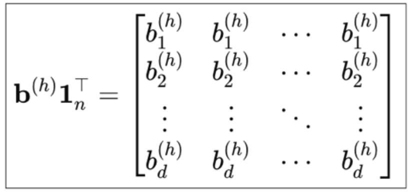
α-th sample, *i*-th element

From input layer to hidden layer (batch form)

Pre-activation

$$\mathbf{Z}^{(h)} = \mathbf{W}^{(h)}\mathbf{X}^{(\text{in})} + \mathbf{b}^{(h)}\mathbf{1}_n^\top \in \mathbb{R}^{d\times n}$$

- $\mathbf{1}_n \in \mathbb{R}^n$ is a vector of ones
- Bias is **broadcast across all examples**



$$\mathbf{b}^{(h)}\mathbf{1}_n^\top = \begin{bmatrix} b_1^{(h)} & b_1^{(h)} & \cdots & b_1^{(h)} \\ b_2^{(h)} & b_2^{(h)} & \cdots & b_2^{(h)} \\ \vdots & \vdots & \ddots & \vdots \\ b_d^{(h)} & b_d^{(h)} & \cdots & b_d^{(h)} \end{bmatrix}$$

Element-wise:

$$Z_{j,\alpha}^{(h)} = \sum_{i=1}^{m} w_{j,i}^{(h)}\, X_{i,\alpha}^{(\text{in})} + b_j^{(h)}$$

Activation

$$\mathbf{A}^{(h)} = \sigma\left(\mathbf{Z}^{(h)}\right) \in \mathbb{R}^{d\times n}$$

From hidden layer to output layer (batch form)

Pre-activation

$$\mathbf{Z}^{(\mathrm{out})} = \mathbf{W}^{(\mathrm{out})}\mathbf{A}^{(h)} + \mathbf{b}^{(\mathrm{out})}\mathbf{1}_n^\top \in \mathbb{R}^{t\times n}$$

Output activation

$$\mathbf{A}^{(\mathrm{out})} = \sigma\left(\mathbf{Z}^{(\mathrm{out})}\right) \in \mathbb{R}^{t\times n}$$

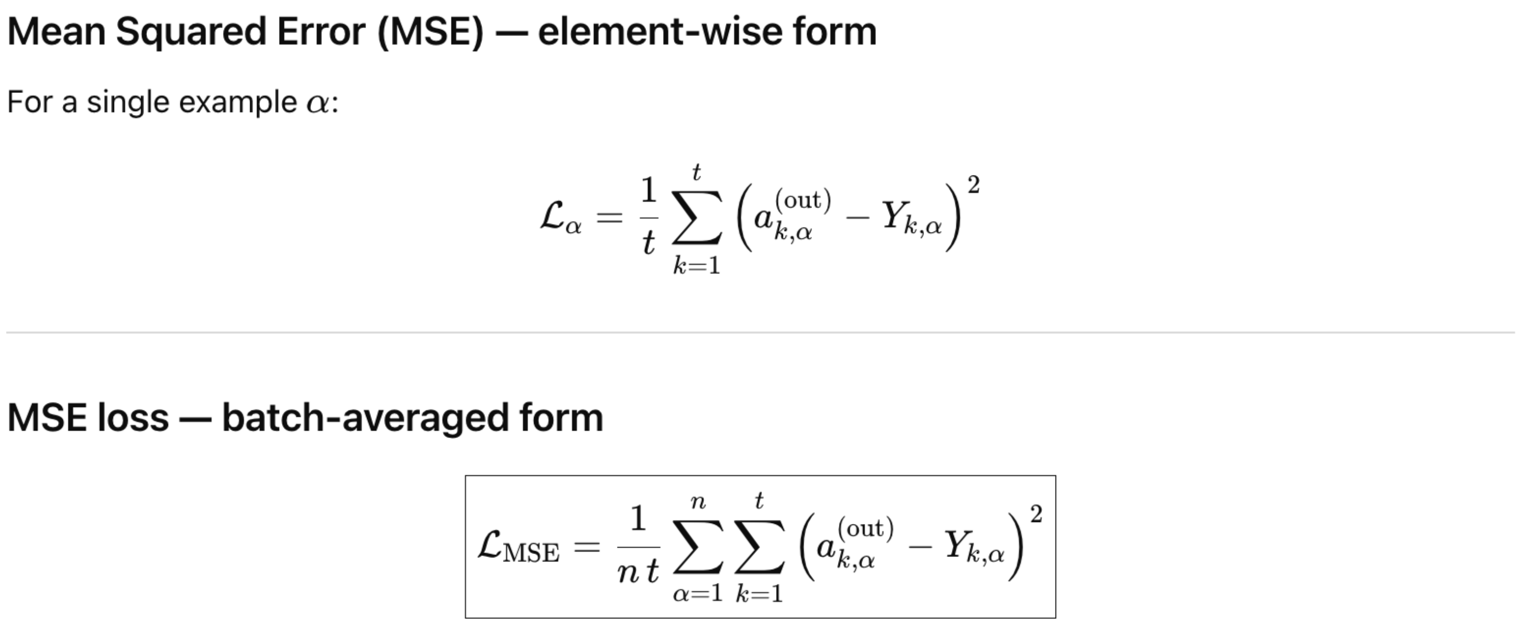
One-line forward pass (batch version)

$$\boxed{\mathbf{A}^{(\mathrm{out})} = \sigma\left(\mathbf{W}^{(\mathrm{out})}\,\sigma\left(\mathbf{W}^{(h)}\mathbf{X}^{(\mathrm{in})} + \mathbf{b}^{(h)}\mathbf{1}_n^\top\right) + \mathbf{b}^{(\mathrm{out})}\mathbf{1}_n^\top\right)}$$

Loss function

- True labels for the n-samples

$$\mathbf{Y} = \begin{bmatrix} | & | & & | \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_n \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{t\times n} \qquad Y_{k,\alpha} \qquad k = 1,\ldots,t,\ \alpha = 1,\ldots,n$$



Mean Squared Error (MSE) — element-wise form

For a single example α:

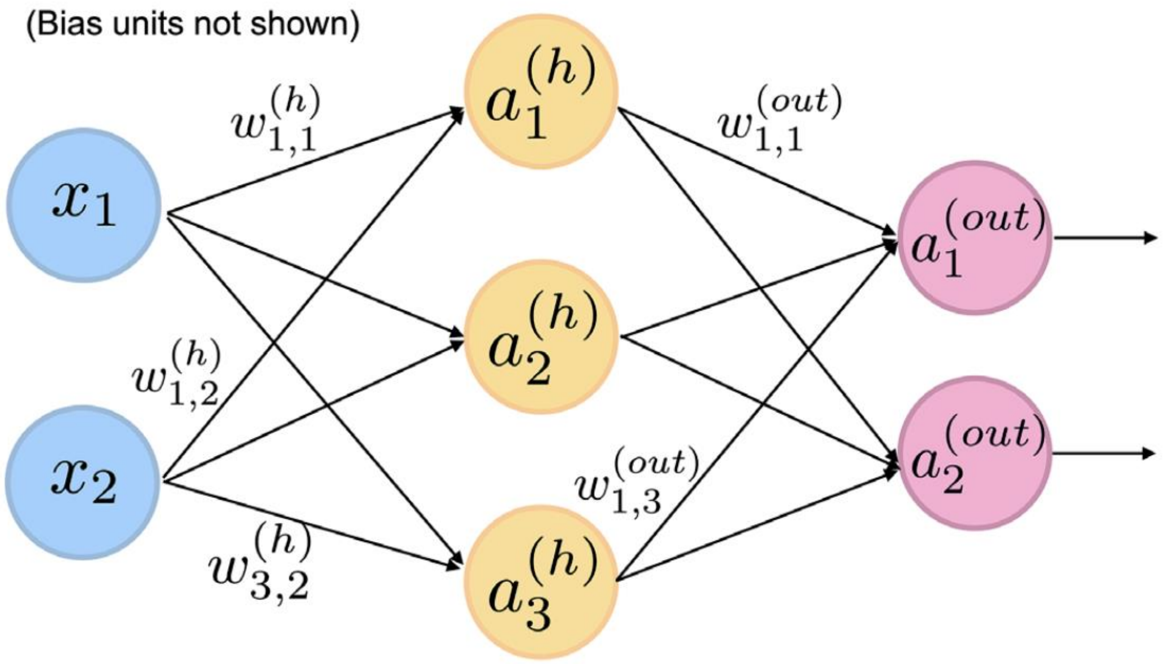
$$\mathcal{L}_\alpha = \frac{1}{t}\sum_{k=1}^{t}\left(a_{k,\alpha}^{(\text{out})} - Y_{k,\alpha}\right)^2$$

MSE loss — batch-averaged form

$$\boxed{\mathcal{L}_{\text{MSE}} = \frac{1}{n\,t}\sum_{\alpha=1}^{n}\sum_{k=1}^{t}\left(a_{k,\alpha}^{(\text{out})} - Y_{k,\alpha}\right)^2}$$

How to update the weights?

- Consider a simple neural network with m=2, d=3, t=2



Forward propagating the input features of a neural network

In total, the number of parameters to optimize is d*m+d+t*d+t

Here, 3*2+3+2*3+2=17

To update the weights, we need to know the gradients

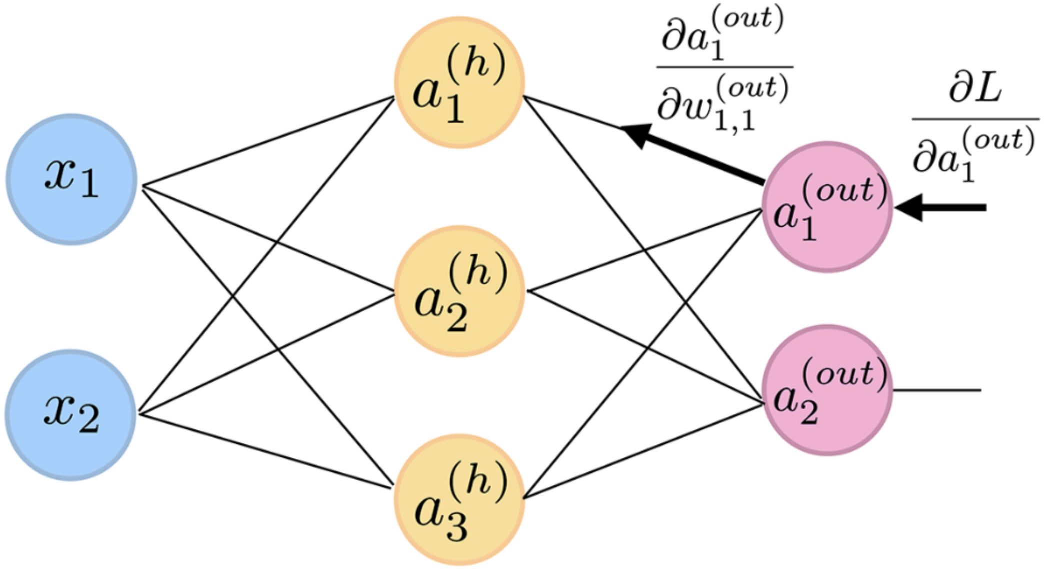
$$w_{j,k}^{(\text{out})} := w_{j,k}^{(\text{out})} - \eta \frac{\partial \mathcal{L}}{\partial w_{j,k}^{(\text{out})}}$$

$$w_{i,j}^{(h)} := w_{i,j}^{(h)} - \eta \frac{\partial \mathcal{L}}{\partial w_{i,j}^{(h)}}$$

Backpropagating the error

- The first output layer weight $w_{1,1}^{(out)}$ was used to compute the output neuron value $a_1^{(out)}$ which was then used to compute the Loss function

- Using the chain rule,



x_1
x_2
$a_1^{(h)}$
$a_2^{(h)}$
$a_3^{(h)}$
$a_1^{(out)}$
$a_2^{(out)}$
$\frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$
$\frac{\partial L}{\partial a_1^{(out)}}$

Gradient for output layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial w_{1,1}^{(out)}}$$

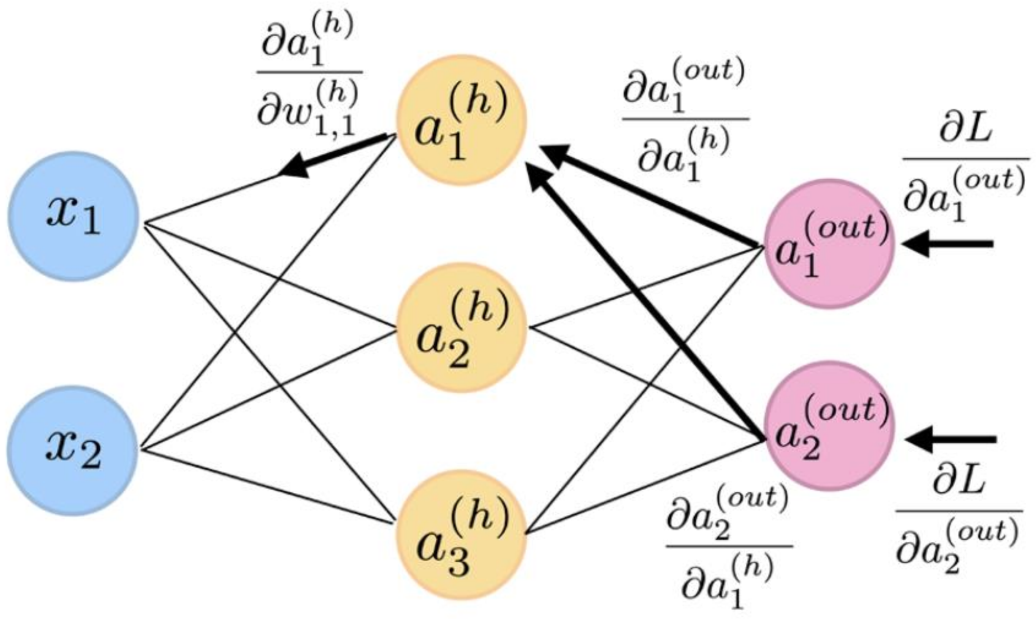
More explicitly,

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \boxed{\frac{\partial L}{\partial a_1^{(out)}}} \cdot \boxed{\frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}} \cdot \boxed{\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}}$$

Backpropagating the error

- The first hidden layer weight $w_{1,1}^{(h)}$ was used to compute the hidden neuron value $a_1^{(h)}$, which was then used to compute the output neuron values $a_1^{(out)}$, $a_2^{(out)}$ and the Loss function

- Using the chain rule,



$\frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}}$
x_1
x_2
$a_1^{(h)}$
$a_2^{(h)}$
$a_3^{(h)}$
$\frac{\partial a_1^{(out)}}{\partial a_1^{(h)}}$
$a_1^{(out)}$
$a_2^{(out)}$
$\frac{\partial L}{\partial a_1^{(out)}}$
$\frac{\partial a_2^{(out)}}{\partial a_1^{(h)}}$
$\frac{\partial L}{\partial a_2^{(out)}}$

Gradient for hidden layer weight:

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}} + \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}}$$

More explicitly,

$$\frac{\partial L}{\partial w_{1,1}^{(h)}} = \boxed{\frac{\partial L}{\partial a_1^{(out)}}} \cdot \boxed{\frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}} \cdot \boxed{\frac{\partial z_1^{(out)}}{\partial a_1^{(h)}}} \cdot \boxed{\frac{\partial a_1^{(h)}}{\partial z_1^{(h)}}} \cdot \boxed{\frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}} + \boxed{\frac{\partial L}{\partial a_2^{(out)}}} \cdot \boxed{\frac{\partial a_2^{(out)}}{\partial z_2^{(out)}}} \cdot \boxed{\frac{\partial z_2^{(out)}}{\partial a_1^{(h)}}} \cdot \boxed{\frac{\partial a_1^{(h)}}{\partial z_1^{(h)}}} \cdot \boxed{\frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}}$$

Derivative of the MSE loss

Ignoring the mini-batch dimension, the MSE loss function is defined as

$$\mathcal{L}_{\text{MSE}} = \frac{1}{t} \sum_{k=1}^{t} \left(y_k - a_k^{(\text{out})} \right)^2 ,$$

Its derivative with respect to a particular output prediction is

$$\boxed{\frac{\partial \mathcal{L}_{\text{MSE}}}{\partial a_k^{(\text{out})}} = \frac{2}{t} \left(a_k^{(\text{out})} - y_k \right) , \qquad k = 1, \ldots, t}$$

Derivative of the sigmoid function

- For hidden layer

For output layer, it is the same

$$\frac{\partial a_j^{(h)}}{\partial z_j^{(h)}} = \frac{\partial}{\partial z_j^{(h)}} \left(\frac{1}{1 + e^{z_j^{(h)}}} \right)$$

$$= \left(\frac{1}{1 + e^{z_j^{(h)}}} \right) \left(1 - \frac{1}{1 + e^{z_j^{(h)}}} \right)$$

$$\boxed{\frac{\partial a_j^{(h)}}{\partial z_j^{(h)}} = a_j^{(h)} \left(1 - a_j^{(h)} \right)}$$

$$\boxed{\frac{\partial a_k^{(\mathrm{out})}}{\partial z_k^{(\mathrm{out})}} = a_k^{(\mathrm{out})} \left(1 - a_k^{(\mathrm{out})} \right)}$$

Derivative of the weight sum (pre-activation)

$$z_j^{(h)} = \sum_{i=1}^{m} w_{j,i}^{(h)} \, x_i^{(\text{in})} + b_j^{(h)} \quad (j = 1, \ldots, d)$$

$$z_k^{(\text{out})} = \sum_{j=1}^{d} w_{k,j}^{(\text{out})} \, a_j^{(h)} + b_k^{(\text{out})} \quad (k = 1, \ldots, t)$$

- With respect to the previous layer neuron values

$$\boxed{\frac{\partial z_j^{(h)}}{\partial x_i^{(\text{in})}} = w_{j,i}^{(h)}}$$

$$\boxed{\frac{\partial z_k^{(\text{out})}}{\partial a_j^{(h)}} = w_{k,j}^{(\text{out})}}$$

- With respect to weights

$$\boxed{\frac{\partial z_j^{(h)}}{\partial w_{j,i}^{(h)}} = x_i^{(\text{in})}}$$

$$\boxed{\frac{\partial z_k^{(\text{out})}}{\partial w_{k,j}^{(\text{out})}} = a_j^{(h)}}$$

Putting all together

- The derivative of the loss w.r.t the **output layer** weights and the bias

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{k,j}^{(\text{out})}} = \delta_k^{(\text{out})}\, a_j^{(h)}}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b_k^{(\text{out})}} = \delta_k^{(\text{out})}}$$

where

$$\boxed{\delta_k^{(\text{out})} := \frac{\partial \mathcal{L}}{\partial z_k^{(\text{out})}} = \frac{2}{t}\left(a_k^{(\text{out})} - y_k\right) a_k^{(\text{out})}\left(1 - a_k^{(\text{out})}\right)}$$

Putting all together

- The derivative of the loss w.r.t the **hidden layer** weights and the bias

$$\boxed{\frac{\partial \mathcal{L}}{\partial w_{j,i}^{(h)}} = \delta_j^{(h)}\, x_i^{(\text{in})}}$$

$$\boxed{\frac{\partial \mathcal{L}}{\partial b_j^{(h)}} = \delta_j^{(h)}}$$

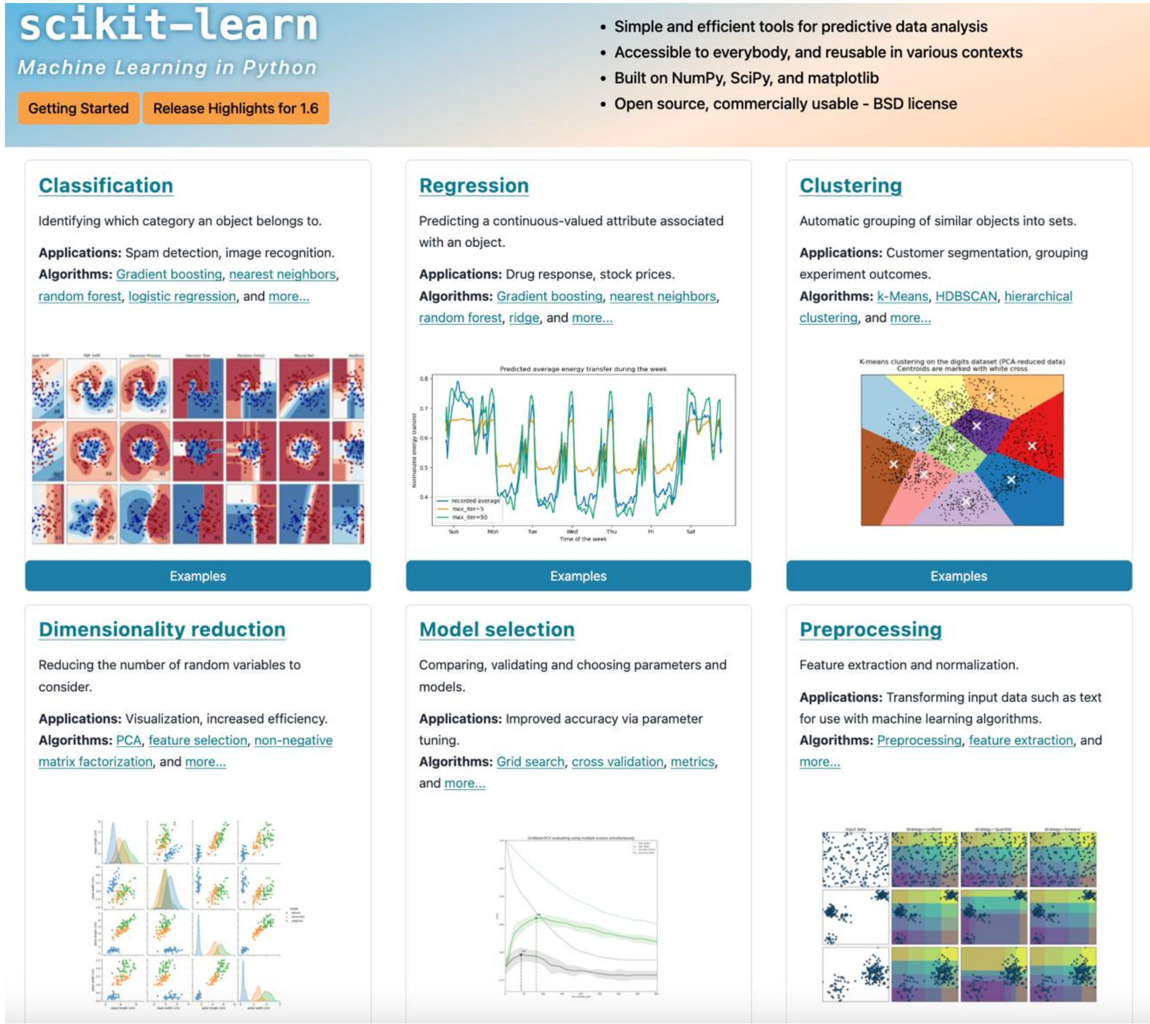
where

$$\boxed{\delta_j^{(h)} := \frac{\partial \mathcal{L}}{\partial z_j^{(h)}} = \left(\sum_{k=1}^{t} \delta_k^{(\text{out})}\, w_{k,j}^{(\text{out})}\right) a_j^{(h)} \left(1 - a_j^{(h)}\right)}$$

Introduction to Scikit-learn

Scikit-learn

- Offers a user-friendly and consistent interface for using popular ML algorithms efficiently and productively


scikit-learn
Machine Learning in Python
Getting Started
Release Highlights for 1.6
Predicted average energy transfer during the week
K-means clustering on the digits dataset (PCA-reduced data)
Centroids are marked with white cross
Examples
Examples
Examples

- Simple and efficient tools for predictive data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying which category an object belongs to.

Applications: Spam detection, image recognition.
Algorithms: Gradient boosting, nearest neighbors, random forest, logistic regression, and more...

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, stock prices.
Algorithms: Gradient boosting, nearest neighbors, random forest, ridge, and more...

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, grouping experiment outcomes.
Algorithms: k-Means, HDBSCAN, hierarchical clustering, and more...

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, increased efficiency.
Algorithms: PCA, feature selection, non-negative matrix factorization, and more...

Model selection

Comparing, validating and choosing parameters and models.

Applications: Improved accuracy via parameter tuning.
Algorithms: Grid search, cross validation, metrics, and more...

Preprocessing

Feature extraction and normalization.

Applications: Transforming input data such as text for use with machine learning algorithms.
Algorithms: Preprocessing, feature extraction, and more...

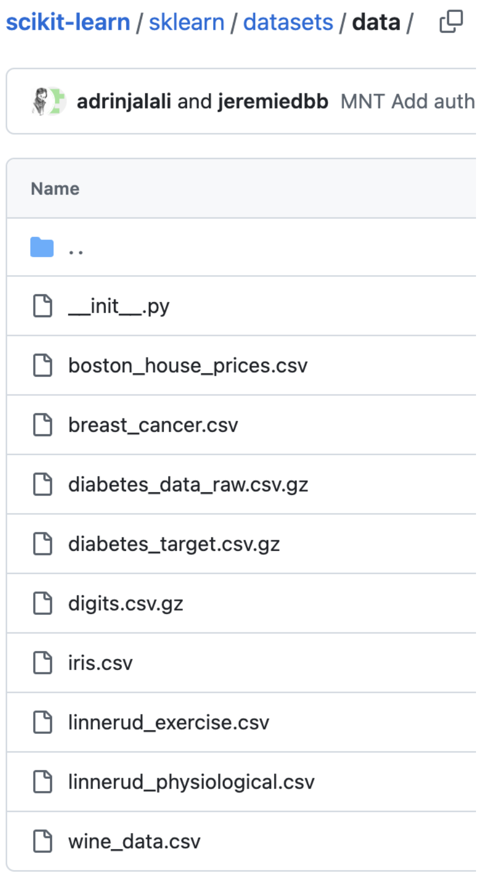
https://scikit-learn.org/stable/

- ✓ Classification
- ✓ Regression
- ✓ Clustering
- ✓ Decision trees
- ✓ Ensemble methods
- ✓ Nearest neighbors
- ✓ Support vector machines
- ✓ Dimensionality reduction
- ✓ Model selection
- ✓ Preprocessing

```
pip install scikit-learn
```

Datasets in scikit-learn

- Small Toy Datasets (CSV-based datasets)



scikit-learn / sklearn / datasets / data /

adrinjalali and jeremiedbb MNT Add auth

Name
..
__init__.py
boston_house_prices.csv
breast_cancer.csv
diabetes_data_raw.csv.gz
diabetes_target.csv.gz
digits.csv.gz
iris.csv
linnerud_exercise.csv
linnerud_physiological.csv
wine_data.csv

- Use load_*() to access

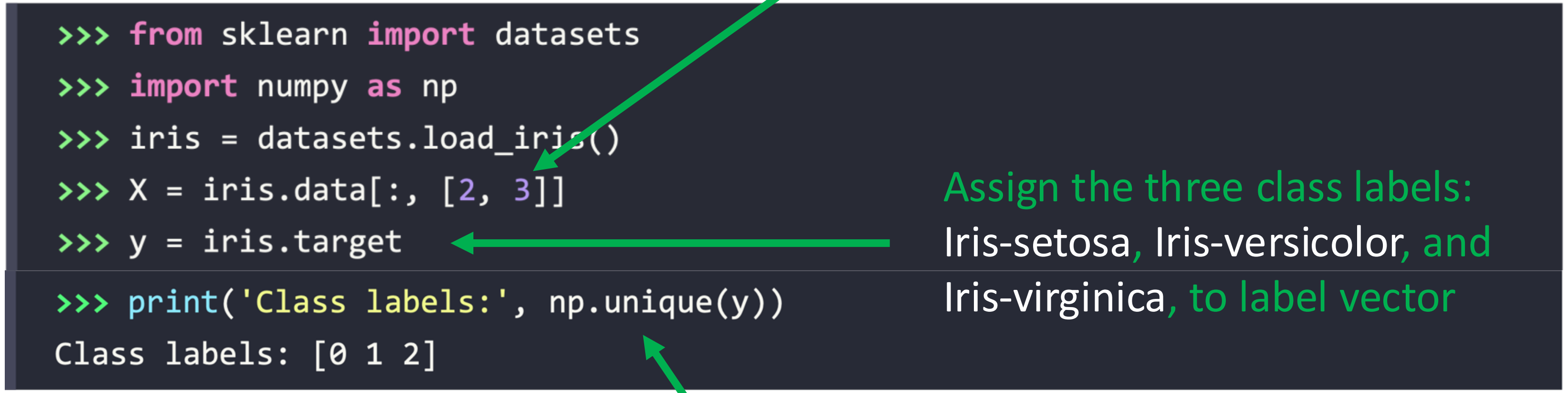
- Scikit-learn can also fetch larger real-world datasets from external sources (e.g., OpenML, UCI, or LIBSVM) via fetch_*() functions

- Additionally, scikit-learn provides functions to generate artificial datasets

Load and extract Iris-flower dataset

- Instead of implementing the perceptron rule and Adaline in Python by ourselves, we use the scikit-learn library

Assign the petal length and petal width of the 150 flower examples to the feature matrix



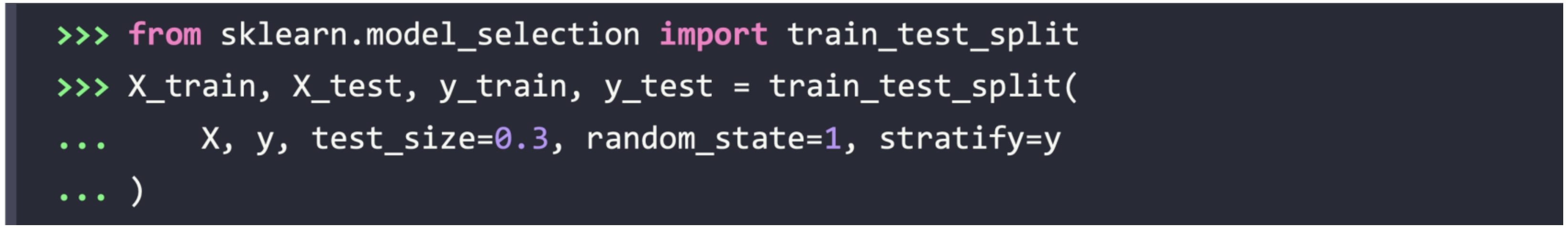
```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

Assign the three class labels: Iris-setosa, Iris-versicolor, and Iris-virginica, to label vector

Return the three labels as integers
(a common convention among most ML libraries)

Separate the training and test examples

- To evaluate how well a trained model performs on **unseen data**, split the dataset into separate **training and test datasets**



```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y
... )
```

- Import the **train_test_split** function, from the **model_selection** module, which is a part of the **scikit-learn** library

- The **train_test_split** function shuffles the dataset internally before splitting

- The **number of test dataset** is test_size*total number of dataset = 0.3*150=45

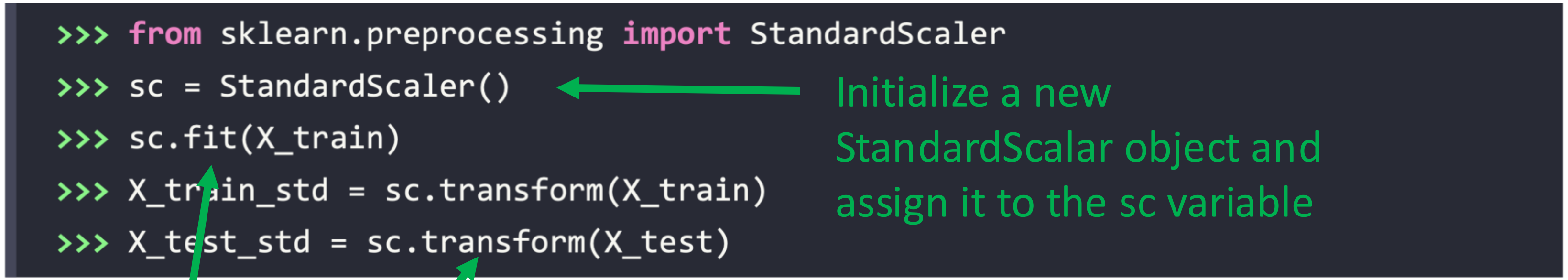
- **random_state=1**: to provide a fixed random seed for internal pseudo-random number generator (which is used to shuffle the dataset) → useful to reproduce the results in the future

Feature scaling

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

standardization

- **Feature scaling** with **StandardScalar** class from scikit-learn's **preprocessing** module



```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

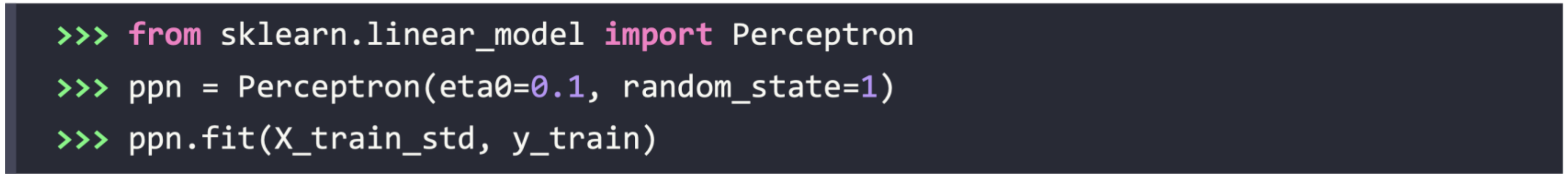
Initialize a new StandardScalar object and assign it to the sc variable

- Using the **fit** method, the sample ***mean*** and ***standard deviation*** are estimated, for each feature variable

- Using the **transform** method, we standardize the data using the estimated mean and std.

Training a perceptron

- Now train a perceptron model
- Scikit-learn supports multiclass classification [via the one-versus-rest (OvR) method]



```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

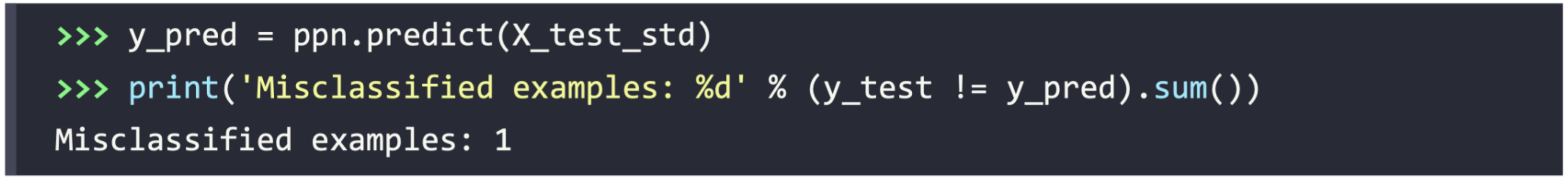
- Import the **Perceptron** class from the **linear_model** module

- Initialize the new Perceptron object (and provide the learning rate 0.1 and a random seed 1)

- The **fit** method trains the model

Predictions and performance metrics

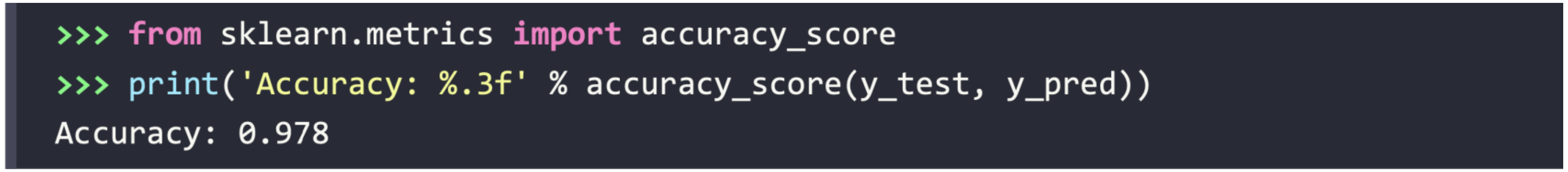
- Make predictions via the **predict** method



```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified examples: %d' % (y_test != y_pred).sum())
Misclassified examples: 1
```

The **misclassification error** on the test data set is 1/45, approximately 2.2%
Or
The **classification accuracy** is 1 - error = 97.8%

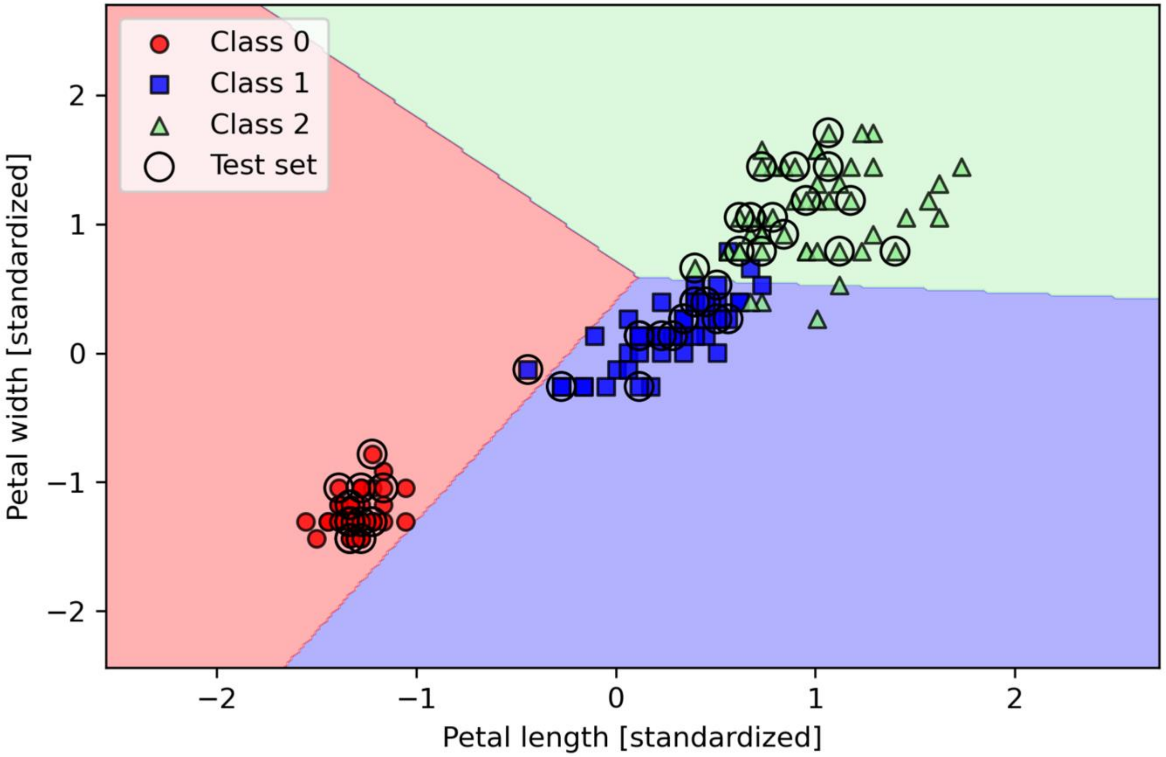
- You can also use the **metrics** module



```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Accuracy: 0.978
```

Visualization

- Visualizing the decision boundaries of a multi-class perceptron model
- See demo code: **demo_sklearn_perceptron.ipynb**


● Class 0
■ Class 1
△ Class 2
○ Test set
Petal width [standardized]
Petal length [standardized]