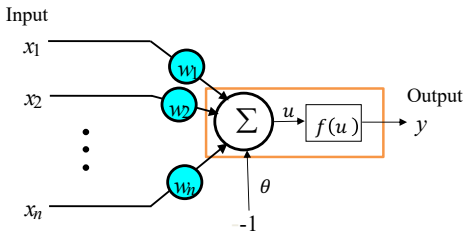
Chapter 1

Fundamentals of Neural Networks

Neural networks and deep learning

Artificial Neuron



Input vector $\boldsymbol{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$
weight vector $\boldsymbol{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$
n is the number of inputs.

Artificial Neuron

An **artificial neuron** is the basic unit of neural networks.

Basic elements of an artificial neuron:

• A set of **input** signals: the input is a vector $\boldsymbol{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$ where n is the number (or the dimension) of input signals. Inputs are also referred to as **features**.

• Inputs are connected to the neuron via synaptic connections whose strengths are represented by their **weights**.

• The weight vector $\boldsymbol{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$ where w_i is the synaptic weight connecting i th input of the neuron.

Notation

Vectors are denoted in bold and
written as horizontally with a transpose (^{T}).

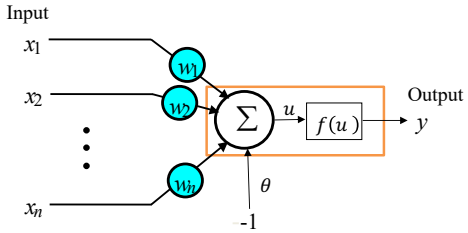
$$\boldsymbol{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$$

Example

$$\boldsymbol{x} = \begin{pmatrix} 1.2 \\ -0.5 \\ 3.5 \\ 2.1 \end{pmatrix} = (1.2 \quad -0.5 \quad 3.5 \quad 2.1)^T$$

$$\boldsymbol{x}^T = (1.2 \quad -0.5 \quad 3.5 \quad 2.1)$$

Artificial Neuron



The total **synaptic input** u to the neuron is given by the sum of the products of the inputs and their corresponding connecting weights minus the **threshold** of the neuron.

The total synaptic input to a neuron, u is given by

$$u = w_1x_1 + w_2x_2 + \cdots w_nx_n - \theta = \sum_{i=1}^{n} w_ix_i - \theta$$

where θ is the threshold of the neuron.

By using vector notations:

$$u = \boldsymbol{w}^T\boldsymbol{x} - \theta$$

Artificial Neuron

The **activation function** f relates synaptic input to the activation of the neuron.

$f(u)$ denotes the **activation** of the neuron.

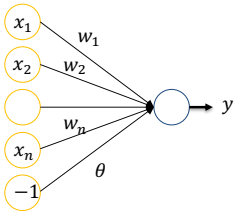
For some neurons, the **output** y is equal to the activation of the neuron.

$$y = f(u)$$

Note that activation is not generally equal to the output of the neuron.

Bias vs Threshold

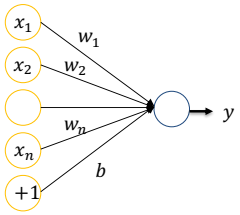
The threshold is often considered as a weight with a fixed input of -1. Often the threshold is represented as a **bias** b that receives constant +1 input.



Threshold

$u = \boldsymbol{w}^T\boldsymbol{x} - \theta$

$y = f(u)$



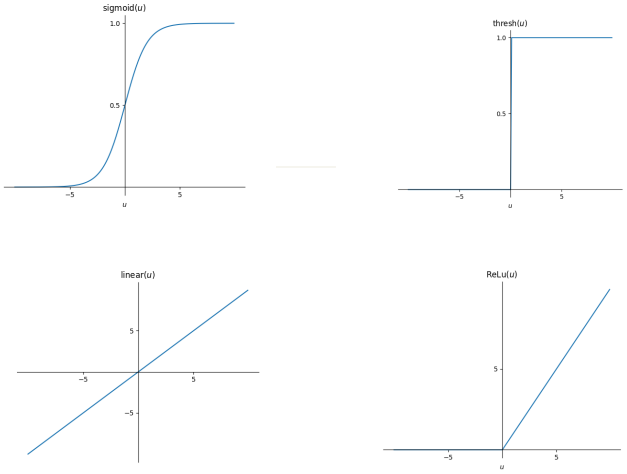
Bias

$u = \boldsymbol{w}^T\boldsymbol{x} + b$

$y = f(u)$

Bias $b = -\theta$

Activation functions



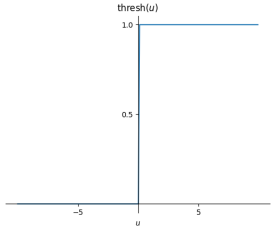
Activation functions

For ***threshold*** **(unit step*)* activation function**, the activation is given by

$$f(u) = theshold(u) = 1(u > 0)$$

where $1(\cdot)$ is the *Indictor function* or *Unit-step function:*

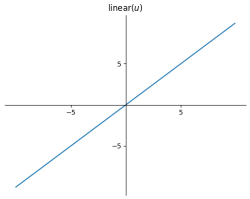
$$1(x) = \begin{cases} 1, & x \text{ is True} \\ 0, & x \text{ is False} \end{cases}$$



Activation functions

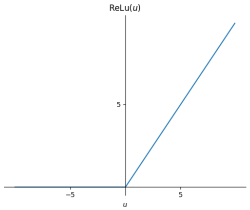
A neuron with ***linear*** activation function can be written as

$$f(u) = linear(u) = u$$



The ***ReLU*** **(rectified-linear unit)** activation function can be written as

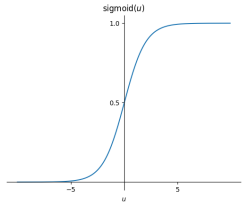
$$f(u) = relu(u) = \max\{0, u\}$$



Sigmoid activation function

The sigmoidal is known as the **logistic function** or simply **sigmoid function**

$$f(u) = sigmoid(u) = \frac{1}{1 + e^{-u}}$$



In general, the ***sigmoid*** activation function can be written as

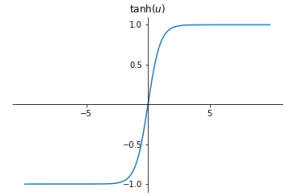
$$f(u) = \frac{a}{1 + e^{-bu}}$$

a is the gain (amplitude) and b is the slope.

But often, $a = 1.0$ and $b = 1.0$.

Tanh activation function

$$f(u) = tanh(u) = \frac{e^{u} - e^{-u}}{e^{u} + e^{-u}}$$

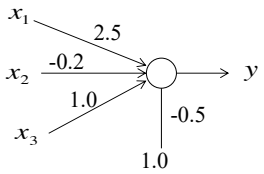


Tanh activation function has the same shape as sigmoidal and spans from -1 and +1. It is also known as **bipolar sigmoidal**.

Sigmoidal is the most pervasive and biologically plausible activation function. Since sigmoid function is *differentiable*, it leads to mathematically attractive neuronal models.

Example 1

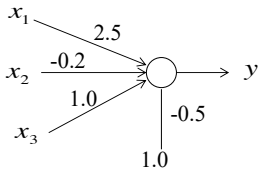
The artificial neuron in the figure receives 3-dimensional inputs $\boldsymbol{x} = (x_1 \quad x_2 \quad x_3)^T$ and has an activation function given by $f(u) = \frac{0.8}{1+e^{-1.2u}}$.



Find the synaptic input and the output of the neuron for inputs: $\begin{pmatrix} 0.8 \\ 2.0 \\ -0.5 \end{pmatrix}$ and $\begin{pmatrix} -0.4 \\ 1.5 \\ 1.0 \end{pmatrix}$.

Example 1

$\boldsymbol{w} = \begin{pmatrix} 2.5 \\ -0.2 \\ 1.0 \end{pmatrix}, \qquad b = -0.5$



Consider $\boldsymbol{x} = \begin{pmatrix} 0.8 \\ 2.0 \\ -0.5 \end{pmatrix}$

Synaptic input $u = \boldsymbol{w}^T\boldsymbol{x} + b = (2.5 \quad -0.2 \quad 1.0)\begin{pmatrix} 0.8 \\ 2.0 \\ -0.5 \end{pmatrix} - 0.5 = 0.6$

Output = $\mathrm{y} = f(u) = \frac{0.8}{1+e^{-1.2u}} = \frac{0.8}{1+e^{-1.2\times 0.6}} = 0.538$

Similarly, for $\boldsymbol{x} = \begin{pmatrix} -0.4 \\ 1.5 \\ 1.0 \end{pmatrix}$, $u = -0.8$ and output $\mathrm{y} = f(u) = 0.222$

PyTorch 2.0

PyTorch/Tensorflow is about processing of **tensors**. Tensor is a multidimensional array.

Rank refers to the number of dimensions and **shape** gives the sizes of each dimension of the tensor.

```
3. # a rank 0 tensor; a scalar with shape [],

[1., 2., 3.] # a rank 1 tensor; a vector with shape [3]

[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]

[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```


Computational Graph

PyTorch program involves building and evaluation of a **computational graph.**

A computational graph is a series of tensor **operations** arranged into a graph.

- Nodes of the graph represent tensor operations
- Edges represent values (tensors) that follow through the graph

Computational graph of a neuron

multiply
add

$u = \boldsymbol{w}^T\boldsymbol{x} + b$
$\mathrm{y} = f(u)$

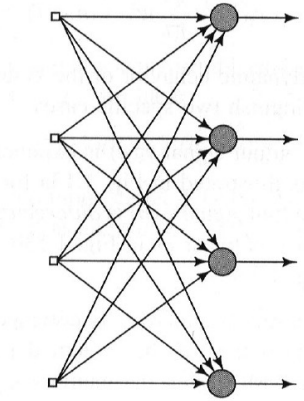
Torch Implementation of Example 1

```
import torch

# a class for neuron
class Neuron():
   # initiate a neuron class with weights and biases (initiate the object)
   def __init__(self):
       self.w = torch.tensor([2.5, -0.2, 1.0])
       self.b = torch.tensor(-0.5)

  # evaluate the neuron (implement a function)
  def __call__(self, x):
      u = torch.inner(self.w, x) + self.b
      y = 0.8/(1+torch.exp(-1.2*u))
   return u, y

# create a neuron
neuron = Neuron()

# evaluate
u, y = neuron(torch.tensor([0.8, 2.0, -0.5]))

# print: u = 0.600, y = 0.538
```


NN Architectures: neuron layers

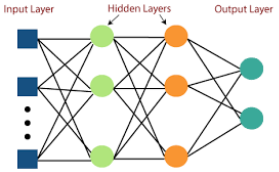
Single – layer of neurons

- Comprised of an input layer of source units that inject into an output layer of neurons.
- A fully-connected layer



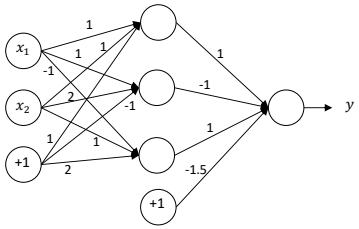
NN architectures: multilayer feedforward networks

- Comprised of more than one layer of neurons. Layers between input source nodes and output layer is referred to as *hidden layers*.
- Multilayer neural networks can handle *more complicated* and *larger scale problems* better than single-layer networks.
- However, training multilayer network may be *more difficult* and *time-consuming*.



Three-layer network

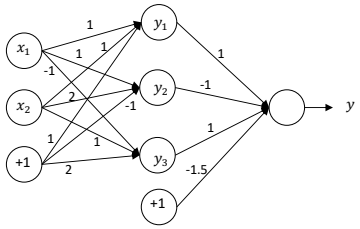
Example 2



Two-layer neural network receives 2-dimensional inputs $(x_1, x_2) \in \boldsymbol{R}^2$ and has one output neuron and three hidden neurons. All the neurons have unit step activation functions. The weights of the connections are given in the figure. Find the space of inputs for which the output $y = 1.0$.

Find the output for inputs $(0.0, 0.0)$, $(2.0, 2.0)$, and $(-1.0, 1.0)$

Example 2



Synaptic inputs:

$$u_1 = x_1 + x_2 + 1$$

Output $y_1 = 1(u_1 > 0)$

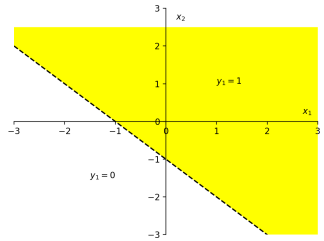
$$u_2 = x_1 + 2x_2 - 1$$
$$y_2 = 1(u_2 > 0)$$

$$u_3 = -x_1 + x_2 + 2$$
$$y_3 = 1(u_3 > 0)$$

$$u = y_1 - y_2 + y_3 - 1.5$$
$$y = 1(u > 0)$$

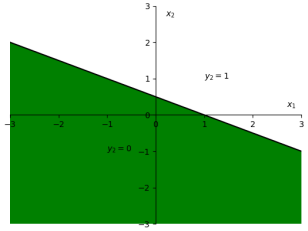
$$y_1 = f(u_1) = 1(u_1 > 0)$$

Boundary: $u_1 = x_1 + x_2 + 1 = 0 \rightarrow x_2 = -x_1 - 1$

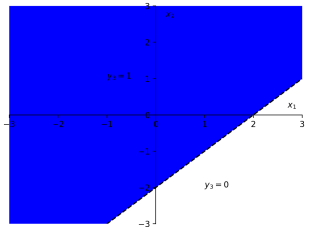


The boundary line is obtained by setting $u_1 = 0$; and for one side of the boundary, $y = 1$ and on other side $y_1 = 0$.

$$u_2 = 2x_2 + x_1 - 1 = 0 \rightarrow x_2 = -0.5x_1 + 0.5$$



$$u_3 = x_2 - x_1 + 2 = 0 \rightarrow x_2 = x_1 - 2$$



Output layer neuron:

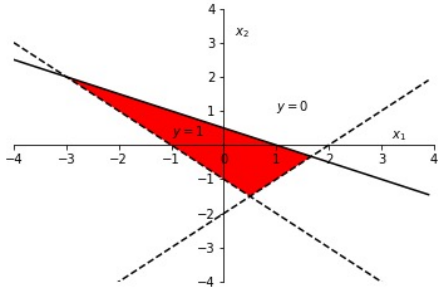
$$u = y_1 - y_2 + y_3 - 1.5$$
$$y = 1(u > 0)$$

Note that $y_1, y_2, y_3 \in \{0, 1\}$

y_1	y_2	y_3	u	y
0	0	0	-1.5	0
0	0	1	-0.5	0
0	1	0	-2.5	0
0	1	1	-1.5	0
1	0	0	-0.5	0
1	0	1	0.5	1
1	1	0	-1.5	0
1	1	1	-0.5	0

$$Y = Y_1 \overline{Y}_2 Y_3$$

$y = 1$ region is given by the intersection of regions: y_1=1, y_2=0, and y_3=1.



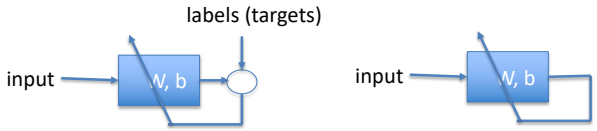
$$x = (0.0, 0.0) \rightarrow y = 1$$
$$x = (2.0, 2.0) \rightarrow y = 0$$
$$x = (-1.0, 1.0) \rightarrow y = 1$$

Note that networks of *discrete perceptrons* (neurons with threshold activation functions) can implement Boolean functions.

Training (or learning) of neural networks

Neural networks attain their operating characteristics through **learning** (or **training**) with training examples. During training, the weights or the strengths of connections are gradually adjusted iteratively to achieve their desirable labels (or **targets**).

Training may be either **supervised** or **unsupervised**.



Supervised Learning **Unsupervised Learning**

Supervised learning

Learning of a neuron or neural network is usually performed in order to minimize a **cost function** (**loss function** or **error function**).

The cost function $J(\boldsymbol{W}, \boldsymbol{b})$ is of an artificial neuron is typically a multi-dimensional function that depends on weights $\boldsymbol{W}$ and the biases $\boldsymbol{b}$. The neuron learning attempts to find the optimal weights $\boldsymbol{W}^*$ and biases $\boldsymbol{b}^*$ that minimize the error function:

$$\boldsymbol{W}^*, \boldsymbol{b}^* = arg \min_{\boldsymbol{W},\boldsymbol{b}} J(\boldsymbol{W}, \boldsymbol{b})$$

Given a set of training patterns, the parameters (weights and biases) minimizing cost function are learned in an iterative procedure. In each iteration, small changes of weights $\Delta\boldsymbol{W}$ and biases $\Delta\boldsymbol{b}$ are made:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \Delta\boldsymbol{W}$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \Delta\boldsymbol{b}$$

Supervised and unsupervised learning

Supervised Learning:
For each training input pattern, the network is presented with the correct **target label** (the desired output).

Unsupervised Learning:
For each training input pattern, the network adjusts weights *without knowing* the correct target.

In unsupervised training, the network **self-organizes** to classify similar input patterns into clusters.

Gradient descent learning

The grading descent procedure states that the weights $\boldsymbol{W}$ (and biases $\boldsymbol{b}$) are updated during learning by searching in the direction of and proportional the **negative gradient** of the cost function.

That is, the change of the weight vector:

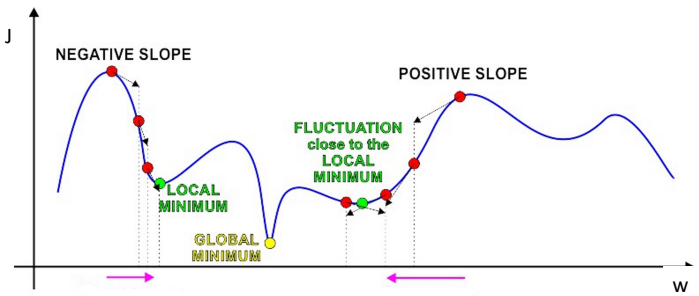
$$\Delta\boldsymbol{W} \propto -\frac{\partial J(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{w}}$$
$$\Delta\boldsymbol{W} = -\alpha\frac{\partial J(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}}$$

where $\frac{\partial J(\boldsymbol{W},\boldsymbol{b})}{\partial \boldsymbol{W}}$ is the gradient (partial derivative) of cost with respect to weight $\boldsymbol{W}$ and α is *learning factor* or *learning rate*. $\alpha \in (0.0, 1.0]$.

The **gradient descent equations** for learning the weights is given by substituting above in

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \Delta\boldsymbol{W}$$

Gradient Descent Leaning



Gradient descent learning

The **gradient descent equations** for the weights is given by

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}}$$

Similarly, for the bias

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}}$$

Notation: $\nabla_{\boldsymbol{W}} J = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{W}}$ and $\nabla_{\boldsymbol{b}} J = \frac{\partial J(\boldsymbol{W}, \boldsymbol{b})}{\partial \boldsymbol{b}}$.

Gradient descent learning is given by

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J$$

Note: Will drop the arguments in J.

Gradient descent learning

Given a set of training examples
Initialize weight $\boldsymbol{W}$ and bias $\boldsymbol{b}$
Set the learning parameter α
Iterate until convergence:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J$$

Convergence is achieved by observing one of the following:

1. No changes in weights and biases
2. No difference between the outputs and targets
3. No decrease in the cost function J

Training dataset

Training data are also referred to as **training examples** or **training patterns.** For supervised learning, a training pattern consists of a pair consisting of input pattern and the corresponding target.

A training dataset is a set of training examples: $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$
or $\{(\boldsymbol{x}_1, d_1), (\boldsymbol{x}_2, d_2), \cdots (\boldsymbol{x}_P, d_P)\}$

$\boldsymbol{x}_p$ is the input (features) and d_p is the target (desired label) of p th training pattern. P is the number of examples in the dataset.

The input is usually n-dimensional and written as:

$$\boldsymbol{x}_p = (x_{p1}, x_{p2}, \cdots x_{pn})^T$$

Stochastic Gradient Descent (SGD) learning

Given training examples $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$
Set learning factor α
Initialize $(\boldsymbol{W}, \boldsymbol{b})$
Iterate until convergence:
  for each pattern $(\boldsymbol{x}_p, d_p)$:
    $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J_p$
    $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J_p$

- In each **epoch** or cycle of iteration, the learning takes place individually over every pattern
- The cost J_p is individually computed from the output and the target of the p th training pattern.

Batches of Data

Inputs are often presented as a batch in a **data matrix** $\boldsymbol{X}$ and a **target vector** $\boldsymbol{d}$. The input datapoints (or patterns) are written as rows in the data matrix and the targets are written into a single vector in the target vector.

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$.

Data matrix:

$$\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_P^T \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ x_{P1} & x_{P2} & \cdots & x_{Pn} \end{pmatrix}$$

Target vector:

$$\boldsymbol{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$$

(Batch) Gradient descent learning

Given a set of training patterns: $(\boldsymbol{X}, \boldsymbol{d})$
Set learning factor α
Initialize $(\boldsymbol{W}, \boldsymbol{b})$
Iterate until convergence:
  $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J$
  $b \leftarrow b - \alpha \nabla_b J$

The cost J is computed using <u>all</u> the training patterns.
That is, using $(\boldsymbol{X}, \boldsymbol{d})$
In each epoch, the weights are updated once considering all the input patterns.

Summary

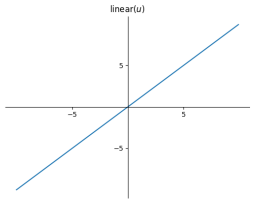
- Analogy between biological and artificial neurons
- Transfer function of artificial neuron:
  $$u = \boldsymbol{w}^T \boldsymbol{x} + b$$
  $$y = f(u)$$
- Types of activation functions: sigmoid, threshold, linear, ReLU, and tanh.
- Given inputs, to find the outputs for simple feedforward networks
- Supervised and unsupervised learning
- Gradient descent learning:
  $$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J$$
  $$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J$$
- Stochastic gradient descent (SGD) and batch gradient descent (GD)

Chapter 2

Regression and classification

Neural networks and deep learning

Linear neuron



Synaptic input u to a neuron is given by

$$u = \boldsymbol{w}^T \boldsymbol{x} + b$$

A linear neuron has a *linear activation function*. That is,

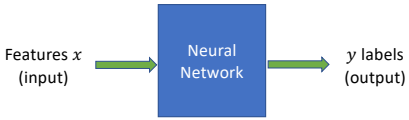
$$y = f(u) = u$$

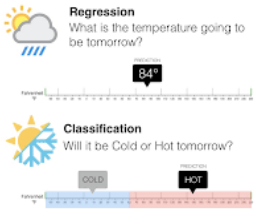
A linear neuron with weights $\boldsymbol{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$ and bias b has an output:

$$y = \boldsymbol{w}^T \boldsymbol{x} + b$$

where input $\boldsymbol{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T \in \boldsymbol{R}^n$ and output $y \in \boldsymbol{R}$.

Regression and classification





Primarily, neural network are used to *predict* output **labels** from input **features**.

Prediction tasks can be classified into two categories:

- **Regression:** the labels are continuous (age, income, height, etc.)
- **Classification:** the labels are discrete (sex, digits, type of flowers, etc.),

Training finds network weights and biases that are optimal for **prediction** of labels from features.

Linear neuron performs linear regression

Representing a dependent (output) variable as a linear combination of independent (input) variables is known as **linear regression**.

The output of a linear neuron can be written as

$$y = w_1 x_1 + w_2 x_2 \cdots + w_n x_n + b$$

where $x_1, x_2, \cdots x_n$ are the inputs. That is, a linear neuron performs linear regression and the weights and biases (that is, b and $w_1, \ldots w_n$) act as regression coefficients. The above function forms a **hyperplane** in Euclidean space $\boldsymbol{R}^n$.

Given a training examples $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$ where input $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and target $d_p \in \boldsymbol{R}$, training a linear neuron finds a linear mapping $\phi: \boldsymbol{R}^n \rightarrow \boldsymbol{R}$ given by:

$$y = \boldsymbol{w}^T \boldsymbol{x} + b$$

Stochastic gradient descent (SGD) for linear neuron

The **cost function** J for regression is usually given as the *square error* (s.e.) between neuron outputs and targets.

Given a training pattern $(\boldsymbol{x}, d), \frac{1}{2}$ square error cost J is defined as

$$J = \frac{1}{2}(d - y)^2$$

where y is neuron output for input pattern $\boldsymbol{x}$ and d is the target label.

$$y = \boldsymbol{w}^T\boldsymbol{x} + b$$

The ½ in the cost function is introduced to simplify learning equations and does not affect the optimal values of the parameters (weights and bias).

SGD for linear neuron

$$J = \frac{1}{2}(d - y)^2$$
$$y = u = \boldsymbol{w}^T\boldsymbol{x} + b$$

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial y} = -(d - y) \quad \text{(A)}$$

$$\nabla_{\boldsymbol{w}} J = \frac{\partial J}{\partial \boldsymbol{w}} = \frac{\partial J}{\partial u}\frac{\partial u}{\partial \boldsymbol{w}} \quad \text{(B)}$$

SGD for linear neuron

$$u = \boldsymbol{w}^T\boldsymbol{x} + b = w_1x_1 + w_2x_2 \cdots + w_nx_n + b$$

$$\frac{\partial u}{\partial \boldsymbol{w}} = \begin{pmatrix} \frac{\partial u}{\partial w_1} \\ \frac{\partial u}{\partial w_2} \\ \vdots \\ \frac{\partial u}{\partial w_n} \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \boldsymbol{x} \quad \text{(C)}$$

$$\frac{\partial u}{\partial \boldsymbol{w}} = \boldsymbol{x}$$

Substituting (A) and (C) in (B),

$$\nabla_{\boldsymbol{w}} J = -(d - y)\boldsymbol{x} \quad \text{(D)}$$

Similarly, since $\frac{\partial u}{\partial b} = 1$,

$$\nabla_b J = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b} = -(d - y) \quad \text{(E)}$$

SGD for linear neuron

Gradient learning equations:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}} J$$
$$b \leftarrow b - \alpha\nabla_b J$$

By substituting from (D) and (E), SGD equations for a linear neuron are given by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d - y)\boldsymbol{x}$$
$$b \leftarrow b + \alpha(d - y)$$

SGD algorithm for linear neuron

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$

Set learning parameter α

Initialize $\boldsymbol{w}$ and b

Repeat until convergence:

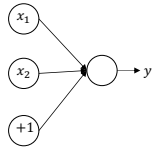
For every training pattern $(\boldsymbol{x}_p, d_p)$:

$$y_p = \boldsymbol{w}^T \boldsymbol{x}_p + b$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d_p - y_p)\boldsymbol{x}_p$$

$$b \leftarrow b + \alpha(d_p - y_p)$$

Example 1



Let's initialize weights randomly and biases to zeros

$\boldsymbol{w} = \begin{pmatrix} 0.92 \\ 0.71 \end{pmatrix}$ and $b = 0.0$

$\alpha = 0.01$

Need to shuffle the patterns before every epoch.

Example 1: SGD on a linear neuron

Train a linear neuron to perform the following mapping, using stochastic gradient descent (SGD) learning:

$\boldsymbol{x}^T = (x_1, x_2)$	$\boldsymbol{d}$
$(0.54, -0.96)$	1.33
$(0.27, 0.50)$	0.45
$(0.00, -0.55)$	0.56
$(-0.60, 0.52)$	-1.66
$(-0.66, -0.82)$	-1.07
$(0.37, 0.91)$	0.30

Use a learning factor $\alpha = 0.01$.

Example 1: epoch 1

Epoch 1 begins

Shuffle the patterns ….

First pattern $p = 1$ is applied:

$\boldsymbol{x}_p = \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix}$ and $d_p = 1.33$

$$y_p = \boldsymbol{w}^T \boldsymbol{x}_p + b = (0.92 \quad 0.71)\begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix} + 0.0 = -0.19$$

s.e. = $(d_p - y_p)^2 = 2.292$

$$\boldsymbol{w} = \boldsymbol{w} + \alpha(d_p - y_p)\boldsymbol{x}_p = \begin{pmatrix} 0.92 \\ 0.71 \end{pmatrix} + 0.01 \times (1.33 + 0.19)\begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix} = \begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$$

$$b = b + \alpha(d_p - y_p) = 0 + 0.01 \times (1.33 + 0.19) = 0.02$$

Example 1: epoch 1

Second pattern $p = 2$ is applied:
$\boldsymbol{x}_p = \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix}$ and $d_p = -1.07$

$y_p = \boldsymbol{w}^T \boldsymbol{x}_p + b = (0.93 \quad 0.70) \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix} + 0.02 = -0.19$
s.e. = $(d_p - y_p)^2 = 0.01$

$\boldsymbol{w} = \boldsymbol{w} + \alpha(d_p - y_p)\boldsymbol{x}_p = \begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix} + 0.01 \times (-1.07 + 0.19) \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix} = \begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$
$b = b + \alpha(d_p - y_p) = 0.02 + 0.01 \times (1.33 + 0.19) = 0.02$

Iterations continues for patterns $p = 3, \cdots 6$.

the second epoch starts …..
Shuffle the patterns
Apply patterns $p = 1,2, \cdots 6$

Training epochs continue until convergence.

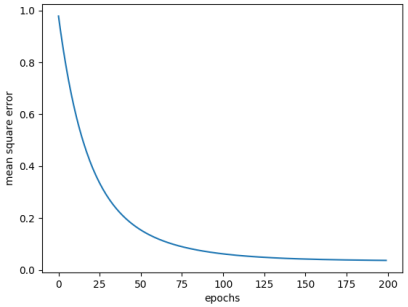
Example 1: epoch 200

$\boldsymbol{x}_p$	y_p	s.e.	$\boldsymbol{w}$	b
$\boldsymbol{x}_1 = \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix}$	1.49	0.03	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	−0.01
$\boldsymbol{x}_2 = \begin{pmatrix} 0.00 \\ -0.55 \end{pmatrix}$	0.22	0.12	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	−0.01
$\boldsymbol{x}_3 = \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix}$	−0.98	0.01	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	−0.01
$\boldsymbol{x}_4 = \begin{pmatrix} 0.37 \\ 0.91 \end{pmatrix}$	0.33	0.00	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	−0.01
$\boldsymbol{x}_5 = \begin{pmatrix} 0.27 \\ 0.50 \end{pmatrix}$	0.30	0.02	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	−0.01
$\boldsymbol{x}_6 = \begin{pmatrix} -0.60 \\ 0.52 \end{pmatrix}$	−1.45	0.04	$\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$	−0.01

Example 1: epoch 1

$\boldsymbol{x}_p$	y_p	s.e.	$\boldsymbol{w}$	b
$\boldsymbol{x}_1 = \begin{pmatrix} 0.54 \\ -0.96 \end{pmatrix}$	−0.19	2.29	$\begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$	0.02
$\boldsymbol{x}_2 = \begin{pmatrix} -0.66 \\ -0.82 \end{pmatrix}$	−1.17	0.01	$\begin{pmatrix} 0.93 \\ 0.70 \end{pmatrix}$	0.03
$\boldsymbol{x}_3 = \begin{pmatrix} 0.00 \\ -0.55 \end{pmatrix}$	−0.37	0.87	$\begin{pmatrix} 0.93 \\ 0.69 \end{pmatrix}$	0.03
$\boldsymbol{x}_4 = \begin{pmatrix} 0.27 \\ 0.50 \end{pmatrix}$	0.62	0.03	$\begin{pmatrix} 0.92 \\ 0.69 \end{pmatrix}$	0.02
$\boldsymbol{x}_5 = \begin{pmatrix} -0.60 \\ 0.52 \end{pmatrix}$	−0.17	2.21	$\begin{pmatrix} 0.93 \\ 0.69 \end{pmatrix}$	0.01
$\boldsymbol{x}_6 = \begin{pmatrix} 0.37 \\ 0.91 \end{pmatrix}$	0.98	0.45	$\begin{pmatrix} 0.93 \\ 0.68 \end{pmatrix}$	0.00

Example 1



m.s.e. = $\frac{1}{6}\sum_{p=1}^{6}(d_p - y_p)^2$

Example 1

At convergence:

$$\boldsymbol{w} = \begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix}$$
$$b = -0.013$$

Mean square error = 0.037

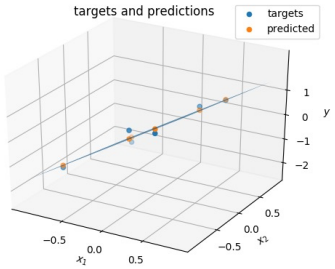
The regression equation:

$$y = \boldsymbol{x}^T\boldsymbol{w} + b = (x_1 \quad x_2)\begin{pmatrix} 2.00 \\ -0.44 \end{pmatrix} - 0.013$$

The mapping learnt by the linear neuron:

$$y = 2.00x_1 - 0.44x_2 - 0.013$$

Example 1



The mapping portrays a hyperplane in the 3-dimensional space:

$$y = 2.00x_1 - 0.44x_2 - 0.013$$

Example 1

inputs $\boldsymbol{x} = (\boldsymbol{x_1}, \boldsymbol{x_2})$	***predictions*** $y = 2.00x_1 - 0.44x_2 - 0.01$	***targets*** ***d***
$(0.54, -0.96)$	1.49	1.33
$(0.27, 0.50)$	0.30	0.45
$(0.00, -0.55)$	0.22	0.56
$(-0.60, 0.52)$	-1.45	-1.66
$(-0.66, -0.82)$	-0.98	-1.07
$(0.37, 0.91)$	0.33	0.30

Gradient descent (GD) for linear neuron

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$, cost function J is given by the sum of square errors (s.s.e.):

$$J = \frac{1}{2}\sum_{p=1}^{P}(d_p - y_p)^2$$

where y_p is the neuron output for input pattern $\boldsymbol{x}_p$.

$$J = \sum_{p=1}^{P} J_p \qquad \text{(F)}$$

where $J_p = \frac{1}{2}(d_p - y_p)^2$ is the square error for the pth pattern.

GD for linear neuron

From (F):

$$\nabla_{\boldsymbol{w}} J = \sum_{p=1}^{P} \nabla_{\boldsymbol{w}} J_p$$

$$= -\sum_{p=1}^{P} (d_p - y_p)\boldsymbol{x}_p \qquad \text{from (D)}$$

$$= -\left((d_1 - y_1)\boldsymbol{x}_1 + (d_2 - y_2)\boldsymbol{x}_2 + \cdots + (d_P - y_P)\boldsymbol{x}_P\right)$$

$$= -\begin{pmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \cdots & \boldsymbol{x}_P \end{pmatrix} \begin{pmatrix} (d_1 - y_1) \\ (d_2 - y_2) \\ \vdots \\ (d_P - y_P) \end{pmatrix}$$

$$= -\boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y}) \qquad \text{(G)}$$

where $\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1{}^T \\ \boldsymbol{x}_2{}^T \\ \vdots \\ \boldsymbol{x}_P{}^T \end{pmatrix}$ is the data matrix, $\boldsymbol{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$ is the target vector, and $\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix}$ is the output vector.

GD for linear neuron

Similarly, $\nabla_b J$ can be obtained by considering inputs of +1 and substituting a vector of +1 in (G):

$$\nabla_b J = -\mathbf{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y}) \qquad \text{(H)}$$

where $\mathbf{1}_P = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$ has P elements of 1.

The output vector $\boldsymbol{y}$ for the batch of P patterns is given by

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_P \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1{}^T\boldsymbol{w} + b \\ \boldsymbol{x}_2{}^T\boldsymbol{w} + b \\ \vdots \\ \boldsymbol{x}_P{}^T\boldsymbol{w} + b \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1{}^T \\ \boldsymbol{x}_2{}^T \\ \vdots \\ \boldsymbol{x}_P{}^T \end{pmatrix} \boldsymbol{w} + b \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \boldsymbol{X}\boldsymbol{w} + b\,\mathbf{1}_P$$

GD for linear neuron

Substituting (G) and (H) in gradient descent equations:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}} J$$
$$b \leftarrow b - \alpha\nabla_b J$$

We get GD learning equations for the linear neuron as

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y})$$
$$b \leftarrow b + \alpha\mathbf{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y})$$

And α is the learning factor.

Where:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + b\,\mathbf{1}_P$$

GD for linear neuron

Given a training dataset$(\boldsymbol{X}, \boldsymbol{d})$
Set learning parameter α
Initialize $\boldsymbol{w}$ and b
Repeat until convergence:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + b\mathbf{1}_P$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y})$$
$$b \leftarrow b + \alpha\mathbf{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y})$$

GD and SGD for a linear neuron

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J = \frac{1}{2}\sum_{p=1}^{P}(d_p - y_p)^2$	$J = \frac{1}{2}(d_p - y_p)^2$
$\boldsymbol{y} = \boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$y_p = u_p = \boldsymbol{x}_p{}^T\boldsymbol{w} + b$
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y})$	$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d_p - y_p)\boldsymbol{x}_p$
$b \leftarrow b + \alpha\boldsymbol{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y})$	$b \leftarrow b + \alpha(d_p - y_p)$

SGD for perceptron

Cost function J is given by

$$J = \frac{1}{2}(d - y)^2$$

where $y = f(u)$ and $u = \boldsymbol{w}^T\boldsymbol{x} + b$

The gradient with respect to the synaptic input:

$$\frac{\partial J}{\partial u} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial u} = -(d - y)f'(u)$$

From (C), $\frac{\partial u}{\partial \boldsymbol{w}} = \boldsymbol{x}$ and $\frac{\partial u}{\partial b} = 1$.

$$\nabla_{\boldsymbol{w}} J = \frac{\partial J}{\partial u}\frac{\partial u}{\partial \boldsymbol{w}} = -(d - y)f'(u)\boldsymbol{x} \quad \text{(I)}$$

$$\nabla_b J = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b} = -(d - y)f'(u) \quad \text{(J)}$$

Perceptron

Perceptron is a neuron having a **sigmoid** activation function and has an output

.

$$y = f(u)$$

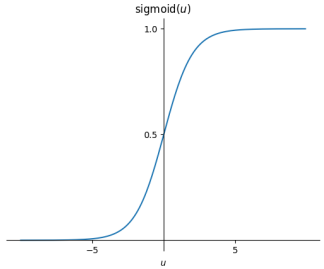
Where:

$$f(u) = \frac{1}{1 + e^{-u}} = sigmoid(u)$$

And $u = \boldsymbol{w}^T\boldsymbol{x} + b$

The square error is used as cost function for learning.

Perceptron performs a *non-linear regression* of inputs.



SGD for perceptron

Gradient learning equations:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}} J$$
$$b \leftarrow b - \alpha\nabla_b J$$

Substituting gradients from (I) and (J), SGD equations for a perceptron are given by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d - y)f'(u)\boldsymbol{x}$$
$$b \leftarrow b + \alpha(d - y)f'(u)$$

SGD algorithm for perceptron

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$

Set learning parameter α

Initialize $\boldsymbol{w}$ and b

Repeat until convergence:

For every training pattern $(\boldsymbol{x}_p, d_p)$:

$u_p = \boldsymbol{w}^T \boldsymbol{x}_p + b$

$y_p = f(u_p) = \frac{1}{1+e^{-u_p}}$

$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(d_p - y_p) f'(u_p) \boldsymbol{x}_p$

$b \leftarrow b + \alpha(d_p - y_p) f'(u_p)$

GD for perceptron

From (F):

$$\nabla_{\boldsymbol{w}} J = \sum_{p=1}^{P} \nabla_{\boldsymbol{w}} J_p$$

$$= -\sum_{p=1}^{P} (d_p - y_p) f'(u_p) \boldsymbol{x}_p \qquad \text{From (J)}$$

$$= -\left((d_1 - y_1) f'(u_1) \boldsymbol{x}_1 + (d_2 - y_2) f'(u_2) \boldsymbol{x}_2 + \cdots + (d_P - y_P) f'(u_P) \boldsymbol{x}_P\right)$$

$$= -(\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \cdots \quad \boldsymbol{x}_P) \begin{pmatrix} (d_1 - y_1) f'(u_1) \\ (d_2 - y_2) f'(u_2) \\ \vdots \\ (d_P - y_P) f'(u_P) \end{pmatrix}$$

$$= -\boldsymbol{X}^T (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u}) \qquad \text{(K)}$$

where $\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_P^T \end{pmatrix}$, $\boldsymbol{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}$, $\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix}$, and $f'(\boldsymbol{u}) = \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_P) \end{pmatrix}$

GD for perceptron

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$, cost function J is given by the sum of square errors (s.s.e) over all the patterns:

$$J = \frac{1}{2} \sum_{p=1}^{P} (d_p - y_p)^2 = \sum_{p=1}^{P} J_p \qquad \text{(F)}$$

where $J_p = \frac{1}{2}(d_p - y_p)^2$ is the square error for the pth pattern.

GD for perceptron

Substituting $\boldsymbol{X}^T$ by $\boldsymbol{1}_P{}^T$ in (K), we get

$$\nabla_{\boldsymbol{b}} J = -\boldsymbol{1}_P{}^T (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u}) \qquad \text{(L)}$$

where $\boldsymbol{1}_P = (1 \quad 1 \quad \cdots \quad 1)^T$.

The gradient descent learning is given by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} J$$
$$b \leftarrow b - \alpha \nabla_b J$$

Substituting (K) and (L), we get the learning equations:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{X}^T (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$
$$b \leftarrow b + \alpha \boldsymbol{1}_P{}^T (\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$

Note that $\cdot$ is the element-wise product.

GD for perceptron

Given a training dataset $(\boldsymbol{X}, \boldsymbol{d})$
Set learning parameter α
Initialize $\boldsymbol{w}$ and b
Repeat until convergence:

$$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$$
$$\boldsymbol{y} = f(\boldsymbol{u}) = \frac{1}{1+e^{-u}}$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$
$$b \leftarrow b + \alpha \boldsymbol{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$

Derivatives of Sigmoid

The activation function of the (continuous) perceptron is *sigmoid function* (i.e., unipolar sigmoidal function with $a = 1.0$ and $b = 1.0$) :

$$y = f(u) = \frac{1}{1+e^{-u}}$$

The derivative is given by

$$f'(u) = \frac{-1}{(1+e^{-u})^2}\frac{\partial(e^{-u})}{\partial u} = \frac{e^{-u}}{(1+e^{-u})^2} = \frac{1}{1+e^{-u}} - \frac{1}{(1+e^{-u})^2} = y(1-y)$$

For *Tanh* function (bipolar sigmoid):

$$y = f(u) = \frac{e^{+u} - e^{-u}}{e^{+u} + e^{-u}}$$

$$f'(u) = \frac{(e^{+u}+e^{-u})(e^{+u}+e^{-u}) - (e^{+u}-e^{-u})(e^{+u}-e^{-u})}{(e^{+u}+e^{-u})^2} = \left(1 - \left(\frac{e^{+u}-e^{-u}}{e^{+u}+e^{-u}}\right)^2\right) = 1 - y^2$$

Gradient descent for perceptron

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J = \frac{1}{2}\sum_{p=1}^{P}(d_p - y_p)^2$	$J = \frac{1}{2}(d_p - y_p)^2$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$	$u_p = \boldsymbol{x}_p{}^T\boldsymbol{w} + b$
$\boldsymbol{y} = f(\boldsymbol{u})$	$y_p = f(u_p)$
$\boldsymbol{w} = \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$	$\boldsymbol{w} = \boldsymbol{w} + \alpha(d_p - y_p)f'(u_p)\boldsymbol{x}_p$
$b = b + \alpha \boldsymbol{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$	$b = b + \alpha(d_p - y_p)f'(u_p)$

Example 2

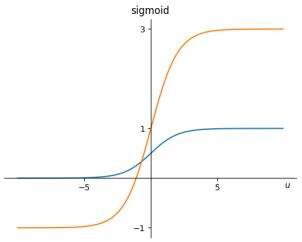
Design a perceptron to learn the following mapping by using gradient descent (GD):

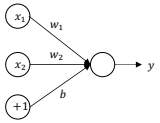
$\boldsymbol{x} = (x_1, x_2)$	d
$(0.77, 0.02)$	2.91
$(0.63, 0.75)$	0.55
$(0.50, 0.22)$	1.28
$(0.20, 0.76)$	-0.74
$(0.17, 0.09)$	0.88
$(0.69, 0.95)$	0.30
$(0.00, 0.51)$	-0.28

Use learning factor $\alpha = 0.01$.

Example 2

$$\boldsymbol{X} = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} \text{ and } \boldsymbol{d} = \begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix}$$





Initially, $\boldsymbol{w} = \begin{pmatrix} 0.81 \\ 0.61 \end{pmatrix}$ and $b = 0.0$

$\alpha = 0.01$

Output $y \in [-0.74, 2.91] \subset [-1.0, 3.0]$

Note that the sigmoidal should have an amplitude = 4 and shifted downwards by 1.0.

So, the activation function should be

$$y = f(u) = \frac{4.0}{1 + e^{-u}} - 1.0$$

$$f'(u) = \frac{4e^{-u}}{(1+e^{-u})^2} = (y+1)\frac{e^{-u}}{(1+e^{-u})} = (y+1)\left(1 - \frac{1}{1+e^{-u}}\right) = \frac{1}{4}(y+1)(3-y)$$

Example 2

$$f'(\boldsymbol{u}) = \frac{1}{4}(\boldsymbol{y}+\boldsymbol{1}) \cdot (\boldsymbol{3}-\boldsymbol{y}) = \frac{1}{4}\left(\begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix} + \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix}\right) \cdot \left(\begin{pmatrix} 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \\ 3.0 \end{pmatrix} - \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix}\right) = \begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix}$$

Example 2

Epoch 1 begins …

$$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \end{pmatrix} \begin{pmatrix} 0.81 \\ 0.61 \end{pmatrix} + 0.0 \begin{pmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{pmatrix} = \begin{pmatrix} 0.64 \\ 0.97 \\ 0.54 \\ 0.63 \\ 0.19 \\ 1.14 \\ 0.32 \end{pmatrix}$$

$$\boldsymbol{y} = f(\boldsymbol{u}) = \frac{4.0}{1 + e^{-\boldsymbol{u}}} - 1.0 = \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix}$$

$$m.s.e. = \frac{1}{7}\sum_{p=1}^{7}(d_p - y_p)^2 = \frac{1}{7}\left((2.91 - 1.61)^2 + (0.55 - 1.90)^2 + \cdots\right) = 2.11$$

Example 2

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$

$$= \begin{pmatrix} 0.81 \\ 0.61 \end{pmatrix} + 0.01 \begin{pmatrix} 0.77 & 0.63 & 0.50 & 0.20 & 0.17 & 0.69 & 0.00 \\ 0.02 & 0.75 & 0.22 & 0.76 & 0.09 & 0.95 & 0.51 \end{pmatrix} \left(\begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix} - \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix}\right) \cdot \begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix} = \begin{pmatrix} 0.80 \\ 0.57 \end{pmatrix}$$

$$b = b + \alpha \boldsymbol{1}_P{}^T(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$

$$= 0.0 + 0.01 \times (1.0 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0 \quad 1.0) \left(\begin{pmatrix} 2.91 \\ 0.55 \\ 1.28 \\ -0.74 \\ 0.88 \\ 0.30 \\ -0.28 \end{pmatrix} - \begin{pmatrix} 1.61 \\ 1.90 \\ 1.53 \\ 1.61 \\ 1.19 \\ 2.03 \\ 1.31 \end{pmatrix}\right) \cdot \begin{pmatrix} 0.90 \\ 0.80 \\ 0.93 \\ 0.91 \\ 0.99 \\ 0.73 \\ 0.98 \end{pmatrix} = -0.05$$

Example 2

iter	u	y	$f'(u)$	*mse*	w	b
1	$\begin{pmatrix}0.64\\0.97\\0.54\\0.63\\0.19\\1.14\\0.32\end{pmatrix}$	$\begin{pmatrix}1.61\\1.90\\1.53\\1.61\\1.19\\2.03\\1.31\end{pmatrix}$	$\begin{pmatrix}0.90\\0.80\\0.93\\0.91\\0.99\\0.73\\0.98\end{pmatrix}$	2.11	$\begin{pmatrix}0.80\\0.57\end{pmatrix}$	-0.05
2	$\begin{pmatrix}0.57\\0.88\\0.47\\0.54\\1.04\\1.95\\0.24\end{pmatrix}$	$\begin{pmatrix}1.56\\1.83\\1.46\\1.52\\1.13\\1.95\\1.24\end{pmatrix}$	$\begin{pmatrix}0.92\\0.83\\0.95\\0.93\\1.00\\0.77\\0.99\end{pmatrix}$	1.96	$\begin{pmatrix}0.79\\0.52\end{pmatrix}$	-0.11
1500	$\begin{pmatrix}2.05\\-0.45\\0.56\\-1.94\\-0.16\\-0.85\\-1.89\end{pmatrix}$	$\begin{pmatrix}2.54\\0.56\\1.55\\-0.50\\0.84\\0.20\\-0.48\end{pmatrix}$	$\begin{pmatrix}0.40\\0.95\\0.92\\0.44\\0.99\\0.84\\0.45\end{pmatrix}$	0.046	$\begin{pmatrix}3.35\\-2.80\end{pmatrix}$	-0.47

Example 2

At convergence:

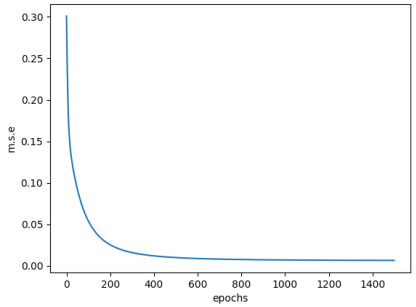
$$\boldsymbol{w} = \begin{pmatrix}3.35\\-2.80\end{pmatrix}$$
$$b = -0.47$$

Mean square error = 0.01

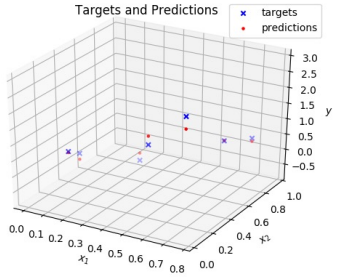
$$u = \boldsymbol{x}^T\boldsymbol{w} + b = \begin{pmatrix}x_1 & x_2\end{pmatrix}\begin{pmatrix}3.35\\-2.80\end{pmatrix} - 0.47 = 3.35x_1 - 2.8x_2 - 0.47$$

$$y = \frac{4.0}{1 + e^{-u}} - 1.0$$
$$y = \frac{4.0}{1 + e^{-3.35x_1 + 2.8x_2 + 0.47}} - 1.0$$

Example 2



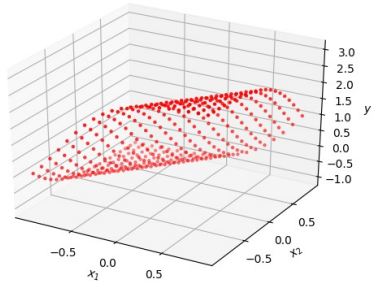
Example 2



Example 2

Non-linear function leant by the perceptron

$$y = \frac{4.0}{1 + e^{-3.35x_1 + 2.8x_2 + 0.47}} - 1.0$$



Classification Example

Classification is to identify or distinguish classes or groups of objects.

Example: To identify *ballet dancers* from *rugby players.*

Two distinctive *features* that can aid in classification:

- ***weight***
- ***height***

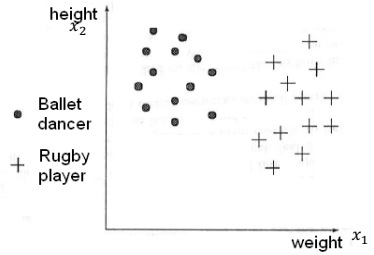


Figure: A 2-dimensional feature space

Let x_1 denote weight and x_2 denote height. Every individual is represented as a point $\boldsymbol{x} = (x_1, x_2)$ in the feature feature space.

Classification Example

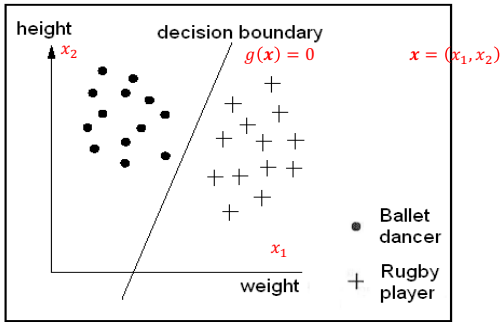


Figure: A linear classification decision boundary

Decision boundary

The **decision boundary** of the classifier, $g(\boldsymbol{x}) = 0$ where function $g(\boldsymbol{x})$ is referred to as the **discriminant function**.

A classifier finds a decision boundary separating the two classes in the feature space. On one side of the decision boundary, discriminant function is positive and on other side, discriminant function is negative.

Therefore, the following class definition may be employed:

If $g(\mathbf{x}) > 0$ => Ballet dancer

If $g(\mathbf{x}) \leq 0$ => Rugby player

Linear Classifier

If the two classes can be separated by a straight line, the classification is said to be **linearly separable**. For linear separable classes, one can design **a linear classifier**.

A linear classifier implements discriminant function or a decision boundary that is represented by a straight line (hyper plane) in the multidimensional **feature space**. Generally, the feature space is multidimensional. In the multidimensional space, a straight line or **hyperplane** is indicated by a linear sum of coordinates.

Given an input (features), $\boldsymbol{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$. A linear description function is given by

$$g(\boldsymbol{x}) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

where $\mathbf{w} = (w_1 \quad w_2 \quad \cdots \quad w_n)^T$ are the coefficient/weights and w_0 is the constant term.

Discrete perceptron as a linear two-class classifier

The linear discriminant function can be implemented by the synaptic input to a neuron

$$g(\boldsymbol{x}) = u = \boldsymbol{w}^T \boldsymbol{x} + b$$

And with a threshold activation function $1(u)$:

$$u = g(\boldsymbol{x}) > 0 \ \rightarrow y = 1 \ \rightarrow class1$$
$$u = g(\boldsymbol{x}) \leq 0 \ \rightarrow y = 0 \rightarrow class2$$

That is, two-class linear classifier (or a dichotomizer) can be implemented with an artificial neuron with a threshold (unit step) activation function (**discrete perceptron**).

The output, 0 or 1, of the binary neuron represents the **label** of the class.

Classification error

In classification, the error is expressed as total mismatches between the target and output labels.

$$\text{Classification error} = \sum_{p=1}^{P} 1(d_p \neq y_p)$$

Logistic regression neuron

A **logistic regression neuron** performs a binary classification of inputs. That is, it classifies inputs into two classes with labels '0' and '1'.

The activation function of the logistic regression neuron is given by the sigmoid function:

$$f(u) = \frac{1}{1 + e^{-u}}$$

where $u = \boldsymbol{w}^T \boldsymbol{x} + b$ is the synaptic input to the neuron.

The activation of a logistic regression neuron gives the probability of the neuron output belonging to class '1'.

$$P(y = 1|\boldsymbol{x}) = f(u)$$

Then

$$P(y = 0|\boldsymbol{x}) = 1 - P(y = 1|\boldsymbol{x}) = 1 - f(u)$$

Logistic Regression Neuron

A logistic regression neuron receives an input $\boldsymbol{x} \in \boldsymbol{R}^n$ and produces a class label $y \in \{0, 1\}$ as the output.

$$f(u) = \frac{1}{1 + e^{-u}}$$

When $u = 0$, $f(u) = P(y = 1|\boldsymbol{x}) = P(y = 0|\boldsymbol{x}) = 0.5$.
That is, $y = 1$ if $f(u) > 0.5$, else $y = 0$.

The output y of the neuron is given by:

$$y = 1(f(u) > 0.5) = 1(u > 0)$$

Note that for logistic neuron, the output and activation are different. It finds a linear boundary $u = 0$ separating the two classes.

SGD for logistic regression neuron

Given a training pattern $(\boldsymbol{x}, d)$ where $\boldsymbol{x} \in \boldsymbol{R}^n$ and $d \in \{0,1\}$.

The cost function for classification is given by the **cross-entropy**:

$$J = -d\log\big(f(u)\big) - (1 - d)\log\big(1 - f(u)\big)$$

where $u = \boldsymbol{w}^T\boldsymbol{x} + b$ and $f(u) = \frac{1}{1+e^{-u}}$.

The cost function J is minimized using the gradient descent procedure.

$$J = \begin{cases} -\log\big(f(u)\big) & \text{if } d = 1 \\ -\log\big(1 - f(u)\big) & \text{if } d = 0 \end{cases}$$

SGD for logistic regression neuron

$$J = -d\ log\big(f(u)\big) - (1 - d)log\big(1 - f(u)\big)$$

where $u = \boldsymbol{w}^T\boldsymbol{x} + b$ and $f(u) = \frac{1}{1+e^{(-u)}}$.

Gradient with respect to u :

$$\frac{\partial J}{\partial u} = -\frac{\partial}{\partial f(u)}\Big(dlog\big(f(u)\big) + (1 - d)log\big(1 - f(u)\big)\Big)\frac{\partial f(u)}{\partial u}$$
$$= -\left(\frac{d}{f(u)} - \frac{(1 - d)}{1 - f(u)}\right)f'(u)$$

Substituting $f'(u) = f(u)\big(1 - f(u)\big)$ for sigmoid activation function,

$$\frac{\partial J}{\partial u} = -\frac{d - f(u)}{f(u)\big(1 - f(u)\big)}f(u)\big(1 - f(u)\big) = -\big(d - f(u)\big)$$

SGD for logistic regression neuron

Substituting $\frac{\partial J}{\partial u}$, $\frac{\partial u}{\partial \boldsymbol{w}} = \boldsymbol{x}$, and $\frac{\partial u}{\partial b} = 1$.

$$\nabla_{\boldsymbol{w}}J = \frac{\partial J}{\partial u}\frac{\partial u}{\partial \boldsymbol{w}} = -\big(d - f(u)\big)\boldsymbol{x} \quad \text{(A)}$$

$$\nabla_b J = \frac{\partial J}{\partial u}\frac{\partial u}{\partial b} = -\big(d - f(u)\big) \quad \text{(B)}$$

Gradient learning equations:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}}J$$
$$b \leftarrow b - \alpha\nabla_b J$$

Substituting $\nabla_{\boldsymbol{w}}J$ and $\nabla_b J$ for logistic regression neuron

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\big(d - f(u)\big)\boldsymbol{x}$$
$$b \leftarrow b + \alpha\big(d - f(u)\big)$$

SGD for logistic regression neuron

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$
Set learning rate α
Initialize $\boldsymbol{w}$ and b
Iterate until convergence:
For every pattern $(\boldsymbol{x}_p, d_p)$:

$$u_p = \boldsymbol{w}^T \boldsymbol{x}_p + b$$
$$f(u_p) = \frac{1}{1+e^{-u_p}}$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\left(d_p - f(u_p)\right)\boldsymbol{x}_p$$
$$b \leftarrow b + \alpha\left(d_p - f(u_p)\right)$$

GD for logistic regression neuron

Given a training dataset $\{(\boldsymbol{x}_p, d_p)\}_{p=1}^{P}$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and $d_p \in \{0,1\}$.

The cost function for logistic regression is given by the *cross-entropy* (or *negative log-likelihood*) over all the training patterns:

$$J = -\sum_{p=1}^{P} d_p \, log\left(f(u_p)\right) + (1-d_p) log\left(1 - f(u_p)\right)$$

where $u_p = \boldsymbol{w}^T \boldsymbol{x}_p + b$ and $f(u_p) = \frac{1}{1+e^{-u_p}}$.

The cost function J can be written as

$$J = \sum_{p=1}^{P} J_p$$

where $J_p = -d_p log\left(f(u_p)\right) - (1-d_p) log\left(1 - f(u_p)\right)$ is cross-entropy due to p th pattern.

GD for logistic regression neuron

$$\nabla_{\boldsymbol{w}} J = \sum_{p=1}^{P} \nabla_{\boldsymbol{w}} J_p$$
$$= -\sum_{p=1}^{P} \left(d_p - f(u_p)\right)\boldsymbol{x}_p \quad \text{From (A)}$$
$$= -\left((d_1 - f(u_1))\boldsymbol{x}_1 + (d_2 - f(u_2))\boldsymbol{x}_2 + \cdots + (d_P - f(u_P))\boldsymbol{x}_P\right)$$
$$= -(\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \cdots \quad \boldsymbol{x}_P)\begin{pmatrix}(d_1 - f(u_1))\\(d_2 - f(u_2))\\\vdots\\(d_P - f(u_P))\end{pmatrix}$$
$$= -\boldsymbol{X}^T(\boldsymbol{d} - f(\boldsymbol{u}))$$

where $\boldsymbol{X} = \begin{pmatrix}\boldsymbol{x}_1^T\\\boldsymbol{x}_2^T\\\vdots\\\boldsymbol{x}_P^T\end{pmatrix}$, $\boldsymbol{d} = \begin{pmatrix}d_1\\d_2\\\vdots\\d_P\end{pmatrix}$, and $f(\boldsymbol{u}) = \begin{pmatrix}f(u_1)\\f(u_2)\\\vdots\\f(u_P)\end{pmatrix}$

GD for logistic regression neuron

By substituting $\boldsymbol{1}_P$ for $\boldsymbol{X}$ in above equation:

$$\nabla_{\boldsymbol{b}} J = -\boldsymbol{1}_P^T(\boldsymbol{d} - f(\boldsymbol{u}))$$

Substituting the gradients in

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha\nabla_{\boldsymbol{w}} J$$
$$b \leftarrow b - \alpha\nabla_b J$$

the gradient descent learning for logistic regression neuron is given by

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\boldsymbol{X}^T(\boldsymbol{d} - f(\boldsymbol{u}))$$
$$b \leftarrow b + \alpha\boldsymbol{1}_P^T(\boldsymbol{d} - f(\boldsymbol{u}))$$

Note that $\boldsymbol{y}$ in the discrete perceptron is now replaced with $f(\boldsymbol{u})$ in logistic regression learning equations.

GD for logistic regression neuron

Given training data $(\boldsymbol{X}, \boldsymbol{d})$
Set learning rate α
Initialize $\boldsymbol{w}$ and b
Iterate until convergence:

$$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\mathbf{1}_P$$
$$f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - f(\boldsymbol{u}))$$
$$b \leftarrow b + \alpha \mathbf{1}_P^T(\boldsymbol{d} - f(\boldsymbol{u}))$$

Learning for logistic regression neuron

GD	SGD
$(\boldsymbol{X}, \boldsymbol{d})$	$(\boldsymbol{x}_p, d_p)$
$J = -\sum_{p=1}^{P} d_p \log\left(f(u_p)\right) + (1-d_p)\log\left(1-f(u_p)\right)$	$J_p = -d_p \log\left(f(u_p)\right) - (1-d_p)\log\left(1-f(u_p)\right)$
$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\mathbf{1}_P$	$u_p = \boldsymbol{w}^T\boldsymbol{x}_p + b$
$f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$	$f(u_p) = \frac{1}{1+e^{-u_p}}$
$\boldsymbol{y} = 1(f(\boldsymbol{u}) > 0.5)$	$y_p = 1(f(u_p) > 0.5)$
$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - f(\boldsymbol{u}))$	$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\left(d_p - f(u_p)\right)\boldsymbol{x}_p$
$b \leftarrow b + \alpha \mathbf{1}_P^T(\boldsymbol{d} - f(\boldsymbol{u}))$	$b \leftarrow b + \alpha\left(d_p - f(u_p)\right)$

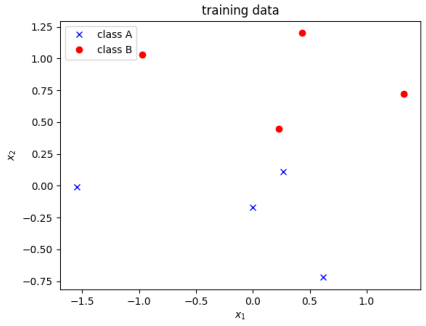
Example 3: GD for logistic regression neuron

Train a logistic regression neuron to perform the following classification, using GD:

$$(1.33 \quad 0.72) \rightarrow class\ B$$
$$(-1.55 \quad -0.01) \rightarrow class\ A$$
$$(0.62 \quad -0.72) \rightarrow class\ A$$
$$(0.27 \quad 0.11) \rightarrow class\ A$$
$$(0.0 \quad -0.17) \rightarrow class\ A$$
$$(0.43 \quad 1.2) \rightarrow class\ B$$
$$(-0.97 \quad 1.03) \rightarrow class\ B$$
$$(0.23 \quad 0.45) \rightarrow class\ B$$

User a learning factor $\alpha = 0.04$.

Example 3



Example 3

Let $y = 1$ for class A and $y = 0$ for class B.

$$\boldsymbol{X} = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & -0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \text{ and } \boldsymbol{d} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Initially, $w = \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix}$, $b = 0.0$ and $\alpha = 0.4$

Example 3

Epoch 1:

$$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1} = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & 0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.0 \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.04 \\ -1.2 \\ 0.46 \\ 0.21 \\ 0.00 \\ 0.36 \\ -0.73 \\ 0.19 \end{pmatrix}$$

$$f(\boldsymbol{u}) = \frac{1}{1 + e^{(-\boldsymbol{u})}} = \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix}$$

Example 3

$$\boldsymbol{y} = 1(f(\boldsymbol{u}) > 0.5) = 1\left(\begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} > 0.5\right) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} \qquad \boldsymbol{d} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Classification error = $\sum_{p=1}^{8} 1(d_p \neq y_p) = 5$

Cross-entropy = $-\sum_{p=1}^{P} d_p \, log\left(f(u_p)\right) + (1 - d_p) log\left(1 - f(u_p)\right)$

$= -log(1 - f(u_1)) - logf(u_2) - logf(u_3) - \cdots - log(1 - f(u_8))$

$= -log(1 - 0.74) - log0.23 - log0.61 - \cdots - log(1 - 0.55)$

= 6.653

Example 3

$$\boldsymbol{w} = \boldsymbol{w} + \alpha \boldsymbol{X}^T(\boldsymbol{d} - f(\boldsymbol{u}))$$

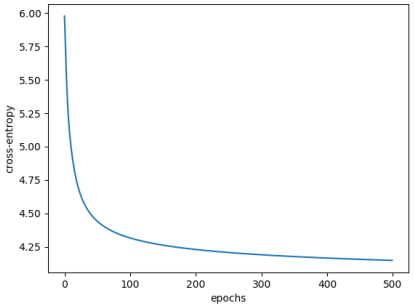
$$= \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.04 \begin{pmatrix} 1.33 & -1.55 & 0.62 & 0.27 & 0 & 0.43 & -0.97 & 0.23 \\ 0.72 & -0.01 & -0.72 & 0.11 & -0.17 & 1.2 & 1.03 & 0.45 \end{pmatrix} \left(\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} \right)$$

$$= \begin{pmatrix} 0.69 \\ -0.2 \end{pmatrix}$$

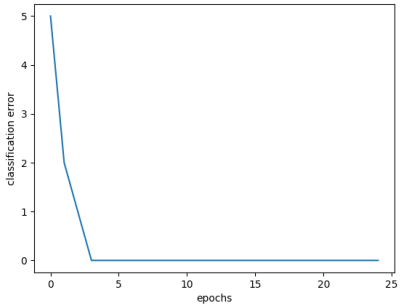
$$b = b + \alpha \boldsymbol{1}_P^T(\boldsymbol{d} - f(\boldsymbol{u}))$$

$$= 0.0 + 0.04(1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1) \left(\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} - \begin{pmatrix} 0.74 \\ 0.23 \\ 0.61 \\ 0.55 \\ 0.5 \\ 0.59 \\ 0.33 \\ 0.55 \end{pmatrix} \right) = -0.09$$

Example 3



Example 3



Example 3

At convergence, $\boldsymbol{w} = \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix}, b = 4.47$

The decision boundary is given by: $u = \boldsymbol{x}^T\boldsymbol{w} + b = 0$

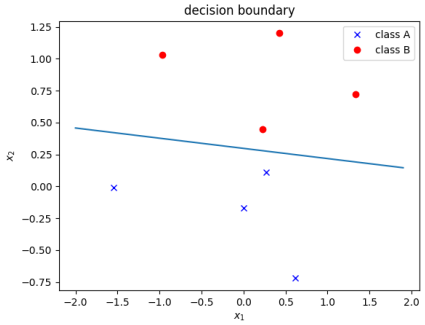
$$(x_1 \quad x_2)^T \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix} + 4.47 = 0$$

$$-1.20x_1 - 15.02x_2 + 4.47 = 0$$

Example 3

Decision boundary:

$$-1.20x_1 - 15.02x_2 + 4.47 = 0$$

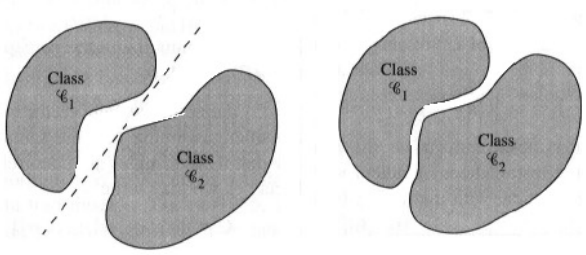


Limitations of logistic regression neuron

As long as the neuron is a *linear combiner* followed by a *non-linear activation function*, then regardless of the form of non-linearity used, the neuron can perform pattern classification *only* on *linearly separable* patterns.

Linear separability requires that the patterns to be classified must be sufficiently separated from each other to ensure that the decision boundaries are hyperplanes.

Limitations of logistic regression neuron



(a) Linearly Separable Pattern

(b) Non-Linearly Separable Pattern

Discrete perceptron and logistic regression neuron can create only linear decision boundaries.

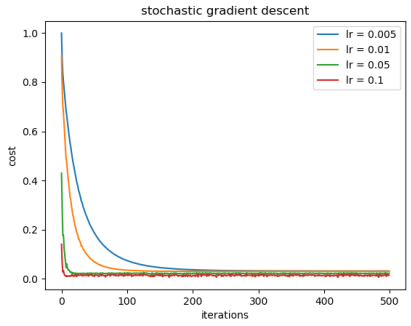
Example 4: effects of leaning rate

Design a perceptron to learn the following mapping by using gradient descent (GD):

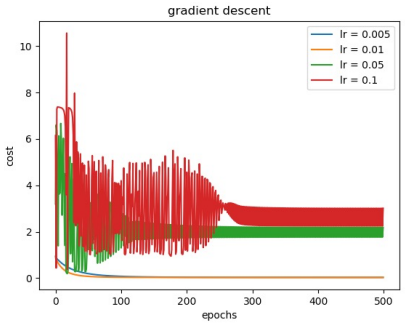
$\boldsymbol{x} = (x_1, x_2)$	d
$(0.77, 0.02)$	2.91
$(0.63, 0.75)$	0.55
$(0.50, 0.22)$	1.28
$(0.20, 0.76)$	-0.74
$(0.17, 0.09)$	0.88
$(0.69, 0.95)$	0.30
$(0.00, 0.51)$	-0.28

Use learning factor $\alpha = 0.01$.

Example 4: Learning rates with SGD



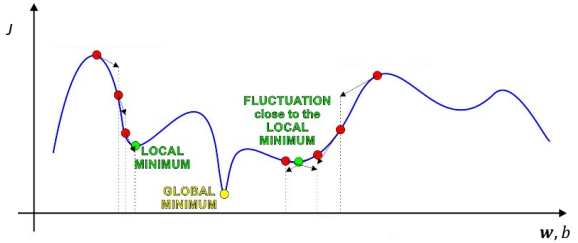
Example 4: Learning rates with GD



Learning rates

- At higher learning rates, convergence is faster but may not be stable.
- The *optimal learning rate* is the largest rate at which learning does not diverge.
- Generally, SGD convergence to a better solution (lower error) as it capitalizes on randomness of data. SGD takes a longer time to converge
- Usually, GD can use a higher learning rate compared to SGD; The time for one add/multiply computation per a weight update is less when patterns are trained in a (small) batch.
- In practice, *mini-batch SGD* is used. Then, the time to train a network is dependent upon
  - the learning rate
  - the batch size

Local minima problem in gradient descent learning



Algorithm may get stuck in a local minimum of error function depending on the initial weights. Gradient descent gives a suboptimal solution and does not guarantee the optimal solution.

Summary: types of neurons

Role	Neuron
Regression (one dimensional)	**Linear neuron**
	Perceptron
Classification (two classes)	**Logistic regression neuron**

Summary: GD for neurons

$$(\boldsymbol{X}, \boldsymbol{d})$$
$$\boldsymbol{u} = \boldsymbol{X}\boldsymbol{w} + b\boldsymbol{1}_P$$
$$\boldsymbol{w} = \boldsymbol{w} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{u}} J$$
$$b = b - \alpha \boldsymbol{1}_P{}^T \nabla_{\boldsymbol{u}} J$$

neuron	$f(\boldsymbol{u}), \boldsymbol{y}$	$\nabla_{\boldsymbol{u}} J$
Logistic regression neuron	$f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$ $\boldsymbol{y} = 1(f(\boldsymbol{u}) > 0.5)$	$-(\boldsymbol{d} - f(\boldsymbol{u}))$
Linear neuron	$\boldsymbol{y} = \boldsymbol{u}$	$-(\boldsymbol{d} - \boldsymbol{y})$
Perceptron	$\boldsymbol{y} = f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}$	$-(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$

Summary: SGD for neurons

$$(\boldsymbol{x}_p, d_p)$$
$$u_p = \boldsymbol{w}^T \boldsymbol{x}_p + b$$
$$\boldsymbol{w} = \boldsymbol{w} - \alpha \nabla_{u_p} J \boldsymbol{x}_p$$
$$b = b - \alpha \nabla_{u_p} J$$

neuron	$f(u_p), y_p$	$\nabla_{u_p} J$
Logistic regression neuron	$f(u_p) = \frac{1}{1+e^{-u_p}}$ $y_p = 1(f(u_p) > 0.5)$	$-(d_p - f(u_p))$
Linear neuron	$y_p = u_p$	$-(d_p - y_p)$
Perceptron	$y_p = f(u_p) = \frac{1}{1+e^{-u_p}}$	$-(d_p - y_p) \cdot f'(u_p)$

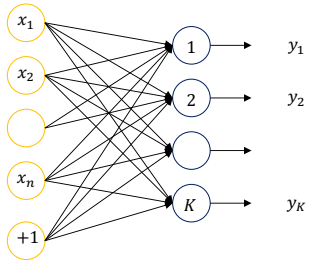
Chapter 3

Neuron Layers

Neural networks and deep learning

Weight matrix of a layer

Consider a layer of K neurons.



Let $\boldsymbol{w}_k$ and b_k denote the weight vector and bias of k th neuron. Weights connected to a neuron layer is given by a weight matrix:

$$\boldsymbol{W} = (\boldsymbol{w}_1 \quad \boldsymbol{w}_2 \quad \cdots \quad \boldsymbol{w}_K)$$

where columns are given by weight vectors of individual neurons.

And a bias vector $\boldsymbol{b}$ where each element corresponds to a bias of a neuron:

$$\boldsymbol{b} = (b_1, b_2, \cdots b_K)^T$$

Synaptic input at a layer for single input

Given an input pattern $\boldsymbol{x} \in \boldsymbol{R}^n$ to a layer of K neurons.

Synaptic input u_k to kth neuron:

$$u_k = \boldsymbol{w}_k{}^T \boldsymbol{x} + b_k$$

$\boldsymbol{w_k}$ and b_k denote the weight vector and bias of kth neuron.

Synaptic input vector $\boldsymbol{u}$ to the layer :

$$\boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{pmatrix} = \begin{pmatrix} \boldsymbol{w}_1^T \boldsymbol{x} + b_1 \\ \boldsymbol{w}_2^T \boldsymbol{x} + b_2 \\ \vdots \\ \boldsymbol{w}_K^T \boldsymbol{x} + b_k \end{pmatrix} = \begin{pmatrix} \boldsymbol{w}_1^T \\ \boldsymbol{w}_2^T \\ \vdots \\ \boldsymbol{w}_K^T \end{pmatrix} \boldsymbol{x} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_k \end{pmatrix} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$$

$$\boldsymbol{u} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$$

where $\boldsymbol{W}$ is the weight matrix and $\boldsymbol{b}$ is the bias vector of the layer.

Activation at a layer for batch input

The synaptic input to the layer due to a batch of patterns:

$$\boldsymbol{U} = \boldsymbol{X}\,\boldsymbol{W} + \boldsymbol{B}$$

where rows of $\boldsymbol{U}$ corresponds to synaptic inputs of the layer, corresponding to individual input patterns:

Activation of the layer:

$$f(\boldsymbol{U}) = \begin{pmatrix} f(\boldsymbol{u}_1^T) \\ f(\boldsymbol{u}_2^T) \\ \vdots \\ f(\boldsymbol{u}_P^T) \end{pmatrix} = \begin{pmatrix} f(\boldsymbol{u}_1)^T \\ f(\boldsymbol{u}_2)^T \\ \vdots \\ f(\boldsymbol{u}_P)^T \end{pmatrix}$$

where activations due to individual patterns are written as rows.

Synaptic input to a layer for batch input

Given a set $\{\boldsymbol{x}_p\}_{p=1}^P$ input patterns to a layer of K neurons where $\boldsymbol{x}_p \in \boldsymbol{R}^n$.

Synaptic input $\boldsymbol{u}_p$ to the layer for an input pattern $\boldsymbol{x}_p$:

$$\boldsymbol{u}_p = \boldsymbol{W}^T \boldsymbol{x}_p + \boldsymbol{b}$$

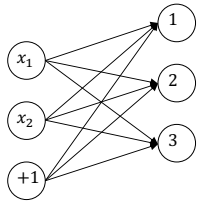
The synaptic input matrix $\boldsymbol{U}$ to the layer for P patterns: $(\boldsymbol{AB})^T = \boldsymbol{B}^T \boldsymbol{A}^T$

$$\boldsymbol{U} = \begin{pmatrix} \boldsymbol{u}_1^T \\ \boldsymbol{u}_2^T \\ \vdots \\ \boldsymbol{u}_P^T \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1{}^T \boldsymbol{W} + \boldsymbol{b}^T \\ \boldsymbol{x}_2{}^T \boldsymbol{W} + \boldsymbol{b}^T \\ \vdots \\ \boldsymbol{x}_P{}^T \boldsymbol{W} + \boldsymbol{b}^T \end{pmatrix} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_P^T \end{pmatrix} \boldsymbol{W} + \begin{pmatrix} \boldsymbol{b}^T \\ \boldsymbol{b}^T \\ \vdots \\ \boldsymbol{b}^T \end{pmatrix} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$

where rows of $\boldsymbol{U}$ are synaptic inputs corresponding to individual input patterns.

The matrix $\boldsymbol{B} = \begin{pmatrix} \boldsymbol{b}^T \\ \boldsymbol{b}^T \\ \vdots \\ \boldsymbol{b}^T \end{pmatrix}$ has bias vector propagated as rows.

Example 1: activations and outputs of a perceptron layer



A perceptron layer of 3 neurons shown in the figure receives 2-dimensional inputs $(x_1, x_2)^T$, and has a weight matrix $\boldsymbol{W}$ and a bias vector $\boldsymbol{b}$ given by

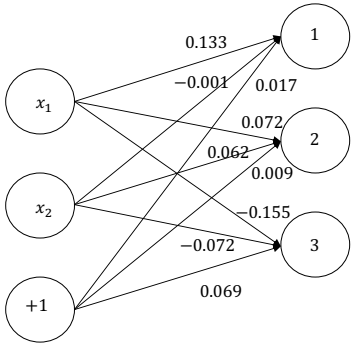
$$\boldsymbol{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} 0.017 \\ 0.009 \\ 0.069 \end{pmatrix}$$

Using batch processing, find the output for input patterns:

$\begin{pmatrix} 0.5 \\ -1.66 \end{pmatrix}$, $\begin{pmatrix} -1.0 \\ -0.51 \end{pmatrix}$, $\begin{pmatrix} 0.78 \\ -0.65 \end{pmatrix}$, and $\begin{pmatrix} 0.04 \\ -0.2 \end{pmatrix}$.

Example 1

$$\boldsymbol{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} 0.017 \\ 0.009 \\ 0.069 \end{pmatrix}.$$



Example 1

$$\boldsymbol{U} = \begin{pmatrix} 0.085 & -0.059 & 0.111 \\ -0.115 & 0.094 & 0.26 \\ 0.121 & 0.024 & -0.005 \\ 0.022 & -0.001 & 0.077 \end{pmatrix}$$

For a perceptron layer

$$\boldsymbol{Y} = f(\boldsymbol{U}) = \frac{1}{1+e^{-\boldsymbol{U}}} = \begin{pmatrix} 0.521 & 0.485 & 0.527 \\ 0.471 & 0.476 & 0.565 \\ 0.530 & 0.506 & 0.499 \\ 0.506 & 0.500 & 0.519 \end{pmatrix}$$

2nd neuron output for 3rd input pattern

For example, third row corresponding to 3rd input:

$$\boldsymbol{x} = \begin{pmatrix} 0.78 \\ -0.65 \end{pmatrix}$$

And the corresponding output

$$\boldsymbol{y} = \begin{pmatrix} 0.530 \\ 0.506 \\ 0.499 \end{pmatrix}$$

Example 1

$$\boldsymbol{W} = \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} \text{ and } \boldsymbol{B} = \begin{pmatrix} 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \end{pmatrix}.$$

Input as a batch of four patterns:

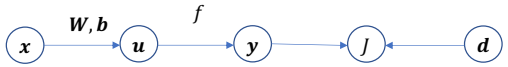
$$\boldsymbol{X} = \begin{pmatrix} 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}$$

The synaptic input to the layer:

$$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$$
$$= \begin{pmatrix} 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix} \begin{pmatrix} 0.133 & 0.072 & -0.155 \\ -0.001 & 0.062 & -0.072 \end{pmatrix} + \begin{pmatrix} 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \\ 0.017 & 0.009 & 0.069 \end{pmatrix}$$
$$= \begin{pmatrix} 0.085 & -0.059 & 0.111 \\ -0.115 & 0.094 & 0.26 \\ 0.121 & 0.024 & -0.005 \\ 0.022 & -0.001 & 0.077 \end{pmatrix}$$

SGD for single layer

Computational graph processing input $(\boldsymbol{x}, \boldsymbol{d})$:



J denotes the cost function.

Need to compute gradients $\nabla_{\boldsymbol{W}} J$ and $\nabla_{\boldsymbol{b}} J$ to learn weight matrix $\boldsymbol{W}$ and bias vector $\boldsymbol{b}$.

SGD for single layer

Consider k th neuron at the layer:

$$u_k = \boldsymbol{x}^T \boldsymbol{w}_k + b_k$$

And

$$\frac{\partial u_k}{\partial \boldsymbol{w}_k} = \boldsymbol{x}$$

The gradient of the cost with respect to the weight connected to kth neuron:

$$\nabla_{\boldsymbol{w}_k} J = \frac{\partial J}{\partial u_k}\frac{\partial u_k}{\partial \boldsymbol{w}_k} = \boldsymbol{x}\frac{\partial J}{\partial u_k} \quad \text{(A)}$$

$$\nabla_{b_k} J = \frac{\partial J}{\partial u_k}\frac{\partial u_k}{\partial b_k} = \frac{\partial J}{\partial u_k} \quad \text{(B)}$$

SGD for single layer

Similarly, by substituting $\frac{\partial J}{\partial b_k} = \frac{\partial J}{\partial u_k}$ from (B):

$$\nabla_{\boldsymbol{b}} J = \begin{pmatrix} \frac{\partial J}{\partial b_1} \\ \frac{\partial J}{\partial b_2} \\ \vdots \\ \frac{\partial J}{\partial b_K} \end{pmatrix} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix} = \nabla_{\boldsymbol{u}} J \quad \text{(D)}$$

$$\nabla_{\boldsymbol{b}} J = \nabla_{\boldsymbol{u}} J$$

SGD for single layer

Gradient of J with respect to $\boldsymbol{W} = (\boldsymbol{w}_1 \quad \boldsymbol{w}_2 \quad \cdots \quad \boldsymbol{w}_K)$:

$$\begin{aligned} \nabla_{\boldsymbol{W}} J &= (\nabla_{\boldsymbol{w}_1} J \quad \nabla_{\boldsymbol{w}_2} J \quad \cdots \quad \nabla_{\boldsymbol{w}_K} J) \\ &= \left(\boldsymbol{x}\frac{\partial J}{\partial u_1} \quad \boldsymbol{x}\frac{\partial J}{\partial u_2} \quad \cdots \quad \boldsymbol{x}\frac{\partial J}{\partial u_K}\right) \\ &= \boldsymbol{x}\left(\frac{\partial J}{\partial u_1} \quad \frac{\partial J}{\partial u_2} \quad \cdots \quad \frac{\partial J}{\partial u_K}\right) \\ &= \boldsymbol{x}(\nabla_{\boldsymbol{u}} J)^T \end{aligned}$$

From (A)

where

$$\nabla_{\boldsymbol{u}} J = \frac{\partial J}{\partial \boldsymbol{u}} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix}$$

That is, $\nabla_{\boldsymbol{W}} J = \boldsymbol{x}(\nabla_{\boldsymbol{u}} J)^T$ (C)

SGD for single layer

From (C) and (D),

$$\nabla_{\boldsymbol{W}} J = \boldsymbol{x}(\nabla_{\boldsymbol{u}} J)^T$$
$$\nabla_{\boldsymbol{b}} J = \nabla_{\boldsymbol{u}} J$$

That is, by computing gradient $\nabla_{\boldsymbol{u}} J$ with respect to synaptic input $\boldsymbol{u}$, the gradient of cost J with respect to the weights and biases is obtained.

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{x}(\nabla_{\boldsymbol{u}} J)^T$$
$$\boldsymbol{b} = \boldsymbol{b} - \alpha \nabla_{\boldsymbol{u}} J$$

GD for single layer

Given a set of patterns $\{(\boldsymbol{x}_p, \boldsymbol{d}_p)\}_{p=1}^{P}$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$
and $\boldsymbol{d}_p \in \boldsymbol{R}^K$ for regression and $d_p \in \{1, 2, \cdots K\}$ for classification.

The cost J is given by the sum of cost due to individual patterns:

$$J = \sum_{p=1}^{P} J_p$$

Where Then,

$$\nabla_{\boldsymbol{W}} J = \sum_{p=1}^{P} \nabla_{\boldsymbol{W}} J_p$$

GD for single layer

$$J = \sum_{p=1}^{P} J_p$$

$$\begin{aligned}
\nabla_{\boldsymbol{b}} J &= \sum_{p=1}^{P} \nabla_{\boldsymbol{b}} J_p \\
&= \textstyle\sum_{p=1}^{P} \nabla_{\boldsymbol{u}_p} J_p && \text{Substituting from (D)} \\
&= \textstyle\sum_{p=1}^{P} \nabla_{\boldsymbol{u}_p} J && \text{Since } \nabla_{\boldsymbol{u}_p} J = \nabla_{\boldsymbol{u}_p} J_p. \\
&= \nabla_{\boldsymbol{u}_1} J + \nabla_{\boldsymbol{u}_2} J + \cdots + \nabla_{\boldsymbol{u}_P} J \\
&= (\nabla_{\boldsymbol{u}_1} J \quad \nabla_{\boldsymbol{u}_2} J \quad \cdots \quad \nabla_{\boldsymbol{u}_P} J) \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \\
&= (\nabla_{\boldsymbol{U}} J)^T \mathbf{1}_P && \text{(F)}
\end{aligned}$$

where $\mathbf{1}_P = (1, 1, \cdots 1)^T$ is a vector of P ones.

GD for single layer

Substituting $\nabla_{\boldsymbol{W}} J_p = \boldsymbol{x}_p (\nabla_{\boldsymbol{u}_p} J_p)^T$ from (C) :

$$\begin{aligned}
\nabla_{\boldsymbol{W}} J &= \sum_{p=1}^{P} \boldsymbol{x}_p (\nabla_{\boldsymbol{u}_p} J_p)^T \\
&= \textstyle\sum_{p=1}^{P} \boldsymbol{x}_p (\nabla_{\boldsymbol{u}_p} J)^T && \text{since } \nabla_{\boldsymbol{u}_p} J = \nabla_{\boldsymbol{u}_p} J_p. \\
&= \boldsymbol{x}_1 (\nabla_{\boldsymbol{u}_1} J)^T + \boldsymbol{x}_2 (\nabla_{\boldsymbol{u}_2} J)^T + \cdots + \boldsymbol{x}_P (\nabla_{\boldsymbol{u}_P} J)^T \\
&= (\boldsymbol{x}_1 \quad \boldsymbol{x}_2 \quad \cdots \quad \boldsymbol{x}_P) \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J)^T \\ (\nabla_{\boldsymbol{u}_2} J)^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J)^T \end{pmatrix} \\
&= \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J && \text{(E)}
\end{aligned}$$

Note that $\boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1{}^T \\ \boldsymbol{x}_2{}^T \\ \vdots \\ \boldsymbol{x}_P{}^T \end{pmatrix}$ and $\boldsymbol{U} = \begin{pmatrix} \boldsymbol{u}_1{}^T \\ \boldsymbol{u}_2{}^T \\ \vdots \\ \boldsymbol{u}_P{}^T \end{pmatrix}$

GD for single layer

From (E) and (F):

$$\nabla_{\boldsymbol{W}} J = \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$$
$$\nabla_{\boldsymbol{b}} J = (\nabla_{\boldsymbol{U}} J)^T \mathbf{1}_P$$

That is, by computing gradient $\nabla_{\boldsymbol{U}} J$ with respect to synaptic input, the weights and biases can be updated.

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$$
$$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{U}} J)^T \mathbf{1}_P$$

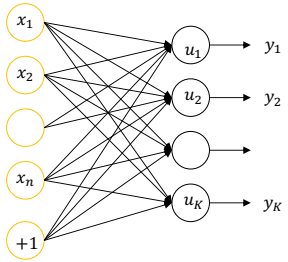
Learning a single layer

Learning a layer of neurons	
SGD	$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_{\boldsymbol{u}} J)^T$ $\boldsymbol{b} = \boldsymbol{b} - \alpha \nabla_{\boldsymbol{u}} J$
GD	$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$ $\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{U}} J)^T \boldsymbol{1}_P$

To learn a given layer, we need to compute
$\nabla_{\boldsymbol{u}} J$ for SGD and $\nabla_{\boldsymbol{U}} J$ for GD.

Those gradients with respect to synaptic inputs are dependent on the types of neurons in the layer.

Perceptron layer



$$y_k = f(u_k) = \frac{1}{1 + e^{-u_k}}$$

A layer of perceptrons performs **multidimensional non-linear regression** and learns a multidimensional non-linear mapping:

$$\phi: \boldsymbol{R}^n \rightarrow \boldsymbol{R}^K$$

SGD for perceptron layer

Given a training pattern $(\boldsymbol{x}, \boldsymbol{d})$
Note $\boldsymbol{x} = (x_1, x_2, \cdots x_n)^{\boldsymbol{T}} \in \boldsymbol{R}^n$ and $\boldsymbol{d} = (d_1, d_2, \cdots d_K)^{\boldsymbol{T}} \in \boldsymbol{R}^K$.

The square-error cost function:

$$J = \frac{1}{2} \sum_{k=1}^{K} (d_k - y_k)^2$$

where $y_k = f(u_k) = \frac{1}{1+e^{-u_k}}$ and $u_k = \boldsymbol{x}^T \boldsymbol{w}_k + b_k$.

Gradient of J with respect to u_k:

$$\frac{\partial J}{\partial u_k} = \frac{\partial J}{\partial y_k} \frac{\partial y_k}{\partial u_k} = -(d_k - y_k) \frac{\partial y_k}{\partial u_k} = -(d_k - y_k) f'(u_k) \qquad \text{(G)}$$

SGD for perceptron layer

Substituting $\nabla_{u_k} J = \frac{\partial J}{\partial u_k} = -(d_k - y_k) f'(u_k)$ from (G):

$$\nabla_{\boldsymbol{u}} J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = -\begin{pmatrix} (d_1 - y_1) f'(u_1) \\ (d_2 - y_2) f'(u_2) \\ \vdots \\ (d_K - y_K) f'(u_K) \end{pmatrix} = -(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u}) \qquad \text{(H)}$$

and '$\cdot$' denotes element-wise multiplication.

$$\nabla_{\boldsymbol{u}} J = -(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$

SGD for perceptron layer

Given a training dataset $\{(\boldsymbol{x}, \boldsymbol{d})\}$
Set learning parameter α
Initialize $\boldsymbol{W}$ and $\boldsymbol{b}$
Repeat until convergence:
For every pattern $(\boldsymbol{x}, \boldsymbol{d})$:

$$\boldsymbol{u} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{y} = f(\boldsymbol{u}) = \frac{1}{1+e^{-u}}$$
$$\nabla_{\boldsymbol{u}} J = -(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$$
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_{\boldsymbol{u}} J)^T$$
$$\boldsymbol{b} = \boldsymbol{b} - \alpha \nabla_{\boldsymbol{u}} J$$

GD for perceptron layer

$$\boldsymbol{U} = \begin{pmatrix} \boldsymbol{u}_1{}^T \\ \boldsymbol{u}_2{}^T \\ \vdots \\ \boldsymbol{u}_P{}^T \end{pmatrix} \rightarrow \nabla_{\boldsymbol{U}} J = \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J)^T \\ (\nabla_{\boldsymbol{u}_2} J)^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J_1)^T \\ (\nabla_{\boldsymbol{u}_2} J_2)^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J_P)^T \end{pmatrix}$$

From (H), substitute $\nabla_{\boldsymbol{u}} J = -(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$:

$$\nabla_{\boldsymbol{U}} J = -\begin{pmatrix} \left((\boldsymbol{d}_1 - \boldsymbol{y}_1) \cdot f'(\boldsymbol{u}_1)\right)^T \\ \left((\boldsymbol{d}_2 - \boldsymbol{y}_2) \cdot f'(\boldsymbol{u}_2)\right)^T \\ \vdots \\ \left((\boldsymbol{d}_P - \boldsymbol{y}_P) \cdot f'(\boldsymbol{u}_P)\right)^T \end{pmatrix} = -\begin{pmatrix} \left(\boldsymbol{d}_1{}^T - \boldsymbol{y}_1{}^T\right) \cdot f'(\boldsymbol{u}_1{}^T) \\ \left(\boldsymbol{d}_2{}^T - \boldsymbol{y}_2{}^T\right) \cdot f'(\boldsymbol{u}_2{}^T) \\ \vdots \\ \left(\boldsymbol{d}_P{}^T - \boldsymbol{y}_P{}^T\right) \cdot f'(\boldsymbol{u}_P{}^T) \end{pmatrix}$$

$$\nabla_{\boldsymbol{U}} J = -(\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$$

where $\boldsymbol{D} = \begin{pmatrix} \boldsymbol{d}_1^T \\ \boldsymbol{d}_2^T \\ \vdots \\ \boldsymbol{d}_P^T \end{pmatrix}$, $\boldsymbol{Y} = \begin{pmatrix} \boldsymbol{y}_1^T \\ \boldsymbol{y}_2^T \\ \vdots \\ \boldsymbol{y}_P^T \end{pmatrix}$, and $\boldsymbol{U} = \begin{pmatrix} \boldsymbol{u}_1^T \\ \boldsymbol{u}_2^T \\ \vdots \\ \boldsymbol{u}_P^T \end{pmatrix}$

GD for perceptron layer

Given a training dataset $\{(\boldsymbol{x}_p, \boldsymbol{d}_p)\}_{p=1}^P$

Note $\boldsymbol{x}_p = (x_{p1}, x_{p2}, \cdots x_{pn})^T \in \boldsymbol{R}^n$ and $\boldsymbol{d}_p = (d_{p1}, d_{p2}, \cdots d_{pK})^T \in \boldsymbol{R}^K$.

The cost function J is given by the sum of square errors (s.s.e.):

$$J = \frac{1}{2} \sum_{p=1}^{P} \sum_{k=1}^{K} (d_{pk} - y_{pk})^2$$

J can be written as the sum of cost due to individual patterns:

$$J = \sum_{p=1}^{P} J_p$$

where $J_p = \frac{1}{2} \sum_{k=1}^{K} (d_{pk} - y_{pk})^2$ is the square error for the p th pattern.

GD for perceptron layer

Given a training dataset $(\boldsymbol{X}, \boldsymbol{D})$
Set learning parameter α
Initialize $\boldsymbol{W}$ and $\boldsymbol{b}$
Repeat until convergence:

$$\boldsymbol{U} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$
$$\boldsymbol{Y} = f(\boldsymbol{U}) = \frac{1}{1+e^{-U}}$$
$$\nabla_{\boldsymbol{U}} J = -(\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$$
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$$
$$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{U}} J)^T \boldsymbol{1}_P$$

Learning a perceptron layer

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$	$\boldsymbol{u} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
$\boldsymbol{Y} = f(\boldsymbol{U})$	$\boldsymbol{y} = f(\boldsymbol{u})$
$\nabla_{\boldsymbol{U}} J = -(\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$	$\nabla_{\boldsymbol{u}} J = -(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$
$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$	$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_{\boldsymbol{u}} J)^T$
$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{U}} J)^T \boldsymbol{1}_P$	$\boldsymbol{b} = \boldsymbol{b} - \alpha \nabla_{\boldsymbol{u}} J$

Softmax layer

The K neurons in the softmax layer performs K class classification and represent K classes.

The activation of each neuron k estimates the probability $P(y = k|x)$ that the input $\boldsymbol{x}$ belongs to the class k:

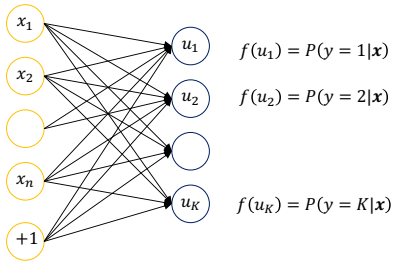
$$P(y = k|\boldsymbol{x}) = f(u_k) = \frac{e^{u_k}}{\sum_{k'=1}^{K} e^{u_{k'}}}$$

where $u_k = \boldsymbol{w}_k{}^T \boldsymbol{x} + b_k$, and $\boldsymbol{w}_k$ is weight vector and b_k is bias of neuron k.

The above activation function f is known as **softmax activation function**.

Softmax layer

Softmax layer is the extension of logistic regression to **multiclass classification** problem, which is also known as *multinomial logistic regression*.



Each neuron in the softmax layer corresponds to one class label. The activation of a neuron gives the probability of the input belonging to that class label. Output is the class label having the maximum probability.

Softmax layer

The output y denotes the class label of the input pattern, which is given by

$$y = \underset{k}{\text{argmax}}\, P(y = k|\boldsymbol{x}) = \underset{k}{\text{argmax}}\, f(u_k)$$

That is, the class label is assigned to the class with the maximum activation.

SGD for softmax layer

Given a training pattern $(\boldsymbol{x}, d)$ where $\boldsymbol{x} \in \boldsymbol{R}^n$ and $d \in \{1,2,\cdots K\}$.

The cost function for learning is by the *multiclass cross-entropy:*

$$J = -\sum_{k=1}^{K} 1(d = k) log\big(f(u_k)\big)$$

where u_k is the synaptic input to the k the neuron.

The cost function can also be written as

$$J = -log\big(f(u_d)\big)$$

where d is the target label of input $\boldsymbol{x}$.

Note that the logarithm here is natural: $log = log_e$

SGD for softmax layer

If $k = d$:

$$\frac{\partial f(u_d)}{\partial u_k} = \frac{\partial}{\partial u_k}\left(\frac{e^{u_k}}{\sum_{k'=1}^{K} e^{u_{k'}}}\right)$$
$$= \frac{\left(\sum_{k'=1}^{K} e^{u_{k'}}\right)e^{u_k} - e^{u_k}e^{u_k}}{\left(\sum_{k'=1}^{K} e^{u_{k'}}\right)^2}$$
$$= \frac{e^{u_k}}{\sum_{k'=1}^{K} e^{u_{k'}}}\left(1 - \frac{e^{u_k}}{\sum_{k'=1}^{K} e^{u_{k'}}}\right)$$
$$= f(u_k)\big(1 - f(u_k)\big)$$
$$= f(u_d)\big(1(k = d) - f(u_k)\big)$$

$$\frac{\partial\left(\sum_{k'=1}^{K} e^{u_{k'}}\right)}{\partial u_k} = e^{u_k}$$

If $k \neq d$:

$$\frac{\partial f(u_d)}{\partial u_k} = \frac{\partial}{\partial u_k}\left(\frac{e^{u_d}}{\sum_{k'=1}^{K} e^{u_{k'}}}\right)$$
$$= -\frac{e^{u_d}e^{u_k}}{\left(\sum_{k'=1}^{K} e^{u_{k'}}\right)^2}$$
$$= -f(u_d)f(u_k)$$
$$= f(u_d)\big(1(k = d) - f(u_k)\big)$$

$$1(k = d) = 0$$

SGD for softmax layer

$$J = -log\big(f(u_d)\big)$$

The gradient with respect to u_k is given by

$$\frac{\partial J}{\partial u_k} = -\frac{1}{f(u_d)}\frac{\partial f(u_d)}{\partial u_k} \qquad \text{(I)}$$

where

$$\frac{\partial f(u_d)}{\partial u_k} = \frac{\partial}{\partial u_k}\left(\frac{e^{u_d}}{\sum_{k'=1}^{K} e^{u_{k'}}}\right)$$

The above differentiation need to be considered separately for $k = d$ and for $k \neq d$.

SGD for softmax layer

$$\frac{\partial f(u_d)}{\partial u_k} = f(u_d)\big(1(d = k) - f(u_k)\big)$$

Substituting in (I):

$$\nabla_{u_k} J = \frac{\partial J}{\partial u_k} = -\frac{1}{f(u_d)}\frac{\partial f(u_d)}{\partial u_k} = -\big(1(d = k) - f(u_k)\big)$$

Gradient J with respect to $\boldsymbol{u}$:

$$\nabla_{\boldsymbol{u}} J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = -\begin{pmatrix} 1(d = 1) - f(u_1) \\ 1(d = 2) - f(u_2) \\ \vdots \\ 1(d = K) - f(u_K) \end{pmatrix} = -\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big) \qquad \text{(J)}$$

where $\boldsymbol{k} = (1 \quad 2 \quad \cdots \quad K)^T$

SGD for softmax layer

For a softmax layer:

$$\nabla_{\boldsymbol{u}} J = -\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big)$$

where:

$$1(\boldsymbol{k} = d) = \begin{pmatrix} 1(d = 1) \\ 1(d = 2) \\ \vdots \\ 1(d = K) \end{pmatrix} \text{ and } f(\boldsymbol{u}) = \begin{pmatrix} f(u_1) \\ f(u_2) \\ \vdots \\ f(u_K) \end{pmatrix}$$

Note that $1(\boldsymbol{k} = d)$ is a one-hot vector where the element corresponding to the target label d is '1' and elsewhere is '0'.

GD for softmax layer

J can be written as the sum of cost due to individual patterns:

$$J = \sum_{p=1}^{P} J_p$$

where $J_p = -log\left(f\left(u_{pd_p}\right)\right)$ is the cross-entropy for the p th pattern.

GD for softmax layer

Given a set of patterns $\{(\boldsymbol{x}_p, d_{\boldsymbol{p}})\}_{p=1}^{P}$ where $\boldsymbol{x}_p \in \boldsymbol{R}^n$ and $d_p \in \{1,2,\cdots K\}$.

The cost function of the *softmax layer* is given by the *multiclass cross-entropy*:

$$J = -\sum_{p=1}^{P}\left(\sum_{k=1}^{K} 1(d_p = k)\log\left(f(u_{pk})\right)\right)$$

where u_{pk} is the synaptic input to the k the neuron for input $\boldsymbol{x}_p$.

The cost function J can also be written as

$$J = -\sum_{p=1}^{P} log\left(f\left(u_{pd_p}\right)\right)$$

GD for softmax layer

$$\nabla_{\boldsymbol{U}} J = \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J)^T \\ (\nabla_{\boldsymbol{u}_2} J)^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J_1)^T \\ (\nabla_{\boldsymbol{u}_2} J_2)^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J_P)^T \end{pmatrix}$$

Substituting $\nabla_{\boldsymbol{u}} J = -\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big)$ from (J):

$$\nabla_{\boldsymbol{U}} J = -\begin{pmatrix} \big(1(\boldsymbol{k} = d_1) - f(\boldsymbol{u}_1)\big)^T \\ \big(1(\boldsymbol{k} = d_2) - f(\boldsymbol{u}_2)\big)^T \\ \vdots \\ \big(1(\boldsymbol{k} = d_P) - f(\boldsymbol{u}_P)\big)^T \end{pmatrix}$$

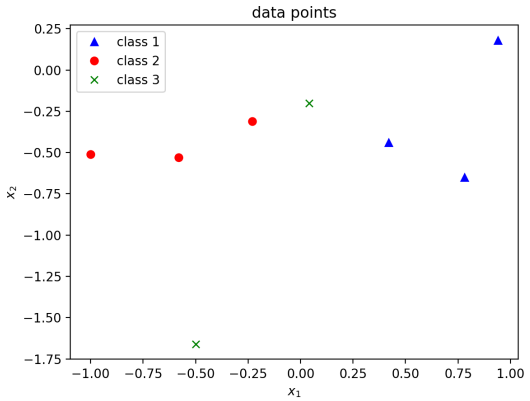
$$\nabla_{\boldsymbol{U}} J = -\big(\boldsymbol{K} - f(\boldsymbol{U})\big)$$

where $\boldsymbol{K} = \begin{pmatrix} 1(\boldsymbol{k} = d_1)^T \\ 1(\boldsymbol{k} = d_2)^T \\ \vdots \\ 1(\boldsymbol{k} = d_P)^T \end{pmatrix}$ is a matrix with every row is a one-hot vector.

Learning a softmax layer

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}, d)$
$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$	$\boldsymbol{u} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
$f(\boldsymbol{U}) = \frac{e^{\boldsymbol{U}}}{\sum_{k'=1}^{K} e^{\boldsymbol{U}_{k'}}}$	$f(\boldsymbol{u}) = \frac{e^{u_k}}{\sum_{k'=1}^{K} e^{u_{k'}}}$
$\boldsymbol{y} = \underset{k}{\text{argmax}}\, f(\boldsymbol{U})$	$y = \underset{k}{\text{argmax}}\, f(\boldsymbol{u})$
$\nabla_{\boldsymbol{U}} J = -\big(\boldsymbol{K} - f(\boldsymbol{U})\big)$	$\nabla_{\boldsymbol{u}} J = -\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big)$
$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$	$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_{\boldsymbol{u}} J)^T$
$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{U}} J)^T \boldsymbol{1}_P$	$\boldsymbol{b} = \boldsymbol{b} - \alpha \nabla_{\boldsymbol{u}} J$

Example 2



Example 2: GD of a softmax layer

Train a softmax regression layer of neurons to perform the following classification:

$$
\begin{aligned}
(0.94 \quad 0.18) &\rightarrow class\ A \\
(-0.58 \quad -0.53) &\rightarrow class\ B \\
(-0.23 \quad -0.31) &\rightarrow class\ B \\
(0.42 \quad -0.44) &\rightarrow class\ A \\
(0.5 \quad -1.66) &\rightarrow class\ C \\
(-1.0 \quad -0.51) &\rightarrow class\ B \\
(0.78 \quad -0.65) &\rightarrow class\ A \\
(0.04 \quad -0.20) &\rightarrow class\ C
\end{aligned}
$$

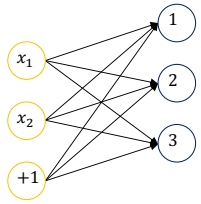
Use a learning factor $\alpha = 0.05$.

Example 2

Let $y = \begin{cases} 1, \text{for } class\ A \\ 2, \text{for } class\ B \\ 3, \text{for } class\ C \end{cases}$

$$
\boldsymbol{X} = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}, \boldsymbol{d} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \\ 1 \\ 3 \end{pmatrix}
$$

$$
\boldsymbol{K} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$



Example 2

Initialize $\boldsymbol{W} = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix},$

Then, $\boldsymbol{B} = \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$

1st epoch starts …

$$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B} = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix} \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Example 2

$$\boldsymbol{y} = \underset{k}{\text{argmax}}\{f(\boldsymbol{U})\} = \underset{k}{\text{argmax}} \left\{ \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix} \right\} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 3 \\ 2 \\ 3 \\ 3 \end{pmatrix}$$

Errors = $\sum_{p=1}^{8} 1(d_p \neq y_p) = 2$

$\boldsymbol{d} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \\ 1 \\ 3 \end{pmatrix}$

Entropy, $J = -\sum_{p=1}^{8} log\left(f\left(u_{pd_p}\right)\right)$

$= -log(0.44) - log(0.42) - \cdots - log(0.35)$

$= 7.26$

Example 2

$$\boldsymbol{U} = \begin{pmatrix} 0.86 & 0.11 & 0.64 \\ -0.84 & -0.28 & -0.49 \\ -0.41 & -0.16 & -0.22 \\ -0.01 & -0.21 & 0.17 \\ -0.86 & -0.82 & -0.06 \\ -1.15 & -0.27 & -0.75 \\ 0.11 & -0.31 & 0.35 \\ -0.12 & -0.10 & -0.02 \end{pmatrix}$$

$$f(u_{12}) = \frac{e^{0.11}}{e^{0.86} + e^{0.11} + e^{0.64}}$$

$$f(\boldsymbol{U}) = \frac{e^{(\boldsymbol{U})}}{\sum_{k=1}^{K} e^{(\boldsymbol{U})}} = \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix}$$

Example 2

$$\nabla_{\boldsymbol{U}} J = -(\boldsymbol{K} - f(\boldsymbol{U}))$$

$$= -\left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix} \right)$$

$$= \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix}$$

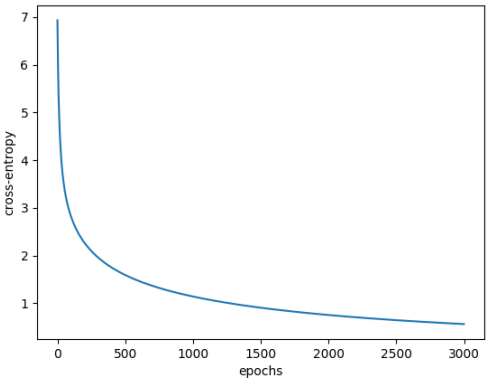
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_U J$$

$$= \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix} - 0.05 \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.2 \end{pmatrix}^T \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix}$$

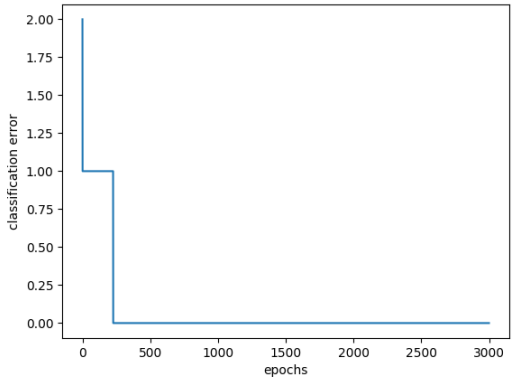
$$= \begin{pmatrix} 0.85 & -0.06 & 0.63 \\ 0.76 & 0.50 & 0.22 \end{pmatrix}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_U J)^T \mathbf{1}_P = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix} - 0.05 \begin{pmatrix} -0.56 & 0.21 & 0.35 \\ 0.24 & -0.58 & 0.34 \\ 0.29 & -0.63 & 0.35 \\ -0.67 & 0.27 & 0.40 \\ 0.23 & 0.24 & -0.48 \\ 0.20 & -0.51 & 0.31 \\ -0.65 & 0.22 & 0.43 \\ 0.32 & 0.33 & -0.65 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.03 \\ 0.02 \\ -0.05 \end{pmatrix}$$

Example 2



Example 2



Example 2

At convergence at 3000 iterations:

$$\boldsymbol{W} = \begin{pmatrix} 14.22 & -13.04 & 0.00 \\ 4.47 & -2.05 & -0.95 \end{pmatrix}$$

$$\boldsymbol{b} = \begin{pmatrix} -0.53 \\ -0.47 \\ 1.00 \end{pmatrix}$$

Entropy = 0.562

Errors = 0

Initialization of weights

Random initialization is inefficient

At initialization, it is desirable that weights are small and near zero
- to operate in the linear region of the activation function
- to preserve the variance of activations and gradients.

Two methods:
- Using a unform distribution within specified limits
- Using a truncated normal distribution

Initialization from a truncated normal distribution

$$w \sim truncated\_normal\left[mean = 0, std = \frac{gain}{\sqrt{n_{in}}}\right]$$

In the truncated normal, the samples that are two s.d. away from the center are discarded and resampled again.

This is also known as **Kaiming normal initialization**.

Initialization from a uniform distribution (Xavier/Glorat uniform Initialization)

$Uniformly$ draws weight samples $w \sim U(-a, +a)$:

where

$$a = gain \times \sqrt{\frac{6}{n_{in}+n_{out}}}$$

n_{in} is the number of input nodes and n_{out} is the number of neurons in the layer.

activation	linear	sigmoid	Tanh	ReLU	Leaky ReLU
gain	1	1	5/3	$\sqrt{2}$	$\sqrt{\frac{1}{1+slope^2}}$

Iris dataset

Iris dataset:
https://archive.ics.uci.edu/ml/datasets/Iris

Three classes of iris flower:

Setosa Versicolour Virginica

Four features:
Sepal length, sepal width, petal length, petal width

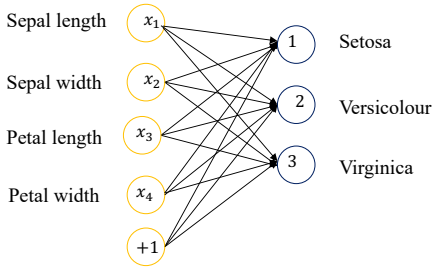
Iris dataset

150 data points, 50 for each class

Features:
[-7.43333333e-01 4.46000000e-01 -2.35866667e+00 -9.98666667e-01]
[-9.43333333e-01 -5.40000000e-02 -2.35866667e+00 -9.98666667e-01]
[-1.14333333e+00 1.46000000e-01 -2.45866667e+00 -9.98666667e-01]

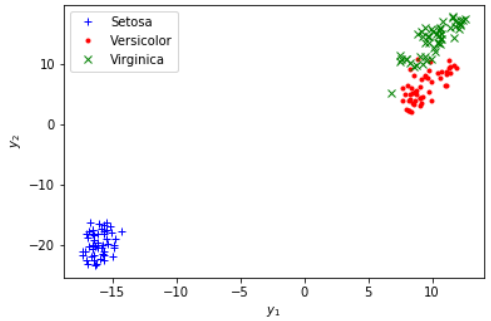
Labels:
[0 0 0 0 0 0 0 0 0 0 ..1 1 1 1 1 1 1 1 1 1….. 2 2 2 2 2 2 2 2 2 2]

Example 3: Softmax classification of iris data

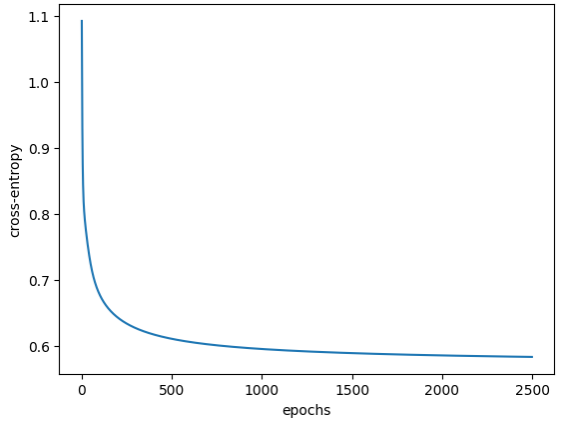


Iris data

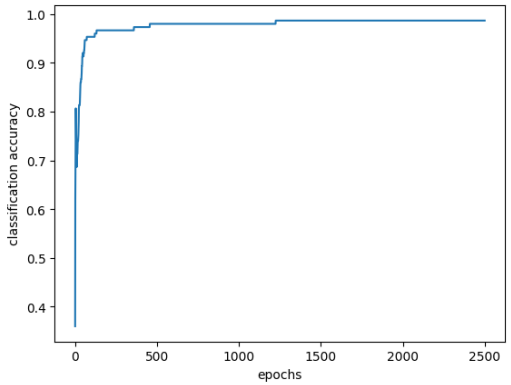
Display of data points after dimensionality reduction (from Four dimensions to Two) by t-SNE.



Example 3



Example 3



Final classification error = 5

Summary: GD for layers

$$(\boldsymbol{X}, \boldsymbol{D})$$
$$\boldsymbol{U} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T(\boldsymbol{\nabla}_{\boldsymbol{U}} J)$$
$$\boldsymbol{b} = \boldsymbol{b} - \alpha(\boldsymbol{\nabla}_{\boldsymbol{U}} J)^T \boldsymbol{1}_P$$

layer	$f(\boldsymbol{U}), \boldsymbol{Y}$	$\boldsymbol{\nabla}_{\boldsymbol{U}} J$
Linear neuron layer	$\boldsymbol{Y} = f(\boldsymbol{U}) = \boldsymbol{U}$	$-(\boldsymbol{D} - \boldsymbol{Y})$
Perceptron layer	$\boldsymbol{Y} = f(\boldsymbol{U}) = \frac{1}{1 + e^{-\boldsymbol{U}}}$	$-(\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$
Softmax layer	$f(\boldsymbol{U}) = \frac{e^{\boldsymbol{U}}}{\sum_{k=1}^{K} e^{\boldsymbol{U}_k}}$ $\boldsymbol{y} = \underset{k}{\text{argmax}}\, f(\boldsymbol{U})$	$-(\boldsymbol{K} - f(\boldsymbol{U}))$

Revision: neurons and layers

Classification	Logistic neurons
Two-class	Logistic regression neuron
Multiclass	Softmax layer

Regression	Linear	Non-linear
One dimensional	Linear neuron	Perceptron
Multi-dimensional	Linear neuron layer	Perceptron layer

Summary: SGD for layers

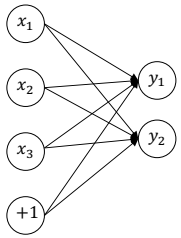
$$(\boldsymbol{x}, \boldsymbol{d})$$
$$\boldsymbol{u} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{x}(\boldsymbol{\nabla}_{\boldsymbol{u}} J)^T$$
$$\boldsymbol{b} = \boldsymbol{b} - \alpha(\boldsymbol{\nabla}_{\boldsymbol{u}} J)$$

layer	$f(\boldsymbol{u}), \boldsymbol{y}$	$\boldsymbol{\nabla}_{\boldsymbol{u}} J$
Linear neuron layer	$\boldsymbol{y} = f(\boldsymbol{u}) = \boldsymbol{u}$	$-(\boldsymbol{d} - \boldsymbol{y})$
Perceptron layer	$\boldsymbol{y} = f(\boldsymbol{u}) = \frac{1}{1 + e^{-\boldsymbol{u}}}$	$-(\boldsymbol{d} - \boldsymbol{y}) \cdot f'(\boldsymbol{u})$
Softmax layer	$f(\boldsymbol{u}) = \frac{e^{u}}{\sum_{k'=1}^{K} e^{k'}}$ $\boldsymbol{y} = \underset{k}{\text{argmax}}\, f(\boldsymbol{u})$	$-\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big)$

Thank you.

Example 4

$$\boldsymbol{X} = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix} \text{ and } \boldsymbol{D} = \begin{pmatrix} 0.37 & 0.47 \\ 0.36 & 0.38 \\ 0.35 & 0.25 \\ 0.48 & 0.42 \\ 0.36 & 0.29 \\ 0.44 & 0.52 \\ 0.60 & 0.52 \\ 0.28 & 0.37 \end{pmatrix}$$



Output $y_1, y_2 \in [0, 1]$

So, activation function for both neurons:

$$f(u) = \frac{1}{1 + e^{-u}}$$
$$f'(u) = y(1 - y)$$

Learning factor $\alpha = 0.1$.

Weights and biases are initialized:

$$\boldsymbol{W} = \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

Example 4: GD of a perceptron layer

Design a perceptron layer to perform the following mapping using GD learning:

$\boldsymbol{x} = (x_1, x_2, x_3)$	$\boldsymbol{d} = (d_1, d_2)$
(0.77, 0.02, 0.63)	(0.37, 0.47)
(0.75, 0.50, 0.22)	(0.36, 0.38)
(0.20, 0.76, 0.17)	(0.35, 0.25)
(0.09, 0.69, 0.95)	(0.48 0.42)
(0.00, 0.51, 0.81)	(0.36, 0.29)
(0.61, 0.72, 0.29)	(0.44 0.52)
(0.92, 0.71, 0.54)	(0.60, 0.52)
(0.14, 0.37, 0.67)	(0.28, 0.37)

Use $\alpha = 0.1$.

Example 4

Epoch 1:

$$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B}$$

$$\boldsymbol{U} = \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix} \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 0.03 & 0.06 \\ 0.03 & 0.06 \\ 0.02 & 0.05 \\ 0.03 & 0.07 \\ 0.02 & 0.05 \\ 0.03 & 0.07 \\ 0.04 & 0.09 \\ 0.02 & 0.05 \end{pmatrix}$$

$$\boldsymbol{Y} = f(\boldsymbol{U}) = \frac{1}{1+e^{-U}} = \begin{pmatrix} 0.51 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \end{pmatrix}$$

Mean square error $= \frac{1}{8}\sum_{p=1}^{8}\sum_{k=1}^{2}(d_{pk} - y_{pk})^2 = \frac{1}{8}\sum_{p=1}^{8}(d_{p1} - y_{p1})^2 + (d_{p2} - y_{p2})^2 = 0.04$

Example 4

$$f'(\boldsymbol{U}) = \boldsymbol{Y} \cdot (1 - \boldsymbol{Y}) = \begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix}$$

$$\nabla_{\boldsymbol{U}} J = -(\boldsymbol{D} - \boldsymbol{Y}) \cdot f'(\boldsymbol{U})$$

$$= -\left(\begin{pmatrix} 0.37 & 0.47 \\ 0.36 & 0.38 \\ 0.35 & 0.25 \\ 0.48 & 0.42 \\ 0.36 & 0.29 \\ 0.44 & 0.52 \\ 0.60 & 0.52 \\ 0.28 & 0.37 \end{pmatrix} - \begin{pmatrix} 0.51 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \\ 0.51 & 0.52 \\ 0.51 & 0.52 \\ 0.50 & 0.51 \end{pmatrix} \right) \cdot \begin{pmatrix} 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \\ 0.25 & 0.25 \end{pmatrix} = \begin{pmatrix} 0.04 & 0.01 \\ 0.04 & 0.03 \\ 0.04 & 0.07 \\ 0.01 & 0.03 \\ 0.04 & 0.06 \\ 0.02 & 0.00 \\ -0.02 & 0.00 \\ 0.06 & 0.04 \end{pmatrix}$$

Example 4

Epoch	Y	mse	W	b
2	$\begin{pmatrix} 0.50 & 0.51 \\ 0.50 & 0.51 \\ 0.50 & 0.50 \\ 0.50 & 0.51 \\ 0.50 & 0.50 \\ 0.50 & 0.51 \\ 0.50 & 0.51 \\ 0.50 & 0.50 \end{pmatrix}$	0.036	$\begin{pmatrix} 0.02 & 0.03 \\ -0.01 & 0.02 \\ 0.00 & 0.01 \end{pmatrix}$	$\begin{pmatrix} -0.04 \\ -0.04 \end{pmatrix}$
10000	$\begin{pmatrix} 0.34 & 0.46 \\ 0.39 & 0.42 \\ 0.32 & 0.29 \\ 0.48 & 0.40 \\ 0.36 & 0.34 \\ 0.45 & 0.42 \\ 0.59 & 0.55 \\ 0.31 & 0.33 \end{pmatrix}$	0.003	$\begin{pmatrix} 1.08 & 1.14 \\ 1.42 & 0.40 \\ 1.14 & 0.84 \end{pmatrix}$	$\begin{pmatrix} -2.24 \\ -1.57 \end{pmatrix}$

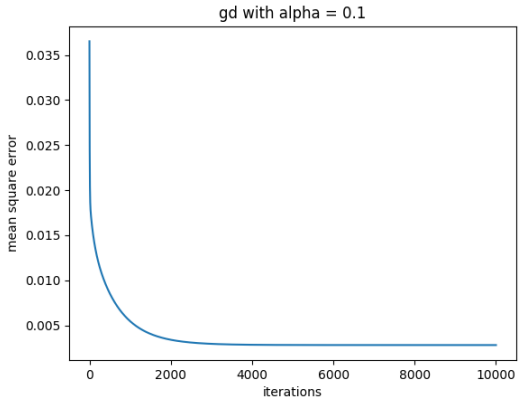
Example 4

$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{U}} J$$

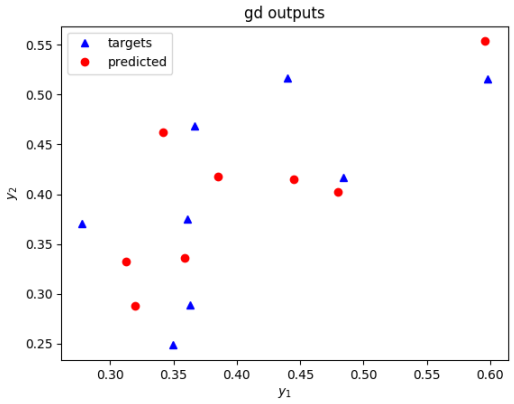
$$= \begin{pmatrix} 0.03 & 0.04 \\ 0.01 & 0.04 \\ 0.02 & 0.04 \end{pmatrix} - 0.1 \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.22 \\ 0.20 & 0.76 & 0.17 \\ 0.09 & 0.69 & 0.95 \\ 0.00 & 0.51 & 0.81 \\ 0.61 & 0.72 & 0.29 \\ 0.92 & 0.71 & 0.54 \\ 0.14 & 0.37 & 0.67 \end{pmatrix}^T \begin{pmatrix} 0.04 & 0.01 \\ 0.04 & 0.03 \\ 0.04 & 0.07 \\ 0.01 & 0.03 \\ 0.04 & 0.06 \\ 0.02 & 0.00 \\ -0.02 & 0.00 \\ 0.06 & 0.04 \end{pmatrix} = \begin{pmatrix} 0.02 & 0.04 \\ 0.00 & 0.03 \\ 0.01 & 0.03 \end{pmatrix}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{U}} J)^T \mathbf{1}_P = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix} - 0.1 \begin{pmatrix} 0.03 & 0.01 \\ 0.03 & 0.03 \\ 0.04 & 0.06 \\ 0.00 & 0.02 \\ 0.03 & 0.05 \\ 0.01 & 0.00 \\ -0.02 & 0.00 \\ 0.05 & 0.03 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -0.02 \\ -0.02 \end{pmatrix}$$

Example 4



Example 4



Chapter 4

Deep neural networks

Neural networks and deep learning

Chain rule of differentiation

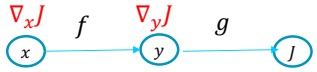
Let $x, y,$ and $J \in \boldsymbol{R}$ be one-dimensional variables and

$$J = g(y)$$
$$y = f(x)$$

Chain rule of differentiation states that:

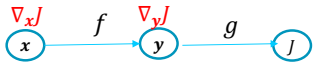
$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial x}$$

$$\nabla_x J = \left(\frac{\partial y}{\partial x}\right)\nabla_y J$$



Note the transfer of gradient of J from y to x.

Chain rule in multidimensions



$\boldsymbol{x} = (x_1, x_2, \cdots x_n) \in \boldsymbol{R}^n$, $\boldsymbol{y} = (y_1, y_2, \cdots y_K) \in \boldsymbol{R}^K$, $J \in \boldsymbol{R}$, and

$$\boldsymbol{y} = f(\boldsymbol{x})$$
$$J = g(\boldsymbol{y})$$

Then, the chain rule of differentiation states that:

$$\nabla_x J = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_y J$$

The matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is known as the **Jacobian** of the function f where $\boldsymbol{y} = f(\boldsymbol{x})$.

Chain rule in multidimensions

$$\nabla_x J = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_y J$$

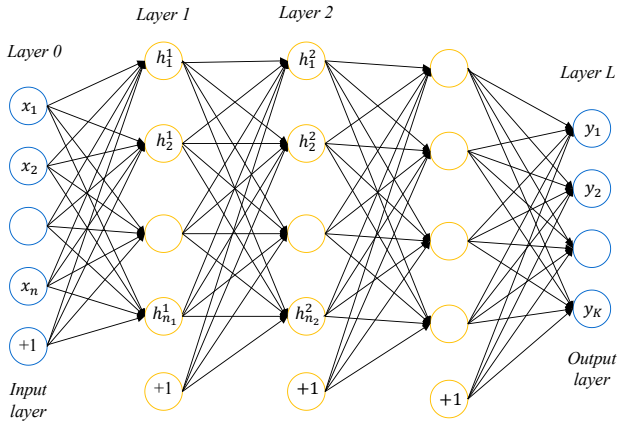
where

$$\nabla_x J = \frac{\partial J}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial J}{\partial x_1} \\ \frac{\partial J}{\partial x_2} \\ \vdots \\ \frac{\partial J}{\partial x_n} \end{pmatrix} \text{ and } \nabla_y J = \frac{\partial J}{\partial \boldsymbol{y}} = \begin{pmatrix} \frac{\partial J}{\partial y_1} \\ \frac{\partial J}{\partial y_2} \\ \vdots \\ \frac{\partial J}{\partial y_K} \end{pmatrix},$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \frac{\partial y_K}{\partial x_2} & \cdots & \frac{\partial y_K}{\partial x_n} \end{pmatrix}$$

Note that differentiation of a scalar by a vector results in a vector and differentiation of a vector by a vector results in a matrix.

Deep neural networks (DDN): Also known as feedforward networks (FFN)



L layer DNN

Example 1: find Jacobian of a function

Let $\boldsymbol{x} = (x_1, x_2, x_3)$, $\boldsymbol{y} = (y_1, y_2)$, and $\boldsymbol{y} = f(\boldsymbol{x})$ where f is given by

$$y_1 = 5 - 2x_1 + 3x_3$$
$$y_2 = x_1 + 5{x_2}^2 + {x_3}^3 - 1$$

Find the Jacobian of f.

$$\frac{\partial y_1}{\partial x_1} = -2, \quad \frac{\partial y_1}{\partial x_2} = 0, \quad \frac{\partial y_1}{\partial x_3} = 3$$
$$\frac{\partial y_2}{\partial x_1} = 1, \quad \frac{\partial y_2}{\partial x_2} = 10x_2, \quad \frac{\partial y_2}{\partial x_3} = 3{x_3}^2$$

Jacobian:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{pmatrix} = \begin{pmatrix} -2 & 0 & 3 \\ 1 & 10x_2 & 3{x_3}^2 \end{pmatrix}$$

Feedforward Networks (FFN)

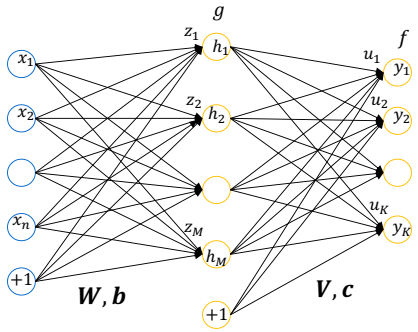
Feedforward networks (FFN) consists of several layers of neurons where activations propagate from input layer to output layer. The layers between the input and output layers are referred to as **hidden layers**.

The number of layers is referred to as the **depth** of the feedforward network. When a network has many hidden layers of neurons, feedforward networks are referred to as **deep neural networks (DNN)**. Learning in deep neural networks is referred to as **deep learning**. The number of neurons in a layer is referred to as the **width** of that layer.

The hidden layers are usually composed of perceptrons (sigmoidal units) or ReLU units and the output layer is usually

- A linear neuron layer for regression
- A softmax layer for classification

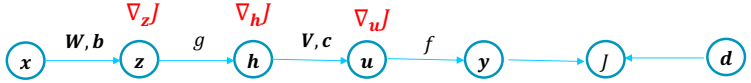
Two-layer FFN



Input $\boldsymbol{x} = (x_1 \quad x_2 \quad \cdots \quad x_n)^T$
Hidden-layer output $\boldsymbol{h} = (h_1 \quad h_2 \quad \cdots \quad h_M)^T$
Output $\boldsymbol{y} = (y_1 \quad y_2 \quad \cdots \quad y_K)^T$

$\boldsymbol{W}, \boldsymbol{b}$ – weight and bias of the hidden layer
$\boldsymbol{V}, \boldsymbol{c}$ – weight and bias of the output layer
M is the number of hidden layer neurons

Backpropagation of gradients

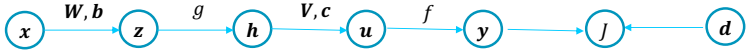


Since the targets appear at the output, the error gradient at the output layer is $\nabla_{\boldsymbol{u}} J$ is known. Therefore, output weights and bias, $\boldsymbol{V}, \boldsymbol{c}$, can be learnt.

To learn hidden layer weights and biases, the gradients at the output layer are to be backpropagated to hidden layers.

Forward propagation of activations: single input pattern

Consider an input pattern $(\boldsymbol{x}, \boldsymbol{d})$ to 2-layer FFN:



Synaptic input $\boldsymbol{z}$ to hidden layer:

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$$

Output $\boldsymbol{h}$ of hidden layer:

$$\boldsymbol{h} = g(\boldsymbol{z})$$

g is hidden layer activation function.

Synaptic input $\boldsymbol{u}$ to output layer:

$$\boldsymbol{u} = \boldsymbol{V}^T \boldsymbol{h} + \boldsymbol{c}$$

Output $\boldsymbol{y}$ of output layer:

$$\boldsymbol{y} = f(\boldsymbol{u})$$

Derivatives

$$\boldsymbol{u} = \boldsymbol{V}^T \boldsymbol{h} + \boldsymbol{c}$$

Consider synaptic input to u_k to the kth neuron at the output layer. Let weight vector $\boldsymbol{v}_k = (v_{k1} \quad v_{k2} \quad \cdots \quad v_{kM})^T$ and bias c_k.

The synaptic input u_k due to $\boldsymbol{h}$ is given by

$$u_k = \boldsymbol{v}_k{}^T \boldsymbol{h} + c_k = v_{k1} h_1 + v_{k2} h_2 + \cdots + v_{kM} h_M + c_k$$

$$\frac{\partial u_k}{\partial h_j} = v_{kj} \qquad for\ all\ j = 1,2, \ldots K$$

Therefore

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{h}} = \begin{pmatrix} \frac{\partial u_1}{\partial h_1} & \frac{\partial u_1}{\partial h_2} & \cdots & \frac{\partial u_1}{\partial h_M} \\ \frac{\partial u_2}{\partial h_1} & \frac{\partial u_2}{\partial h_2} & \cdots & \frac{\partial u_2}{\partial h_M} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial u_K}{\partial h_1} & \frac{\partial u_K}{\partial h_2} & \cdots & \frac{\partial u_K}{\partial h_K} \end{pmatrix} = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1M} \\ v_{21} & v_{22} & \cdots & v_{2M} \\ \vdots & \vdots & \vdots & \vdots \\ v_{K1} & v_{K2} & \cdots & v_{KM} \end{pmatrix} = \boldsymbol{V}^T$$

That is,

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{h}} = \boldsymbol{V}^T \qquad \text{(A)}$$

Derivatives

$$\boldsymbol{y} = f(\boldsymbol{u})$$

Considering kth neuron:

$$y_k = f(u_k)$$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} = \begin{pmatrix} \frac{\partial y_1}{\partial u_1} & \frac{\partial y_1}{\partial u_2} & \cdots & \frac{\partial y_1}{\partial u_K} \\ \frac{\partial y_2}{\partial u_1} & \frac{\partial y_2}{\partial u_2} & \cdots & \frac{\partial y_2}{\partial u_K} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial y_K}{\partial u_1} & \frac{\partial y_K}{\partial u_2} & \cdots & \frac{\partial y_K}{\partial u_K} \end{pmatrix} = \begin{pmatrix} f'(u_1) & 0 & \cdots & 0 \\ 0 & f'(u_2) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & f'(u_K) \end{pmatrix} = diag\big(f'(\boldsymbol{u})\big)$$

That is,

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} = diag\big(f'(\boldsymbol{u})\big) \qquad \text{(B)}$$

where $diag\big(f'(\boldsymbol{u})\big)$ is a diagonal matrix composed of derivatives corresponding to individual components of $\boldsymbol{u}$ in the diagonal.

Proof

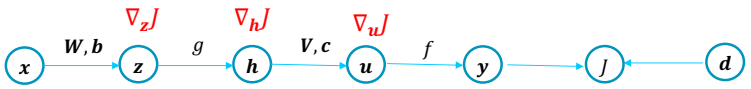
For a vector $\boldsymbol{x}$:

$$diag\big(f'(\boldsymbol{u})\big)\boldsymbol{x} = \begin{pmatrix} f'(u_1) & 0 & \cdots & 0 \\ 0 & f'(u_2) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & f'(u_K) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix} = \begin{pmatrix} f'(u_1)x_1 \\ f'(u_2)x_2 \\ \vdots \\ f'(u_K)x_K \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{pmatrix} \cdot \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_K) \end{pmatrix}$$

That is:

$$diag\big(f'(\boldsymbol{u})\big)\boldsymbol{x} = \boldsymbol{x} \cdot f'(\boldsymbol{u}) = f'(\boldsymbol{u}) \cdot \boldsymbol{x}$$

Back-propagation of gradients: single pattern



Considering output layer,

$$\nabla_{\boldsymbol{u}} J = \begin{cases} -(\boldsymbol{d} - \boldsymbol{y}) & \text{for a linear layer} \\ -\big(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\big) & \text{for a softmax layer} \end{cases}$$

From chain rule of differentiation,

$$\nabla_{\boldsymbol{h}} J = \left(\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{h}}\right)^T \nabla_{\boldsymbol{u}} J = \boldsymbol{V}\nabla_{\boldsymbol{u}} J \qquad \text{From (A)}$$

$$\nabla_{\boldsymbol{z}} J = \left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\right)^T \nabla_{\boldsymbol{h}} J = diag\big(g'(\boldsymbol{z})\big)\boldsymbol{V}\nabla_{\boldsymbol{u}} J = \boldsymbol{V}\nabla_{\boldsymbol{u}} J \cdot g'(\boldsymbol{z})$$

(C), from (B)

Back-propagation of gradients: single input pattern

From (C);

$$\nabla_{\boldsymbol{z}} J = \boldsymbol{V}\nabla_{\boldsymbol{u}} J \cdot g'(\boldsymbol{z})$$

That is, the gradients at output layer are multiplied by $\boldsymbol{V}$ and back-propagated to hidden layer.

Note that hidden-layer activations are multiplied by $\boldsymbol{V}^T$ (Note $\boldsymbol{u} = \boldsymbol{V}^T\boldsymbol{h} + \boldsymbol{c}$) in forward propagation and in back-propagation, the gradients are multiplied by $\boldsymbol{V}$.

Flow of gradients propagated in backward direction, hence named **back-propagation** (backprop) algorithm.

SGD of two-layer FFN

Output layer:

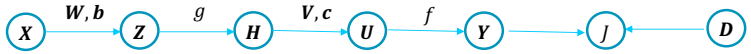
$$\nabla_{\boldsymbol{u}} J = \begin{cases} -(\boldsymbol{d} - \boldsymbol{y}) & \text{for linear layer} \\ -\left(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\right) & \text{for softmax layer} \end{cases}$$

Hidden layer:

$$\nabla_{\boldsymbol{z}} J = \boldsymbol{V} \nabla_{\boldsymbol{u}} J \cdot g'(\boldsymbol{z})$$

Forward propagation of activations: batch of inputs

Computational graph of 2-layer FFN for a batch of patterns $(\boldsymbol{X}, \boldsymbol{D})$:



Synaptic input $\boldsymbol{Z}$ to hidden layer:

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$

Output $\boldsymbol{H}$ of the hidden layer:

$$\boldsymbol{H} = g(\boldsymbol{Z})$$

Synaptic input $\boldsymbol{U}$ to output layer:

$$\boldsymbol{U} = \boldsymbol{H}\boldsymbol{V} + \boldsymbol{C}$$

Output $\boldsymbol{Y}$ of the output layer:

$$\boldsymbol{Y} = f(\boldsymbol{U})$$

SGD for a two-layer FFN

Given a training dataset $\{(\boldsymbol{x}, \boldsymbol{d})\}$
Set learning parameter α
Initialize $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}$
Repeat until convergence:
For every pattern $(\boldsymbol{x}, \boldsymbol{d})$:

$$\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{h} = g(\boldsymbol{z})$$
$$\boldsymbol{u} = \boldsymbol{V}^T \boldsymbol{h} + \boldsymbol{c}$$
$$\boldsymbol{y} = f(\boldsymbol{u})$$

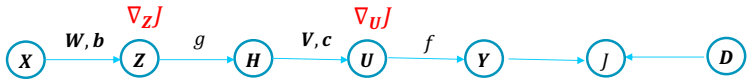
Forward propagation

$$\nabla_{\boldsymbol{u}} J = \begin{cases} -(\boldsymbol{d} - \boldsymbol{y}) \\ -\left(1(\boldsymbol{k} = d) - f(\boldsymbol{u})\right) \end{cases}$$
$$\nabla_{\boldsymbol{z}} J = \boldsymbol{V} \nabla_{\boldsymbol{u}} J \cdot g'(\boldsymbol{z})$$

Backward propagation

$$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \boldsymbol{h} (\nabla_{\boldsymbol{u}} J)^T$$
$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \nabla_{\boldsymbol{u}} J$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{x} (\nabla_{\boldsymbol{z}} J)^T$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{z}} J$$

Back-propagation of gradients: batch of patterns



$$\nabla_{\boldsymbol{U}} J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) & \text{for linear layer} \\ -\left(\boldsymbol{K} - f(\boldsymbol{U})\right) & \text{for softmax layer} \end{cases}$$

$$\nabla_{\boldsymbol{Z}} J = \begin{pmatrix} \left(\nabla_{\boldsymbol{z}_1} J\right)^T \\ \left(\nabla_{\boldsymbol{z}_2} J\right)^T \\ \vdots \\ \left(\nabla_{\boldsymbol{z}_P} J\right)^T \end{pmatrix} = \begin{pmatrix} \left(\boldsymbol{V} \nabla_{\boldsymbol{u}_1} J \cdot g'(\boldsymbol{z}_1)\right)^T \\ \left(\boldsymbol{V} \nabla_{\boldsymbol{u}_2} J \cdot g'(\boldsymbol{z}_2)\right)^T \\ \vdots \\ \left(\boldsymbol{V} \nabla_{\boldsymbol{u}_P} J \cdot g'(\boldsymbol{z}_P)\right)^T \end{pmatrix}$$

Substituting from (C)

Back-propagation of gradients: batch of patterns

$$\nabla_{\boldsymbol{Z}} J = \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J)^T \boldsymbol{V}^T \cdot (g'(\boldsymbol{z}_1))^T \\ (\nabla_{\boldsymbol{u}_2} J)^T \boldsymbol{V}^T \cdot (g'(\boldsymbol{z}_2))^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J)^T \boldsymbol{V}^T \cdot (g'(\boldsymbol{z}_P))^T \end{pmatrix} \qquad (XY)^T = Y^T X^T$$

$$= \begin{pmatrix} (\nabla_{\boldsymbol{u}_1} J)^T \\ (\nabla_{\boldsymbol{u}_2} J)^T \\ \vdots \\ (\nabla_{\boldsymbol{u}_P} J)^T \end{pmatrix} \boldsymbol{V}^T \cdot \begin{pmatrix} (g'(\boldsymbol{z}_1))^T \\ (g'(\boldsymbol{z}_2))^T \\ \vdots \\ (g'(\boldsymbol{z}_P))^T \end{pmatrix}$$

$$= (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$$

$$\nabla_{\boldsymbol{Z}} \boldsymbol{J} = (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$$

GD for a two-layer FFN

Given a training dataset $(\boldsymbol{X}, \boldsymbol{D})$
Set learning parameter α
Initialize $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}$
Repeat until convergence:

Forward propagation

$$\boldsymbol{Z} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$
$$\boldsymbol{H} = g(\boldsymbol{Z})$$
$$\boldsymbol{U} = \boldsymbol{H}\boldsymbol{V} + \boldsymbol{C}$$
$$\boldsymbol{Y} = f(\boldsymbol{U})$$

Backward propagation

$$\nabla_{\boldsymbol{U}} J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) \\ -(\boldsymbol{K} - f(\boldsymbol{U})) \end{cases}$$
$$\nabla_{\boldsymbol{Z}} J = (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$$

$$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \boldsymbol{H}^T \nabla_{\boldsymbol{U}} J$$
$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha (\nabla_{\boldsymbol{U}} J)^T \boldsymbol{1}_P$$
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_{\boldsymbol{Z}} J$$
$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha (\nabla_{\boldsymbol{Z}} J)^T \boldsymbol{1}_P$$

GD of two-layer FFN

Output layer:

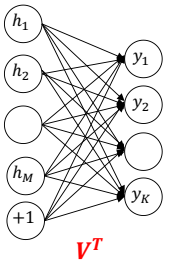
$$\nabla_{\boldsymbol{U}} J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) & \text{for linear layer} \\ -(\boldsymbol{K} - f(\boldsymbol{U})) & \text{for softmax layer} \end{cases}$$

Hidden layer:

$$\nabla_{\boldsymbol{Z}} J = (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z}) \qquad \text{(D)}$$

Back-propagation

$\boldsymbol{H}$ —$\boldsymbol{V}$→ $\boldsymbol{U} = \boldsymbol{H}\boldsymbol{V} + \boldsymbol{C}$



$\nabla_{\boldsymbol{Z}} \boldsymbol{J} = (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$ ←$\boldsymbol{V}^T$— $\nabla_{\boldsymbol{U}} J$

The error gradient can be seen as propagating from the output layer to the hidden layer and so learning in feedforward networks is known as the *back-propagation* algorithm

Learning in two-layer FFN

GD	SGD
$(\boldsymbol{X}, \boldsymbol{D})$	$(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{Z} = \boldsymbol{XW} + \boldsymbol{B}$	$\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
$\boldsymbol{H} = g(\boldsymbol{Z})$	$\boldsymbol{h} = g(\boldsymbol{z})$
$\boldsymbol{U} = \boldsymbol{HV} + \boldsymbol{C}$	$\boldsymbol{u} = \boldsymbol{V}^T\boldsymbol{h} + \boldsymbol{c}$
$\boldsymbol{Y} = f(\boldsymbol{U})$	$\boldsymbol{y} = f(\boldsymbol{u})$
$\nabla_U J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) \\ -(\boldsymbol{K} - f(\boldsymbol{U})) \end{cases}$	$\nabla_u J = \begin{cases} -(\boldsymbol{d} - \boldsymbol{y}) \\ (1(\boldsymbol{k} = d) - f(\boldsymbol{u})) \end{cases}$
$\nabla_Z J = (\nabla_U J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$	$\nabla_z J = \boldsymbol{V}\nabla_u J \cdot g'(\boldsymbol{z})$
$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_Z J$	$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{x}(\nabla_z J)^T$
$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha(\nabla_Z J)^T \boldsymbol{1}_P$	$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_z J$
$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \boldsymbol{H}^T \nabla_U J$	$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \boldsymbol{h}(\nabla_u J)^T$
$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha(\nabla_U J)^T \boldsymbol{1}_P$	$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \nabla_u J$

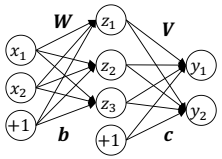
Example 2: Two-layer FFN

Design a two-layer FFN, using gradient descent to perform the following mapping. Use a learning factor = 0.05 and three perceptrons in the hidden-layer.

Inputs $\boldsymbol{x} = (x_1, x_2)$	Targets $\boldsymbol{d} = (d_1, d_2)$
$(0.77, 0.02)$	$(0.44, -0.42)$
$(0.63, 0.75)$	$(0.84, 0.43)$
$(0.50, 0.22)$	$(0.09, -0.72)$
$(0.20, 0.76)$	$(-0.25, 0.35)$
$(0.17, 0.09)$	$(-0.12\ -0.13)$
$(0.69, 0.95)$	$(0.24, 0.03)$
$(0.00\ , 0.51)$	$(0.30, 0.20)$
$(0.81, 0.61)$	$(0.61, 0.04)$

Example 2

$$\boldsymbol{X} = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \text{ and } \boldsymbol{D} = \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix}$$



Output layer is a linear neuron layer

Hidden layer is a sigmoidal layer

Initialized (weights using a uniform distribution):

$$\boldsymbol{W} = \begin{pmatrix} -3.97 & 1.10 & 0.42 \\ 2.79 & -2.64 & 3.13 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}, \boldsymbol{V} = \begin{pmatrix} 3.58 & -1.58 \\ -3.58 & -1.75 \\ -3.38 & 2.88 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$$

Example 2

Epoch 1:

$$\boldsymbol{Z} = \boldsymbol{XW} + \boldsymbol{B} = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.20 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \begin{pmatrix} -3.97 & 1.10 & 0.42 \\ 2.79 & -2.64 & 3.13 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -3.00 & 0.8 & 0.39 \\ -0.42 & -1.27 & 2.61 \\ -1.35 & -0.04 & 0.91 \\ 1.34 & -1.79 & 2.46 \\ -0.42 & -0.05 & 0.35 \\ -0.05 & -1.76 & 3.27 \\ 1.42 & -1.34 & 1.60 \\ -1.51 & -0.72 & 2.25 \end{pmatrix}$$

$$\boldsymbol{H} = g(\boldsymbol{Z}) = \frac{1}{1 + e^{-\boldsymbol{z}}} = \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.40 & 0.49 & 0.59 \\ 0.49 & 0.15 & 0.96 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix}$$

Example 2

$$\boldsymbol{Y} = \boldsymbol{H}\boldsymbol{V} + \boldsymbol{C} = \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.40 & 0.49 & 0.59 \\ 0.49 & 0.15 & 0.96 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix} \begin{pmatrix} 3.58 & -1.18 \\ -3.58 & -1.78 \\ -3.38 & 2.88 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \\ 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -4.32 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix}$$

$$\nabla_{\boldsymbol{U}} J = -(\boldsymbol{D} - \boldsymbol{Y}) = -\left(\begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix} - \begin{pmatrix} -4.32 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix} \right) = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix}$$

Example 2

Output layer:

$$\nabla_{\boldsymbol{V}} J = \boldsymbol{H}^T \nabla_{\boldsymbol{U}} J = \begin{pmatrix} -6.23 & 3.22 \\ -8.81 & 2.85 \\ -17.07 & 7.40 \end{pmatrix}$$

$$\nabla_{\boldsymbol{c}} J = (\nabla_{\boldsymbol{U}} J)^T \mathbf{1}_P = \begin{pmatrix} -21.83 \\ 8.81 \end{pmatrix}$$

Hidden layer:

$$\nabla_{\boldsymbol{W}} J = \boldsymbol{X}^T \nabla_{\boldsymbol{Z}} J = \begin{pmatrix} -8.43 & 7.81 & 7.79 \\ -8.21 & 4.56 & 3.76 \end{pmatrix}$$

$$\nabla_{\boldsymbol{b}} J = (\nabla_{\boldsymbol{Z}} J)^T \mathbf{1}_P = \begin{pmatrix} -15.22 \\ 13.11 \\ 13.92 \end{pmatrix}$$

Example 2

$$g'(\boldsymbol{Z}) = \boldsymbol{H} \cdot (\mathbf{1} - \boldsymbol{H}) = \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

$$\nabla_{\boldsymbol{Z}} J = (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$$

$$= \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 1.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix} \begin{pmatrix} 3.58 & -1.58 \\ -3.58 & -1.75 \\ -3.38 & 2.88 \end{pmatrix}^T \cdot \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

Example 2

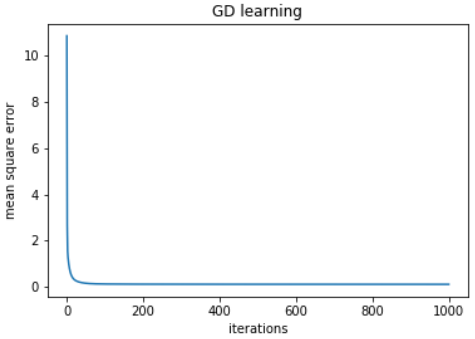
$$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \nabla_{\boldsymbol{V}} J = \begin{pmatrix} 3.89 & -1.74 \\ -3.14 & -1.89 \\ -2.53 & 2.51 \end{pmatrix}$$

$$\boldsymbol{c} \leftarrow \boldsymbol{c} - \alpha \nabla_{\boldsymbol{c}} J = \begin{pmatrix} 1.09 \\ -0.44 \end{pmatrix}$$

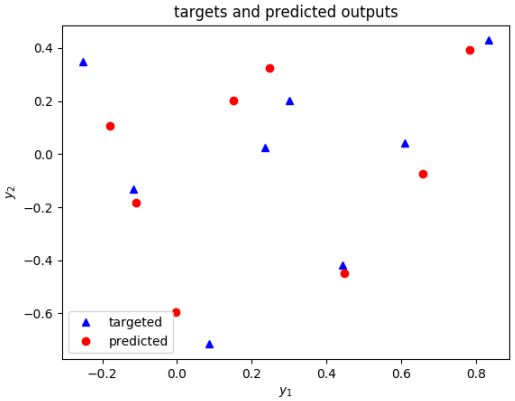
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J = \begin{pmatrix} -3.55 & 0.72 & 0.03 \\ 3.21 & -2.87 & 2.94 \end{pmatrix}$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \nabla_{\boldsymbol{b}} J = \begin{pmatrix} 0.76 \\ -0.66 \\ -0.70 \end{pmatrix}$$

Example 2



Example 2



After 20,000 iterations

Example 2

After 1,000 epochs,

m. s. e = 0.107

$$\boldsymbol{W} = \begin{pmatrix} -2.04 & -0.52 & -1.88 \\ 3.55 & -2.6 & 2.43 \end{pmatrix}$$

$$\boldsymbol{b} = \begin{pmatrix} 0.22 \\ -0.93 \\ 0.15 \end{pmatrix}$$

$$\boldsymbol{V} = \begin{pmatrix} 2.14 & -1.14 \\ -2.93 & -1.78 \\ 3.59 & 2.28 \end{pmatrix}$$

$$\boldsymbol{c} = \begin{pmatrix} 1.25 \\ -0.43 \end{pmatrix}$$

Preprocessing of inputs

If inputs have similar variations, better approximation of inputs or prediction of outputs is achieved. Mainly, there are two approaches to normalization of inputs.

Suppose ith input $x_i \in [x_{i,min}, x_{i,max}]$ and has a mean μ_i and a standard deviation σ_i.

If $\tilde{x}_i$ denotes the normalized input.

1. **Scaling** the inputs such that $\tilde{x}_i \in [0,1]$:

$$\tilde{x}_i = \frac{x_i - x_{i,min}}{x_{i,max} - x_{i,min}}$$

2. **Normalizing** the input to have standard normal distributions $\tilde{x}_i \sim N(0,1)$:

$$\tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

Preprocessing of Outputs

Linear activation function:
The convergence is usually improved if each output is **normalized** to have zero mean and unit standard deviation: $\tilde{y}_k \sim N(0,1)$

$$\tilde{y}_k = \frac{y_k - \mu_k}{\sigma_k}$$

Sigmoid activation function:
Since sigmoidal activation range from 0 to 1.0, you can **scale** $\tilde{y}_k \in [0,1]$:

$$\tilde{y}_k = \frac{y_k - y_{k,min}}{y_{k,max} - y_{k,min}}$$

$$= \frac{1}{y_{k,max} - y_{k,min}} y_k - \frac{y_{k,min}}{y_{k,max} - y_{k,min}}$$

Example 3: Two-layer FFN predicting housing prices in California

Thirteen input variables, One output variable

We use FFN with one hidden layer with 10 neuron
Network size: [8, 10, 1]

```
class FFN(nn.Module):
  def __init__(self, no_features, no_hidden, no_labels):
    super().__init__()
    self.relu_stack = nn.Sequential(
      nn.Linear(no_features, no_hidden),
      nn.ReLU(),
      nn.Linear(no_hidden, no_labels),
    )

  def forward(self, x):
    logits = self.relu_stack(x)
    return logits
```

California housing dataset

https://developers.google.com/machine-learning/crash-course/california-housing-data-description

9 variables

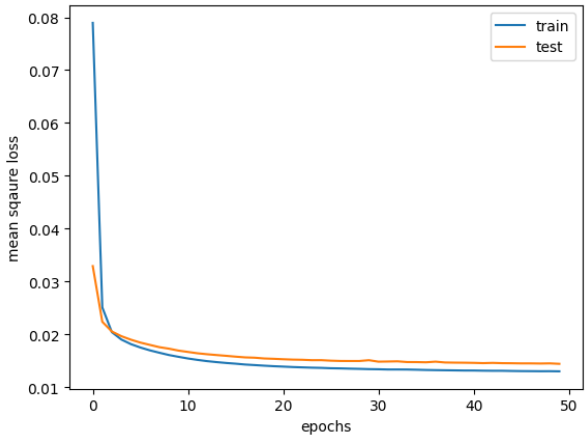
The problem is to predict the housing prices using the other 8 variables.

20540 samples

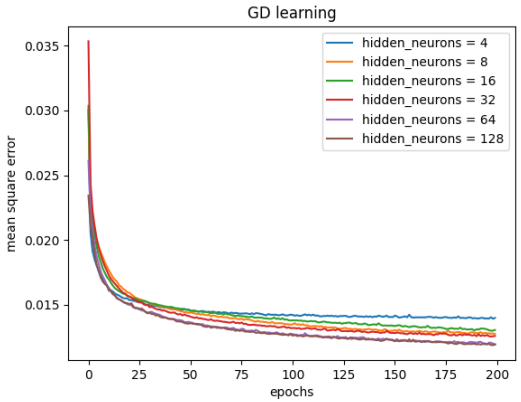
Train: 14448 samples
Test: 6192 samples

longitude	A measure of how far west a house is; a more negative value is farther west
latitude	A measure of how far north a house is; a higher value is farther north
housingMedianAge	Median age of a house within a block; a lower number is a newer building
totalRooms	Total number of rooms within a block
totalBedrooms	Total number of bedrooms within a block
population	Total number of people residing within a block
households	Total number of households, a group of people residing within a home unit, for a block
medianIncome	Median income for households within a block of houses (measured in tens of thousands of US Dollars)
medianHouseValue	Median house value for households within a block (measured in US Dollars)

Example 3a



Example 3b: no of hidden neurons

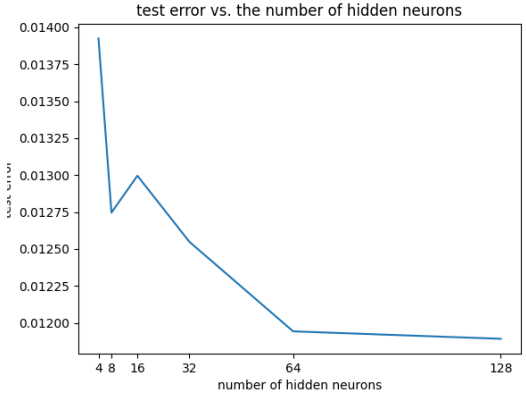


Width of hidden Layers

The number of parameters of the network increases with the **width** of layers. Therefore, the network attempts to remember the training patterns with increasing number of parameters. In other words, the network aims at minimizing the training error at the expense of its generalization ability on unseen data.

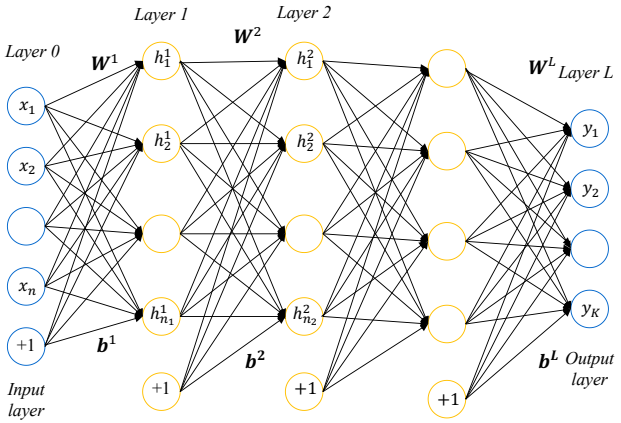
As the number of hidden units increases, the test error decreases initially but tends to increase at some point. The optimal number of hidden units is often determined empirically (that is, by trial and error).

Example 3b: width of the hidden layer



Optimum number of hidden neurons 64

Deep neural networks (DNN)



Depth = L

DNN notations

Input layer $l = 0$:
Width = n
Input $\boldsymbol{x}, \boldsymbol{X}$

Hidden layers $l = 1, 2, \cdots L-1$
Width: n_l
Weight matrix $\boldsymbol{W}^l$, bias vector $\boldsymbol{b}^l$
Synaptic input $\boldsymbol{u}^l, \boldsymbol{U}^l$
Activation function f^l
Output $\boldsymbol{h}^l, \boldsymbol{H}^l$

Output layer $l = L$
Width: K
Synaptic input $\boldsymbol{u}^L, \boldsymbol{U}^L$
Activation function f^L
Output $\boldsymbol{y}, \boldsymbol{Y}$
Desired output $\boldsymbol{d}, \boldsymbol{D}$

Back-propagation of gradients in DNN: single pattern

if $l = L$:

$$\nabla_{\boldsymbol{u}^l} J = \begin{cases} -(\boldsymbol{d} - \boldsymbol{y}) & \text{for linear layer} \\ -\left(1(\boldsymbol{k} = d) - f^L(\boldsymbol{u}^L)\right) & \text{for softmax layer} \end{cases}$$

else:

$$\nabla_{\boldsymbol{u}^l} J = \boldsymbol{W}^{l+1}\left(\nabla_{\boldsymbol{u}^{l+1}} J\right) \cdot f^{l\,\prime}\left(\boldsymbol{u}^l\right) \qquad \text{from (C)}$$

$$\nabla_{\boldsymbol{W}^l} J = \boldsymbol{h}^{l-1}\left(\nabla_{\boldsymbol{u}^l} J\right)^T$$
$$\nabla_{\boldsymbol{b}^l} J = \nabla_{\boldsymbol{u}^l} J$$

Gradients are backpropagated from the output layer to the input layer

Forward propagation of activation in DNN: single pattern

Input $(\boldsymbol{x}, \boldsymbol{d})$
$\boldsymbol{u}^1 = \boldsymbol{W}^{1\,T}\boldsymbol{x} + \boldsymbol{b}^1$
For layers $l = 1, 2, \cdots, L-1$:
$\quad \boldsymbol{h}^l = f^l(\boldsymbol{u}^l)$
$\quad \boldsymbol{u}^{l+1} = \boldsymbol{W}^{l+1\,T}\boldsymbol{h}^l + \boldsymbol{b}^{l+1}$
$\boldsymbol{y} = f^L(\boldsymbol{u}^L)$

Forward propagation of activation in DNN: batch of patterns

Input $(\boldsymbol{X}, \boldsymbol{D})$
$\boldsymbol{U}^1 = \boldsymbol{X}\boldsymbol{W}^1 + \boldsymbol{B}^1$
For layers $l = 1, 2, \cdots, L-1$:
$\quad \boldsymbol{H}^l = f^l(\boldsymbol{U}^l)$
$\quad \boldsymbol{U}^{l+1} = \boldsymbol{H}^l\boldsymbol{W}^{l+1} + \boldsymbol{B}^{l+1}$
$\boldsymbol{Y} = f^L(\boldsymbol{U}^L)$

Back-propagation of gradients in DNN: batch of patterns

If $l = L$:

$$\nabla_{\boldsymbol{U}^l} J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) \\ -\left(\boldsymbol{K} - f^L(\boldsymbol{U}^L)\right) \end{cases}$$

Else:

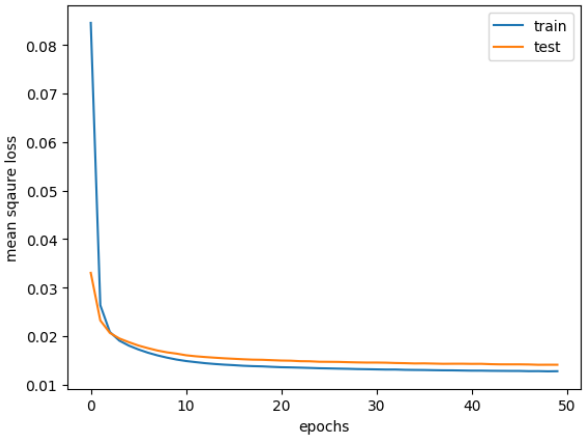
$$\nabla_{\boldsymbol{U}^l} J = \left(\nabla_{\boldsymbol{U}^{l+1}} J\right) \boldsymbol{W}^{l+1^T} \cdot f^{l'}(\boldsymbol{U}^l) \qquad \text{from (D)}$$

$$\nabla_{\boldsymbol{W}^l} J = \boldsymbol{H}^{l-1^T} \left(\nabla_{\boldsymbol{U}^l} J\right)$$

$$\nabla_{\boldsymbol{b}^l} J = \left(\nabla_{\boldsymbol{U}^l} J\right)^T \mathbf{1}_P$$

Gradients are backpropagated from the output layer to the input layer

Example 4



Example 4: DNN on California Housing data

California housing data:

https://developers.google.com/machine-learning/crash-course/california-housing-data-description

Predicting housing price from other 8 variables

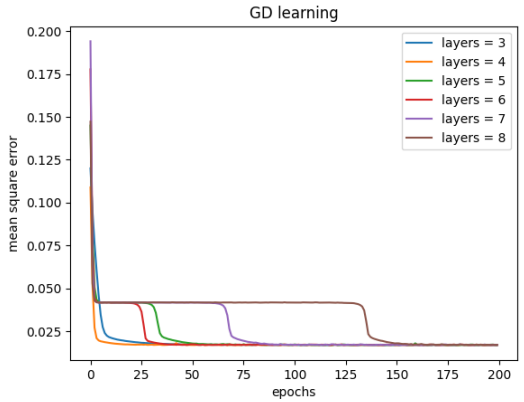
DNN with two hidden layers:
[8, 10, 5, 1]

```
relu_stack = nn.Sequential(
        nn.Linear(no_features, no_hidden1),
        nn.ReLU(),
        nn.Linear(no_hidden1, no_hidden2),
        nn.ReLU(),
        nn.Linear(no_hidden2, no_labels),
    )
```

Example 4b: Varying the depth of DNN

Architectures:
One hidden layer: [8, 5, 1]
Two hidden layers: [8, 5, 5, 1]
Three hidden layers: [8, 5, 5, 5, 1]
Four hidden layers: [8, 5, 5, 5, 5, 1]
Five hidden layers: [8, 5, 5, 5, 5, 5, 1]

Example 4

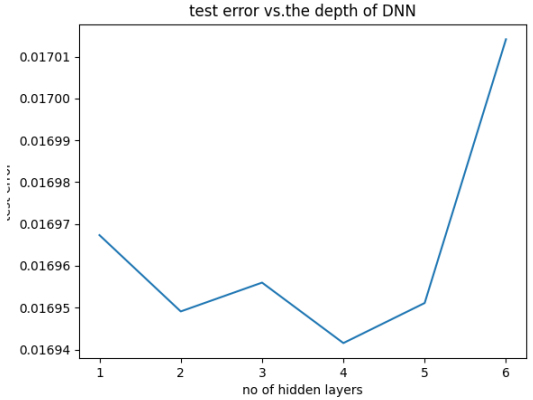


Depth of DNN

The deep networks extract features at different levels of complexity for regression or classification. However, the **depth** or the number of layers that you can have for the networks depend on the number of training patterns available. The deep networks have more parameters (weights and biases) to learn, so need more data to train. Deep networks can learn complex mapping accurately if sufficient training data is available.

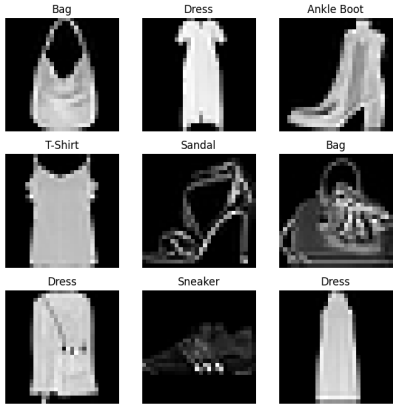
The optimal number of layers is determined usually through experiments. The optimal architecture minimizes the error (training, test, and validation).

Example 4



Optimum number of hidden layers = 4

Fashion MNIST dataset

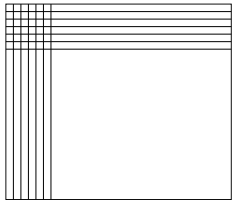


IThe Fasion-MNIST database of gray level images of fashion items from 10 classes:
Ihttps://github.com/zalandoresearch/fashion-mnist

Each image is 28x28 size.
Intensities are in the range [0, 255].

Training set: 60,000
Test set: 10,000

MNIST images



An image is divided into rows and columns and defined by its pixels.

Size of the image = rows x columns pixels

Pixels of **grey-level image** are assigned intensity values: For example, integer values between 0 and 255 assigned as intensities (grey-values) for pixels with 0 representing 'black' and 255 representing 'white'.

Color images has three color channels: red, green, and blue. A pixel in a color image is a vector (r, g, b) denoting intensity in red, green, and blue channels.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.softmax_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.softmax_relu_stack(x)
        return logits

model = NeuralNetwork()
```


Example 5: Classification of Fashion-MNIST images

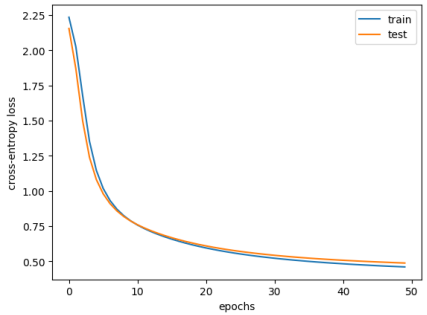
No of inputs $n = 28 \times 28 = 784$ (after flattening)
Inputs were normalized to [0.0, 1.0]

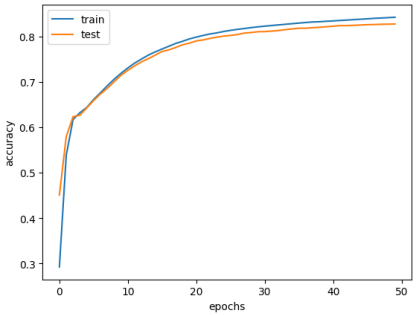
Use a 3-layer FFN
- Hidden-layer-1 is a perceptron layer
- Hidden-layer-2 is perceptron layer
- Output-layer is a softmax layer

Input-layer size $n = 784$
Hidden-layer-1 size $n_1 = 512$
Hidden-layer-2 size $n_2 = 512$
Output-layer size $K = 10$

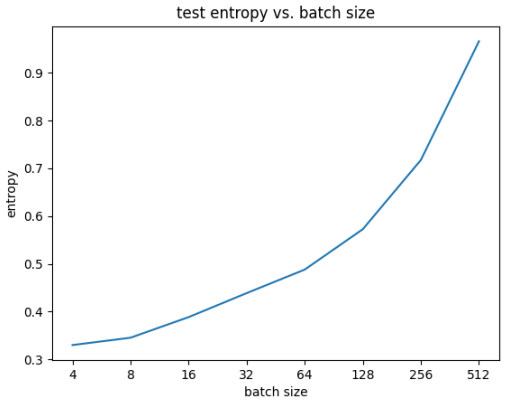
Training:
Batch size = 64
Learning rate $\alpha = 0.001$

Example 5

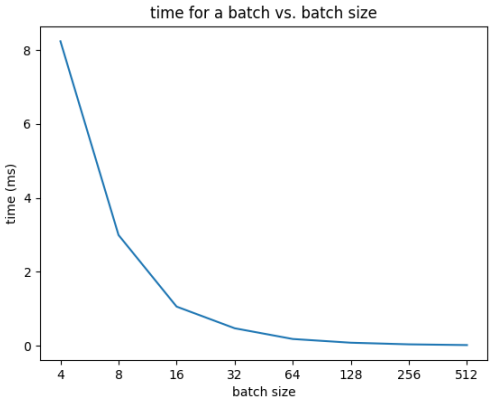




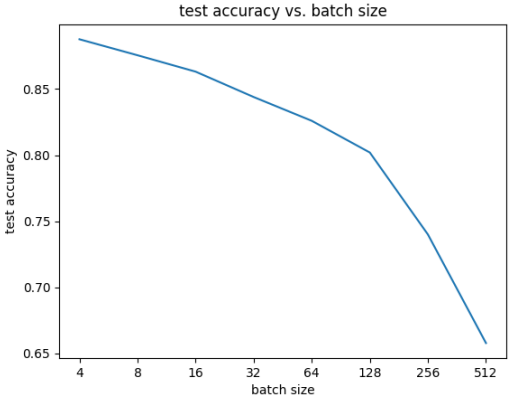
Example 5b: Effect of batch size



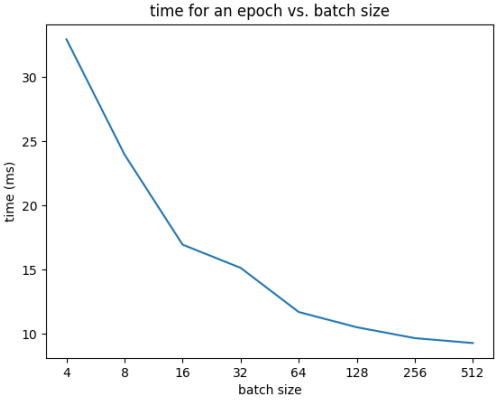
Example 5: Effect of batch size



Example 5: Effect of batch size



Example 5: Effect of batch size



Mini-batch SGD

In practice, gradient descent is performed on *mini-batch* updates of gradients within a *batch* or *block* of data of size B. In mini-batch SGD, the data is divided into blocks and the gradients are evaluated on blocks in an epoch in random order.

$B = 1$: stochastic (online) gradient descent
$B = P$ (size of training data): (batch) gradient descent
$1 < B < P$ → mini-batch stochastic gradient descent

When B increases, more add-multiply operations per second, taking advantages of parallelism and matrix computations. On the other hand, as B increases, the number of computations per update (of weights, biases) increases.

Therefore, the curve of the time for weight update against batch size usually take a U-shape curve. There exists an optimal value of B – that depends on the sizes of the caches as well.

Selection of batch size

For SGD, it is desirable to randomly sample the patterns from training data in each epoch. In order to efficiently sample blocks, the training patterns are shuffled at the beginning of every training epoch and then blocks are sequentially fetched from memory.

Typical batch sizes: 16, 32, 64, 128, and 256.

The batch size is dependent on the size of caches of CPU and GPUs.

Summary

- Chain rule for backpropagation of gradients: $\nabla_x J = \left(\frac{\partial y}{\partial x}\right)^T \nabla_y J$
- FFN with one hidden layer (Shallow FFN)
- Backpropagation for FFN with one hidden layer:

$$\nabla_Z J = (\nabla_U J) V^T \cdot g'(Z)$$

- Backpropagation learning for deep FFN (DNN)
- Training deep neural networks (GD and SGD):
  - Forward propagation of activation
  - Backpropagation of gradients
  - Updating weights
- Parameters to be decided: depth, width, and batch size

Chapter 5

Model selection and overfitting

Neural networks and deep learning

Model Selection

In neural networks, there exist several free parameters: learning rate, batch size, no of layers, number of neurons, etc.

Every set of parameters of the network leads to a specific model.

How do we determine the "optimum" parameter(s) or the model for a given regression or classification problem?

Selecting the best model with the best parameter values

Performance estimates

How do we measure the performance of the network?

Some metrics:

1. Mean-square error/Root-mean square error for **regression** — the mean-squared error or its square root. A measure of the deviation from actual.

$$MSE = \frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(d_{pk} - y_{pk}\right)^2 \text{ and } RMSE = \sqrt{\frac{1}{P}\sum_{p=1}^{P}\sum_{k=1}^{K}\left(d_{pk} - y_{pk}\right)^2}$$

2. Classification error of a **classifier**.

$$classification\ error = \sum_{p=1}^{P} 1\left(d_p \neq y_p\right)$$

where d_p is the target and y_p is the predicted output of pattern p. $1(\cdot)$ is the indicator function.

True error or apparent error?

Apparent error (training error): the error on the training data. What the learning algorithm tries to optimize.

True error: the error that will be obtained in use (i.e., over the whole *sample space*). What we want to optimize *but* unknown.

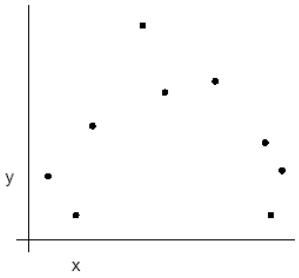
However, the **apparent error** is not always a good estimate of the **true error**. It is just an optimistic measure of it.

Test error: (out-of-sample error) an estimate of the true error obtained by testing the network on some independent data.

Generally, a larger test set helps provide a greater confidence on the accuracy of the estimate.

Example: Regression

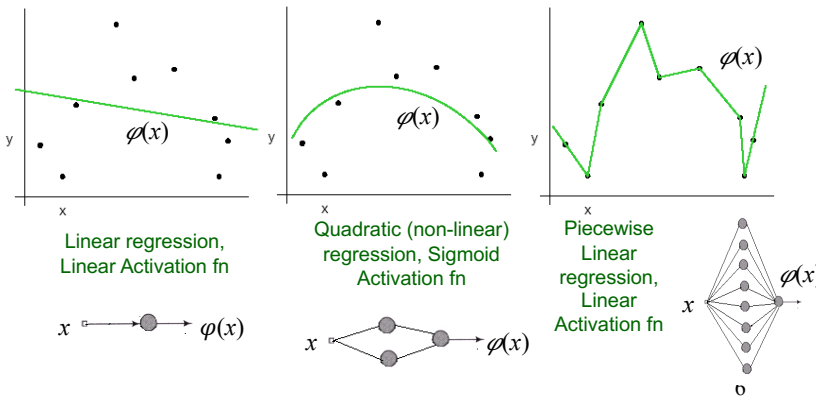
Given the following sample data:



Approximate $y \approx \varphi(x)$ using an NN

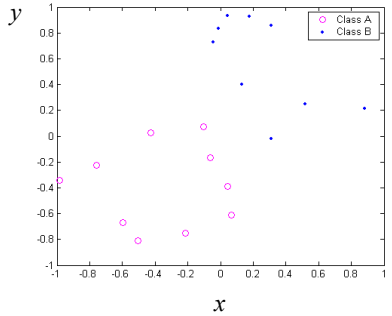
Example: Regression

Which one is the best approximation model?



Example: Classification

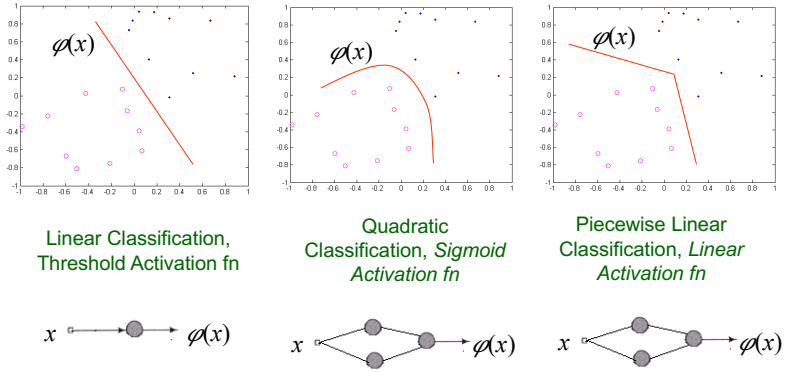
Given the following sample data:



Classify the following data using an NN.

Example: Classification

Which one is the best classification model?



Estimation of true error

Intuition: Choose the model with the best fit to the data?

Meaning: Choose the model that provides the lowest error rate on the entire sample population. Of course, that error rate is the *true error rate*.

"However, to choose a model, we must first know how to **estimate the error** of a model."

The entire **sample population** is often unavailable and only **example data** is available.

Validation

In real applications, we only have access to a finite set of examples, usually smaller than we wanted.

Validation is the approach to use the entire example data available to build the model and estimate the error rate. The validation uses a part of the data to select the model, which is known as the *validation set.*

Validation attempts to solve fundamental problems encountered:

- The model tends to *overfit* the training data. It is not uncommon to have 100% correct classification on training data.
- There is no way of knowing how well the model performs on *unseen data*
- The *error rate estimate* will be overly optimistic (usually lower than the true error rate) . Need to get an unbiased estimate.

Validation Methods

An effective approach is to split the entire data into subsets, i.e., Training/Validation/Testing datasets.

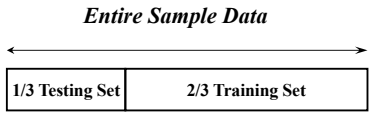
Some Validation Methods:

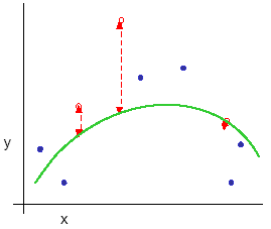
- The Holdout (1/3 - 2/3 rule for test and train partitions)
- Re-sampling techniques
  - Random Subsampling
  - K-fold Cross-Validation
  - Leave one out Cross-Validation
- Three-way data splits (train-validation-test partitions)

Holdout Method

Split entire dataset into two sets:

- **Training set** (blue): used to train the classifier
- **Testing set** (red): used to estimate the error rate of the trained classifier on unseen data samples

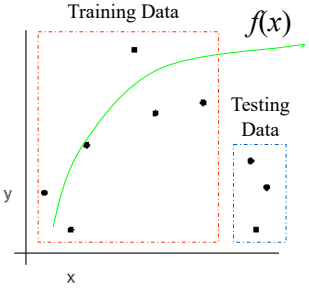




Holdout Method

The holdout method has two basic drawbacks:

- By setting some samples for testing, the training dataset becomes smaller
- Use of a single train-and-test experiment, could lead to misleading estimate if an "unfortunate" split happens



An "unfortunate" split: training data may not cover the space of testing data

Random Sampling Methods

Limitations of the holdout can be overcome with a family of resampling methods at the expense of more computations:

- Random Subsampling
- K-Fold Cross-Validation
- Leave-one-out (LOO) Cross-Validation

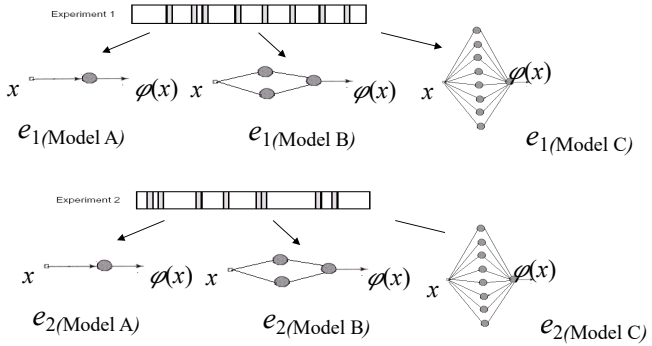
K Data Splits Random SubSampling

Random Subsampling performs *K* data splits of the dataset for training and testing.

Each split randomly selects a (fixed) no. of examples. For each data split we retrain the classifier from scratch with the training data. Let the error estimate obtained for *i*th split (experiments) be e_i.

$$Average\ test\ error = \frac{1}{K}\sum_{i=1}^{K} e_i$$

Example: K=2 Data Splits Random SubSampling



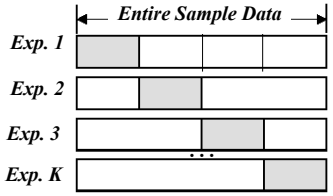
$$Test\ error_{(M)} = \frac{1}{2}(e_{1(M)} + e_{2(M)})$$

Choose the model with the best test error, i.e., lowest average test error!

K-fold Cross-Validation

Create a *K*-fold partition of the the dataset:

• For each of *K* experiments, use *K*-1 folds for training and the remaining one-fold for testing

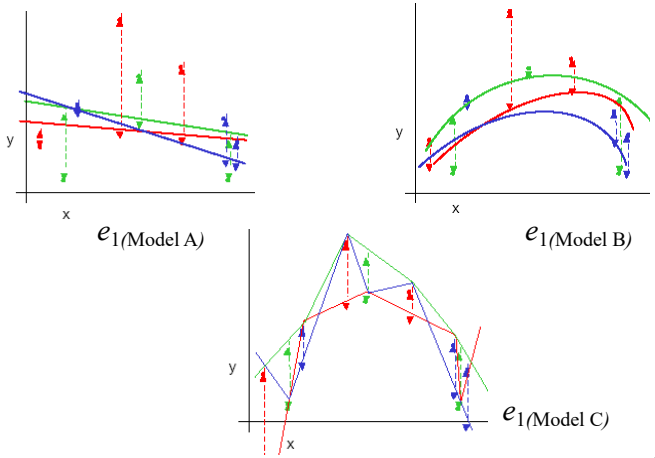


Let the error estimate for *i*th experiment on test partition be e_i.

Cross-validation error $\frac{1}{K}\sum_{i=1}^{K} e_i$

K-fold cross validation is similar to Random Subsampling. The *advantage* of *K*-Fold Cross validation is that all examples in the dataset are eventually used for both training and testing

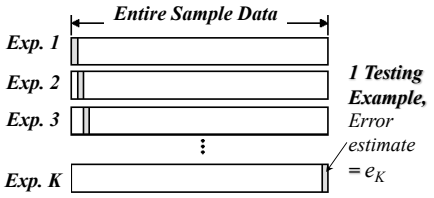
Example: 3-fold Cross-Validation



Leave-One-Out (LOO) Cross-Validation

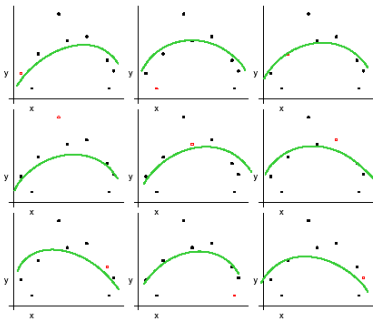
Leave-One-Out is the degenerate case of *K*-Fold Cross Validation, where *K* is chosen as the total number of examples:

- For a dataset with N examples, perform N experiments, i.e., $N = K$.
- For each experiment use N-1 examples for training and the remaining one example for testing.

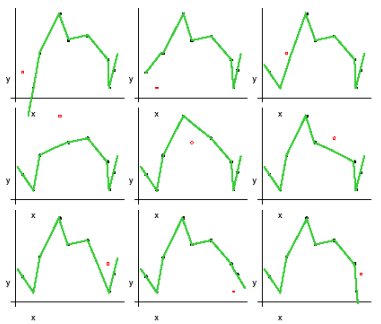


LOO CV error $\frac{1}{K}\sum_{i=1}^{K} e_i$

Example: Leave-One-Out Cross-Validation



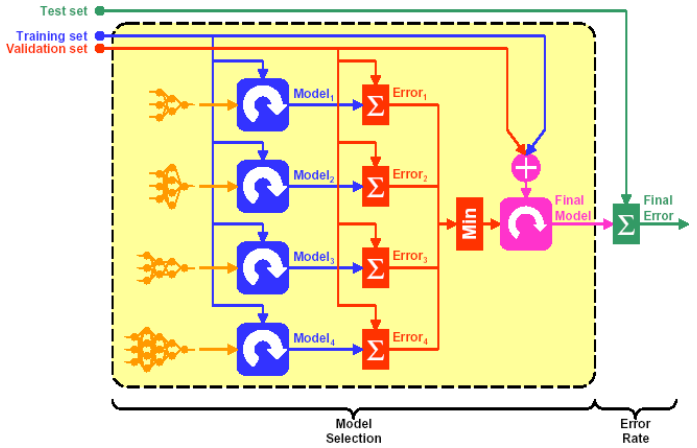
LOOCV Quadratic Approximation



LOOCV Piecewise Linear Approximation

Three-Way Data Splits Method

Dataset is partitioned in to training set, **validation set**, and testing set.



Three-Way Data Splits Method

If model selection and true error estimates are to be computed simultaneously, the data needs to be divided into three disjoint sets:

• **Training set**: examples for *learning* to fit the parameters of several possible classifiers. In the case of DNN, we would use the training set to find the "optimal" weights with the gradient descent rule.

• **Validation set**: examples to *determine* the error J_m of different models m, using the validation set. The optimal model m^* is given by

$$m^* = \underset{m}{\text{arg}min}\, J_m$$

• **Training + Validation set:** combine examples used to re-train/redesign $model_{m^*}$ and find new "optimal" weights and biases.

• **Test set**: examples used only to *assess* the performance of a *trained model* m^*. We will use the test data to estimate the error rate after we have trained the final model with train + validation data.

Three-Way Data Splits Method

Why separate test and validation sets?

• The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is also used to select in the process of final model selection.

• After assessing the final model, an independent test set is required to estimate the performance of the final model.

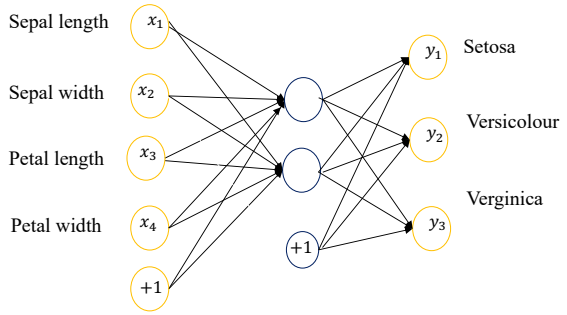
"NO FURTHERING TUNNING OF THE MODEL IS ALLOWED!"

Examples 1-3: Iris dataset

https://archive.ics.uci.edu/ml/datasets/Iris

Three classes of iris flower: Setosa, versicolour, and virginica
Four features: Sepal length, sepal width, petal length, petal width
150 data points

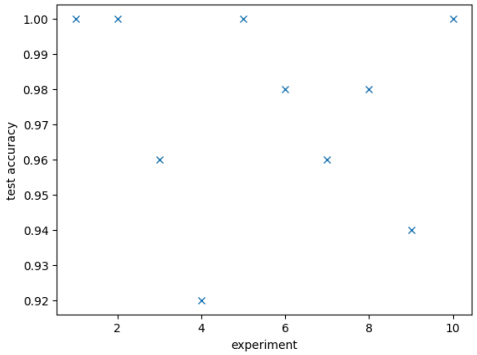
DNN with one hidden layer. **Determine number of hidden neurons?**



Example 1a: Random subsampling

150 data points
In each experiment, 50 points for testing and 100 for training

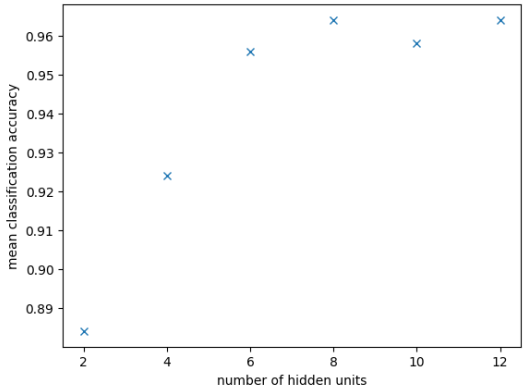
Example: 5 hidden neurons, 10 experiments



Mean accuracy = 97.4%

Example 1b

For different number of hidden units, misclassification errors in 10 experiments

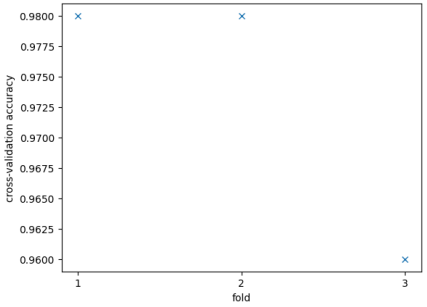


Optimum number of hidden units = 8
Accuracy = 96.4%

Example 2a: Cross validation

150 data points
3-fold cross validation: 50 data points in one fold.
Two folds are used for training and the remaining fold for testing

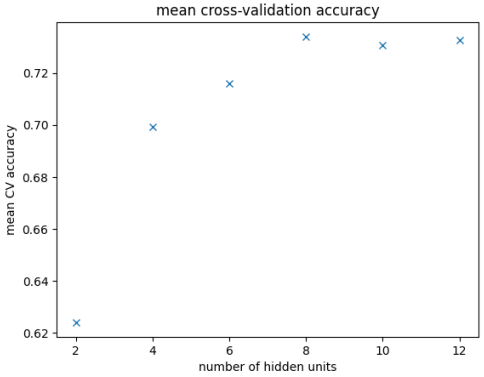
Example: hidden number of units = 5



3-fold cross-validation (CV) accuracy = 97.3%

Example 2b

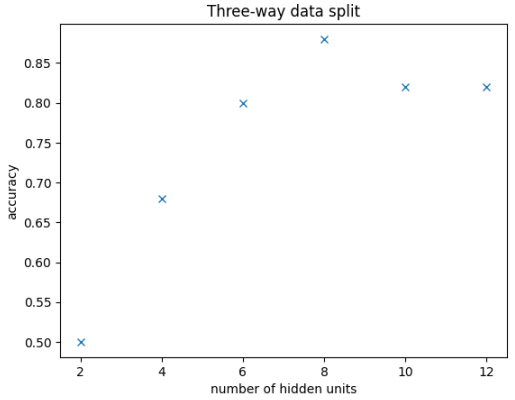
Mean CV error for 10 experiments for different number of hidden units:



Optimum number of hidden neurons = 8
Cross-validation accuracy = 73.4%

Example 3a: Three-way data splits

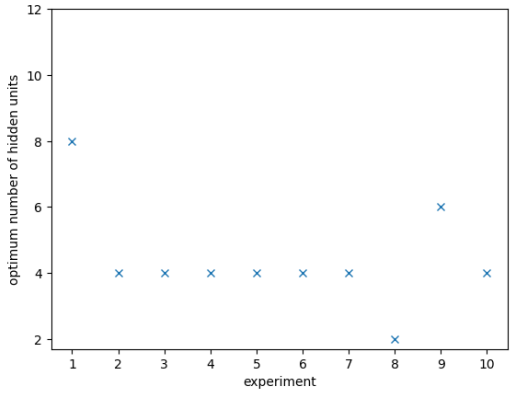
150 data points
50 data points each for training, for validation, and for testing.



Optimum number of hidden neurons = 8
Accuracy = 88.0%

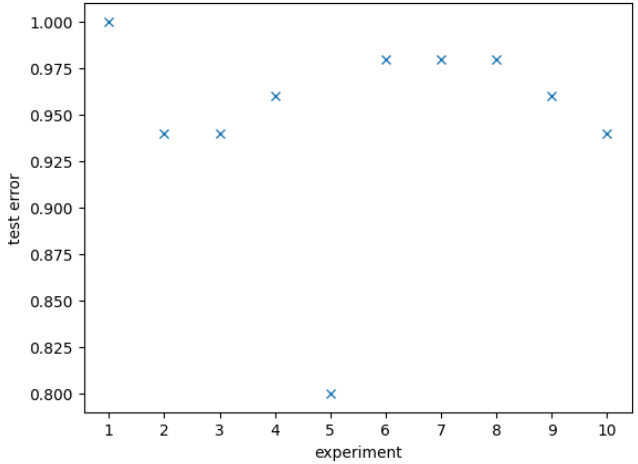
Example 3b

One way to further improve the results is to repeat it for many experiments, say for 10 experiments:



Optimum number of hidden neurons = 4

Example 3b



Test accuracy = 93.4% (average)

Model Complexity

Complex models: models with many adjustable weights and biases will

- more likely to be able to solve the required task,
- more likely to memorize the training data without solving the task.

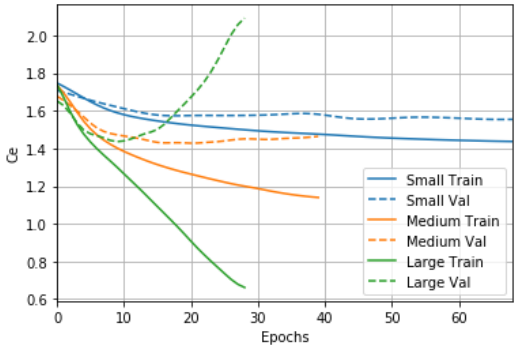
Simple models: The simpler the model that can learn from the training data is more likely to generalize over the entire sample space. May not learn the problem well.

This is the fundamental trade-off:

• Too simple — cannot do the task because not enough parameters to learn. e.g., 5 hidden neurons. (*underfitting*)

• Too complex — cannot generalize from small and noisy datasets well, e.g., 20 hidden neurons. (*overfitting*)

Overfitting and underfitting



Small model is **underfitting**
Medium model seems just right
Large model is **overfitting**

Overfitting

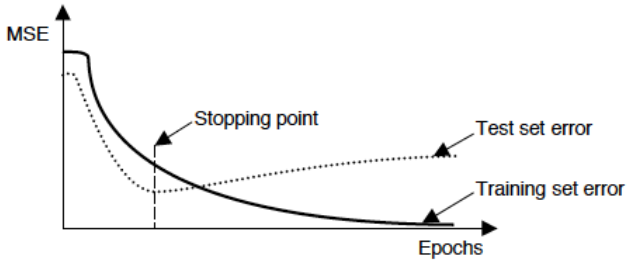
Overfitting is one of the problems that occur during training of neural networks, which drives the training error of the network to a very small value at the expense of the test error. The network learns to respond correctly to the training inputs by remembering them too much but is unable to generalize to produce correct outputs to novel inputs.

- Overfitting happens when the amount of training data is inadequate in comparison to the number of network parameters to learn.
- Overfitting occurs when the weights and biases become too large and are fine-tuned to remember the training patterns too much.
- Even your model is right, too much training can cause overfitting.

Methods to overcome overfitting

1. Early stopping
2. Regularization of weights
3. Dropouts

Early stopping



Training of the network is to be stopped when the validation error starts increasing. Early stopping can be used in test/validation by stopping when the validation error is minimum.

Regularization of weights

During overfitting, some weights attain large values to reduce training error, jeopardizing its ability to generalizing. In order to avoid this, a *penalty* term (*regularization* term) is added to the cost function.

For a network with weights $\boldsymbol{W} = \{w_{ij}\}$ and bias $\boldsymbol{b}$, the penalized (or regularized) cost function $J_1(\boldsymbol{W}, \boldsymbol{b})$ is defined as

$$J_1 = J + \beta_1 \sum_{i,j} |w_{ij}| + \beta_2 \sum_{ij} (w_{ij})^2$$

where $J(\boldsymbol{W}, \boldsymbol{b})$ is the standard cost function (i.e., m.s.e. or cross-entropy),

$$L^1 - norm = \sum_{i,j} |w_{ij}|$$

$$L^2 - norm = \sum_{ij} (w_{ij})^2$$

And β_1 and β_2 are known as L^1 and L^2 regularization (penalty) constants, respectively. These penalties discourage weights from attaining large values.

L2 regularization of weights

Regularization is usually not applied on bias terms. L^2 regularization is most popular on weights.

$$J_1 = J + \beta_2 \sum_{ij} (w_{ij})^2$$

Gradient of the regularized cost wrt weights:

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta_2 \frac{\partial (\sum_{ij} {w_{ij}}^2)}{\partial \boldsymbol{W}} \qquad \text{(A)}$$

L2 regularization of weights

$$L^2 \text{ norm} = \sum_{ij} (w_{ij})^2 = {w_{11}}^2 + {w_{12}}^2 + \cdots {w_{Kn}}^2$$

$$\frac{\partial \left(\sum_{ij} (w_{ij})^2\right)}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial \sum (w_{ij})^2}{\partial w_{11}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{12}} & \cdots & \frac{\partial \sum (w_{ij})^2}{\partial w_{1k}} \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{21}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{22}} & \cdots & \frac{\partial \sum (w_{ij})^2}{\partial w_{2k}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{n1}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{n2}} & \cdots & \frac{\partial \sum (w_{ij})^2}{\partial w_{nK}} \end{pmatrix} = 2\boldsymbol{W}$$

L2 regularization of weights

Substituting in (A):

$$\nabla_{\boldsymbol{W}} J_1 = \nabla_{\boldsymbol{W}} J + \beta_2 \frac{\partial (\sum_{ij} {w_{ij}}^2)}{\partial \boldsymbol{W}} = \nabla_{\boldsymbol{W}} J + 2\beta_2 \boldsymbol{W}$$

For gradient decent learning that uses regularized cost function:

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \nabla_{\boldsymbol{W}} J_1$$

Substituting above:

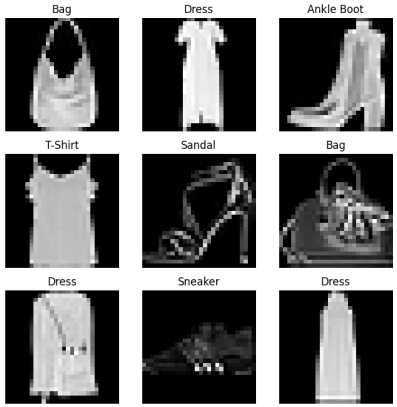
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha(\nabla_{\boldsymbol{W}} J + \beta \boldsymbol{W})$$

where $\beta = 2\beta_2$

β is known as the *weight decay parameter*.

That is for L^2 regularization, the weight matrix is weighted by decay parameter and added to the gradient term.

Fashion MNIST dataset



https://github.com/zalandoresearch/fashion-mnist

Example 4: Early stopping and weight decay

DNN with [784, 400, 400, 400, 10] architecture.
Let's implement L2 regularization with $\beta = 0.0001$
Use early stopping to terminate learning.

```
class NeuralNetwork(nn.Module):
    def __init__(self, hidden_size=500):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(32*32*3, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 10),
            nn.Softmax(dim=1)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```


Example 4

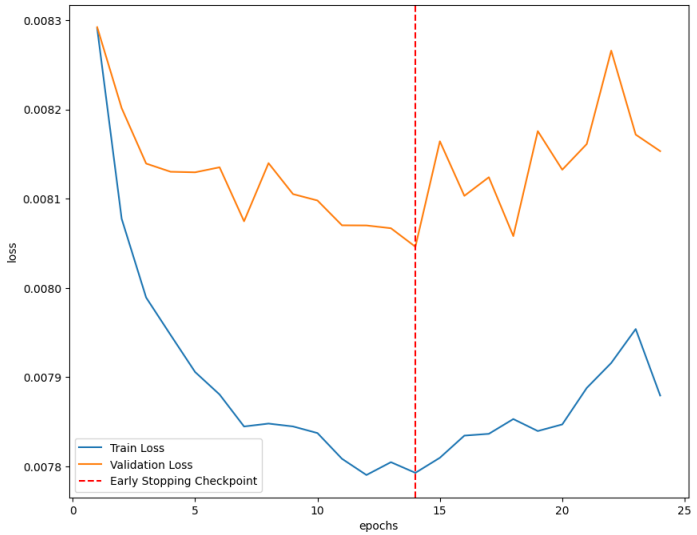
```
class EarlyStopper:
    def __init__(self, patience=5, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.min_validation_loss = np.inf

    def early_stop(self, validation_loss):
        if validation_loss < self.min_validation_loss:
            self.min_validation_loss = validation_loss
            self.counter = 0
        elif validation_loss > (self.min_validation_loss + self.min_delta):
            self.counter += 1
            if self.counter >= self.patience:
                return True
        return False
```


Example 4

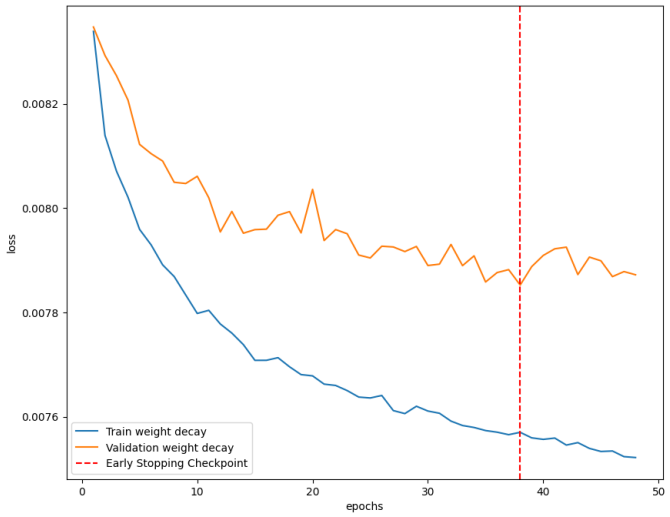
```
model = NeuralNetwork()
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, weight_decay=0.001)
early_stopper = EarlyStopper(patience=patience, min_delta)
```

```
if early_stopper.early_stop(test_loss):
        print("Done!")
        break
```

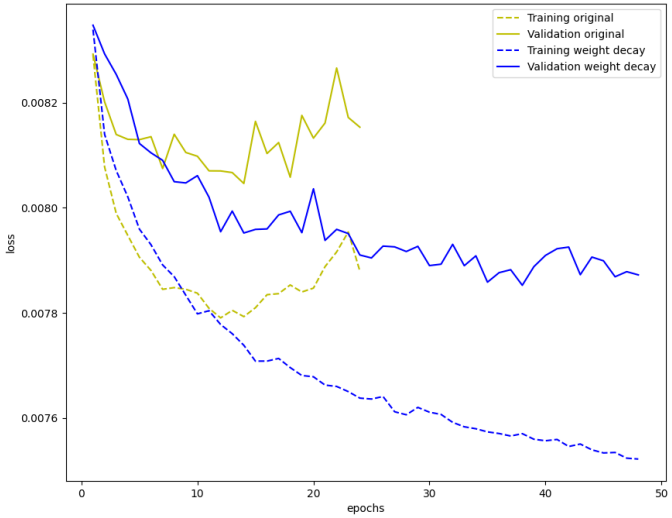

Example 4



Example 4



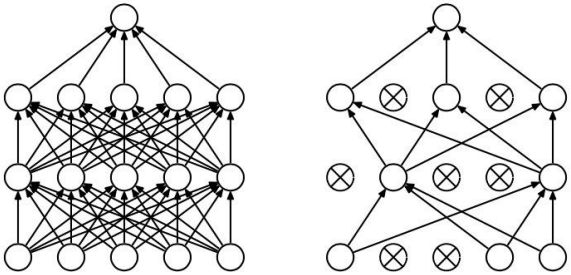
Example 4



Dropouts

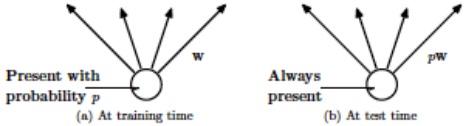
Overfitting can be avoided by training only a fraction of weights in each iteration. The key idea of 'dropouts' is to randomly drop neurons (along with their connections) from the networks during training.

This prevents neurons from co-adapting and thereby reduces overfitting.



Fully connected network Network with dropouts

Dropouts



At the training time, the units (neurons) are present with a probability p and presented to the next layer with weight $\boldsymbol{W}$ to the next layer.

This results in a scenario that at test time, the weights are always present, and presented to the network with weights multiplied by probability p. That is, The output at the test time is multiplied by $\frac{1}{p}$.

Applying dropouts result in a 'thinned network' that consists of only neurons that survived. This minimizes the redundancy in the network.

Dropout ratio

Dropout ratio is the fraction of neurons to be dropped out at one forward step.

Dropout ratio (p) has to be specified with nn.Dropout() in torch after activation function.

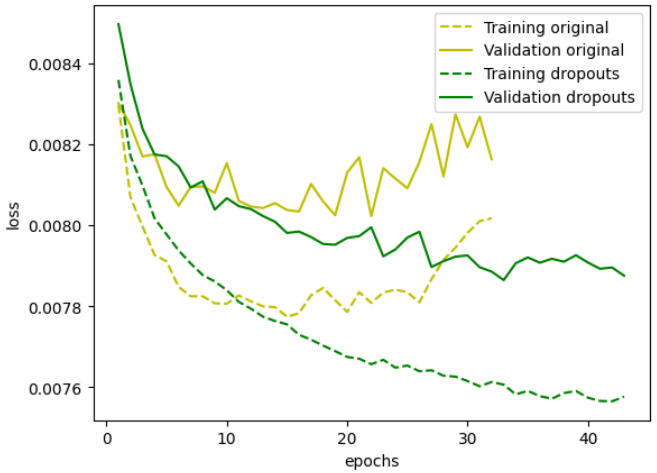
```
nn.Dropout(p=0.2)
```


Example 5: dropouts

When training with dropouts, we train on a thinned network dropping out units in each mini-batch.

```
class NeuralNetwork_dropout(nn.Module):
  def __init__(self, hidden_size = 100, drop_out=0.5):
    super(NeuralNetwork_dropout, self).__init__()
    self.flatten = nn.Flatten()
    self.linear_relu_stack = nn.Sequential(
      nn.Linear(32*32*3, hidden_size),
      nn.ReLU(),
      nn.Linear(hidden_size, hidden_size),
      nn.ReLU(),
      nn.Linear(hidden_size, hidden_size),
      nn.ReLU(),
      nn.Dropout(p=drop_out),
      nn.Linear(hidden_size, 10),
      nn.Softmax(dim=1)
    )

  def forward(self, x):
    x = self.flatten(x)
    logits = self.linear_relu_stack(x)
    return logits
```


Example 5



Summary

- **Model selection**
  - Holdout method
  - Resampling methods
    - Random subsampling
    - K-fold cross-validation
    - LOO cross-validation
  - Three-way data split
- **Methods to overcome overfitting**
  - Early stopping
  - Weight regularization
  - Dropouts