Chapter 4

Deep neural networks

Neural networks and deep learning

© Nanyang Technological University

Chain rule of differentiation

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

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

Chain rule of differentiation states that:

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

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

$$\begin{tikzcd} \nabla_x J & \nabla_y J & \\ x \arrow[r, "f"] & y \arrow[r, "g"] & J \end{tikzcd}$$

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

Chain rule in multidimensions

$$\begin{tikzcd} \nabla_{\boldsymbol{x}} J & & \nabla_{\boldsymbol{y}} J & & \\ \boldsymbol{x} \arrow[rr, "f"] & & \boldsymbol{y} \arrow[rr, "g"] & & J \end{tikzcd}$$

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

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

Then, the chain rule of differentiation states that:

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

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

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

where

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

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

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.

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

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

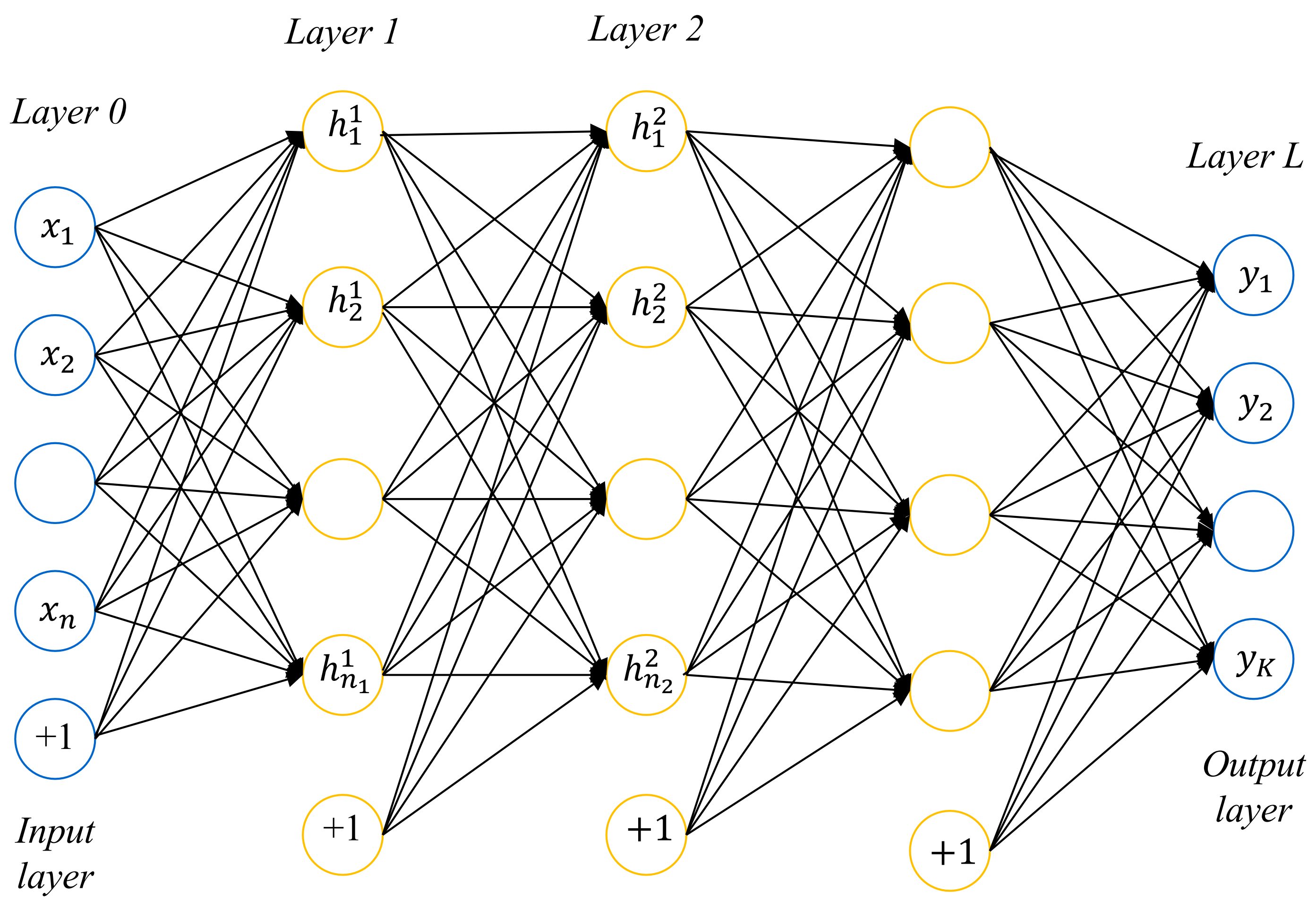
Find the Jacobian of f.

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

Jacobian:

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

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



L layer DNN

Feedforward Networks (FFN)

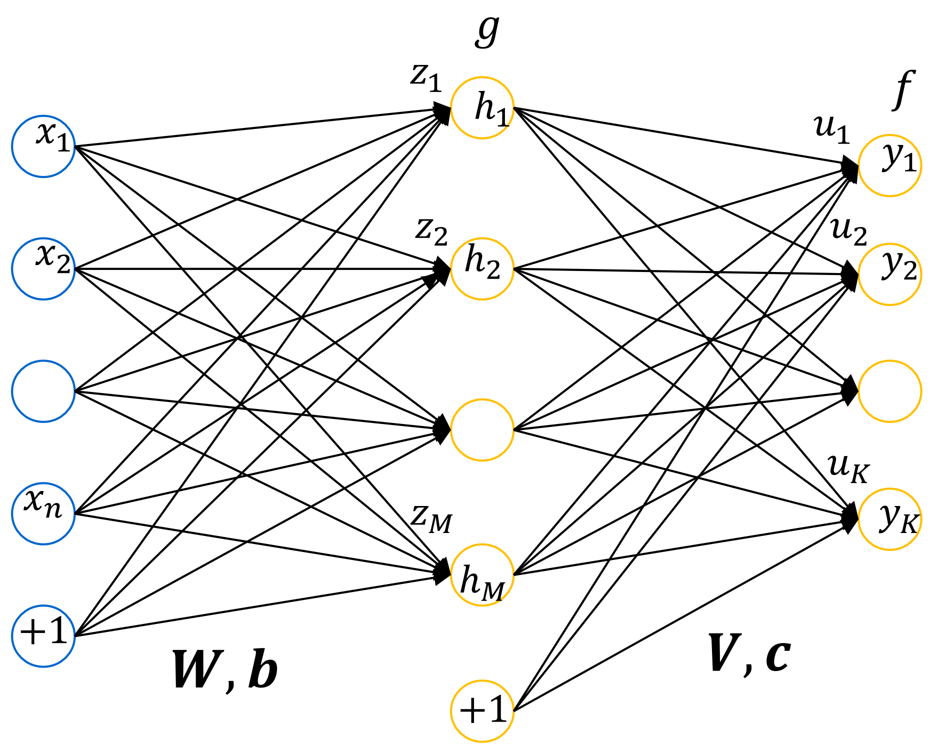
Feedforward networks (FFN) consists of several layers of neurons where activations propagate from input layer to output layer. The layers between the input and output layers are referred to as **hidden layers**.

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

Forward propagation of activations: single input pattern

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

$$\begin{tikzcd}
\boldsymbol{x} \arrow[r, "\boldsymbol{W}, \boldsymbol{b}"] & \boldsymbol{z} \arrow[r, "g"] & \boldsymbol{h} \arrow[r, "\boldsymbol{V}, \boldsymbol{c}"] & \boldsymbol{u} \arrow[r, "f"] & \boldsymbol{y} \arrow[r] & J & \boldsymbol{d} \arrow[l]
\end{tikzcd}$$

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:

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

Backpropagation of gradients

$$\begin{tikzcd}
 & \nabla_{\boldsymbol{z}} J & \nabla_{\boldsymbol{h}} J & \nabla_{\boldsymbol{u}} J & & & \\
\boldsymbol{x} \arrow[r, "\boldsymbol{W}, \boldsymbol{b}"] & \boldsymbol{z} \arrow[r, "g"] & \boldsymbol{h} \arrow[r, "\boldsymbol{V}, \boldsymbol{c}"] & \boldsymbol{u} \arrow[r, "f"] & \boldsymbol{y} \arrow[r] & J & \boldsymbol{d} \arrow[l]
\end{tikzcd}$$

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.

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

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

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

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:

$$y_k = f(u_k)$$

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

That is,

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

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

Back-propagation of gradients: single pattern

$$\begin{tikzcd}
 & \nabla_{\boldsymbol{z}} J & \nabla_{\boldsymbol{h}} J & \nabla_{\boldsymbol{u}} J & & & \\
\boldsymbol{x} \arrow[r, "\boldsymbol{W}, \boldsymbol{b}"] & \boldsymbol{z} \arrow[r, "g"] & \boldsymbol{h} \arrow[r, "\boldsymbol{V}, \boldsymbol{c}"] & \boldsymbol{u} \arrow[r, "f"] & \boldsymbol{y} \arrow[r] & J & \boldsymbol{d} \arrow[l]
\end{tikzcd}$$

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)

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 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:

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

SGD for a two-layer FFN

```
Given a training dataset $\{(\boldsymbol{x}, \boldsymbol{d})\}$
Set learning parameter $\alpha$
Initialize $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{V}, \boldsymbol{c}$
Repeat until convergence:
        For every pattern $(\boldsymbol{x}, \boldsymbol{d})$:
                $\boldsymbol{z} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{b}$
                $\boldsymbol{h} = g(\boldsymbol{z})$                                   Forward propagation
                $\boldsymbol{u} = \boldsymbol{V}^T \boldsymbol{h} + \boldsymbol{c}$
                $\boldsymbol{y} = f(\boldsymbol{u})$

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

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

Forward propagation of activations: batch of inputs

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

$$\begin{tikzcd}
\boldsymbol{X} \arrow[r, "\boldsymbol{W}, \boldsymbol{b}"] & \boldsymbol{Z} \arrow[r, "g"] & \boldsymbol{H} \arrow[r, "\boldsymbol{V}, \boldsymbol{c}"] & \boldsymbol{U} \arrow[r, "f"] & \boldsymbol{Y} \arrow[r] & J & \boldsymbol{D} \arrow[l]
\end{tikzcd}$$

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:

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

Back-propagation of gradients: batch of patterns

$$\begin{tikzcd}
 & \nabla_{Z} J & & & \nabla_{U} J & & & \\
\boldsymbol{X} \arrow[r, "\boldsymbol{W}, \boldsymbol{b}"] & \boldsymbol{Z} \arrow[r, "g"] & \boldsymbol{H} \arrow[r, "\boldsymbol{V}, \boldsymbol{c}"] & & \boldsymbol{U} \arrow[r, "f"] & \boldsymbol{Y} \arrow[r] & J & \boldsymbol{D} \arrow[l]
\end{tikzcd}$$

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

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

Substituting from (C)

Back-propagation of gradients: batch of patterns

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

$$(XY)^T = Y^T X^T$$

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

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

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

GD of two-layer FFN

Output layer:

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

Hidden layer:

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

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:

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

Forward propagation

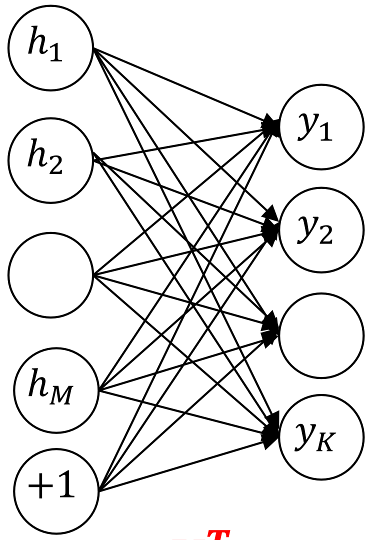
$$\nabla_{\boldsymbol{U}} J = \begin{cases} -(\boldsymbol{D} - \boldsymbol{Y}) \\ -\left(\boldsymbol{K} - f(\boldsymbol{U})\right) \end{cases}$$
$$\nabla_{\boldsymbol{Z}} J = (\nabla_{\boldsymbol{U}} J)\boldsymbol{V}^T \cdot g'(\boldsymbol{Z})$$

Backward propagation

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

Back-propagation

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



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

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

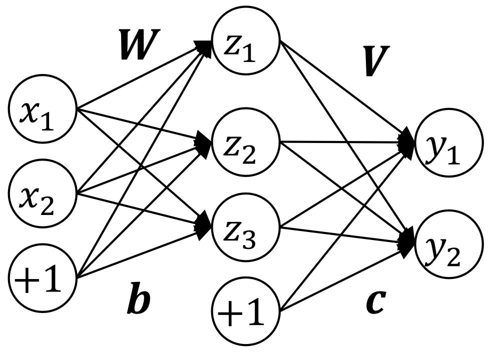
Example 2: Two-layer FFN

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:

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

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

Example 2

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

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

Example 2

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

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

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

Example 2

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

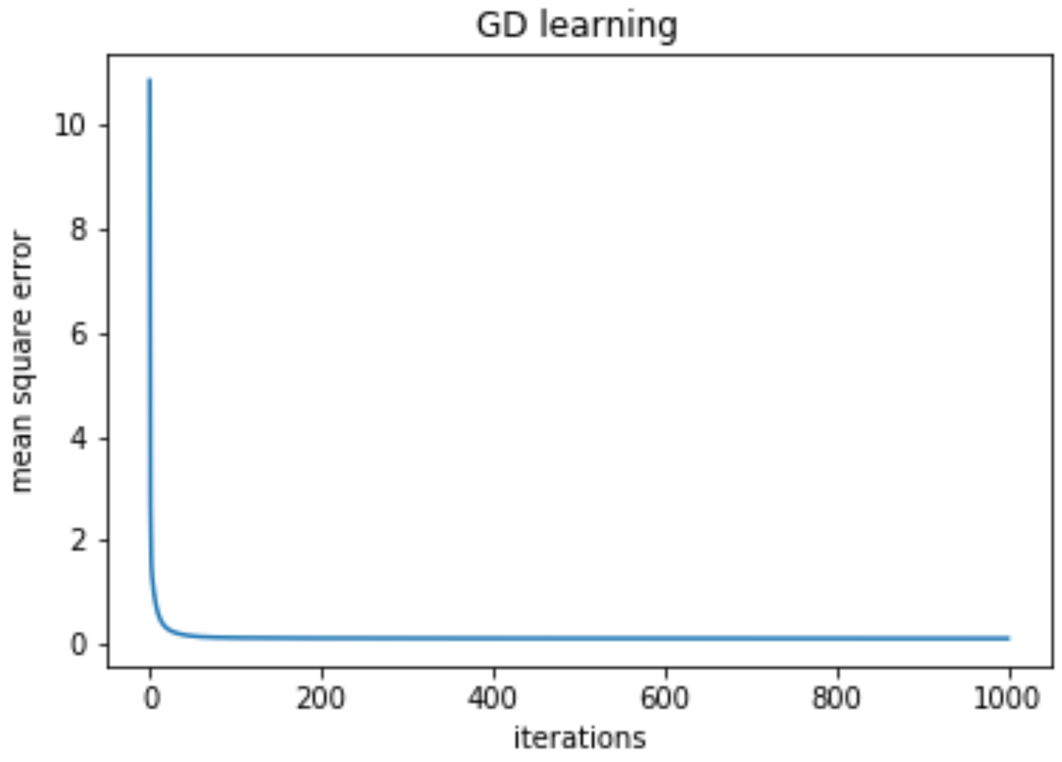
$$\boldsymbol{V} \leftarrow \boldsymbol{V} - \alpha \nabla_{\boldsymbol{V}} J = \begin{pmatrix} 3.89 & -1.74 \\ -3.14 & -1.89 \\ -2.53 & 2.51 \end{pmatrix}$$

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

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

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

Example 2



GD learning
mean square error
iterations
0
2
4
6
8
10
0
200
400
600
800
1000

Example 2

After 1,000 epochs,

m. s. e = 0.107

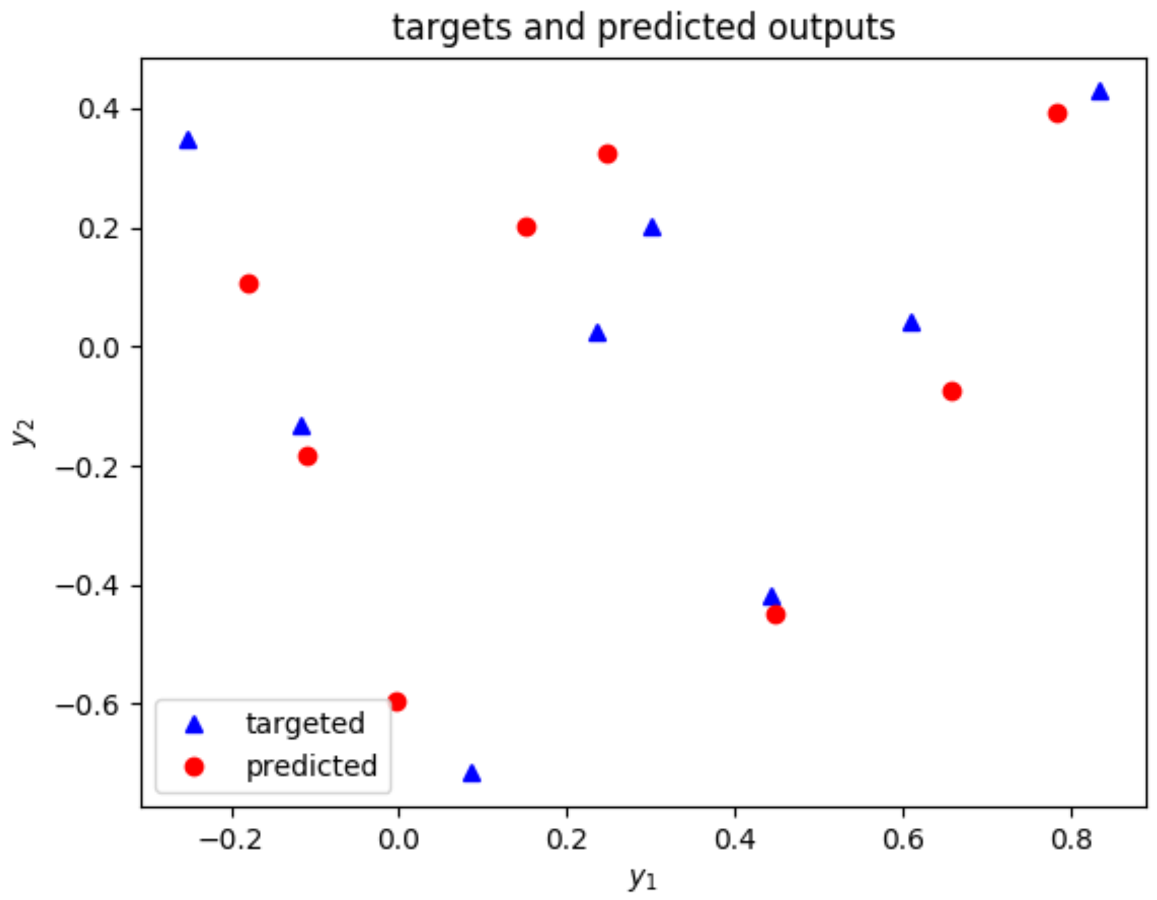
$$\boldsymbol{W} = \begin{pmatrix} -2.04 & -0.52 & -1.88 \\ 3.55 & -2.6 & 2.43 \end{pmatrix}$$

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

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

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

Example 2



After 20,000 iterations

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** $\tilde{y}_k \in [0,1]$:

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

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

California housing dataset

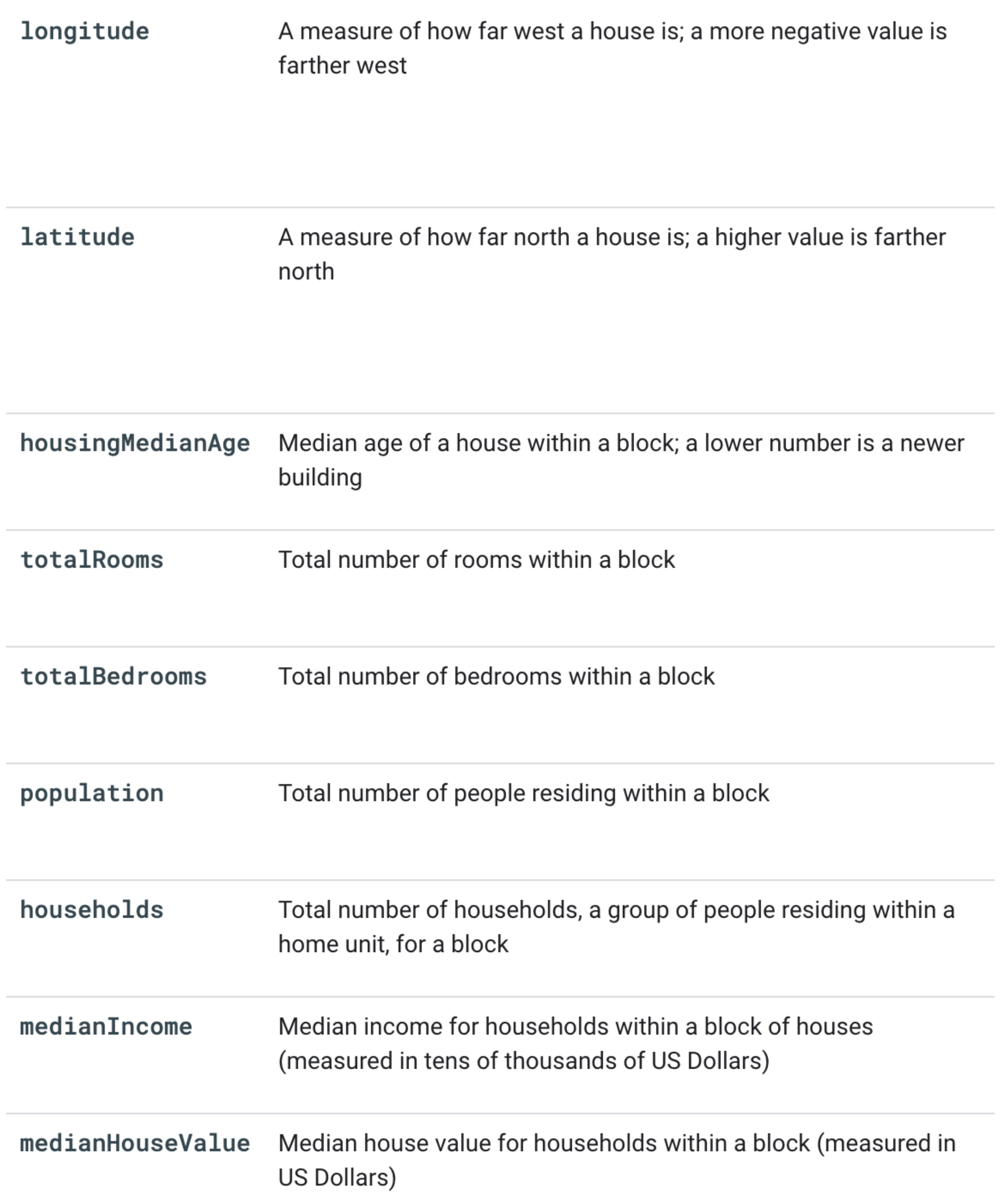
https://developers.google.com/machine-learning/crash-course/california-housing-data-description

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)

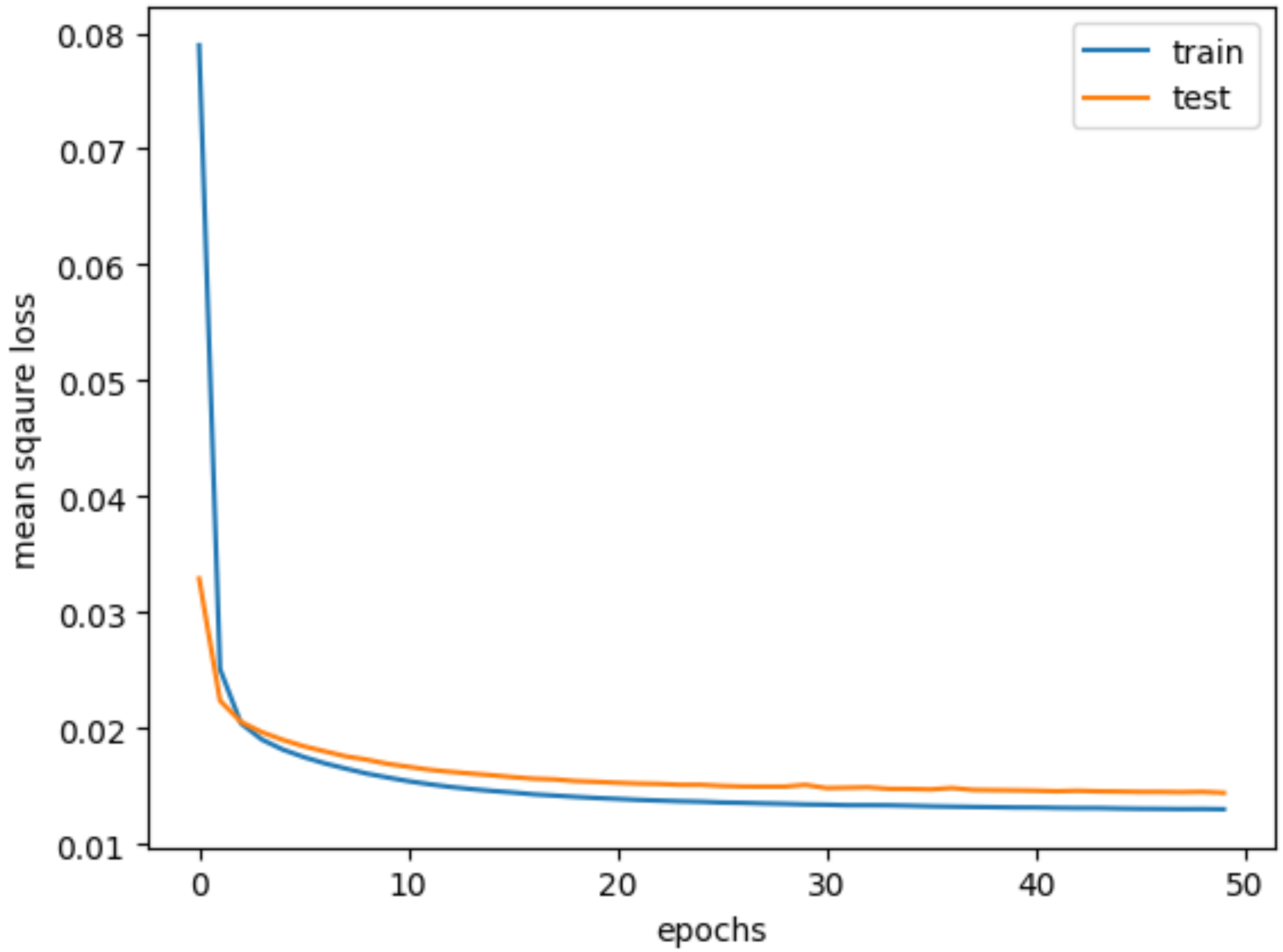
Example 3: Two-layer FFN predicting housing prices in California

Thirteen input variables, One output variable

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

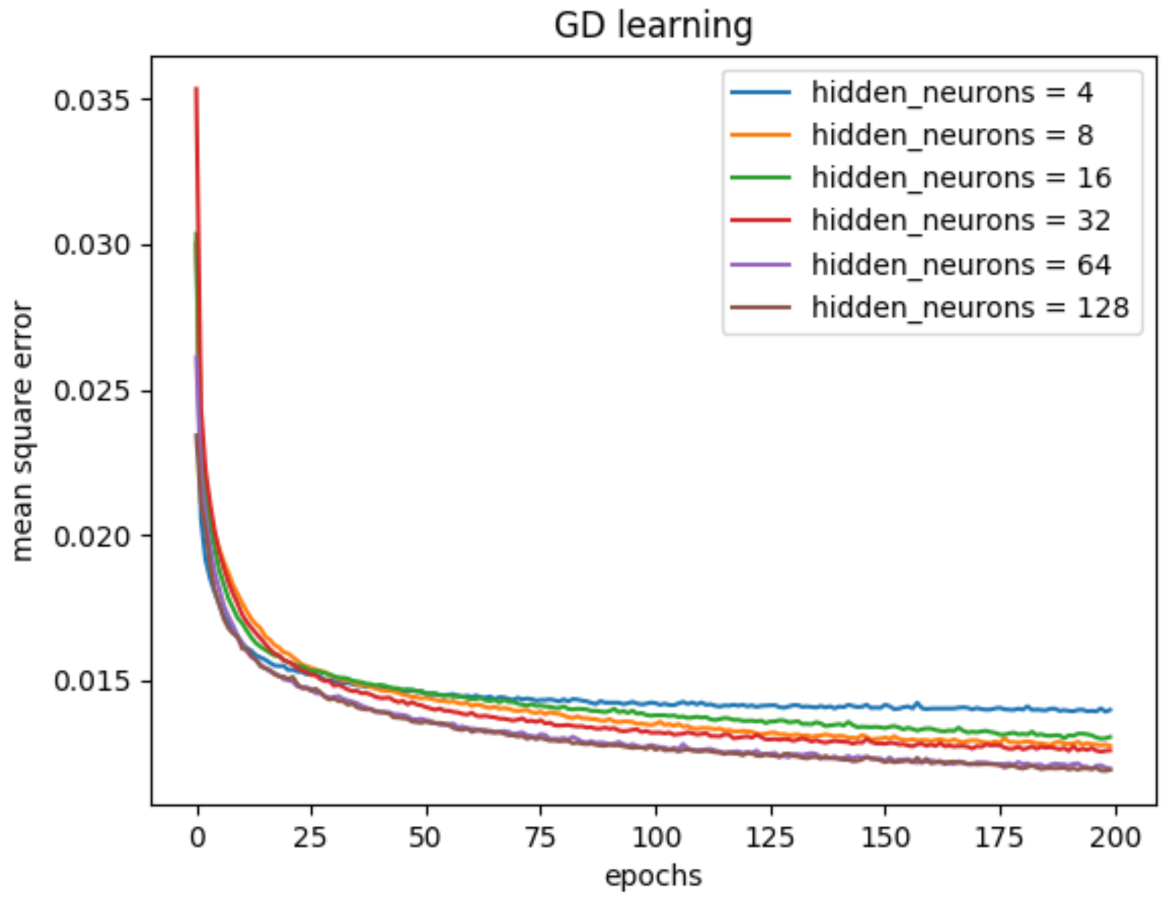
```
class FFN(nn.Module):
    def __init__(self, no_features, no_hidden, no_labels):
        super().__init__()
        self.relu_stack = nn.Sequential(
            nn.Linear(no_features, no_hidden),
            nn.ReLU(),
            nn.Linear(no_hidden, no_labels),
        )

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

Example 3a

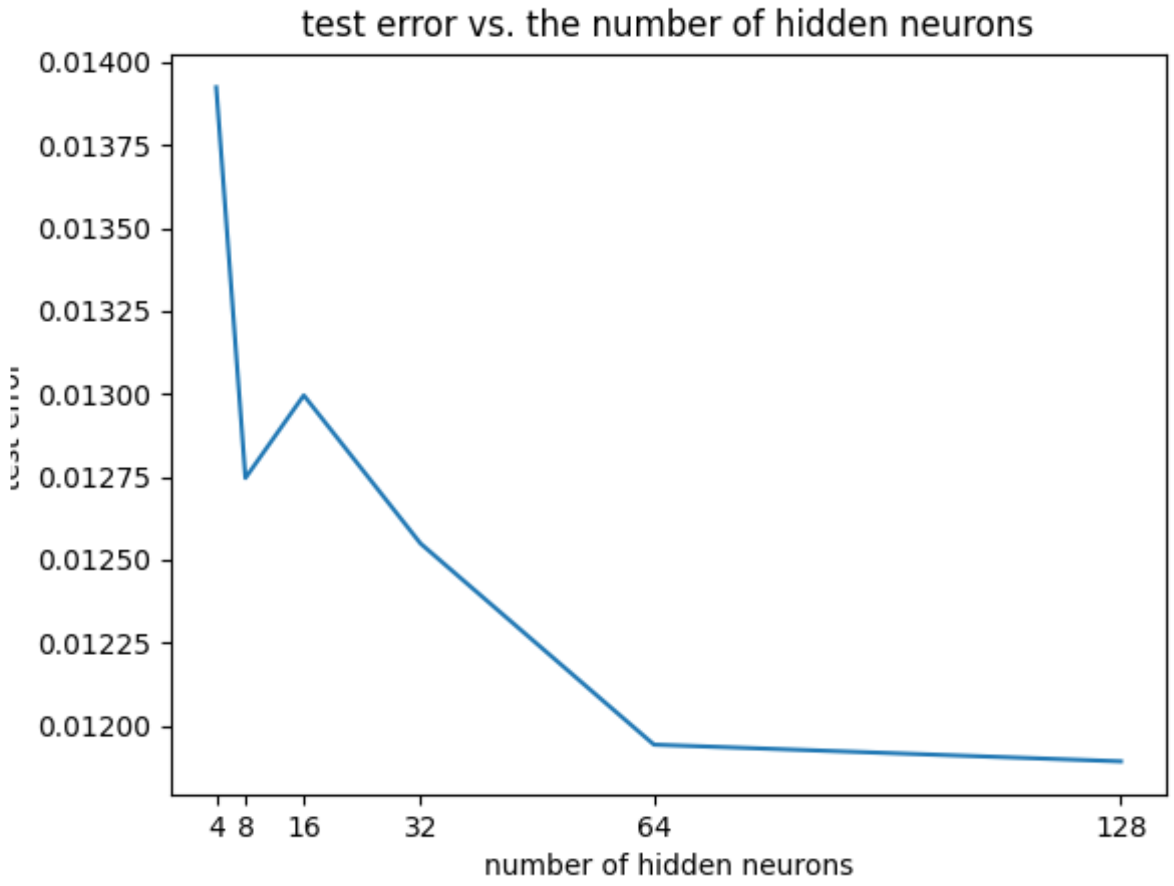


0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
mean sqaure loss
0
10
20
30
40
50
epochs
train
test

Example 3b: no of hidden neurons



Example 3b: width of the hidden layer



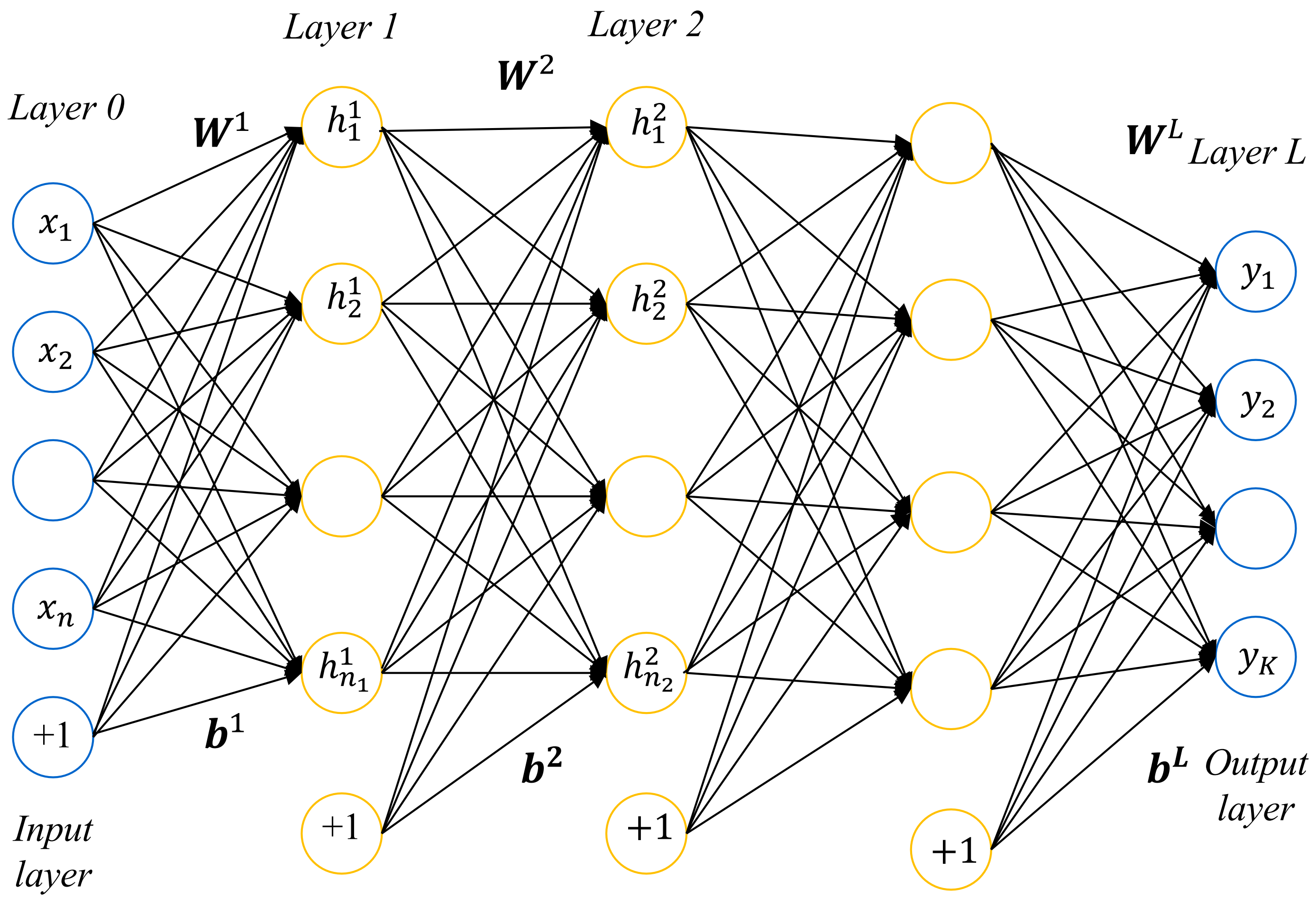
Optimum number of hidden neurons 64

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).

Deep neural networks (DNN)



Depth = L

DNN notations

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

<u>Hidden layers</u> $l = 1, 2, \cdots L - 1$
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 $\boldsymbol{h}^l, \boldsymbol{H}^l$

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

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 $l = 1, 2, \cdots, L-1$:
        $\boldsymbol{h}^l = f^l(\boldsymbol{u}^l)$
        $\boldsymbol{u}^{l+1} = {\boldsymbol{W}^{l+1}}^T \boldsymbol{h}^l + \boldsymbol{b}^{l+1}$
$\boldsymbol{y} = f^L(\boldsymbol{u}^L)$
```

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:

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

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

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

Gradients are backpropagated from the output layer to the input layer

Forward propagation of activation in DNN: batch of patterns

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

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:

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

from (D)

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

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

Gradients are backpropagated from the output layer to the input layer

Example 4: DNN on California Housing data

California housing data:

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

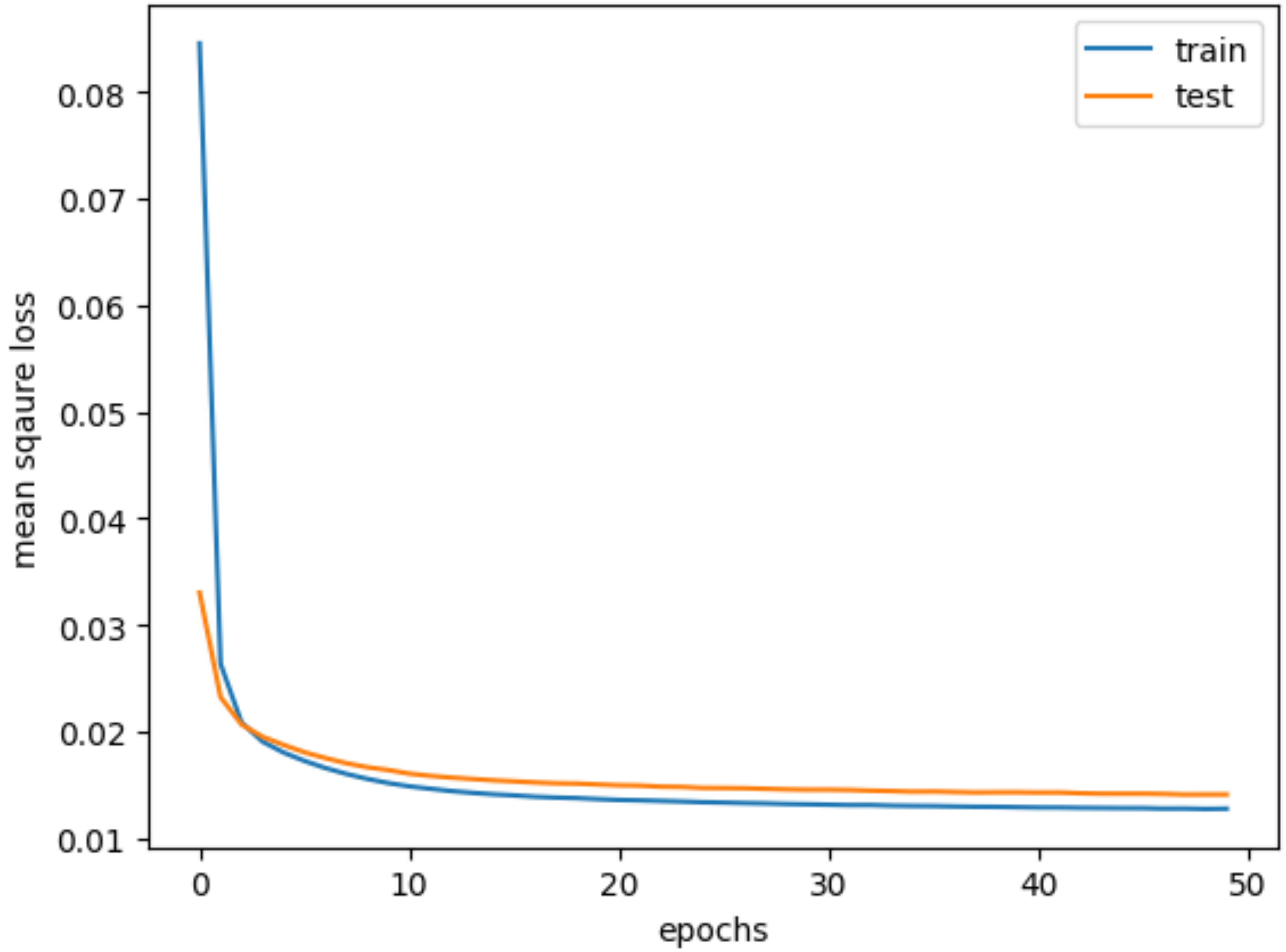
Predicting housing price from other 8 variables

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),
    )
```

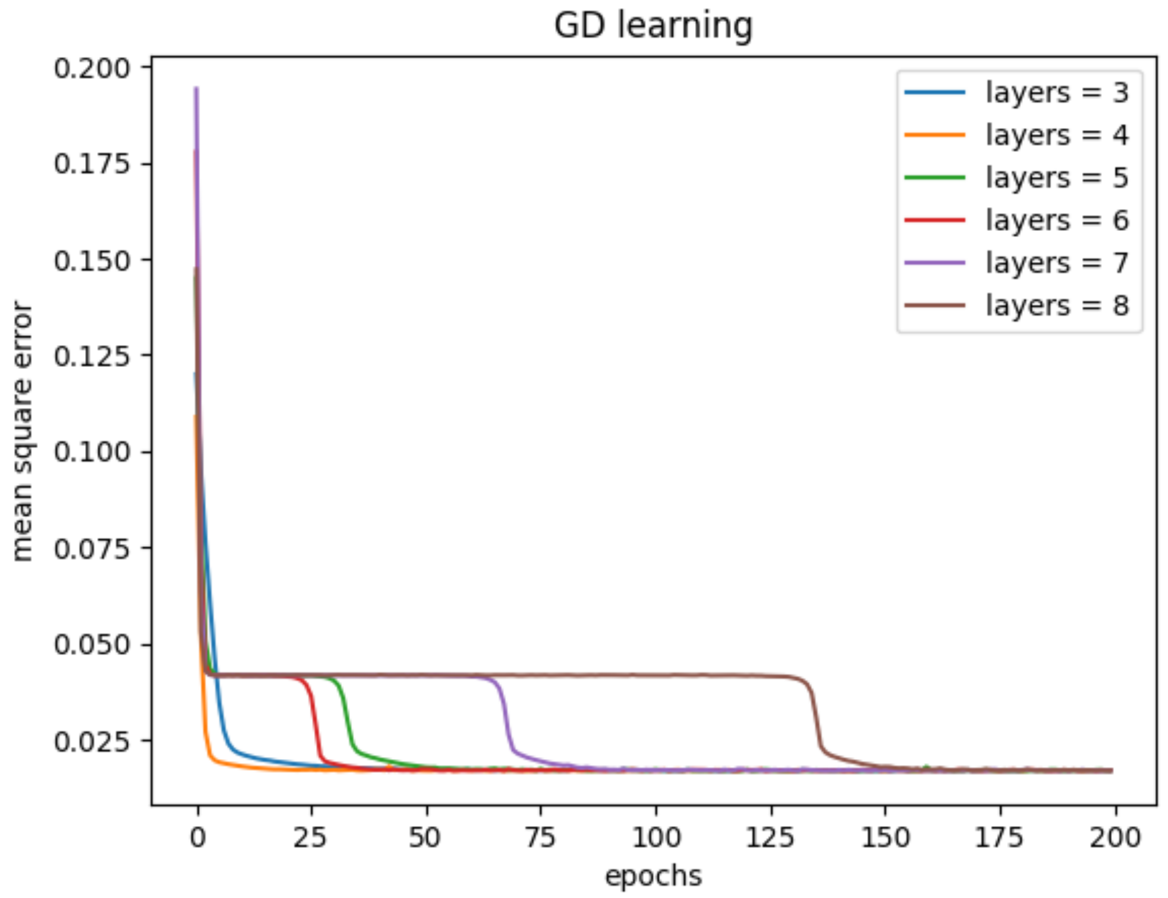
Example 4


train
test
mean sqaure loss
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
10
20
30
40
50
epochs

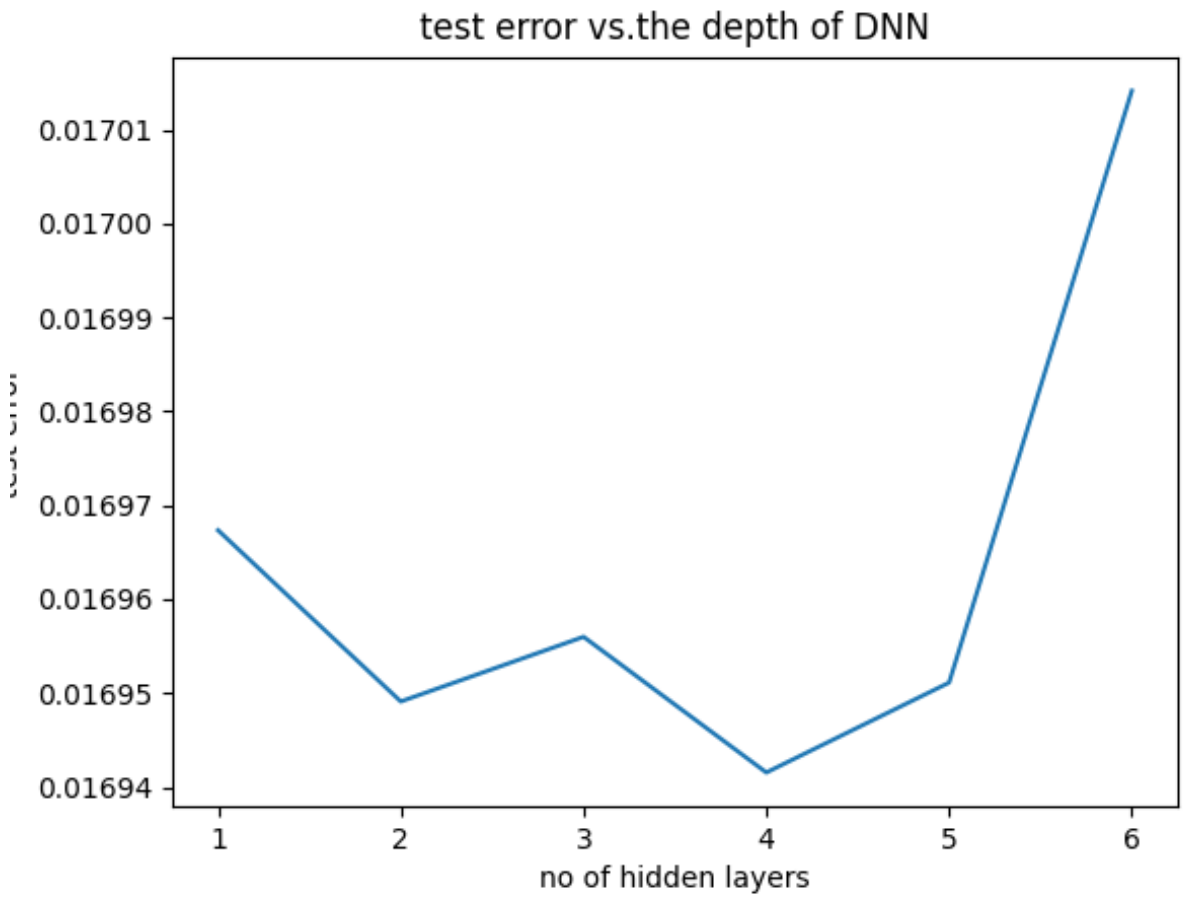
Example 4b: Varying the depth of DNN

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


GD learning
mean square error
epochs
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0
25
50
75
100
125
150
175
200
layers = 3
layers = 4
layers = 5
layers = 6
layers = 7
layers = 8

Example 4



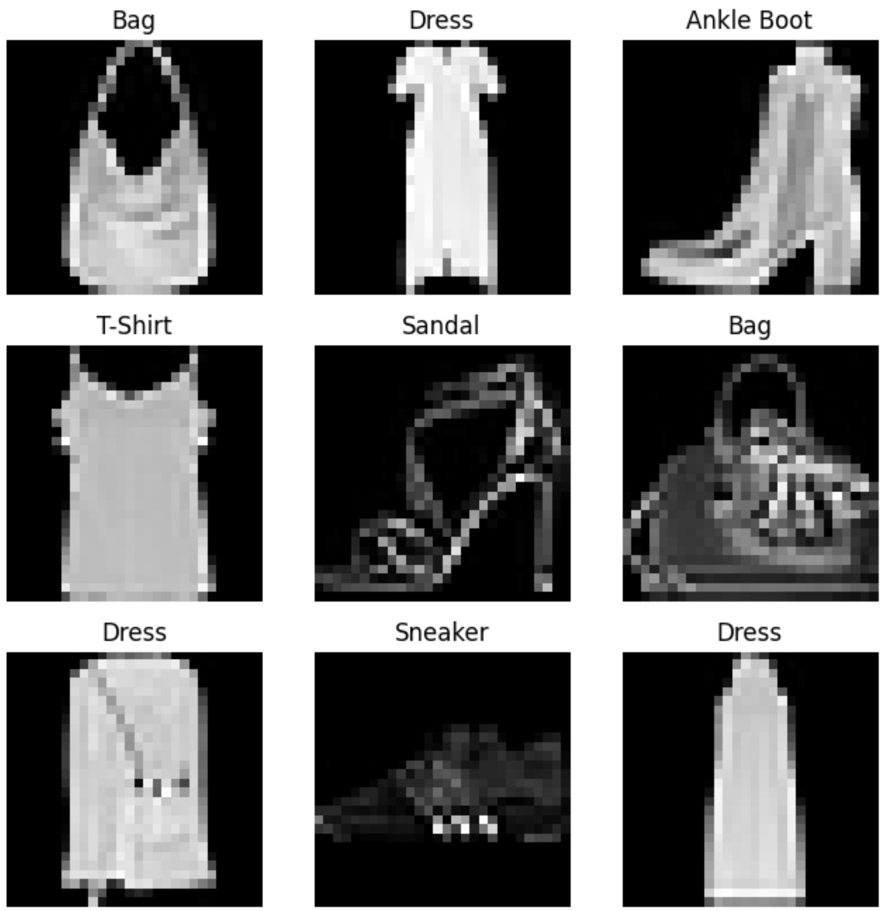
Optimum number of hidden layers = 4

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).

Fashion MNIST dataset



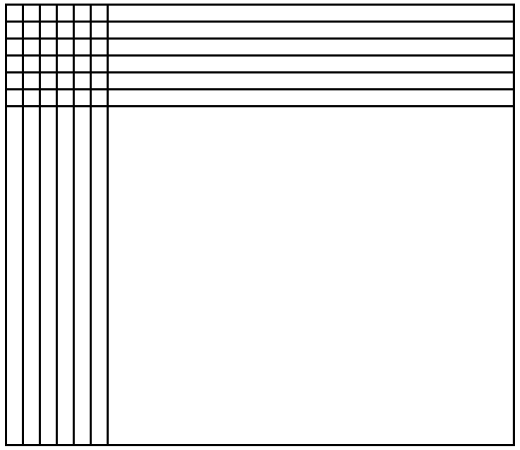
lThe Fasion-MNIST database of gray level images of fashion items from 10 classes:

lhttps://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.

Example 5: Classification of Fashion-MNIST images

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

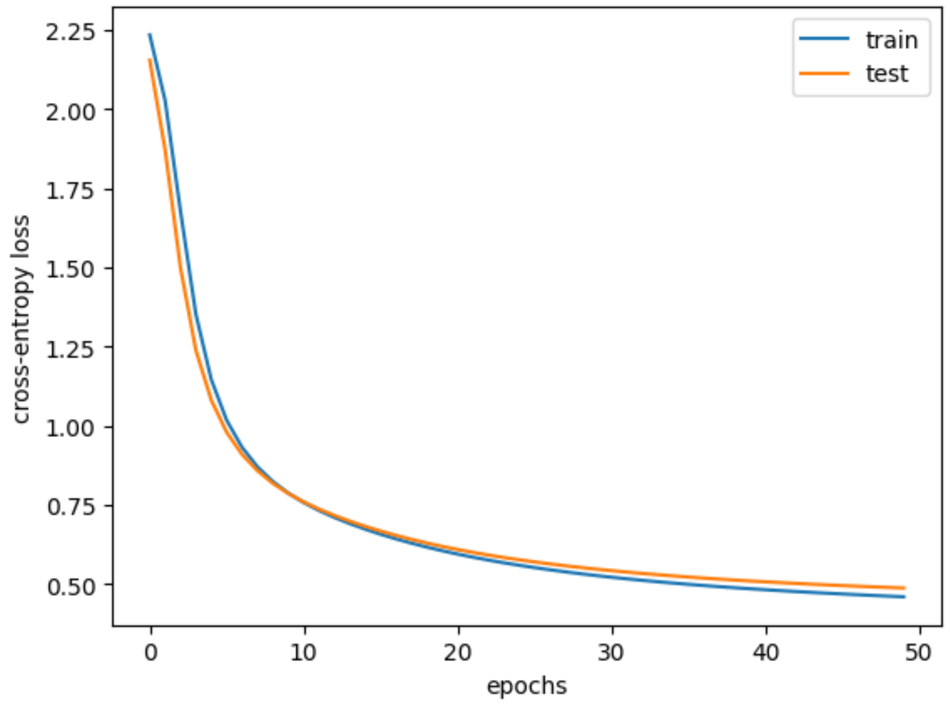
Use a 3-layer FFN
- Hidden-layer-1 is a perceptron layer
- Hidden-layer-2 is perceptron layer
- Output-layer is a softmax layer

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

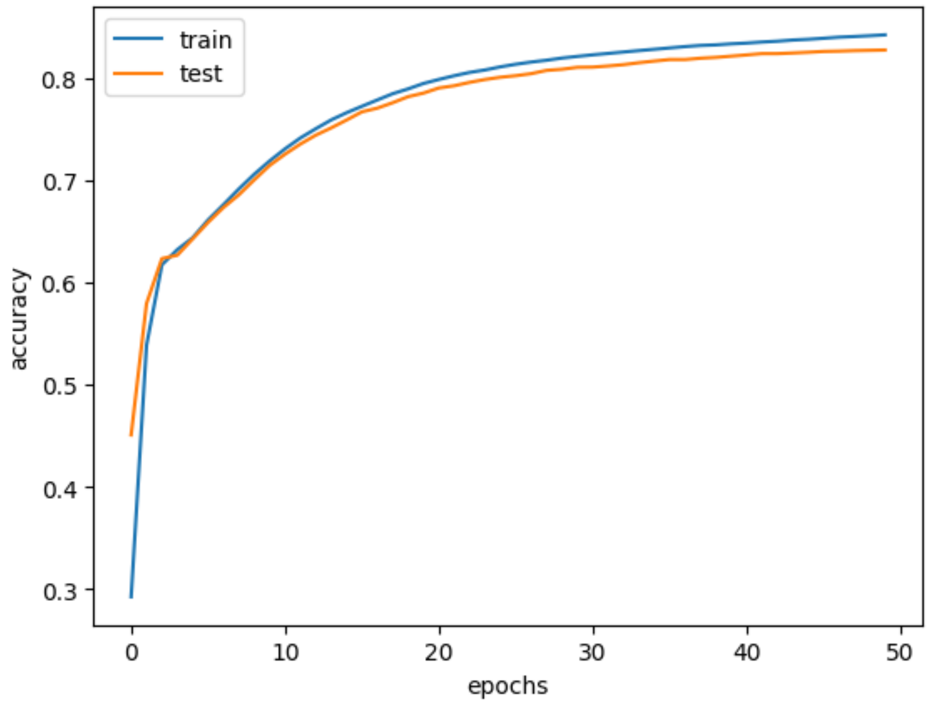
Training:
Batch size = 64
Learning rate $\alpha = 0.001$

```
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

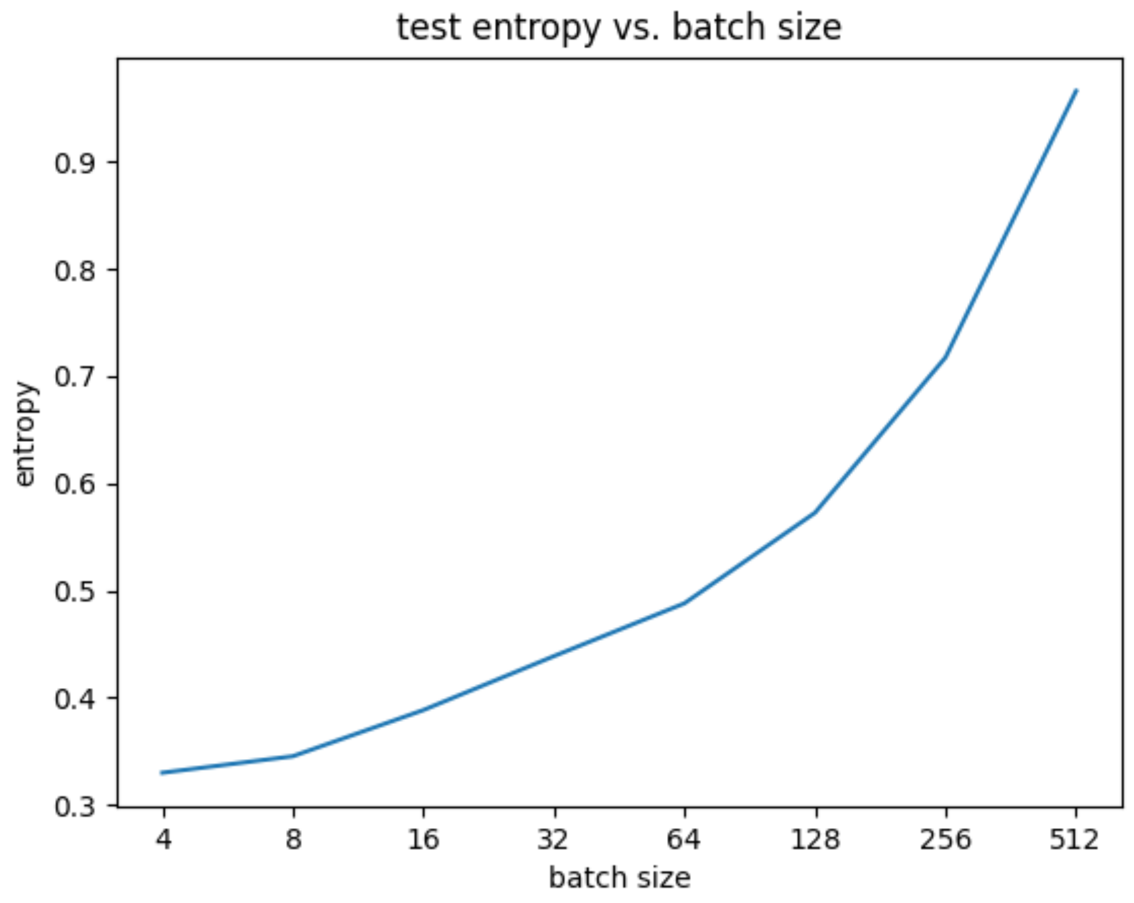


2.25
2.00
1.75
1.50
1.25
1.00
0.75
0.50
cross-entropy loss
0
10
20
30
40
50
epochs
train
test

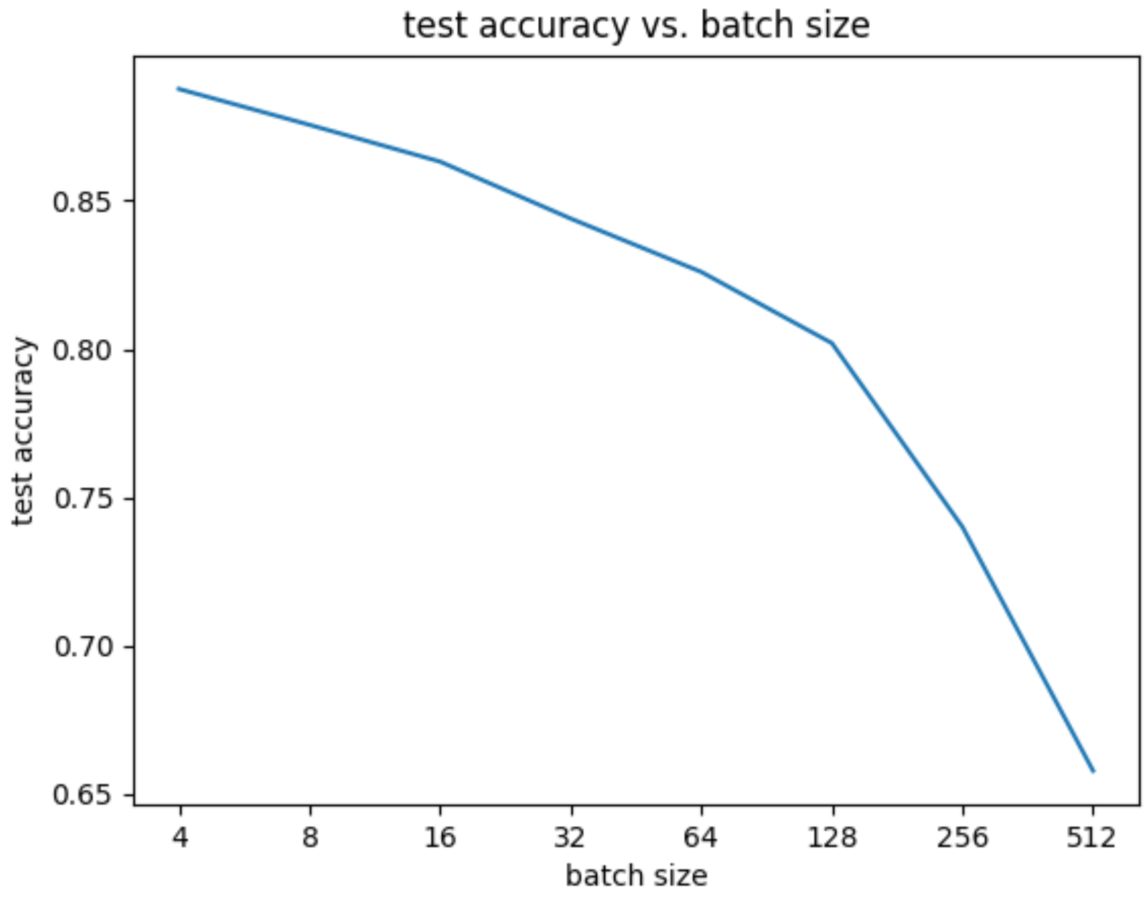


0.8
0.7
0.6
0.5
0.4
0.3
accuracy
0
10
20
30
40
50
epochs
train
test

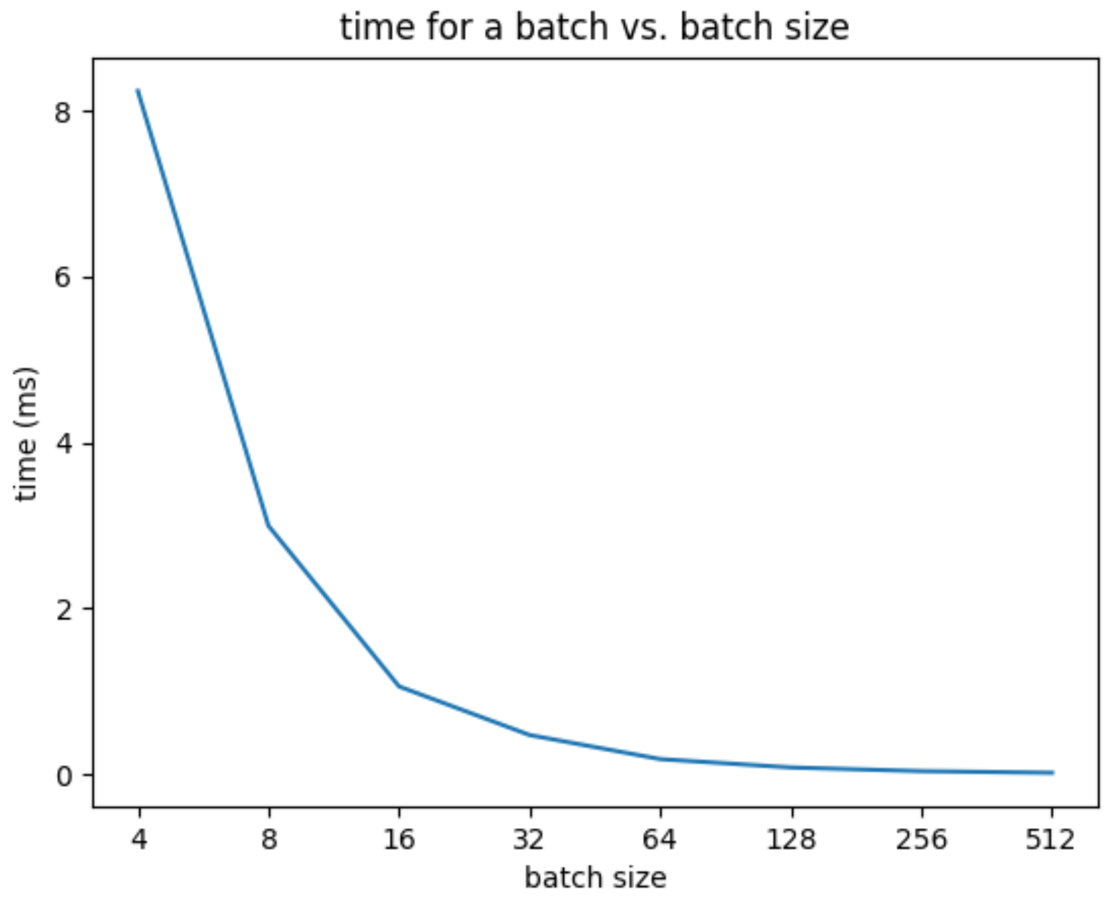
Example 5b: Effect of batch size



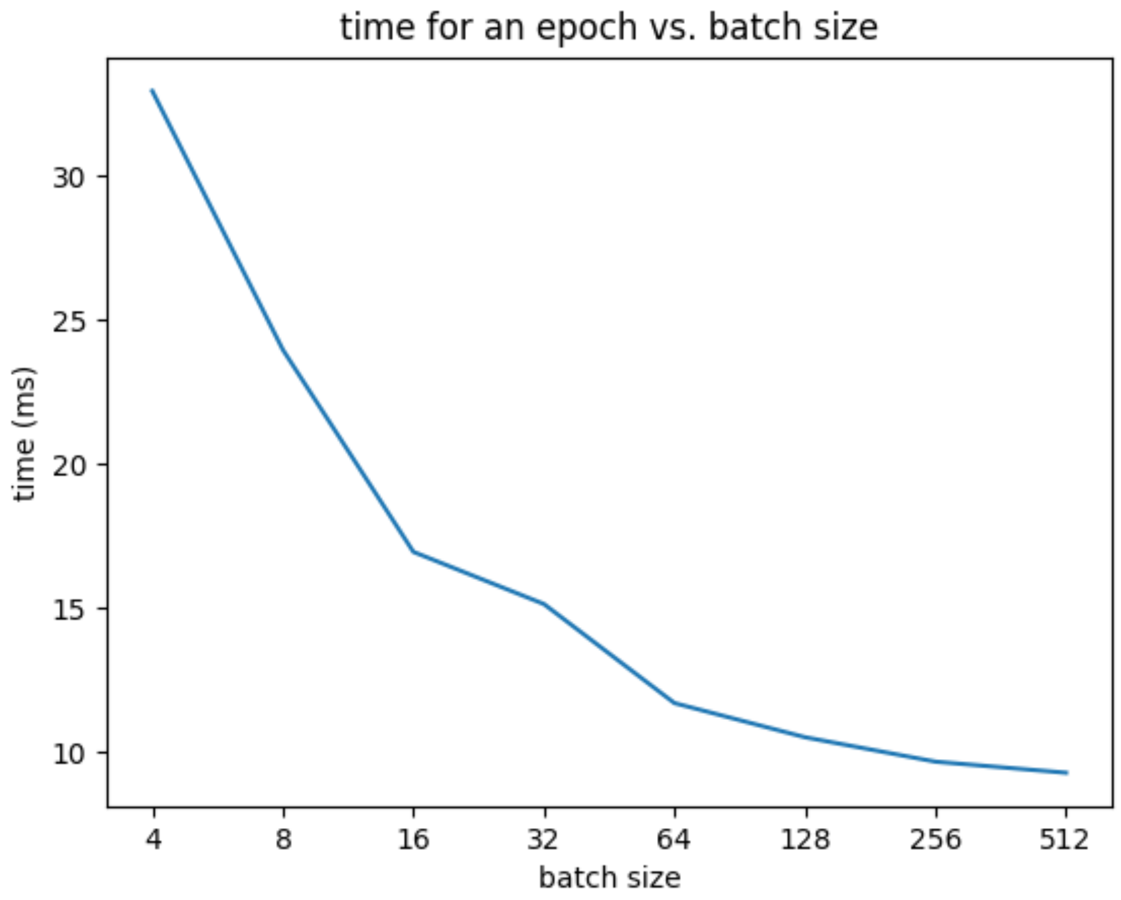
Example 5: Effect of batch size



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
$1 < B < P \rightarrow$ 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: $\nabla_{\boldsymbol{x}} J = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{T} \nabla_{\boldsymbol{y}} J$
- FFN with one hidden layer (Shallow FFN)
- Backpropagation for FFN with one hidden layer:

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

- 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