

1. The two-layer feedforward perceptron network shown in figure 1 has weights and biases initialized as indicated and receives 2-dimensional inputs (x_1, x_2) . The network is to respond with $d_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $d_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ for input patterns $x_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 3 \\ 0 \\ 3 \\ +1 \\ -1 \end{pmatrix}$ and $x_2 = \begin{pmatrix} -2 \\ 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ -1 \\ 3 \end{pmatrix}$, respectively.

Analyse a single feedforward and feedback step for gradient decent learning of the two patterns by doing the following:

- Find the weight matrix \mathbf{W} to the hidden-layer and weight matrix \mathbf{V} to the output-layer, and the corresponding biases.
- Calculate the synaptic input z and output h of the hidden-layer, and the synaptic input x and output y = (y_1, y_2) of the output layer.
- Find the mean square error cost J between the outputs and targets.
- Calculate the gradients ∇_w and ∇_d at the output-layer and the hidden-layer, respectively.
- Compute the new weights and biases.
- Write a program to continue iterations until convergence and find the final weights and biases.

Assume a learning rate of 0.05.

Repeat above (a) – (f) for stochastic gradient decent learning.

$$\text{Weight matrix to the hidden layer, } \mathbf{W} = \begin{pmatrix} 1.0 & 2.0 \\ -2.0 & 0.0 \end{pmatrix}$$

$$\text{Bias vector to the hidden-layer } \mathbf{b} = \begin{pmatrix} 3.0 \\ -1.0 \end{pmatrix}$$

$$\text{Weight matrix to the output-layer, } \mathbf{V} = \begin{pmatrix} 1.0 & 1.0 \\ 0.0 & -2.0 \end{pmatrix}$$

$$\text{Bias vector to the output-layer } \mathbf{c} = \begin{pmatrix} -2.0 \\ 3.0 \end{pmatrix}$$

$$x_1 = \begin{pmatrix} 1.0 \\ 3.0 \\ -2.0 \end{pmatrix} \text{ and } d_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$x_2 = \begin{pmatrix} -2.0 \\ 0 \\ 2 \end{pmatrix} \text{ and } d_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\mathbf{X} = \begin{pmatrix} 1.0 & 3.0 & -2.0 \\ -2.0 & 0 & 2 \end{pmatrix} \text{ and } \mathbf{D} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

Forward propagation:

$$\text{Synaptic input to hidden-layer, } \mathbf{Z} = \mathbf{XW} + \mathbf{B} = \begin{pmatrix} 1.0 & 3.0 \\ -2.0 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1.0 & 2.0 \\ -2.0 & 0.0 \end{pmatrix} + \begin{pmatrix} 3.0 & -1.0 \\ -0.70 & 0.38 \end{pmatrix} = \begin{pmatrix} 5.0 & -5.0 \\ -2.0 & 0.0 \end{pmatrix}$$

$$\text{Output of the hidden layer, } \mathbf{H} = g(\mathbf{Z}) = \frac{1}{1+e^{-z}} = \begin{pmatrix} 0.12 & 0.73 \\ 0.99 & 0.01 \end{pmatrix}$$

$$\text{Synaptic input to output-layer, } \mathbf{U} = \mathbf{HV} + \mathbf{C} = \begin{pmatrix} 0.12 & 0.73 \\ 0.99 & 0.01 \end{pmatrix} \begin{pmatrix} 1.0 & 1.0 \\ -2.0 & 3.0 \end{pmatrix} + \begin{pmatrix} -2.0 & 3.0 \\ -0.99 & 3.98 \end{pmatrix} = \begin{pmatrix} 0.54 & 0.28 \\ 0.56 & 0.31 \end{pmatrix}$$

$$\text{Output of the output layer, } \mathbf{Y} = f(\mathbf{U}) = \frac{1}{1+e^{-u}} = \begin{pmatrix} 0.13 & 0.84 \\ 0.27 & 0.98 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$m, s, e = \frac{1}{2} \sum_{k=1}^2 \sum_{j=1}^2 (d_{pk} - y_{pk})^2 = \frac{1}{2} \left(((0.13 - 0.12)^2 + (1 - 0.84)^2) + ((1 - 0.27)^2 + (0 - 0.98)^2) \right) = 0.77$$

$$f'(U) = Y \cdot (1 - Y) = \begin{pmatrix} 0.13 & 0.84 \\ 0.27 & 0.98 \end{pmatrix} \cdot \begin{pmatrix} 1.0 & 1.0 \\ 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 0.11 & 0.13 \\ 0.20 & 0.02 \end{pmatrix}$$

$$\nabla_U = -(\mathbf{D} - \mathbf{Y}) \cdot f'(U) = -\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0.13 & 0.84 \\ 0.27 & 0.98 \end{pmatrix} \cdot \begin{pmatrix} 0.12 & 0.13 \\ 0.20 & 0.02 \end{pmatrix} = \begin{pmatrix} 0.02 & -0.02 \\ -0.14 & 0.02 \end{pmatrix}$$

$$g'(\mathbf{Z}) = \mathbf{H} \cdot (1 - \mathbf{H}) = \begin{pmatrix} 0.12 & 0.73 \\ 0.99 & 0.01 \end{pmatrix} \cdot \begin{pmatrix} 1.0 & 1.0 \\ -2.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 0.10 & 0.2 \\ 0.01 & 0.01 \end{pmatrix}$$

$$\nabla_Z = (\nabla_U)^\top \cdot g'(\mathbf{Z}) = \begin{pmatrix} 0.02 & -0.02 \\ -0.14 & 0.02 \end{pmatrix} \cdot \begin{pmatrix} 1.0 & 1.0 \\ -2.0 & 0.0 \end{pmatrix} \cdot \begin{pmatrix} 0.10 & 0.2 \\ 0.01 & 0.01 \end{pmatrix} = \begin{pmatrix} -0.001 & 0.01 \\ -0.001 & 0.00 \end{pmatrix}$$

$$\mathbf{V} \leftarrow \mathbf{V} - \alpha \mathbf{H}^\top \nabla_U = \begin{pmatrix} 1.01 & 1.0 \\ 0.0 & -2.0 \end{pmatrix}$$

$$\mathbf{c} \leftarrow \mathbf{c} - \alpha (\nabla_U)^\top \mathbf{1}_P = \begin{pmatrix} -1.99 \\ 3.00 \end{pmatrix}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \mathbf{X}^\top \nabla_Z = \begin{pmatrix} 1.0 & 2.0 \\ -2.0 & 0.0 \end{pmatrix}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha (\nabla_Z)^\top \mathbf{1}_P = \begin{pmatrix} 3.0 \\ -1.0 \end{pmatrix}$$

2. A feedforward neural network with one hidden layer to perform the following classification:

class	inputs
A	(1.0, 1.0), (0.0, 1.0)
B	(3.0, 4.0), (2.0, 2.0)
C	(2.0, -2.0), (-2.0, -3.0)

The network has a hidden layer consisting of three perceptrons and a softmax output layer.

Show one iteration of gradient descent learning and plot learning curves until convergence at a learning rate $\alpha = 0.1$.

Initialize the weights \mathbf{W} and biases \mathbf{b} to the hidden layer, and the weights \mathbf{V} and biases \mathbf{c} to the output layer as follows:

$$\mathbf{W} = \begin{pmatrix} -0.10 & 0.97 & 0.18 \\ -0.70 & 0.38 & 0.93 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

$$\mathbf{V} = \begin{pmatrix} 1.01 & 0.09 & -0.39 \\ 0.79 & -0.45 & -0.22 \\ 0.28 & 0.96 & -0.07 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

Determine the weights and biases at convergence.

Find the class labels predicted by the trained network for patterns:

$$x_1 = \begin{pmatrix} 2.5 \\ 1.5 \end{pmatrix} \text{ and } x_2 = \begin{pmatrix} -1.5 \\ 0.5 \end{pmatrix}$$

Synaptic input to hidden-layer,

$$\mathbf{Z} = \mathbf{XW} + \mathbf{B} = \begin{pmatrix} 1.0 & 1.0 \\ 3.0 & 4.0 \\ 2.0 & 2.0 \\ 2.0 & -2.0 \\ -2.0 & -3.0 \end{pmatrix} \begin{pmatrix} -0.10 & 0.97 & 0.18 \\ -0.70 & 0.38 & 0.93 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} \begin{pmatrix} -0.80 & 1.35 & 1.10 \\ -0.70 & 0.38 & 0.93 \\ -1.59 & 2.70 & 2.21 \\ 1.20 & 1.18 & -1.50 \\ 2.29 & -3.08 & -3.13 \end{pmatrix}$$

$$\text{Output of the hidden layer, } \mathbf{H} = g(\mathbf{Z}) = \frac{1}{1+e^{-z}} = \begin{pmatrix} 0.31 & 0.79 & 0.75 \\ 0.33 & 0.59 & 0.72 \\ 0.04 & 0.99 & 0.99 \\ 0.17 & 0.94 & 0.90 \\ 0.77 & 0.77 & 0.18 \\ 0.91 & 0.04 & 0.04 \end{pmatrix}$$

$$\text{Synaptic input to output-layer, } \mathbf{U} = \mathbf{HV} + \mathbf{C} = \begin{pmatrix} 0.31 & 0.79 & 0.75 \\ 0.33 & 0.59 & 0.72 \\ 0.04 & 0.99 & 0.99 \\ 0.17 & 0.94 & 0.90 \\ 0.77 & 0.77 & 0.18 \\ 0.91 & 0.04 & 0.04 \end{pmatrix} \begin{pmatrix} 1.01 & 0.99 & -0.39 \\ 0.79 & -0.45 & -0.22 \\ 0.28 & 0.96 & -0.07 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 1.15 & 0.40 & -0.34 \\ 1.10 & 0.46 & 0.51 \\ 1.16 & 0.47 & -0.33 \\ 1.43 & 0.09 & -0.48 \\ 0.96 & 0.11 & -0.36 \end{pmatrix}$$

$$\text{Output layer activation } f(\mathbf{U}) = \frac{e^{\mathbf{u}}}{\sum e^{\mathbf{u}}} = \begin{pmatrix} 0.54 & 0.28 & 0.15 \\ 0.56 & 0.31 & 0.14 \\ 0.58 & 0.29 & 0.13 \\ 0.73 & 0.16 & 0.11 \\ 0.59 & 0.25 & 0.16 \end{pmatrix}$$

$$\text{Output } \mathbf{Y} = \arg \max_k f(\mathbf{U}) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 3 \end{pmatrix}, \quad \mathbf{Y} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}, \quad f(\mathbf{U}) = \begin{pmatrix} 0.59 & 0.28 & 0.13 \\ 0.54 & 0.31 & 0.15 \\ 0.56 & 0.29 & 0.13 \\ 0.73 & 0.16 & 0.11 \\ 0.59 & 0.25 & 0.16 \end{pmatrix}$$

Epoch 1:

Apply first pattern $x = \begin{pmatrix} 1.0 \\ 3.0 \end{pmatrix}$ and $d = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$:

Synaptic input to the hidden-layer

$$\mathbf{z} = \mathbf{W}^\top \mathbf{x} + \mathbf{b} = \begin{pmatrix} 1.0 & -2.0 \\ 2.0 & 0.0 \end{pmatrix} \begin{pmatrix} 1.0 \\ 3.0 \end{pmatrix} + \begin{pmatrix} 3.0 \\ -1.0 \end{pmatrix} = \begin{pmatrix} -2.0 \\ 1.0 \end{pmatrix}$$

$$\text{Output of the hidden-layer } \mathbf{h} = g(\mathbf{z}) = \frac{1}{1+e^{-z}} = \begin{pmatrix} 0.12 \\ 0.73 \end{pmatrix}$$

Synaptic input to output-layer

$$\mathbf{u} = \mathbf{V}^\top \mathbf{h} + \mathbf{c} = \begin{pmatrix} -1.88 \\ 1.66 \end{pmatrix}$$

$$\text{Output of the output-layer } \mathbf{y} = f(\mathbf{u}) = \frac{1}{1+e^{-u}} = \begin{pmatrix} 0.13 \\ 0.84 \end{pmatrix}$$

Computing gradients:

$$f'(\mathbf{u}) = \mathbf{y} \cdot (1 - \mathbf{y}) = \begin{pmatrix} 0.13 \\ 0.84 \end{pmatrix} \cdot \begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix} = \begin{pmatrix} 0.11 \\ 0.13 \end{pmatrix}$$

$$\nabla_{\mathbf{u}} = -(\mathbf{d} - \mathbf{y}) \cdot f'(\mathbf{u}) = -\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0.13 \\ 0.84 \end{pmatrix} \cdot \begin{pmatrix} 0.12 \\ 0.88 \end{pmatrix} = \begin{pmatrix} 0.02 \\ -0.02 \end{pmatrix}$$

$$g'(\mathbf{z}) = \mathbf{h} \cdot (1 - \mathbf{h}) = \begin{pmatrix} 0.12 \\ 0.73 \end{pmatrix} \cdot \begin{pmatrix} 1.0 \\ 0.0 \end{pmatrix} = \begin{pmatrix} 0.10 \\ 0.20 \end{pmatrix}$$

$$\nabla_{\mathbf{z}} = \nabla_{\mathbf{u}}^\top \cdot g'(\mathbf{z}) = \begin{pmatrix} 1.0 & 1.0 \\ 0.0 & 0.0 \end{pmatrix} \cdot \begin{pmatrix} 0.12 \\ 0.88 \end{pmatrix} = \begin{pmatrix} -0.001 \\ 0.001 \end{pmatrix}$$

Updating weights:

$$\mathbf{V} \leftarrow \mathbf{V} - \alpha \mathbf{H}^\top \nabla_{\mathbf{u}} = \begin{pmatrix} 1.01 & 1.0 \\ 0.0 & -2.0 \end{pmatrix}$$

$$\mathbf{c} \leftarrow \mathbf{c} - \alpha (\nabla_{\mathbf{u}})^\top \mathbf{1}_P = \begin{pmatrix} -1.99 \\ 3.00 \end{pmatrix}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \mathbf{X}^\top \nabla_{\mathbf{z}} = \begin{pmatrix} 1.0 & 2.0 \\ -2.0 & 0.0 \end{pmatrix}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \alpha (\nabla_{\mathbf{z}})^\top \mathbf{1}_P = \begin{pmatrix} 3.00 \\ -1.00 \end{pmatrix}$$

Training inputs and targets

$$\mathbf{X} = \begin{pmatrix} 1.0 & 1.0 \\ 3.0 & 4.0 \\ 2.0 & 2.0 \\ 2.0 & -2.0 \\ -2.0 & -3.0 \end{pmatrix} \text{ and } \mathbf{D} = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 3 \\ 3 \end{pmatrix}$$

Targets as a one hot matrix:

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

Initial weights and biases:

$$\text{To the hidden layer, } \mathbf{W} = \begin{pmatrix} -0.10 & 0.97 & 0.18 \\ -0.70 & 0.38 & 0.93 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

$$\text{To the output-layer } \mathbf{V} = \begin{pmatrix} 1.01 & 0.09 & -0.39 \\ 0.79 & -0.45 & -0.22 \\ 0.28 & 0.96 & -0.07 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

Learning factor $\alpha = 0.1$

$$\begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} \begin{pmatrix} -0.80 & 1.35 & 1.10 \\ -0.70 & 0.38 & 0.93 \\ -1.59 & 2.70 & 2.21 \\ 1.20 & 1.18 & -1.50 \\ 2.29 & -3.08 & -3.13 \end{pmatrix}$$

$$\begin{pmatrix} 0.31 & 0.79 & 0.75 \\ 0.33 & 0.59 & 0.72 \\ 0.04 & 0.99 & 0.99 \\ 0.17 & 0.94 & 0.90 \\ 0.77 & 0.77 & 0.18 \\ 0.91 & 0.04 & 0.04 \end{pmatrix} \begin{pmatrix} 0.31 & 0.79 & 0.75 \\ 0.33 & 0.59 & 0.72 \\ 0.04 & 0.99 & 0.99 \\ 0.17 & 0.94 & 0.90 \\ 0.77 & 0.77 & 0.18 \\ 0.91 & 0.04 & 0.04 \end{pmatrix} = \begin{pmatrix} 1.15 & 0.40 & -0.34 \\ 1.10 & 0.46 & 0.51 \\ 1.16 & 0.47 & -0.33 \\ 1.43 & 0.09 & -0.48 \\ 0.96 & 0.11 & -0.36 \end{pmatrix}$$

$$\begin{pmatrix} 0.54 & 0.28 & 0.15 \\ 0.56 & 0.31 & 0.14 \\ 0.58 & 0.29 & 0.13 \\ 0.73 & 0.16 & 0.11 \\ 0.59 & 0.25 & 0.16 \end{pmatrix} \begin{pmatrix} 0.54 & 0.28 & 0.15 \\ 0.56 & 0.31 & 0.14 \\ 0.58 & 0.29 & 0.13 \\ 0.73 & 0.16 & 0.11 \\ 0.59 & 0.25 & 0.16 \end{pmatrix} = \begin{pmatrix} 1.15 & 0.40 & -0.34 \\ 1.10 & 0.46 & 0.51 \\ 1.16 & 0.47 & -0.33 \\ 1.43 & 0.09 & -0.48 \\ 0.96 & 0.11 & -0.36 \end{pmatrix}$$

$$\text{Classification error} = \sum I(D \neq Y) = 4$$

$$\text{Entropy } J = -\sum p_i \log(f(u_{p_i}))$$

$$= -(\log(0.59) + \log(0.54) + \log(0.31) + \log(0.29) + \log(0.11) + \log(0.16))$$

$$= 7.63$$

$$\begin{pmatrix} 1.21 & 0.16 & 0.19 \\ 0.22 & 0.24 & 0.20 \\ 0.04 & 0.01 & 0.01 \\ 0.14 & 0.16 & 0.09 \\ 0.03 & 0.18 & 0.15 \\ 0.06 & 0.14 & 0.04 \end{pmatrix}$$

$$\begin{pmatrix} -0.41 & 0.28 & 0.13 \\ -0.46 & 0.31 & 0.15 \\ -0.56 & 0.31 & 0.14 \\ -0.58 & -0.69 & 0.14 \\ -0.53 & -0.71 & 0.13 \\ -0.73 & 0.16 & -0.09 \end{pmatrix}$$

$$\begin{pmatrix} -0.41 & 0.28 & 0.13 \\ -0.46 & 0.31 & 0.15 \\ -0.56 & 0.31 & 0.14 \\ -0.58 & -0.69 & 0.14 \\ -0.53 & -0.71 & 0.13 \\ -0.73 & 0.16 & -0.09 \end{pmatrix} \begin{pmatrix} 1.21 & 0.16 & 0.19 \\ 0.22 & 0.24 & 0.20 \\ 0.04 & 0.01 & 0.01 \\ 0.14 & 0.16 & 0.09 \\ 0.03 & 0.18 & 0.15 \\ 0.06 & 0.14 & 0.04 \end{pmatrix}$$

$$\begin{pmatrix} -0.09 & -0.08 & 0.03 \\ -0.07 & -0.13 & 0.01 \\ 0.02 & 0.01 & -0.01 \\ 0.07 & 0.04 & -0.05 \\ 0.20 & 0.13 & 0.09 \\ 0.08 & 0.02 & 0.02 \end{pmatrix}$$

$$\begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix}$$

$$\begin{pmatrix} -0.16 \\ 0.04 \\ 0.12 \end{pmatrix}$$

$$\begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix} \begin{pmatrix} -0.16 \\ 0.04 \\ 0.12 \end{pmatrix}$$

$$\begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix} = \begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix}$$

$$\begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix} \begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix} = \begin{pmatrix} 0.92 & 0.05 & -0.26 \\ 0.68 & -0.36 & -0.19 \\ 0.22 & 1.05 & -0.10 \end{pmatrix}$$

Autoencoder

Parameters:

- model: This is the autoencoder model to be trained.
- train_loader: The data loader that provides batches of training data.
- optimalizer: The optimization algorithm used to update the weights of the model.
- step: An integer value that indicates the current training step or epoch.
- device: The device (CPU or GPU) on which the model will be trained.

Function Overview

- The function sets the model to training mode using `model.train()`.
- It initializes `train_loss` to keep track of the cumulative loss for the epoch.
- The function then iterates over the `train_loader`, feeding batches of data and corresponding targets.
- It moves the data and targets to the specified device.
- Depending on the value of `step`, it executes a different part of the model:
 - Step 1:** Train the encoder and the first part of the decoder, optimizing the reconstruction loss between the input and its first stage reconstruction y_1 .
 - Step 2:** Continues training by optimizing the reconstruction loss between the first hidden representation H_1 and its reconstruction y_2 .
 - Step 3:** Trains the classifier part of the model, optimizing the classification loss between the final output y_3 and the target labels.
- For each step, the function:
 - Cleans the gradients of the optimizer.
 - Computes the loss using a pre-defined loss function specific to each step (`loss_fn` for steps 1 and 2, `loss_class` for step 3).
 - Accumulates the loss to `train_loss`.
 - Performs backpropagation using `loss.backward()`.
 - Updates the model parameters using `optimizer.step()`.

Return Value:

- The function returns the average training loss for the epoch, which is the cumulative `train_loss` divided by the total number of items in the dataset.

By dividing the training process into steps, the model can first learn to encode and decode the input before focusing on the classification task, which can sometimes lead to better generalization and learning.

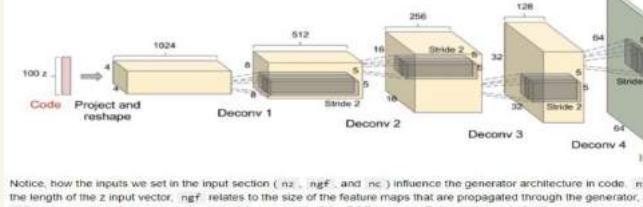
Generally early stopping helps prevent overfitting by stopping the training process before the model starts to memorize the training data too closely. In this case, the early stopping occurred at early stage when the model has not converged on the training data yet. Training Loss is lower than Test Loss and Training Accuracy is higher than Test Accuracy inferring that model's performance on unseen data (a test set) is worse than its performance on the training data, this represents overfitting of the model. We should increase the patience parameter on early stopping to see if the model is going to converge further.

What is a DCGAN?

A DCGAN is a direct extension of the GAN described above, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It was first described by Radford et al. in the paper [Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks](#). The discriminator is made up of *strided convolution* layers, *batch_norm* layers, and LeakyReLU activations. The input is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of *convolutional-transpose* layers, *batch_norm* layers, and *ReLU* activations. The input is a latent vector, z , that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. In the paper, the authors also give some tips about how to setup the optimizers, how to calculate the loss functions, and how to initialize the model weights, all of which will be explained in the coming sections.

Generator

The generator, G , is designed to map the latent space vector (z) to data-space. Since our data are images, converting z to data-space means ultimately creating a RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1]$. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below:



Notice, how the inputs we set in the input section (`nz`, `ngf`, and `nc`) influence the generator architecture in code. `nz` is the length of the z input vector. `ngf` relates to the size of the feature maps that are propagated through the generator, and `nc` is the number of channels in the output image (set to 3 for RGB images). Below is the code for the generator.

Discriminator

As mentioned, the discriminator, D , is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and LeakyReLU layers, and outputs the final probability through a Sigmoid activation function. This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, BatchNorm, and LeakyReLU. The DCGAN paper mentions it is a good practice to use strided convolution rather than pooling to downsample because it lets the network learn its own pooling function. Also batch norm and leaky relu functions promote healthy gradient flow which is critical for the learning process of both G and D .

Loss Functions and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (`BCELoss`) function which is defined in PyTorch as:

$$\mathcal{L}(x, y) = L = (I_1, \dots, I_N)^T, \quad I_n = -[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)]$$

Notice how this function provides the calculation of both log components in the objective function (i.e. $\log(D(x))$ and $\log(1 - D(G(z)))$). We can specify what part of the BCE equation to use with the `y` input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing `y` (i.e. GT labels).

Next, we define our real label as 1 and the fake label as 0. These labels will be used when calculating the losses of D and G and this is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for D and one for G . As specified in the DCGAN paper, both are Adam optimizers with learning rate 0.0002 and Beta1 = 0.5. For keeping track of the generator's learning progression, we will generate a fixed batch of latent vectors that are drawn from a Gaussian distribution (i.e. `fixed_noise`). In the training loop, we will periodically input this `fixed_noise` into G , and over the iterations we will see images form out of the noise.

Scaled Dot Product Attention

The Scaled Dot Product Attention mechanism is a core component of the Transformer architecture, which has been influential in various NLP tasks. This mechanism computes attention scores based on the dot product of the query and key matrices, and then scales the scores to stabilize gradients, especially in deeper models or larger embeddings.

Steps:

- Compute Dimension (d_k):**
 - d_k represents the dimension of the key vectors.
 - It's obtained from the last dimension of the $x_{[k]}$ tensor.
- Calculate Attention Scores (attn_scores):**
 - The attention scores are computed by taking the dot product of the query matrix, Q , and the transposed key matrix, K .
 - In the context of the attention mechanism, the matrices Q , K , and V typically have the shape (batch_size, sequence_length, d_k), where d_k represents the dimension of the key (and also the query) vectors.
 - The operation $Q \cdot K^T$, where K^T would attempt to multiply the last dimension of K with the second-to-last dimension of Q . However, this isn't the desired behavior for the dot product. By transposing the last dimension of K using `K.transpose(-1, -2)`, we modify its shape to (batch_size, d_k , sequence_length). Now, executing `matmul(Q, K.transpose(-1, -2))` multiplies the last dimension of Q (which is d_k) with the second-to-last dimension of the transposed K (also d_k), yielding the desired shape (batch_size, sequence_length, sequence_length).
 - Why -2 and -3 in PyTorch: negative indices for dimension count from the last dimension backward. Specifically, -1 refers to the last dimension, and -2 pertains to the second-to-last dimension. Utilizing negative indices in this context ensure that the code remains general and adaptable to tensors with varying numbers of dimensions.
 - The scores are then scaled down by dividing by the square root of d_k . This scaling helps in stabilizing the gradients, especially when the dimensions of the key vectors are large.
- Compute Attention Weights (attention_weights):**
 - The scaled attention scores pass through a softmax function along the last dimension to produce the attention weight. This ensures that the weights are normalized and sum up to 1 for each query.
- Compute Attention Output ($\text{attn}[i, \text{out}]$):**
 - The attention output is computed by taking the dot product of the attention weights and the value matrix, V . This step essentially takes a weighted sum of the values based on the attention weights, giving more importance to values that are more relevant to the query.

The resulting `attention_out` tensor provides a context-aware representation of the input, emphasizing the most relevant parts of the input for the current query.

Defining the Custom Loss Function

This code defines a custom loss function for an autoencoder that incorporates both Mean Squared Error (MSE) and spatial constraints using Kullback-Leibler (KL) divergence. Here's a detailed breakdown:

Parameters:

- x : The activations from the hidden layer of the autoencoder.
- y : The reconstructed output from the autoencoder.
- $original$: The original input data to the autoencoder.

Mean Squared Error (MSE) Loss:

- The MSE loss measures the difference between the reconstructed output, y , and the original input, $original$.
- `mean_loss(y, original, reduction='sum')` computes the sum of squared differences between the two tensors.
- The result is normalized by the batch size $y.shape[0]$ to get the average loss per data point in the batch.

Spatial Loss:

- The goal of the spatial loss is to ensure that the activations in the hidden layer, x , are sparse, meaning most of the activations are close to zero.
- The desired average activation level is set to `rho`, which is typically a small value (e.g. 0.05).
- `sparsity_target` is a tensor filled with the value of `rho` and has the same size as the hidden layer.
- `sparsity_real` calculates the mean activation level across the batch for each neuron in the hidden layer.
- `KL divergence`, `F1_kl_div(sparsity_target, sparsity_real, reduction='batchwise')`, measures the difference between the desired (sparsity_target) and actual (sparsity_real) activation distributions. It provides a measure how much the real activations deviate from the desired level of sparsity.
- The result of the KL divergence for each neuron is summed up to get the total `sparsity_loss`.

Total Loss:

- The final loss is a combination of the MSE loss and the sparsity loss.
- The sparsity loss is scaled by a factor of 0.5 (or any other desired value) to control its contribution relative to the MSE loss.
- The combined loss ensures that the autoencoder not only reconstructs the input data accurately but also maintains a sparse representation in the hidden layer.

`dataset_root` - the path to the root of the dataset folder. We will talk more about the dataset in the next section.

`workers` - the number of worker threads for loading the data with the `DataLoader`.

`batch_size` - the batch size used in training. The DCGAN paper uses a batch size of 128.

`image_size` - the spatial size of the images used for training. This implementation defaults to 64x64. If another size is desired, the structures of D and G must be changed. See [here](#) for more details.

`nc` - number of color channels in the input images. For color images this is 3.

`nz` - length of latent vector.

`ngf` - relates to the depth of feature maps carried through the generator.

`nrf` - sets the depth of feature maps propagated through the discriminator.

`num_epochs` - number of training epochs to run. Training for longer will probably lead to better results but will also take much longer.

`lr` - learning rate for training. As described in the DCGAN paper, this number should be 0.0002.

`beta1` - beta1 hyperparameter for Adam optimizers. As described in paper, this number should be 0.5.

`ngpu` - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs.

Training

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from the [Goodfellow's paper](#), while abiding by some of the best practices shown in [pytorch-GAN](#). Namely, we will construct different mini-batches for the real and fake images, and also adjust G's objective function to maximize $\log(D(G(z)))$. Training is split up into two main parts: Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1 - Train the Discriminator

Recall the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$. Due to the separate mini-batch suggestion from [pytorch-GAN](#), we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss ($\log(D(x))$), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss ($\log(1 - D(G(z)))$), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

Part 2 - Train the Generator

As stated in the original paper, we want to train the Generator by minimizing $\log(1 - D(G(z)))$ in an effort to generate better fakes. As mentioned, this was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize $\log(D(G(z)))$. In the code we accomplish this by: classifying the Generator output from Part 1 with the discriminator, computing G's loss using `G7`, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels as GT labels for the loss function, but this allows us to use the `log(x)` part of the `BCELoss` (rather than the `log(1 - x)` part) which is exactly what we want.

Finally, we will do some statistic reporting and at the end of each epoch we will push our `fixed_noise` batch through the generator to visually track the progress of G's training. The training statistics report are:

- `Loss_D` - discriminator loss calculated as the sum of losses for all the real and all fake batches ($\log(D(x)) + \log(1 - D(G(z)))$).
- `Loss_G` - generator loss calculated as $\log(D(G(z)))$.
- `D(x)` - average output (across the batch) of the discriminator for all the real batch. This should start close to 1 then theoretically converge to 0.5 when G gets better. Think about why this.
- `D(Gt)` - average discriminator outputs for all the fake batch. The first number is before D is updated and the second number is after D is updated. These numbers should start near 0 and converge to 0.5 as G gets better. Think about why this.

Note: This step might take a while, depending on how many epochs you run and if you removed some data from the dataset.

2. Regression n Classification

Regression and classification



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

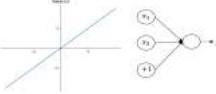
Prediction tasks can be classified into two categories:

Regression: the labels are continuous (age, income, height, etc.)

Classification: the labels are discrete (yes, digit, type of flower, etc.).

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

Linear neuron



Synaptic input u to a neuron is given by

$$u = w^T x + b$$

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

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

A linear neuron with weights $w = (w_1, w_2, \dots, w_n)^T$ and bias b has an output:

$$y = w^T x + b$$

where input $x = (x_1, x_2, \dots, x_n)^T \in R^n$ and output $y \in R$.

Linear neuron performs linear regression

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

$$\begin{aligned} & y = w^T x_1 + b \rightarrow \text{line} \\ & Y = w^T X + b \rightarrow \text{Plane} \end{aligned}$$

The output of a linear neuron can be written

$$y = w^T x_1 + w_2 x_2 + \dots + w_n x_n + b$$

where x_1, x_2, \dots, x_n are the inputs. That is, a linear neuron performs linear regression and the weights and biases (that is, w and b) w_n act as regression coefficients. The above function forms a hyperplane in Euclidean space R^n .

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

$$y = w^T x + b$$

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 (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 x and d is the target label.

$$y = w^T x + b$$

The $\frac{1}{2}$ 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

$$u = w^T x + b = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

$$\frac{\partial u}{\partial w} = \begin{pmatrix} \frac{\partial u}{\partial w_1} \\ \vdots \\ \frac{\partial u}{\partial w_n} \end{pmatrix} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = x$$

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

$$y = w^T x + b$$

<math display

Perceptron

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

$$y = f(u)$$

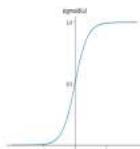
Where:

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

And $y = w^T x + b$

The square error is used as cost function for learning.

Perceptron performs a one-layer regression of inputs.



SGD for perceptron



Cost function J is given by

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

where $y = f(u)$ and $u = w^T x + b$

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

The gradient with respect to the synaptic input:

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

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

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

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

$$\nabla_b J = \frac{\partial J}{\partial b} = -(d - y) f'(u) 1$$

Gradient learning equations:

$$w \leftarrow w - \alpha \nabla_w J$$

$$b \leftarrow b - \alpha \nabla_b J$$

Substituting gradients from (1) and (2), SGD equations for a perceptron are given by

$$w \leftarrow w + \alpha(d - y)f'(u)x$$

$$b \leftarrow b + \alpha(d - y)f'(u)$$

SGD algorithm for perceptron

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

Set learning parameter α

Initialize w and b

Repeat until convergence:

For every training pattern (x_p, d_p) :

$$u_p = w^T x_p + b$$

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

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

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

GD for perceptron

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

$$j = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2 = \sum_{p=1}^P j_p \quad (3)$$

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

From (3):

$$\nabla_w J = \sum_{p=1}^P \nabla_w j_p$$

$$= - \sum_{p=1}^P (d_p - y_p)f'(u_p)x_p \quad \text{From (1)}$$

$$= -[(d_1 - y_1)f'(u_1)x_1 + (d_2 - y_2)f'(u_2)x_2 + \dots + (d_P - y_P)f'(u_P)x_P]$$

$$= -(x_1 \cdot x_2 \cdots x_P) \cdot \begin{pmatrix} (d_1 - y_1)f'(u_1) \\ (d_2 - y_2)f'(u_2) \\ \vdots \\ (d_P - y_P)f'(u_P) \end{pmatrix}$$

$$= -x^T (d - y) \cdot f'(u) \quad \text{Important! Use element-wise product}$$

$$\text{where } X = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_P^T \end{pmatrix}, d = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_P \end{pmatrix}, y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_P \end{pmatrix}, \text{ and } f'(u) = \begin{pmatrix} f'(u_1) \\ f'(u_2) \\ \vdots \\ f'(u_P) \end{pmatrix}$$

Substituting X^T by 1_p^T in (K), we get

$$\nabla_b J = -1_p^T (d - y) \cdot f'(u)$$

where $1_p = (1 \ 1 \ \dots \ 1)^T$.

The gradient descent learning is given by

$$w \leftarrow w - \alpha \nabla_w J$$

$$b \leftarrow b - \alpha \nabla_b J$$

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

$$w \leftarrow w + \alpha X^T (d - y) \cdot f'(u)$$

$$b \leftarrow b + \alpha 1_p^T (d - y) \cdot f'(u)$$

Note that \cdot is the element-wise product.

Given a training dataset $\{(X, d)\}$

Set learning parameter α

Initialize w and b

Repeat until convergence:

$$u = Xw + b1_p$$

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

$$w \leftarrow w + \alpha X^T (d - y) \cdot f'(u)$$

$$b \leftarrow b + \alpha 1_p^T (d - y) \cdot f'(u)$$

Gradient descent for perceptron

GD	SGD
(X, d)	(x_p, d_p)
$J = \frac{1}{2} \sum_{p=1}^P (d_p - y_p)^2$	$J = \frac{1}{2} (d_p - y_p)^2$
$u = Xw + b1_p$	$u_p = x_p^T w + b$
$y = f(u)$	$y_p = f(u_p)$
$w = w + \alpha X^T (d - y) \cdot f'(u)$	$w = w + \alpha (d_p - y_p)f'(u_p)x_p$
$b = b + \alpha 1_p^T (d - y) \cdot f'(u)$	$b = b + \alpha (d_p - y_p)f'(u_p)$

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

$$y = f(u) = \frac{1}{1+e^{-u}} \quad \text{exam! need!}$$

The derivative is given by

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

For Tanh function (bipolar sigmoid):

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

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

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

$$= (0.81) + 0.01 \begin{pmatrix} 0.77 & 0.63 & 0.50 & 0.20 & 0.17 & 0.69 & 0.00 \end{pmatrix} \begin{pmatrix} 0.55 \\ 1.28 \\ 1.53 \\ 0.93 \\ 0.88 \\ 1.19 \\ -0.74 \end{pmatrix} = \begin{pmatrix} 0.80 \\ 0.64 \end{pmatrix}$$

$$b = b + \alpha 1_p^T (d - y) \cdot f'(u)$$

$$= 0.0 + 0.01 \times (1.0 \ 1.0 \ 1.0 \ 1.0 \ 1.0 \ 1.0) \begin{pmatrix} 2.91 \\ 1.55 \\ 1.28 \\ 1.53 \\ 0.93 \\ 0.88 \end{pmatrix} = -0.05$$

$$\text{iter} \quad u \quad y \quad f'(u) \quad mse \quad w \quad b$$

1	$\begin{pmatrix} 0.69 \\ 0.97 \\ 0.54 \\ 1.30 \\ 1.00 \\ 0.93 \end{pmatrix}$	$\begin{pmatrix} 1.81 \\ 1.30 \\ 1.00 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmatrix}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmatrix}$	2.11	$\begin{pmatrix} 0.80 \\ 0.57 \end{pmatrix}$	-0.05
2	$\begin{pmatrix} 0.57 \\ 1.56 \\ 0.92 \\ 1.86 \\ 1.00 \\ 0.95 \end{pmatrix}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmatrix}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmatrix}$	0.79	$\begin{pmatrix} 0.80 \\ 0.52 \end{pmatrix}$	-0.11
3	$\begin{pmatrix} 0.45 \\ 1.56 \\ 0.89 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmatrix}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmatrix}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmatrix}$	0.47	$\begin{pmatrix} 0.80 \\ 0.47 \end{pmatrix}$	-0.17
4	$\begin{pmatrix} 0.33 \\ 1.56 \\ 0.77 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmatrix}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmatrix}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmatrix}$	0.24	$\begin{pmatrix} 0.80 \\ 0.43 \end{pmatrix}$	-0.23
5	$\begin{pmatrix} 0.21 \\ 1.56 \\ 0.65 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmatrix}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmatrix}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmatrix}$	0.12	$\begin{pmatrix} 0.80 \\ 0.41 \end{pmatrix}$	-0.29
6	$\begin{pmatrix} 0.09 \\ 1.56 \\ 0.53 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmatrix}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmatrix}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmatrix}$	0.06	$\begin{pmatrix} 0.80 \\ 0.39 \end{pmatrix}$	-0.35
7	$\begin{pmatrix} -0.01 \\ 1.56 \\ 0.41 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.03	$\begin{pmatrix} 0.80 \\ 0.37 \end{pmath}$	-0.41
8	$\begin{pmatrix} 0.11 \\ 1.56 \\ 0.29 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.02	$\begin{pmatrix} 0.80 \\ 0.36 \end{pmath}$	-0.46
9	$\begin{pmatrix} 0.29 \\ 1.56 \\ 0.17 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.01	$\begin{pmatrix} 0.80 \\ 0.35 \end{pmath}$	-0.51
10	$\begin{pmatrix} 0.47 \\ 1.56 \\ 0.05 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.005	$\begin{pmatrix} 0.80 \\ 0.34 \end{pmath}$	-0.56
11	$\begin{pmatrix} 0.65 \\ 1.56 \\ -0.13 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.002	$\begin{pmatrix} 0.80 \\ 0.33 \end{pmath}$	-0.61
12	$\begin{pmatrix} 0.83 \\ 1.56 \\ -0.31 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.001	$\begin{pmatrix} 0.80 \\ 0.32 \end{pmath}$	-0.66
13	$\begin{pmatrix} 0.01 \\ 1.56 \\ -0.49 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.0005	$\begin{pmatrix} 0.80 \\ 0.31 \end{pmath}$	-0.71
14	$\begin{pmatrix} -0.19 \\ 1.56 \\ -0.67 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.0002	$\begin{pmatrix} 0.80 \\ 0.30 \end{pmath}$	-0.76
15	$\begin{pmatrix} -0.37 \\ 1.56 \\ -0.85 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.0001	$\begin{pmatrix} 0.80 \\ 0.29 \end{pmath}$	-0.81
16	$\begin{pmatrix} -0.55 \\ 1.56 \\ -1.03 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.00005	$\begin{pmatrix} 0.80 \\ 0.28 \end{pmath}$	-0.86
17	$\begin{pmatrix} -0.73 \\ 1.56 \\ -1.21 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.00002	$\begin{pmatrix} 0.80 \\ 0.27 \end{pmath}$	-0.91
18	$\begin{pmatrix} -0.91 \\ 1.56 \\ -1.39 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.00001	$\begin{pmatrix} 0.80 \\ 0.26 \end{pmath}$	-0.96
19	$\begin{pmatrix} -0.09 \\ 1.56 \\ -1.57 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.000005	$\begin{pmatrix} 0.80 \\ 0.25 \end{pmath}$	-0.10
20	$\begin{pmatrix} 0.11 \\ 1.56 \\ -1.75 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.000002	$\begin{pmatrix} 0.80 \\ 0.24 \end{pmath}$	-0.15
21	$\begin{pmatrix} 0.29 \\ 1.56 \\ -1.93 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.000001	$\begin{pmatrix} 0.80 \\ 0.23 \end{pmath}$	-0.20
22	$\begin{pmatrix} 0.47 \\ 1.56 \\ -2.11 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.0000005	$\begin{pmatrix} 0.80 \\ 0.22 \end{pmath}$	-0.25
23	$\begin{pmatrix} 0.65 \\ 1.56 \\ -2.29 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.0000002	$\begin{pmatrix} 0.80 \\ 0.21 \end{pmath}$	-0.30
24	$\begin{pmatrix} 0.83 \\ 1.56 \\ -2.47 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.0000001	$\begin{pmatrix} 0.80 \\ 0.20 \end{pmath}$	-0.35
25	$\begin{pmatrix} 0.01 \\ 1.56 \\ -2.65 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.00000005	$\begin{pmatrix} 0.80 \\ 0.19 \end{pmath}$	-0.40
26	$\begin{pmatrix} -0.19 \\ 1.56 \\ -2.83 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.00000002	$\begin{pmatrix} 0.80 \\ 0.18 \end{pmath}$	-0.45
27	$\begin{pmatrix} -0.37 \\ 1.56 \\ -3.01 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.00000001	$\begin{pmatrix} 0.80 \\ 0.17 \end{pmath}$	-0.50
28	$\begin{pmatrix} -0.55 \\ 1.56 \\ -3.19 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.000000005	$\begin{pmatrix} 0.80 \\ 0.16 \end{pmath}$	-0.55
29	$\begin{pmatrix} -0.73 \\ 1.56 \\ -3.37 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.000000002	$\begin{pmatrix} 0.80 \\ 0.15 \end{pmath}$	-0.60
30	$\begin{pmatrix} -0.91 \\ 1.56 \\ -3.55 \\ 1.86 \\ 0.99 \\ 0.95 \end{pmath}$	$\begin{pmatrix} 1.81 \\ 1.83 \\ 0.93 \\ 1.00 \\ 1.00 \\ 1.00 \end{pmath}$	$\begin{pmatrix} 0.98 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \\ 0.90 \end{pmath}$	0.000000001	$\begin{pmatrix} 0.80 \\ 0.14 \end{pmath}$	-0.65

Example 2

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

$x = (x_1, x_2)$	d

</tbl_r

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**

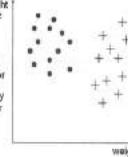


Figure 1: A 2-dimensional feature space

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

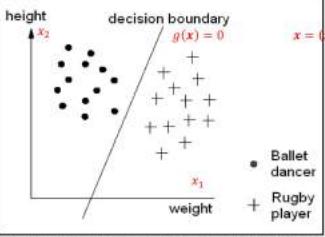


Figure 2: A linear classification decision boundary

Decision boundary

The **decision boundary** of the classifier, $g(\mathbf{x}) = 0$ where function $g(\mathbf{x})$ 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 $g(\mathbf{x}) > 0 \Rightarrow$ Ballet dancer

If $g(\mathbf{x}) \leq 0 \Rightarrow$ 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), $\mathbf{x} = (x_1 \ x_2 \ \dots \ x_n)^T$. A linear description function is given by

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

where $\mathbf{w} = (w_1 \ w_2 \ \dots \ w_n)^T$ 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

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

And with a threshold activation function $1(\cdot)$:

$$u = g(\mathbf{x}) > 0 \rightarrow y = 1 \rightarrow \text{class 1}$$

$$u = g(\mathbf{x}) \leq 0 \rightarrow y = 0 \rightarrow \text{class 2}$$

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.

not differentiable

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

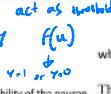
Logistic regression neuron

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

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

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

where $u = \mathbf{w}^T \mathbf{x} + b$ 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'.

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

Then

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

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

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

When $u = 0$, $f(u) = P(y = 1|\mathbf{x}) = P(y = 0|\mathbf{x}) = 0.5$. 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 (\mathbf{x}, d) where $\mathbf{x} \in R^n$ and $d \in \{0, 1\}$.

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

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

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

$$J = -\log(f(u)) \quad \text{if } d = 1 \\ J = -\log(1-f(u)) \quad \text{if } d = 0$$

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

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

Gradient with respect to u :

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

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

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

$$\nabla_b J = \frac{\partial J}{\partial b} = -(\mathbf{d} - f(u))$$

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

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

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

Set learning rate α

Initialize \mathbf{w} and b

Iterate until convergence:

For every pattern (\mathbf{x}_p, d_p) :

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

GD for logistic regression neuron

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

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

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

The cost function J can be written as

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

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

$$\begin{aligned} \nabla_{\mathbf{w}} J &= \sum_{p=1}^P \nabla_{\mathbf{w}} J_p \\ &= -\sum_{p=1}^P (d_p - f(u_p)) \mathbf{x}_p \\ &= -(d_1 - f(u_1)) \mathbf{x}_1 + (d_2 - f(u_2)) \mathbf{x}_2 + \dots + (d_P - f(u_P)) \mathbf{x}_P \\ &= -(x_1 \ x_2 \ \dots \ x_P) \begin{pmatrix} (d_1 - f(u_1)) \\ (d_2 - f(u_2)) \\ \vdots \\ (d_P - f(u_P)) \end{pmatrix} \\ &= -\mathbf{x}^T (\mathbf{d} - f(\mathbf{u})) \end{aligned} \quad \text{From (A)}$$

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

By substituting $\mathbf{1}_P$ for \mathbf{x} in above equation:

$$\nabla_b J = -\mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u}))$$

Substituting the gradients in

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \\ b &\leftarrow b - \alpha \nabla_b \end{aligned}$$

the gradient descent learning for logistic regression neuron is given by

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} + \alpha \mathbf{X}^T (\mathbf{d} - f(\mathbf{u})) \\ b &\leftarrow b + \alpha \mathbf{1}_P^T (\mathbf{d} - f(\mathbf{u})) \end{aligned}$$

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

Given training data (\mathbf{X}, \mathbf{d})

Set learning rate α

Initialize \mathbf{w} and b

Iterate until convergence:

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

Learning for logistic regression neuron

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

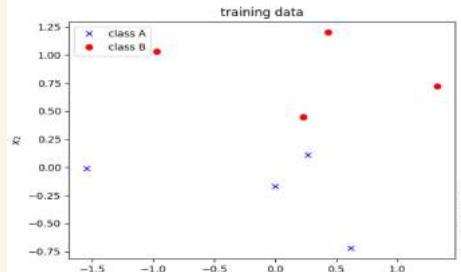
$\mathbf{G} \rightarrow \mathbf{m} \rightarrow \mathbf{matrix}$

Example 3: GD for logistic regression neuron

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

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

User a learning factor $\alpha = 0.04$.



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

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

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

Epoch 1:

$$u = Xw + b = \begin{pmatrix} 1.33 & 0.72 \\ -1.55 & 0.01 \\ 0.62 & -0.72 \\ 0.27 & 0.11 \\ 0.0 & -0.17 \\ 0.43 & 1.2 \\ -0.97 & 1.03 \\ 0.23 & 0.45 \end{pmatrix} \begin{pmatrix} 0.77 \\ 0.02 \end{pmatrix} + 0.0 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1.04 \\ -1.2 \\ 0.46 \\ 0.21 \\ 0.00 \\ 0.36 \\ -0.73 \\ 0.19 \end{pmatrix}$$

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

205?

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

Y5 mismatch

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

blue arrow

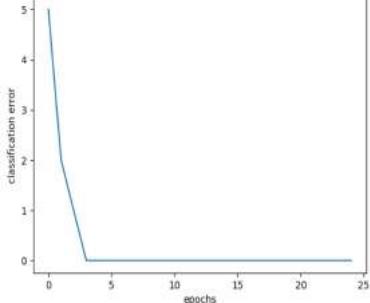
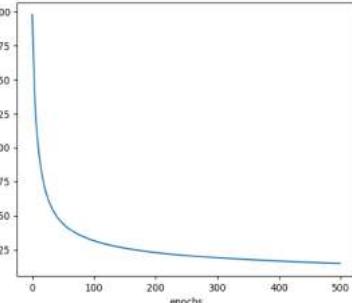
$$\begin{aligned} \text{Cross-entropy} &= -\sum_{p=1}^8 d_p \log(f(u_p)) + (1-d_p) \log(1-f(u_p)) \\ &= -\log(1-f(u_1)) - \log(f(u_2)) - \log(f(u_3)) - \dots - \log(1-f(u_8)) \\ &= -\log(1-0.74) - \log(0.23) - \log(0.61) - \dots - \log(1-0.55) \\ &= 6.653 \end{aligned}$$

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

$$\begin{aligned} &= (0.77) + 0.04 \begin{pmatrix} 1.33 & -1.55 & 0.62 & 0.27 & 0 & 0.43 & -0.97 & 0.23 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\ &= (0.69) \end{aligned}$$

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

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



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

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

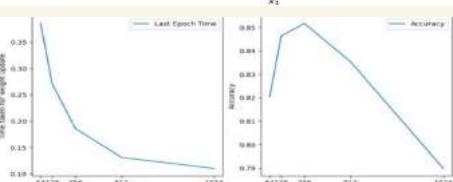
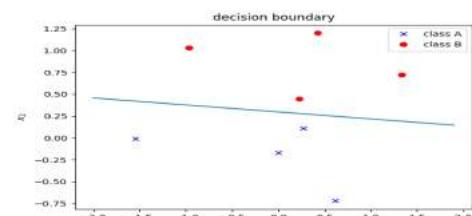
$$(x_1 \ x_2)^T \begin{pmatrix} -1.20 \\ -15.02 \end{pmatrix} + 4.47 = 0$$

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

Decision boundary:

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

if $x_1 = 0 \ x_2 = 0.3$
if $x_2 = 0 \ x_1 = 3.73$



Reason:

Out of the 8 batches given, we can see that time taken doesn't improve much after "batch_size 512" (at the most optimal time) and we can see that accuracy is still quite high before it starts to decrease further as shown in the graph, hence we can conclude that batch_size 512 is the elbow point after which time is less and not too low accuracy (optimal point for both time and accuracy) which is why we choose batch_size 512 as it represents the good balance between accuracy and time taken with the current machines GPU.

Analysis: based on graph :

The accuracy improves steadily as the batch size increases from 64 to 256 (peak). The accuracy is high at batch size 512 and decreases as the batch size increases to 1024. The time taken also increases as the batch size increases, but the decrease in training time between batch size 512 and 1024 is less significant.

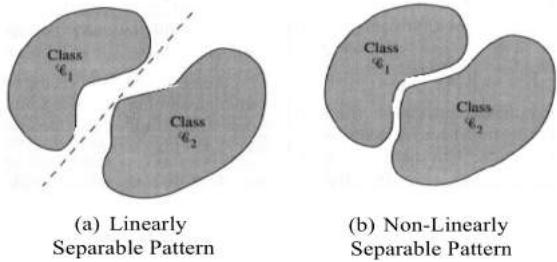
Based on these observations, the elbow point could be around batch size 512. Beyond this point, the increase in batch size may not result in significant improvements in accuracy while the decrease in training time is less pronounced. Therefore, batch size 512 appears to offer a good balance between accuracy and training time in this scenario.

Limitations of logistic regression neuron

Learning rates

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.



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

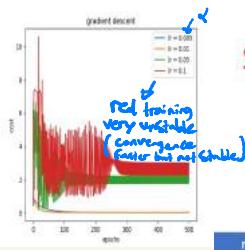
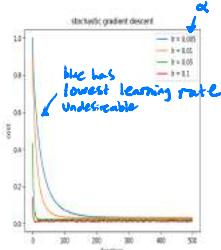
Example 4: effects of leaning rate

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

$x = (x_1, x_2)$	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

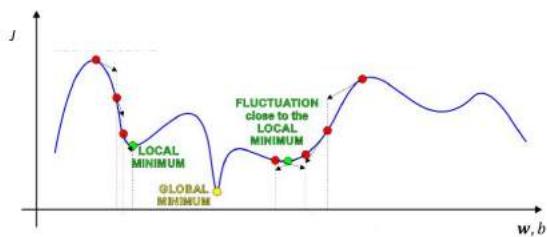
Use learning factor $\alpha = 0.01$.

Example 4: Learning rates with SGD



- 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 converges 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

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

Summary: SGD for neurons

$$(x_p, d_p)$$

$$w = w - \alpha X^T V_w$$

$$b = b - \alpha 1_p^T V_b$$

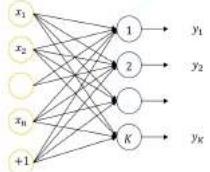
$$(x_p, d_p)$$

$$w = w - \alpha V_w x_p + b$$

$$b = b - \alpha V_b x_p$$

Weight matrix of a layer

Consider a layer of K neurons.



Let \mathbf{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:

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

where columns are given by weight vectors of individual neurons.

And a bias vector \mathbf{b} where each element corresponds to a bias of a neuron:
 $\mathbf{b} = (b_1, b_2, \dots, b_K)^T$

Synaptic input at a layer for single input

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

Synaptic input u_k to k th neuron:

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

\mathbf{w}_k and b_k denote the weight vector and bias of k th neuron.

Synaptic input vector \mathbf{u} to the layer :

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

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

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

Synaptic input at a layer for batch input

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

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

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

The synaptic input matrix \mathbf{U} to the layer for P patterns:

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

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

$$\text{p} \downarrow \text{n} \downarrow \text{m} \downarrow = \text{p} \downarrow \text{n} \downarrow \text{m} \downarrow$$

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

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

Activation at a layer for batch input

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

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

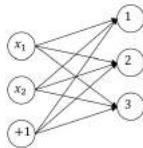
where rows of \mathbf{U} corresponds to synaptic inputs of the layer, corresponding to individual input patterns:

Activation of the layer:

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

where activations due to individual patterns are written as rows.

Example 1: activations and outputs of a perceptron layer

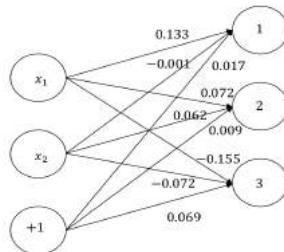


A perceptron layer of 3 neurons shown in the figure receives 2-dimensional inputs $(x_1, x_2)^T$, and has a weight matrix \mathbf{W} and a bias vector \mathbf{b} given by

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

Using batch processing, find the output for input patterns:
 $(0.5, -1.0), (-1.0, 0.78), (-1.66, -0.51), (-0.65, -0.2)$.

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



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

Input as a batch of four patterns:

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

The synaptic input to the layer:

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

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

For a perceptron layer

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

For example, third row corresponding to 3rd input:

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

2nd neuron output for 3rd input pattern

And the corresponding output

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

SGD for single layer

Computational graph processing input (x, d) :



J denotes the cost function.

Need to compute gradients $\nabla_w J$ and $\nabla_b J$ to learn weight matrix W and bias vector b .

Consider k th neuron at the layer:

$$u_k = x^T w_k + b_k$$

And

$$\frac{\partial u_k}{\partial w_k} = x$$

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

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

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

Gradient of J with respect to $W = (w_1 \ w_2 \ \dots \ w_K)$:

$$\begin{aligned} \nabla_w J &= (\nabla_{w_1} J \ \nabla_{w_2} J \ \dots \ \nabla_{w_K} J) \\ &= \left(x \frac{\partial J}{\partial u_1} \ x \frac{\partial J}{\partial u_2} \ \dots \ x \frac{\partial J}{\partial u_K} \right) \quad \text{From (A)} \\ &= x \begin{pmatrix} \frac{\partial J}{\partial u_1} & \frac{\partial J}{\partial u_2} & \dots & \frac{\partial J}{\partial u_K} \end{pmatrix} \\ &= x(\nabla_u J)^T \end{aligned}$$

where

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

That is, $\nabla_w J = x(\nabla_u J)^T$ (C)

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

$$\nabla_b J = \begin{pmatrix} \frac{\partial J}{\partial b_1} \\ \frac{\partial J}{\partial b_2} \\ \vdots \\ \frac{\partial J}{\partial b_K} \end{pmatrix} = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_K} \end{pmatrix} = \nabla_u J \quad (\text{D})$$

$$\nabla_b J = \nabla_u J$$

From (C) and (D),

$$\begin{aligned} \nabla_w J &= x(\nabla_u J)^T \\ \nabla_b J &= \nabla_u J \end{aligned}$$

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

$$\begin{aligned} W &= W - \alpha x(\nabla_u J)^T \\ b &= b - \alpha \nabla_u J \end{aligned}$$

GD for single layer

Given a set of patterns $\{(x_p, d_p)\}_{p=1}^P$ where $x_p \in R^n$ and $d_p \in R^K$ for regression and $d_p \in \{1, 2, \dots, K\}$ 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,

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

Substituting $\nabla_w J_p = x_p(\nabla_{u_p} J_p)^T$ from (C) :

$$\begin{aligned} \nabla_w J &= \sum_{p=1}^P x_p(\nabla_{u_p} J_p)^T \\ &= \sum_{p=1}^P x_p(\nabla_{u_p} J)^T \quad \text{since } \nabla_{u_p} J = \nabla_{u_p} J_p \\ &= x_1(\nabla_{u_1} J)^T + x_2(\nabla_{u_2} J)^T + \dots + x_p(\nabla_{u_p} J)^T \\ &= (x_1 \ x_2 \ \dots \ x_p) \begin{pmatrix} (\nabla_{u_1} J)^T \\ (\nabla_{u_2} J)^T \\ \vdots \\ (\nabla_{u_p} J)^T \end{pmatrix} \\ &= X^T \nabla_u J \end{aligned} \quad (\text{E})$$

Note that $X = \begin{pmatrix} x_1^T \\ x_2^T \\ \vdots \\ x_p^T \end{pmatrix}$ and $U = \begin{pmatrix} u_1^T \\ u_2^T \\ \vdots \\ u_p^T \end{pmatrix}$

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

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

$$\begin{aligned} &= \sum_{p=1}^P \nabla_{u_p} J_p \\ &= \sum_{p=1}^P \nabla_{u_p} J \\ &= \sum_{p=1}^P \nabla_{u_p} J \quad \text{Since } \nabla_{u_p} J = \nabla_{u_p} J_p \\ &= \nabla_{u_1} J + \nabla_{u_2} J + \dots + \nabla_{u_p} J \\ &= (\nabla_{u_1} J \ \nabla_{u_2} J \ \dots \ \nabla_{u_p} J) \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \\ &= (\nabla_u J)^T \mathbf{1}_p \end{aligned} \quad (\text{F})$$

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

From (E) and (F):

$$\begin{aligned} \nabla_w J &= X^T \nabla_u J \\ \nabla_b J &= (\nabla_u J)^T \mathbf{1}_p \end{aligned}$$

That is, by computing gradient $\nabla_u J$ with respect to synaptic input, the weights and biases can be updated.

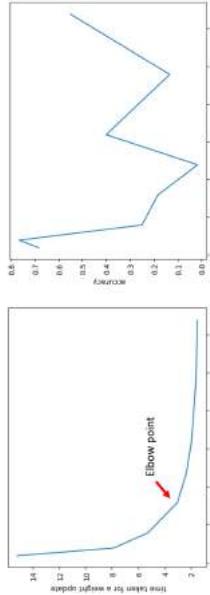
$$\begin{aligned} W &= W - \alpha X^T \nabla_u J \\ b &= b - \alpha (\nabla_u J)^T \mathbf{1}_p \end{aligned}$$

Learning a single layer

Learning a layer of neurons	
SGD	$W = W - \alpha x(\nabla_u J)^T$ $b = b - \alpha \nabla_u J$
GD	$W = W - \alpha X^T \nabla_u J$ $b = b - \alpha (\nabla_u J)^T \mathbf{1}_p$

To learn a given layer, we need to compute $\nabla_u J$ for SGD and $\nabla_w J$ for GD.

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

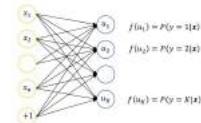


Softmax layer

SGD for softmax layer

GD for softmax layer

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



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

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

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

where u_k is the synaptic input to the k neuron.

The cost function can also be written as

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

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

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

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

The gradient with respect to u_k is given by

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

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 $k \neq d$.

If $k = d$:

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

If $k \neq d$:

$$\begin{aligned} \frac{\partial f(u_d)}{\partial u_k} &= \frac{\partial}{\partial u_k} \left(\frac{e^{u_d}}{\sum_{k'=1}^K e^{u_{k'}}} \right) \\ &= -\frac{e^{u_d} e^{u_k}}{(\sum_{k'=1}^K e^{u_{k'}})^2} \\ &= -f(u_d)f(u_k) \quad 1(k=d) = 0 \\ &= f(u_d)(1(k=d) - f(u_k)) \\ \frac{\partial f(u_d)}{\partial u_k} &= f(u_d)(1(k=d) - f(u_k)) \end{aligned}$$

Substituting in (I):

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

Gradient J with respect to \mathbf{u} :

$$\nabla_u J = \begin{pmatrix} \nabla_{u_1} J \\ \nabla_{u_2} J \\ \vdots \\ \nabla_{u_K} J \end{pmatrix} = -\begin{pmatrix} 1(d=1) - f(u_1) \\ 1(d=2) - f(u_2) \\ \vdots \\ 1(d=K) - f(u_K) \end{pmatrix} = -(1(\mathbf{k}=d) - f(\mathbf{u})) \quad (J)$$

where $\mathbf{k} = (1 \ 2 \ \dots \ K)^T$

For a softmax layer:

$$\nabla_u J = -(1(\mathbf{k}=d) - f(\mathbf{u}))$$

where:

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

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

Given a set of patterns $\{(x_p, d_p)\}_{p=1}^P$, where $x_p \in \mathbb{R}^n$ and $d_p \in \{1, 2, \dots, K\}$.

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

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

where u_{pd_p} is the synaptic input to the k neuron for input x_p .

The cost function J can also be written as

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

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

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

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

$$\nabla_u J = \begin{pmatrix} (\nabla_{u_1} J_p)^T \\ (\nabla_{u_2} J_p)^T \\ \vdots \\ (\nabla_{u_K} J_p)^T \end{pmatrix} = \begin{pmatrix} (\nabla_{u_1} f_p)^T \\ (\nabla_{u_2} f_p)^T \\ \vdots \\ (\nabla_{u_K} f_p)^T \end{pmatrix}$$

Substituting $\nabla_u J = -(1(\mathbf{k}=d) - f(\mathbf{u}))$ from (J):

$$\begin{aligned} \nabla_u J &= -\begin{pmatrix} (1(k=d_1) - f(u_1))^T \\ (1(k=d_2) - f(u_2))^T \\ \vdots \\ (1(k=d_p) - f(u_p))^T \end{pmatrix} \\ \nabla_u J &= -(K - f(\mathbf{u})) \end{aligned}$$

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

Learning a softmax layer

GD	SGD
(\mathbf{X}, \mathbf{D})	(\mathbf{x}, d)
$\mathbf{U} = \mathbf{XW} + \mathbf{B}$	$\mathbf{u} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$
$f(\mathbf{U}) = \frac{e^{\mathbf{u}}}{\sum_{k'=1}^K e^{\mathbf{u}_{k'}}}$	$f(\mathbf{u}) = \frac{e^{\mathbf{u}_d}}{\sum_{k'=1}^K e^{\mathbf{u}_{k'}}}$
$\mathbf{y} = \operatorname{argmax}_k f(\mathbf{U})$	$\mathbf{y} = \operatorname{argmax}_k f(\mathbf{u})$
$\nabla_{\mathbf{U}} J = -(K - f(\mathbf{U}))$	$\nabla_{\mathbf{u}} J = -(1(k=d) - f(\mathbf{u}))$
$\mathbf{W} = \mathbf{W} - \alpha \mathbf{X}^T \nabla_{\mathbf{U}} J$	$\mathbf{W} = \mathbf{W} - \alpha \mathbf{x}^T \nabla_{\mathbf{u}} J$
$\mathbf{b} = \mathbf{b} - \alpha (\nabla_{\mathbf{U}} J)^T \mathbf{1}_P$	$\mathbf{b} = \mathbf{b} - \alpha \nabla_{\mathbf{u}} J$

from torch import nn

```
class SoftmaxLayer(nn.Module):
    def __init__(self, no_inputs, no_outputs):
        super().__init__()
        self.softmax_layer = nn.Sequential(
            nn.Linear(no_inputs, no_outputs), # applies a linear transformation y = w^T x + b
            nn.Softmax(dim=1) # implements softmax; sum up across rows to 1.0
        )
    def forward(self, x):
        logits = self.softmax_layer(x)
        return logits
```

Example 2: GD of a softmax layer

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

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

Use a learning factor $\alpha = 0.05$.

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

$$X = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.20 \end{pmatrix}, d = \begin{pmatrix} 1 \\ 2 \\ 2 \\ 1 \\ 3 \\ 2 \\ 1 \\ 3 \end{pmatrix}$$

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

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

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

1st epoch starts ...

$$U = XW + B = \begin{pmatrix} 0.94 & 0.18 \\ -0.58 & -0.53 \\ -0.23 & -0.31 \\ 0.42 & -0.44 \\ 0.5 & -1.66 \\ -1.0 & -0.51 \\ 0.78 & -0.65 \\ 0.04 & -0.20 \end{pmatrix} \begin{pmatrix} 0.77 & 0.02 & 0.63 \\ 0.75 & 0.50 & 0.23 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

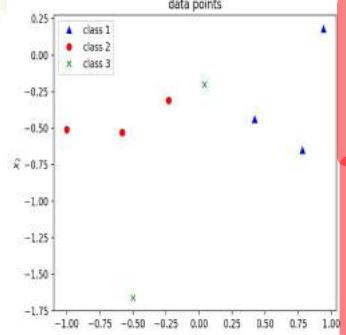
$$U = \begin{pmatrix} 0.86 & 0.11 & 0.64 \\ -0.84 & -0.28 & -0.49 \\ -0.41 & -0.16 & -0.22 \\ -0.01 & -0.21 & 0.17 \\ -0.86 & -0.82 & -0.06 \\ -1.15 & -0.27 & -0.75 \\ 0.11 & -0.31 & 0.35 \\ -0.12 & -0.10 & -0.02 \end{pmatrix}$$

$$f(U) = \frac{e^{(U)}}{\sum_{k=1}^3 e^{(U)}} = \begin{pmatrix} 0.44 & 0.21 & 0.35 \\ 0.24 & 0.42 & 0.34 \\ 0.29 & 0.37 & 0.35 \\ 0.33 & 0.27 & 0.40 \\ 0.23 & 0.24 & 0.52 \\ 0.20 & 0.49 & 0.31 \\ 0.34 & 0.22 & 0.43 \\ 0.32 & 0.33 & 0.35 \end{pmatrix}$$

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

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

$$\begin{aligned} \text{Entropy, } J &= -\sum_{p=1}^8 \log(f(u_{pd_p})) \\ &= -\log(0.44) - \log(0.42) - \dots - \log(0.35) \\ &= 7.26 \end{aligned}$$



TUT 6
Draw decision boundary across classes

At convergence:

$$W = \begin{pmatrix} -0.15 & -3.41 & 4.18 \\ 5.27 & -1.02 & -4.15 \end{pmatrix} \text{ and } b = \begin{pmatrix} -7.82 \\ 5.81 \\ 2.02 \end{pmatrix}$$

Synaptic inputs at the softmax layer for an input $x = (x_1, x_2)$:

$$\text{Neuron of class A, } u_1 = w_1^T x + b_1 = -0.15x_1 + 5.27x_2 - 7.82$$

$$\text{Neuron of class B, } u_2 = w_2^T x + b_2 = -3.41x_1 - 1.02x_2 + 5.81$$

$$\text{Neuron of class C, } u_3 = w_3^T x + b_3 = 4.18x_1 - 4.15x_2 + 2.02$$

Decision boundaries:

$$\text{Between class A and class B is given when } u_1 = u_2$$

$$-0.15x_1 + 5.27x_2 - 7.82 = -3.41x_1 - 1.02x_2 + 5.81$$

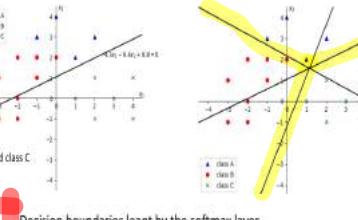
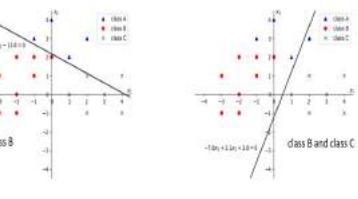
$$3.25x_1 + 6.29x_2 - 13.63 = 0$$

$$\text{Similarly, between class B and class C } u_2 = u_3 :$$

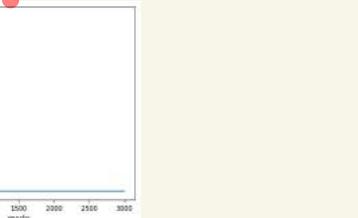
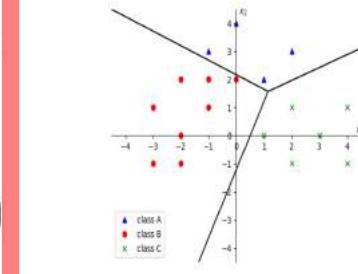
$$-7.59x_1 + 3.13x_2 + 3.79 = 0$$

$$\text{between class A and class C } u_1 = u_3 :$$

$$4.33x_1 - 9.42x_2 + 9.84 = 0$$



Decision boundaries learnt by the softmax layer



At convergence at 3000 iterations:

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

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

$$\text{Entropy} = 0.562$$

$$\text{Errors} = 0$$

4. Deep Neural Networks

Chain rule of differentiation

Let x , y , and $f \in R$ be one-dimensional variables and

$$j = g(y)$$

$$y = f(x)$$

Chain rule of differentiation states that:

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

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



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

Chain rule in multidimensions



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

$$y = f(x)$$

$$J = g(y)$$

Then, the chain rule of differentiation states that:

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

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

$$\nabla_x J = \left(\frac{\partial y_i}{\partial x_j} \right)^T \nabla_y J$$

where

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

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

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 $x = (x_1, x_2, x_3)$, $y = (y_1, y_2)$, and $y = f(x)$ where f is given by

$$y_1 = 5 - 2x_1 + 3x_3$$

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

Find the Jacobian of f .

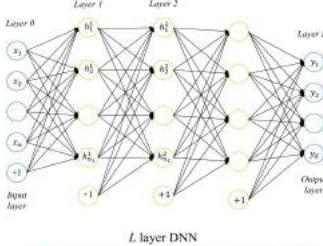
$$\begin{aligned} \frac{\partial y_1}{\partial x_1} &= -2, & \frac{\partial y_1}{\partial x_2} &= 0, & \frac{\partial y_1}{\partial x_3} &= 3 \\ \frac{\partial y_2}{\partial x_1} &= 1, & \frac{\partial y_2}{\partial x_2} &= 10x_2, & \frac{\partial y_2}{\partial x_3} &= 3x_3^2 \end{aligned}$$

Jacobian:

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

Deep neural networks (DDN):

Also known as feedforward networks (FFN)



Layered 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

Forward propagation of activations single input pattern

Consider an input pattern (x, d) to 2-layer FFN:



Synaptic input x to hidden layer:

$$z = W^T x + b$$

Output h of hidden layer:

$$h = g(z)$$

g is hidden layer activation function.

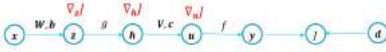
Synaptic input h to output layer:

$$u = V^T h + c$$

Output y of output layer:

$$y = f(u)$$

Backpropagation of gradients



Since the targets appear at the output, the error gradient at the output layer is $\nabla_d J$ is known. Therefore, output weights and bias, V, 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

$$u = V^T h + c$$

Consider synaptic input to u_k to the k th neuron at the output layer. Let weight vector $v_k = (v_{k1}, v_{k2}, \dots, v_{kM})$ and bias c_k .

The synaptic input u_k due to h is given by

$$u_k = v_k^T h + c_k = v_{k1}h_1 + v_{k2}h_2 + \dots + v_{kM}h_M + c_k$$

Therefore

$$\frac{\partial u}{\partial h} = \begin{pmatrix} \frac{\partial u_k}{\partial h_1} & \frac{\partial u_k}{\partial h_2} & \dots & \frac{\partial u_k}{\partial h_M} \\ \frac{\partial u_k}{\partial h_1} & \frac{\partial u_k}{\partial h_2} & \dots & \frac{\partial u_k}{\partial h_M} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial u_k}{\partial h_1} & \frac{\partial u_k}{\partial h_2} & \dots & \frac{\partial u_k}{\partial h_M} \end{pmatrix} = \begin{pmatrix} v_{11} & v_{12} & \dots & v_{1M} \\ v_{21} & v_{22} & \dots & v_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ v_{K1} & v_{K2} & \dots & v_{KM} \end{pmatrix} = V^T$$

That is,

$$\frac{\partial u}{\partial h} = V^T \quad (A)$$

$$y = f(u)$$

Considering k th neuron:

$$y_k = f(u_k)$$

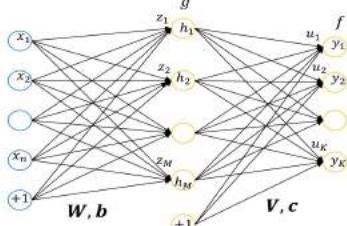
$$\frac{\partial y}{\partial u} = \begin{pmatrix} \frac{\partial y_k}{\partial u_1} & \frac{\partial y_k}{\partial u_2} & \dots & \frac{\partial y_k}{\partial u_K} \\ \frac{\partial y_k}{\partial u_1} & \frac{\partial y_k}{\partial u_2} & \dots & \frac{\partial y_k}{\partial u_K} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_k}{\partial u_1} & \frac{\partial y_k}{\partial u_2} & \dots & \frac{\partial y_k}{\partial u_K} \end{pmatrix} = \begin{pmatrix} f'(u_1) & 0 & \dots & 0 \\ 0 & f'(u_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & f'(u_K) \end{pmatrix} = diag(f'(u))$$

That is,

$$\frac{\partial y}{\partial u} = diag(f'(u)) \quad (B)$$

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

Two-layer FFN



Input $x = (x_1 \ x_2 \ \dots \ x_n)^T$

Hidden-layer output $h = (h_1 \ h_2 \ \dots \ h_M)^T$

Output $y = (y_1 \ y_2 \ \dots \ y_K)^T$

W, b – weight and bias of the hidden layer

V, c – weight and bias of the output layer

M is the number of hidden layer neurons

Back-propagation of gradients: single pattern



Considering output layer,

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

From chain rule of differentiation,

$$\nabla_u J = \left(\frac{\partial u}{\partial h} \right)^T \nabla_v J = V \nabla_w J \quad \text{From (A)}$$

$$\nabla_z J = \left(\frac{\partial h}{\partial z} \right)^T \nabla_h J = \text{diag}(g'(z)) V \nabla_w J = V \nabla_w J \cdot g'(z)$$

(C), from (B)

Proof

For a vector x :

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

That is:

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

Back-propagation of gradients: single input pattern

From (C);

$$\nabla_z J = V \nabla_w J \cdot g'(z)$$

That is, the gradients at output layer are multiplied by V and back-propagated to hidden layer.

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

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

1. Forward Propagation:

- Forward propagation refers to the process of computing the output of the neural network for a given input.
- It involves passing the input data through the network layer by layer, from the input layer to the output layer, while applying the activation functions and using the current values of the parameters (weights and biases).
- Each neuron in the network receives inputs from the previous layer, computes a weighted sum of these inputs, adds a bias term, and applies an activation function to produce its output.
- The output of the last layer (the output layer) is the predicted output of the network for the given input.
- Forward propagation does not involve adjusting the parameters of the network; it is solely concerned with computing predictions based on the current parameter values.

2. Backward Propagation (Backpropagation):

- Backward propagation refers to the process of updating the parameters of the neural network based on the error between the predicted output and the true target values.
- It involves computing the gradients of the loss function with respect to each parameter of the network using techniques such as the chain rule from calculus.
- The gradients indicate how much the loss function would change with respect to small changes in each parameter.
- Using the gradients, the parameters are updated in the opposite direction of the gradient (gradient descent) to minimize the loss function.
- The updated parameters are obtained by subtracting a fraction of the gradient from the current parameter values, scaled by a learning rate hyperparameter.
- Backward propagation is typically performed iteratively for each batch of training data in a process known as stochastic gradient descent (SGD) or its variants, updating the parameters gradually to minimize the J function over the entire training dataset.

SGD of two-layer FFN

Output layer:

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

Output layer:

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

Hidden layer:

$$\nabla_h J = V \nabla_w J \cdot g'(z)$$

Given a training dataset $\{(x, d)\}$

Set learning parameter α

Initialize W, b, V, c

Repeat until convergence:

For every pattern (x, d) :

$$z = W^T x + b$$

$$h = g(z)$$

$$u = V^T h + c$$

$$y = f(u)$$

Forward propagation

Hidden layer:

$$\nabla_h J = (\nabla_w J) V^T \cdot g'(z) \quad (D)$$

Given a training dataset $\{(X, D)\}$

Set learning parameter α

Initialize W, b, V, c

Repeat until convergence:

$Z = XW + B$

$H = g(Z)$

$U = HV + C$

$Y = f(U)$

Forward propagation

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

$$\nabla_z J = (\nabla_u J) V^T \cdot g'(Z)$$

Backward propagation

$$V \leftarrow V - \alpha H^T \nabla_u J$$

$$c \leftarrow c - \alpha (V_u J)^T \mathbf{1}_p$$

$$W \leftarrow W - \alpha X^T \nabla_z J$$

$$b \leftarrow b - \alpha V_u J^T \mathbf{1}_p$$

Back-propagation

Forward propagation of activations: batch of inputs

Computational graph of 2-layer FFN for a batch of patterns (X, D) :



Synaptic input Z to hidden layer:

$$Z = XW + B$$

Output H of the hidden layer:

$$H = g(Z)$$

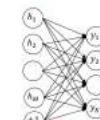
Synaptic input U to output layer:

$$U = HV + C$$

Output Y of the output layer:

$$Y = f(U)$$

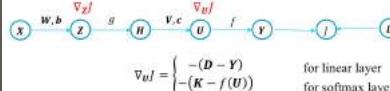
H — V — U — $HV + C$



$$\nabla_z J = (\nabla_u J) V^T \cdot g'(Z) \quad \nabla_u J$$

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

Back-propagation of gradients: batch of patterns



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

$$\nabla_z J = \begin{pmatrix} (\nabla_{z_1} J)^T \\ (\nabla_{z_2} J)^T \\ \vdots \\ (\nabla_{z_p} J)^T \end{pmatrix} = \begin{pmatrix} \left(V \nabla_{u_1} J \cdot g'(z_1) \right)^T \\ \left(V \nabla_{u_2} J \cdot g'(z_2) \right)^T \\ \vdots \\ \left(V \nabla_{u_p} J \cdot g'(z_p) \right)^T \end{pmatrix}$$

Substituting from (C)

$$\nabla_z J = \begin{pmatrix} (\nabla_{u_1} J)^T V^T \cdot (g'(z_1))^T \\ (\nabla_{u_2} J)^T V^T \cdot (g'(z_2))^T \\ \vdots \\ (\nabla_{u_p} J)^T V^T \cdot (g'(z_p))^T \end{pmatrix} \quad (XY)^T = Y^T X^T$$

$$= \begin{pmatrix} (\nabla_{u_1} J)^T \\ (\nabla_{u_2} J)^T \\ \vdots \\ (\nabla_{u_p} J)^T \end{pmatrix} V^T \cdot \begin{pmatrix} (g'(z_1))^T \\ (g'(z_2))^T \\ \vdots \\ (g'(z_p))^T \end{pmatrix}$$

$$= (\nabla_u J) V^T \cdot g'(z)$$

$$\nabla_z J = (\nabla_u J) V^T \cdot g'(Z)$$

Learning in two-layer FFN

GD	SGD
(X, D)	(x, d)
$Z = XW + B$	$z = xW + b$
$H = g(Z)$	$h = g(x)$
$U = HV + C$	$u = V^T h + c$
$Y = f(U)$	$y = f(u)$

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

$$\nabla_z J = (\nabla_u J) V^T \cdot g'(Z)$$

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

$$W \leftarrow W - \alpha X^T \nabla_z J$$

$$b \leftarrow b - \alpha V_u J^T \mathbf{1}_p$$

$$V \leftarrow V - \alpha H^T \nabla_u J$$

$$c \leftarrow c - \alpha (V_u J)^T \mathbf{1}_p$$

FP → Compute output of nn for a given input

→ Pass the input to network to generate pred without adjusting parameters

→ Compute output of each neuron by applying activation func to weighted sums of input

→ Perform once per input to generate pred

Bp → Update parameter of nn based on error between pred and true target by computing gradients and adjust params to minimize loss function

→ Compute gradients of loss func to each neuron using chain rule and update parameter with gradient descent

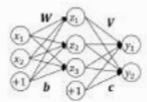
→ Perform iteratively for each batch of training data

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 $x = (x_1, x_2)$	Targets $d = (d_1, d_2)$
(0.77, 0.02)	(0.44, -0.42)
(0.63, 0.75)	(0.84, 0.43)
(0.50, 0.22)	(0.09, -0.72)
(0.26, 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)

$$X = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.26 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \text{ and } D = \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix}$$



Output layer is a linear neuron layer
Hidden layer is a sigmoidal layer

Initialized (weights using a uniform distribution):

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

Epoch 1:

$$Z = XW + b = \begin{pmatrix} 0.77 & 0.02 \\ 0.63 & 0.75 \\ 0.50 & 0.22 \\ 0.26 & 0.76 \\ 0.17 & 0.09 \\ 0.69 & 0.95 \\ 0.00 & 0.51 \\ 0.81 & 0.61 \end{pmatrix} \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.40 & 0.49 & 0.59 \\ 0.49 & 0.15 & 0.96 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} -3.00 & 0.8 & 0.39 \\ -1.32 & -0.27 & 2.61 \\ -1.35 & -0.04 & 0.91 \\ -1.34 & -1.79 & 2.46 \\ -0.05 & -1.76 & 3.27 \\ -1.42 & -1.74 & 1.60 \\ -1.51 & -0.72 & 2.25 \end{pmatrix}$$

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

$$Y = HV + C = \begin{pmatrix} 0.05 & 0.69 & 0.60 \\ 0.40 & 0.22 & 0.93 \\ 0.21 & 0.49 & 0.71 \\ 0.79 & 0.14 & 0.92 \\ 0.40 & 0.49 & 0.59 \\ 0.49 & 0.15 & 0.96 \\ 0.80 & 0.21 & 0.83 \\ 0.18 & 0.33 & 0.91 \end{pmatrix} \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix} + \begin{pmatrix} 0.42 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix} = \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \end{pmatrix}$$

$$\nabla_{\theta} J = -(D - Y) = \begin{pmatrix} 0.44 & -0.42 \\ 0.84 & 0.43 \\ 0.09 & -0.72 \\ -0.25 & 0.35 \\ -0.12 & -0.13 \\ 0.24 & 0.03 \\ 0.30 & 0.20 \\ 0.61 & 0.04 \end{pmatrix} - \begin{pmatrix} -4.32 & 0.44 \\ -2.52 & 1.67 \\ -3.43 & 0.87 \\ -0.79 & 1.15 \\ -2.32 & 0.21 \\ -2.04 & 1.75 \\ -0.67 & 0.76 \\ -3.59 & 1.75 \end{pmatrix} = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 0.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix}$$

$$\nabla_{\theta} J = (\nabla_{\theta} J)^T \cdot g'(Z) = \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix} \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}^T = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 0.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix}^T \cdot \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

$$V_{\theta} J = (\nabla_{\theta} J)^T \cdot g'(Z) = \begin{pmatrix} -4.76 & 0.85 \\ -3.35 & 1.24 \\ -3.52 & 0.59 \\ -0.54 & 0.80 \\ -2.20 & 0.34 \\ -2.28 & 1.72 \\ -0.98 & 0.56 \\ -4.20 & 1.70 \end{pmatrix} = \begin{pmatrix} 0.04 & 0.21 & 0.24 \\ 0.24 & 0.17 & 0.06 \\ 0.16 & 0.25 & 0.20 \\ 0.16 & 0.12 & 0.07 \\ 0.24 & 0.25 & 0.24 \\ 0.25 & 0.13 & 0.04 \\ 0.16 & 0.16 & 0.14 \\ 0.15 & 0.22 & 0.09 \end{pmatrix}$$

Output layer:

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

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

Hidden layer:

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

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

Layer	Weights
L1 (Conv)	96 kernels of size 11x11x3, with a stride of 4 pixels
L2 (Conv)	96 kernels of size 5x5x3, with a stride of 2 pixels
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	16,781,312
L7 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

$$96 \times (4 \times 11 \times 11 \times 3 + 1)$$

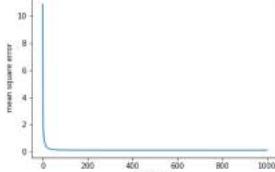
$$V \leftarrow V - \alpha \nabla_{\theta} J = \begin{pmatrix} 3.89 & -1.74 \\ -3.14 & -1.89 \\ -2.53 & 2.51 \end{pmatrix}$$

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

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

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

GD learning



After 1,000 epochs,

m. s. e. = 0.107

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	16,781,312
L7 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

Note: There are no parameters associated with a pooling layer. The pool size, stride, and padding are hyperparameters.

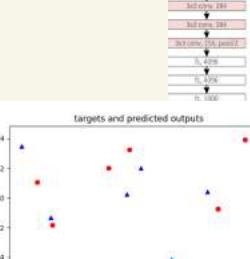
$$W = \begin{pmatrix} -2.04 & -0.52 & -1.88 \\ 3.55 & -2.26 & 2.43 \end{pmatrix}$$

$$b = \begin{pmatrix} 0.22 \\ 0.15 \end{pmatrix}$$

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

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

targets and predicted outputs



After 20,000 iterations

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,342

First FC layer:
Number of neurons = 4096
Number of kernels in the previous Conv Layer = 256
Size (width) of the output image of the previous Conv Layer = 6

Number of parameters = $(6 \times 6 \times 256 \times 4096) + 4096 = 37,752,832$

Layer	Weights
L1 (Conv)	34,944
L2 (Conv)	614,656
L3 (Conv)	885,120
L4 (Conv)	1,327,488
L5 (Conv)	884,992
L6 (FC)	37,752,832
L7 (FC)	16,781,312
L8 (FC)	4,097,000
Conv Subtotal	3,747,200
FC Subtotal	58,631,144
Total	62,378,344

The last FC layer:
Number of neurons = 1000
Number of neurons in the previous FC Layer = 4096
Number of parameters = $(1000 \times 4096) + 1000 = 4,097,000$

Number of parameters = $(1000 \times 4096) + 1000 = 4,097,000$

- Escape from a few layers
- ReLU nonlinearity for solving gradient vanishing (better)
- Data augmentation
- Dropout
- Outperformed all previous models on ILSVRC by 10%

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 the 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,

$$1. \text{ Scaling} \quad \text{the inputs such that } \tilde{x}_i \in [0, 1]: \\ \tilde{x}_i = \frac{x_i - \mu_i}{x_{i,\max} - x_{i,\min}}$$

$$2. \text{ Normalizing} \quad \text{the input to have standard normal distributions } \tilde{x}_i \sim N(0, 1): \\ \tilde{x}_i = \frac{x_i - \mu_i}{\sigma_i}$$

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

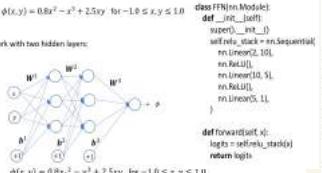
TUT ques

3. Design a feedforward neural network consisting of two-hidden layers to approximate the following function:

$$\phi(x, y) = 0.8x^2 - y^2 + 2.5xy$$

For $-1.0 \leq x, y \leq 1.0$. Use three ReLU neurons at each hidden layer and a linear neuron at the output layer.

- (a) Divide the input space equally into square regions of size 0.25×0.25 and use grid points as data to learn the function ϕ .
- (b) Train the network using gradient descent learning at learning rate $\alpha = 0.01$ and plot the learning curve (mean square error vs. iterations) and the predicted data points.
- (c) Compare the learning curves when learning the function at learning rates $\alpha = 0.005, 0.01, 0.05$, and 0.1 .



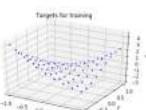
Data points in a grid of size 0.25x0.25:

feedforward neural network with two hidden layers:

```
class FFN(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(2, 10)
        self.ReLU1 = nn.ReLU()
        self.linear2 = nn.Linear(10, 5)
        self.ReLU2 = nn.ReLU()
        self.linear3 = nn.Linear(5, 1)

    def forward(self, x):
        logits = self.linear1(x)
        return logits
```

Training inputs



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]

Feature 1: $\frac{x_1 - \mu_1}{\sigma_1}$ House value

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),
```

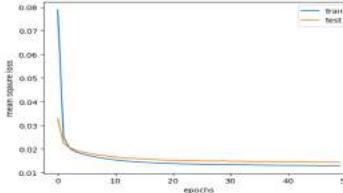
)

def forward(self, x):

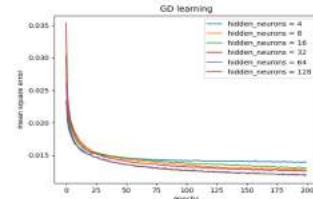
logits = self.relu_stack(x)

return logits

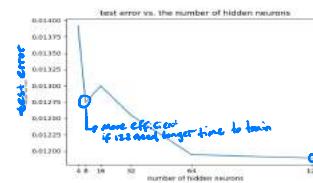
Example 3a



Example 3b: no of hidden neurons



Example 3b: width of the hidden layer



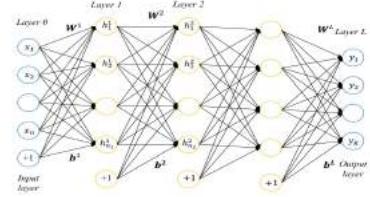
Optimum number of hidden neurons: 64

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)



DNN notations

Input layer $l = 0$

Width: n

Input: x

Hidden layers $l = 1, 2, \dots, L - 1$

Width: n_l

Weight matrix W^l , bias vector b^l

Synaptic input u^l , U^l

Activation function f^l

Output y^l, Y^l

Desired output d, D

Forward propagation of activation in DNN: single pattern

Input (x, d)

$u^l = W^{l,T} x + b^l$

For layers $l = 1, 2, \dots, L - 1$:

$h^l = f^l(u^l)$

$u^{l+1} = W^{l+1,T} h^l + b^{l+1}$

$y^l = f^l(u^l)$

Back-propagation of gradients in DNN: single pattern

If $l = L$:

$$\nabla_w J = \begin{cases} -(d - y) \\ -(1(k=d) - f^L(u^L)) \end{cases}$$

for linear layer
for softmax layer

else:

$$\nabla_w J = W^{l+1} (\nabla_u J) \cdot f'^l(u^l)$$

from (C)

$$\nabla_w J = H^{l-1} (\nabla_u J)^T$$

$$\nabla_b J = \nabla_w J$$

Gradients are backpropagated from the output layer to the input layer

Forward propagation of activation in DNN: batch of patterns

Input (X, D)

$U^l = X W^l + B^l$

For layers $l = 1, 2, \dots, L - 1$:

$H^l = f^l(U^l)$

$U^{l+1} = H^l W^{l+1} + B^{l+1}$

$Y = f^L(U^L)$

Back-propagation of gradients in DNN: batch of patterns

If $l = L$:

$$\nabla_w J = \begin{cases} -(D - Y) \\ -(K - f^L(U^L)) \end{cases}$$

Else:

$$\nabla_w J = (W^{l+1})^T (\nabla_u J) \cdot f'^l(U^l)$$

from (D)

$$\nabla_w J = H^{l-1} (\nabla_u J)^T$$

$$\nabla_b J = (\nabla_w J)^T I_p$$

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(in_features, no_hidden1),

nn.ReLU(),

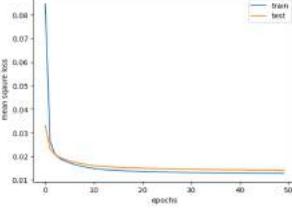
nn.Linear(no_hidden1, no_hidden2),

nn.ReLU(),

nn.Linear(no_hidden2, no_labels),

)

Example 4



Example 4b: Varying the depth of DNN

Architectures:

One hidden layer: [8, 5, 1]

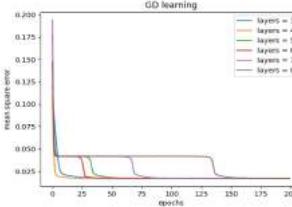
Two hidden layers: [8, 5, 5, 1]

Three hidden layers: [8, 5, 5, 5, 1]

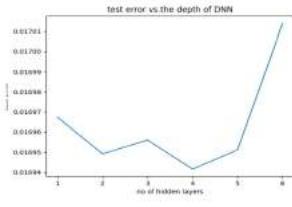
Four hidden layers: [8, 5, 5, 5, 5, 1]

Five hidden layers: [8, 5, 5, 5, 5, 1]

Example 4



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).

at least 1 epoch per parameter

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, 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

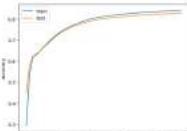
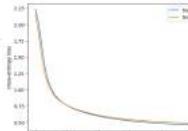
Learning rate $\alpha = 0.001$

```
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)
        )

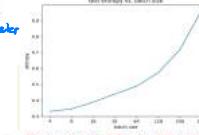
    def forward(self, x):
        x = self.flatten(x)
        logits = self.softmax_relu_stack(x)
        return logits

model = NeuralNetwork()
```

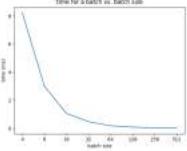
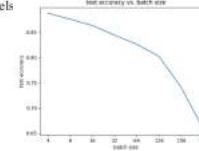
Example 5



Example 5b: 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: $\nabla_Z J = (\frac{\partial J}{\partial x})^T \nabla_y$
- FFN with one hidden layer (Shallow FFN)
- Backpropagation for FFN with one hidden layer:
- 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

5 Model Selection and overfitting

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:

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

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

- Classification error of a classifier. ~~Show me that's why we need to use more~~

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

where d_p is the target and y_p is the predicted output of pattern p . $\frac{1}{K}$ 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.

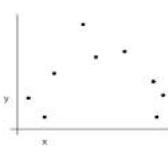
Validation error

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

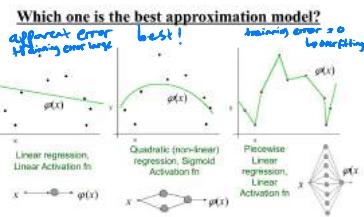
Generally, a larger test set helps provide a greater confidence on the accuracy of the estimate.

Example: Regression

Given the following sample data:

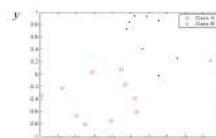


Approximate $y \approx \varphi(x)$ using an NN



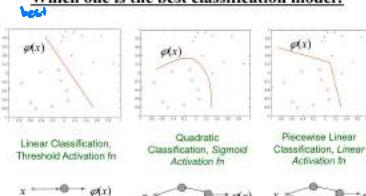
Example: Classification

Given the following sample data:



Classify the following data using an NN.

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)

Solution:

K-fold cross validation

Problem!: there will be overlap

Here will be data that never be used for training

Holdout Method

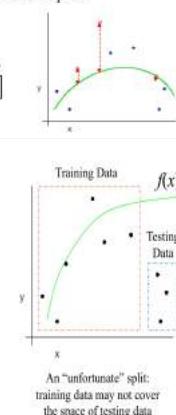
Split entire dataset into two sets:

- 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



The holdout method has two basic drawbacks:

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



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

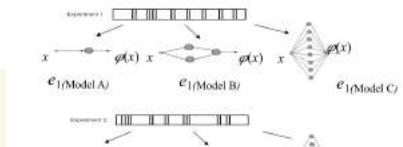
Random Subsampling performs K data splits of the dataset for training and testing.



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 .

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

Example: K=2 Data Splits Random SubSampling



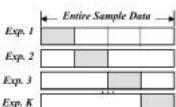
$$\text{Test error}_{(M)} = \frac{1}{2} (e_{1(M)} + e_{2(M)})$$

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 dataset:

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

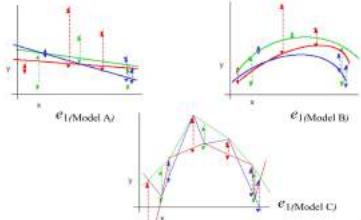


Let the error estimate for i th experiment on test partition be e_i

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

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.

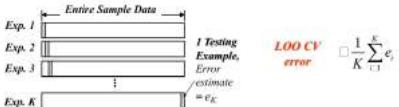
Example: 3-fold Cross-Validation



Leave-One-Out (LOO) Cross-Validation

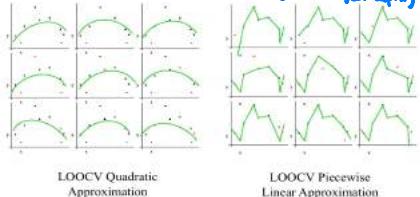
Leave-One-Out is the degenerate case of **K-Fold Cross Validation**, where K is chosen as the total number of examples:

- 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.



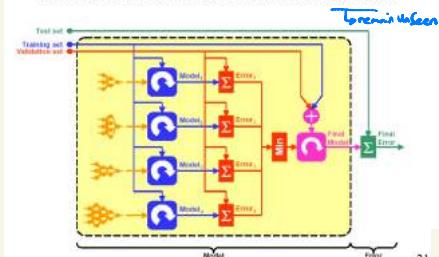
Example: Leave-One-Out Cross-Validation

downback: - limited scope (only for datasets with limited depth)



Three-Way Data Splits Method

Dataset is partitioned into 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.
- Each model m , train the network and find the error e_m [blue box]**
- Validation set:** examples to determine the error J_m of different models m , using the validation set. The optimal model m^* is given by

$$m^* = \underset{m}{\operatorname{argmin}} J_m$$

more

- Training + Validation set:** combine examples used to re-train/redesign model J_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.

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.

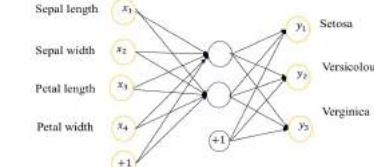
"NO FURTHERING TUNING OF THE MODEL IS ALLOWED!"

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?

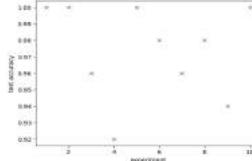


Example 1a: Random subsampling

150 data points

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

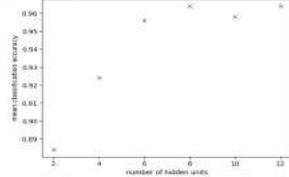
Example: 5 hidden neurons, 10 experiments



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%

↓ because simpler model → avoids overfitting issues run quicker

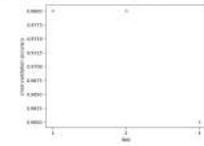
Example 2a: Cross validation

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

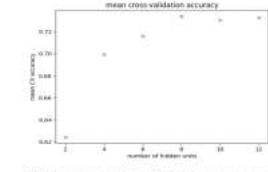
Example: hidden number of units = 5



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

Example 2b

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

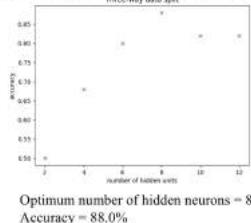


Optimum number of hidden neurons = 8
Cross-validation accuracy = 73.4%

Example 3a: Three-way data splits

150 data points

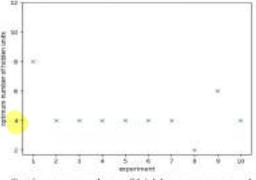
50 data points each for training, for validation, and for testing.



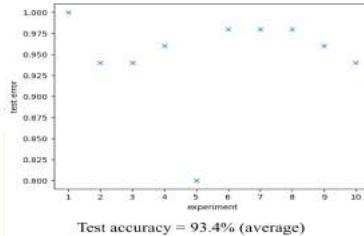
Optimum number of hidden neurons = 8
Accuracy = 88.0%

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



Test accuracy = 93.4% (average)

Model Complexity

Complex models: models with many adjustable weights and biases will

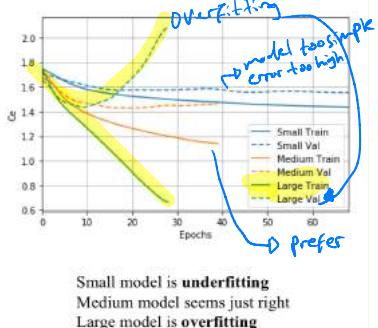
- more likely to be able to solve the required task,
- more likely to memorize the training data without solving the task.

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



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.

architectures - GoogLeNet

- An important lesson - go deeper
- Inception structures (v2, v3, v4)
 - Reduce parameters (4M vs 60M in AlexNet)



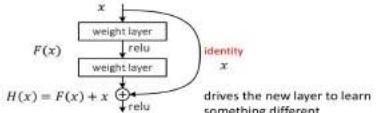
The 1x1 convolutions are performed to reduce the dimensions of input/output.

- Batch normalization
 - Normalizes the activation for each training mini-batch
 - Allows us to use much higher learning rates and be less careful about initialization

• An important lesson - go deeper

• Escape from 100 layers

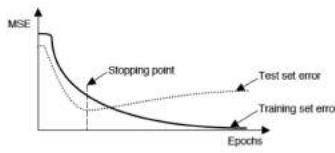
- Residual learning



Methods to overcome overfitting

- Early stopping
- Regularization of weights
- 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 generalize. In order to avoid this, a *penalty term* (*regularization term*) is added to the cost function.

For a network with weights $W = [w_{ij}]$ and bias b , the penalized (or regularized) cost function $J(W, b)$ is defined as

$$J = f + \beta_1 \sum_{ij} |w_{ij}| + \beta_2 \sum_{ij} (w_{ij})^2$$

where $J(W, b)$ is the standard cost function (i.e., m.s.e. or cross-entropy),

$$L^1 - \text{norm} = \sum_{ij} |w_{ij}|$$

$$L^2 - \text{norm} = \left(\sum_{ij} (w_{ij})^2 \right)^{\frac{1}{2}}$$

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.

$$J = f + \beta_2 \sum_{ij} (w_{ij})^2$$

Gradient of the regularized cost wrt weights:

$$\nabla_W J_1 = \nabla_W f + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial W}$$

(A)

$$L^2 \text{ norm} = \sum_{ij} (w_{ij})^2 = w_{11}^2 + w_{12}^2 + \dots + w_{Kk}^2$$

$$\frac{\partial (\sum_{ij} (w_{ij})^2)}{\partial W} = \begin{pmatrix} \frac{\partial \sum (w_{ij})^2}{\partial w_{11}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{12}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{1K}} \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{21}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{22}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{2K}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \sum (w_{ij})^2}{\partial w_{N1}} & \frac{\partial \sum (w_{ij})^2}{\partial w_{N2}} & \dots & \frac{\partial \sum (w_{ij})^2}{\partial w_{NK}} \end{pmatrix} = 2W$$

Substituting in (A):

$$\nabla_W J_1 = \nabla_W f + \beta_2 \frac{\partial (\sum_{ij} w_{ij}^2)}{\partial W} = \nabla_W f + 2\beta_2 W$$

For gradient decent learning that uses regularized cost function:

$$W \leftarrow W - \alpha \nabla_W f$$

Substituting above:

$$W \leftarrow W - \alpha (\nabla_W f + \beta W)$$

where $\beta = 2\beta_2$

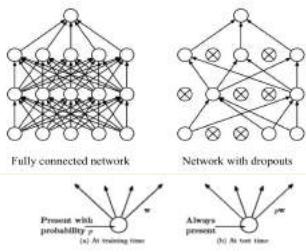
β is known as the *weight decay parameter*.

That is for L^2 regularization, the weight matrix is weighted by decay parameter and added to the gradient term.

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.



At the training time, the units (neurons) are present with a probability p and presented to the next layer with weight 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.

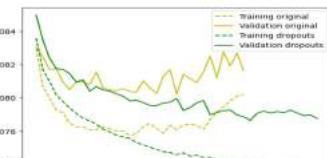
`nn.Dropout(p=0.2)`

Example 5: dropouts

When training with dropouts, we train on a thinned network dropping out units in each mini-batch.

```
class NeuralNetwork_dropout(Module):
    def __init__(self, hidden_size=100, dropout=0.5):
        super(NeuralNetwork_dropout, self).__init__()
        self.linear1 = nn.Linear(1, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, hidden_size)
        self.linear4 = nn.Linear(hidden_size, 10)
        self.Dropout = nn.Dropout(p=0.5)
        self.Softmax = nn.Softmax(dim=1)
    def forward(self, x):
        x = self.linear1(x)
        x = self.Dropout(x)
        x = self.linear2(x)
        x = self.Dropout(x)
        x = self.linear3(x)
        x = self.Dropout(x)
        x = self.linear4(x)
        x = self.Softmax(dim=1)
        return x
```

Example 5



Summary

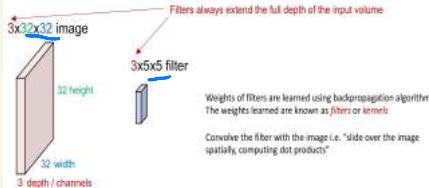
• Model selection

- 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

Convolution layer



Consider a second, green filter

3x32x32 image

3x5x5-filter

convolve (slide) over all spatial locations

Two 1x28x28 activation/feature maps

3x32x32 image

3x5x5-filter

convolve (slide) over all spatial locations

Six 1x28x28 activation/feature maps

2x3x32x32 batch of images

Also the 6-dim bias vector

2x6x28x28 batch of activation/feature maps

NxDxH1xW1 batch of images

Also the D1-dim bias vector

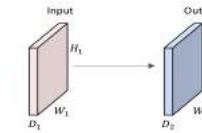
D1xNxDxH1xW1 batch of activation/feature maps

CNN is a sequence of convolutional layers, interspersed with activation functions

Convolution layer - summary

A convolution layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires four hyperparameters
 - Number of filters K
 - Their spatial extent F
 - The stride S
 - The amount of zero padding P
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $D_2 = (D_1 + 2P)/S + 1$
 - $H_2 = (H_1 + 2P)/S + 1$ (i.e., width and height are computed equally by symmetry)
 - $W_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases
- In the output volume, the d -th depth slice (of size $H_2 \times W_2$) is the result of a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias

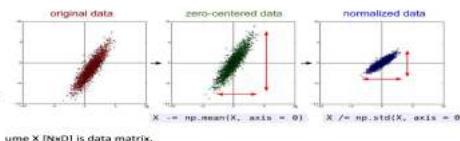


Epoch and mini-batch

- Epoch** – in the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

- Mini-batch** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on **mini-batches**, where the number of data points in a batch is a hyperparameter that we can tune.

Data pre-processing



Optimization

- A CNN as composition of functions $f_w(x) = f_1(\dots(f_2(f_1(x; w_1); w_2) \dots; w_L))$
- Parameters $w = (w_1, w_2, \dots, w_L)$
- Empirical loss function $L(w) = \frac{1}{n} \sum_i l(y_i, f_w(x_i))$

$$w^* = \operatorname{argmin}_w L(w)$$

- In 1-dimension, the derivative of a function gives the slope:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

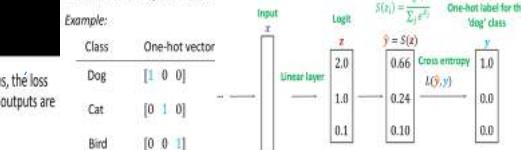
$$f(x) = 2x$$

$$f(x+2) = 2(x+2)$$

$$\frac{x+2 - x}{2 - x} = \frac{2}{2} = 2$$

- In multiple dimensions, the **gradient** is the vector of (partial derivatives along each dimension)
- The direction of steepest descent is the **negative gradient**

One-hot encoded ground truth

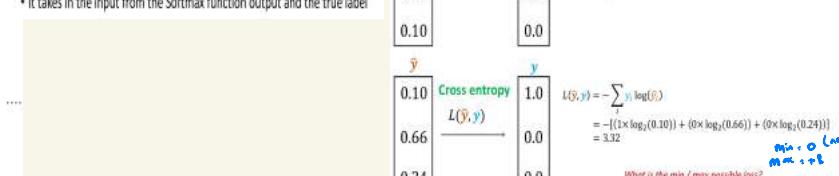


Cross entropy loss

Loss function – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs are correctly predicted by the model outputs.

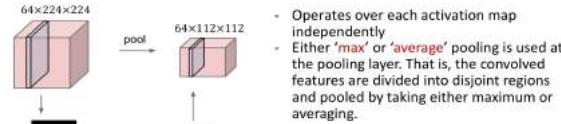
Cross entropy loss (Multinomial Logistic Regression)

- The usual loss function for a multi-class classification problem
- Right after the Softmax function
- It takes in the input from the Softmax function output and the true label



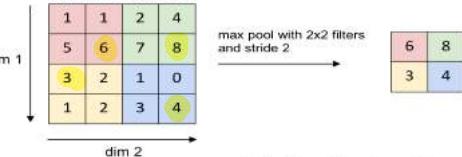
What is the min / max possible loss?

Pooling layer



MAX Pooling

Single depth slice



Consider pooling with non-overlapping windows $\{(l, m)\}_{(l,m)=1/(2M)^2}^{L/2M^2}$ of size $L \times M$

The max pooling output is the maximum of the activation inside the pooling window. Pooling of a feature map y at $p = (l, j)$ produce pooled feature

$$z(l, j) = \max_{i, m} y(i + l, j + m)$$

The mean pooling output is the mean of activations in the pooling window

$$z(l, j) = \frac{1}{L \times M} \sum_{i, m} y(i + l, j + m)$$

Why pooling?

A function f is invariant to g if $f(g(x)) = f(x)$.

• Pooling layers can be used for building inner activations that are (slightly) invariant to small translations of the input.

• Invariance to local translation is helpful if we care more about the presence of a pattern, rather than its exact position.

Example 1

Given an input pattern X :

$$X = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ 0.2 & 0.7 & 0.0 & 0.2 & -0.3 \\ -0.4 & 0.1 & 0.2 & 0.3 & 0.3 \end{pmatrix}$$

The input pattern is received by a convolution layer consisting of one kernel (filter)

$$w = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

and bias = 0.05.

If convolution layer has a sigmoid activation function,

- Find the outputs of the convolution layer if the padding is VALID at strides = 1
- Assume the pooling layer uses max pooling, has a pooling window size of 2x2, and stride = 2, find the activations at the pooling layer.

c) Repeat (a) & (b) using convolution layer with SAME padding

Find the outputs of the convolution layer if the padding is VALID at strides = 1

$$I = \begin{pmatrix} 0.5 & -0.1 & 0.2 & 0.3 & 0.5 \\ 0.8 & 0.1 & -0.5 & 0.5 & 0.1 \\ 0.2 & 0.7 & 0.0 & 0.2 & -0.3 \\ -0.4 & 0.1 & 0.2 & 0.3 & 0.3 \end{pmatrix}; w = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

Synaptic input to the pooling-layer:

$$u(i, j) = \sum_m x(i + l, j + m)w(l, m) + b$$

For VALID padding:

$$\begin{aligned} u(1,1) &= 0.5 \times 0 - 0.1 \times 1 + 0.2 \times 1 + 0.3 \times 1 + 0.1 \times 0 - 0.5 \times 1 - 0.8 \times 1 + 0.2 \times 0 + 0.05 = -0.25 \\ u(2,2) &= -0.1 \times 0 + 0.2 \times 1 + 0.3 \times 1 + 0.1 \times 1 - 0.5 \times 0 + 0.5 \times 1 + 0.2 \times 0 + 0.1 \times 0 + 0.05 = 1.25 \\ u(3,3) &= 0.2 \times 0 + 0.3 \times 1 + 0.5 \times 1 - 0.5 \times 0 + 0.5 \times 1 + 0.0 \times 1 + 0.3 \times 1 - 0.2 \times 0 + 0.05 = 0.75 \\ u(2,1) &= 0.8 \times 0 + 0.1 \times 1 - 0.5 \times 1 + 0.2 \times 0 + 0.0 \times 1 + 0.7 \times 1 + 0.1 \times 1 + 0.2 \times 0 + 0.05 = -0.55 \end{aligned}$$

Example 2

Inputs are digit Images from MNIST database: <http://yann.lecun.com/exdb/mnist/>



First convolution layer consists of three filters:

$$w_1 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, w_2 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 1 \\ -1 & -2 & -1 \end{bmatrix}, w_3 = \begin{bmatrix} 3 & 4 & 5 \\ 4 & 5 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

Find the feature maps at the convolution layer and pooling layer. Assume zero bias

For convolution layer, use a stride = 1 (default) and padding = 'VALID'.

For pooling layer, use a window of size 2x2 and a stride = 2.

Pooling layer - summary

A pooling layer

- Accepts a volume of size $D_1 \times H_1 \times W_1$
- Requires two hyperparameters
 - Their spatial extent F
 - The stride S
- Produces a volume of size $D_2 \times H_2 \times W_2$, where
 - $D_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layers

Synaptic input to the convolution layer:

$$U = \begin{pmatrix} -0.55 & 0.85 & 0.65 \\ 0.75 & 0.05 & 1.05 \end{pmatrix}$$

Output of the convolution layer:

$$f(U) = \frac{1}{1 + e^{-U}} = \begin{pmatrix} 0.384 & 0.701 & 0.512 \\ 0.67 & 0.512 & 0.7 \end{pmatrix}$$

Output of the max pooling layer:

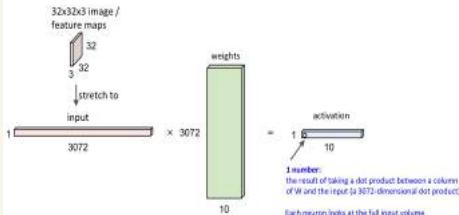
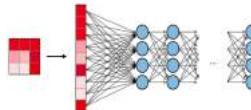
$$(0.794)$$

Now find the outputs of the convolution layer if the padding is SAME at strides = 1

Fully connected layer

The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons.

If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.

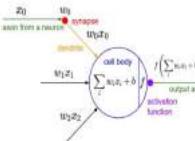


Activation function

Recall that the output of the neuron at (i, j) of the convolution layer

$$y(i, j) = f(u(i, j))$$

where u is the synaptic input and f is an activation function [It aims at introducing non-linearities to the network].



Sigmoid

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

tanh

$$\tanh(x)$$

ReLU

$$\max(0, x)$$

Leaky ReLU

$$\max(0.1x, x)$$

Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

See exp.1.ipynb

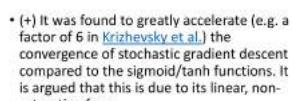
The activation function is usually an abstraction representing the rate of firing in the cell

Activation function – ReLU

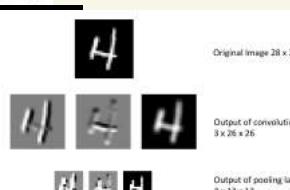


- Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$

$f(x) = \max(0, x)$



- (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.



See exp.2.ipynb

8. Recurrent Neural Networks (RNN)

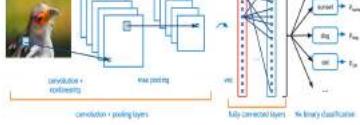
Last week

- CNN Architectures
 - More on convolution
 - How to calculate FLOPs
 - Poarallel convolution
 - Depthwise convolution
 - Depthwise convolution + Poarallel convolution
 - Batch normalization
 - Prevent overfitting
 - Transfer learning
 - Data augmentation

You learn some classic architectures

- You learn how to isolate computation complexity of convolutional layer and how to design a lightweight network
- You learn an important technique to improve the learning of recurrent neural networks
- You learn two important techniques to prevent overfitting in neural networks

Previous: Convolutional Neural Networks (CNN)



How about sequential information?

- Turn a sequence of sound pressures into a sequence of word identities?
- Speech recognition?
- Video prediction?

Outline

- Recurrent Neural Network (RNN)
 - Hidden recurrence
 - Top-down recurrence
- Long Short-Term Memory (LSTM)
 - Long-term dependency
 - Structure of LSTM
- Example Applications

Recurrent Neural Networks (RNN)

Recurrent neural networks (RNN) are designed to process **sequential information**. That is, the data presented in a sequence.

The next data point in the sequence is usually **dependent** on the current data point.

Examples:

- Natural language processing (spoken words and written sentences): The next word in a sentence depends on the word which comes before it.
- Genomic sequences: a nucleotide in a DNA sequence is dependent on its neighbors.

RNN attempts to **capture dependency** among the data points in the sequence.

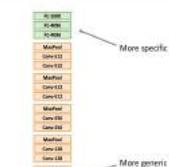
Text Generation with an RNN

QUEENE:
I had thought thou hadst a Roman; for the oracle,
Thus by All bids the man against the word,
Which are so weak of care, by old care done;
Your children were in your holy love,
And the precipitation through the bleeding thron.

in google search
image captions

BISHOP OF ELY:
Marry, and will, my lord, to weep in such a one were prettiest;
Yet now I was adopted heir
Of the world's lamentable day.
To watch the next way with his father with his face?

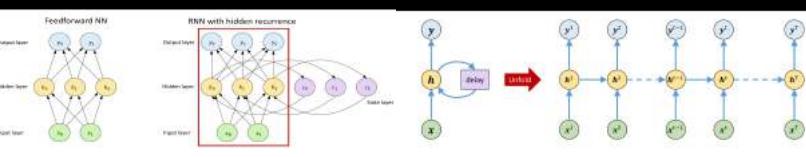
Transfer learning



In a network with an N-dimensional softmax output layer that has been successfully trained toward a supervised classification objective, each output unit will be **specific** to a particular class.

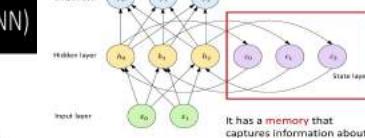
When trained on images, deep networks tend to learn first-layer features that resemble either Gabor filters or color blobs. These first-layer features are **general**.

Recurrent Neural Networks (RNN)



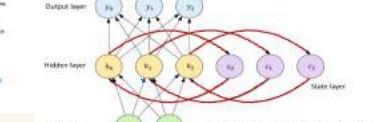
It's called **recurrent** because they perform the same task for every data element (frame) of a sequence, with the output depending on the previous output.

RNN with hidden recurrence



It has a **memory** that captures information about what has been processed so far.

RNN with hidden recurrence

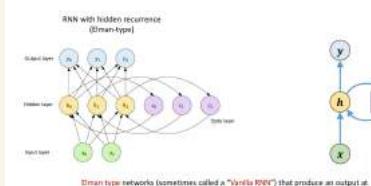


RNN achieves its function by using feedback connections that enables learning of sequential (temporal) information of sequences.

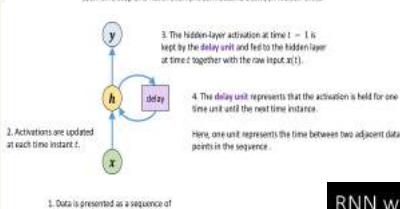
Types of RNN



RNN with hidden recurrence (Elman type)



Elman type networks (sometimes called a "Vanilla RNN") that produce an output at each time step and have recurrent connections between hidden units.



1. Data is presented as a sequence of instance t [time] that is discrete

2. Activations are updated at each time instant t .

3. The hidden-layer activation at time $t = 1$ is kept by the delay unit and fed to the hidden layer at time t together with the raw input $x(t)$.

4. The delay unit represents that the activation is held for one time unit and the next time instance.

Here, one unit represents the time between two adjacent data points in the sequence.

RNN with hidden recurrence

The recurrent connections in the hidden-layer can be unfolded to process sequences of arbitrary length.

By considering the unfolded structure, $h(t)$ is dependent on all the inputs at time t and before time t :

$$h(t) = f^t(x(t), x(t-1), \dots, x(2), x(1))$$

The function f^t takes the whole past sequence $(x(t), x(t-1), \dots, x(2), x(1))$ as input and produce the hidden layer activation.

Not an efficient way! The function f^t is dependent to the sequence length.

Let's represent the past inputs by the hidden-layer activations in the previous instant.

$$h(t) = f^t(x(t), x(t-1), \dots, x(2), x(1))$$

$\underbrace{\hspace{10em}}$

$h(t-1)$



The recurrent structure allows us to factorize f^t into repeated applications of a function f . We can write:

$$\begin{aligned} \text{new state: } h(t) &= f(h(t-1), x(t)) \\ \text{same function: } f &\text{ old state: } h(t-1) \\ \text{input vector at } &\text{ same time step: } x(t) \end{aligned}$$

The folded structure introduces two major advantages:

1. Regardless of the sequence length, the learned model **always has the same size**, rather than specified in terms of a variable-length history of states.
 2. It is possible to use **same transition function** f with the **same parameters** at every time step.
- U:** weight vector that transforms raw inputs to the hidden-layer
- W:** recurrent weight vector connecting previous hidden-layer output to hidden input
- V:** weight vector of the output layer
- b:** bias connected to hidden layer
- c:** bias connected to the output layer

ϕ : the tanh hidden-layer activation function
 σ : the linear/softmax output-layer activation function

Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

Activation of the Elman-type RNN with one hidden-layer is given by:

$$\begin{aligned} h(t) &= \phi(U^T x(t) + W^T h(t-1) + b) \\ y(t) &= \sigma(V^T h(t) + c) \end{aligned}$$

σ is a **softmax** function for classification and a **linear** function for regression.

RNN with hidden recurrence: batch processing

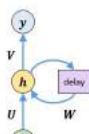
Given P patterns $\{x_p\}_{p=1}^P$ where $x_p = (x_p(t))_{t=1}^T$,

$$X(t) = \begin{pmatrix} x_1(t)^T \\ x_2(t)^T \\ \vdots \\ x_P(t)^T \end{pmatrix}$$

Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t .

Activation of the three-layer Elman-type RNN is given by:

$$\begin{aligned} H(t) &= \phi(X(t)U + H(t-1)W + B) \\ Y(t) &= \sigma(H(t)V + C) \end{aligned}$$



Example 1

A recurrent neural network with hidden recurrence has two input neurons, three hidden neurons, and two output neurons. The parameters of the network are initialized as $U = \begin{pmatrix} -1.0 & 0.5 & -0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix}$, $W = \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 2.0 & -1.0 & 1.5 \\ 1.5 & 0.5 & 0.2 \end{pmatrix}$, and $V = \begin{pmatrix} 0.2 & 0.2 \\ 0.2 & 0.8 \end{pmatrix}$. Bias to the hidden layer $b = \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix}$ and to the output layer $c = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$.

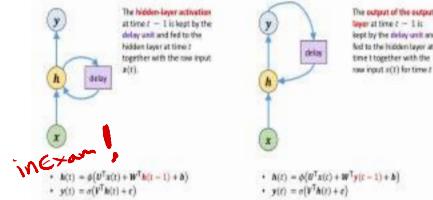
For a sequence of inputs $(x(1), x(2), x(3), x(4))$ where

$$x(1) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, x(2) = \begin{pmatrix} -1 \\ 1 \end{pmatrix}, x(3) = \begin{pmatrix} 0 \\ 3 \end{pmatrix}, x(4) = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

find the output of RNN.

Assume that hidden layer activations are initialized to zero and tanh and sigmoid functions for the hidden and output layer activation functions, respectively.

RNN with top-down recurrence (Jordan type)



Let $x(t)$, $y(t)$, and $h(t)$ be the input, output, and hidden output of the network at time t .

At $t=1$:

$$U = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \\ 1.5 & 0.0 & -0.5 \\ -0.2 & 1.5 & 0.4 \end{pmatrix}, W = \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 2.0 & -1.0 & 1.5 \\ 1.5 & 0.5 & 0.2 \end{pmatrix}$$

$$b = \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix} \text{ and } c = \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}$$

Three hidden neurons and two output neurons

$$h(t) = \phi(U^T x(t) + W^T h(t-1) + b)$$

$$y(t) = \sigma(V^T h(t) + c)$$

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

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

$$\text{Assume } h(0) = (0 \ 0 \ 0)^T.$$

$$\text{At } t=1, x(1) = \begin{pmatrix} 1 \\ 1 \end{pmatrix};$$

$$\begin{aligned} h(1) &= \phi(U^T x(1) + W^T h(0) + b) \\ &= \tanh\left(\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.2 \\ 0.72 \\ -0.9 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} y(1) &= \sigma(V^T h(1) + c) \\ &= \text{sigmoid}\left(\begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 2.0 & -1.0 & 1.5 \\ 1.5 & 0.5 & 0.2 \end{pmatrix}\begin{pmatrix} 0.2 \\ 0.72 \\ -0.9 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.41 \\ 0.37 \end{pmatrix} \end{aligned}$$

$$\text{At } t=2, x(2) = \begin{pmatrix} -1 \\ 1 \end{pmatrix};$$

$$\begin{aligned} h(2) &= \phi(U^T x(2) + W^T h(1) + b) \\ &= \tanh\left(\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \end{pmatrix}\begin{pmatrix} -1 \\ 1 \end{pmatrix} + \begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 2.0 & -1.0 & 1.5 \\ 1.5 & 0.5 & 0.2 \end{pmatrix}\begin{pmatrix} 0.2 \\ 0.72 \\ -0.9 \end{pmatrix} + \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix}\right) \\ &= \begin{pmatrix} 1.0 \\ -0.89 \\ -0.99 \end{pmatrix} \end{aligned}$$

$$\begin{aligned} y(2) &= \sigma(V^T h(2) + c) \\ &= \text{sigmoid}\left(\begin{pmatrix} 2.0 & 1.3 & -1.0 \\ 2.0 & -1.0 & 1.5 \\ 1.5 & 0.5 & 0.2 \end{pmatrix}\begin{pmatrix} 1.0 \\ -0.89 \\ -0.99 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.97 \\ 0.11 \end{pmatrix} \end{aligned}$$

Similarly,

$$\text{at } t=3, x(3) = \begin{pmatrix} 0 \\ 3 \end{pmatrix}, h(3) = \begin{pmatrix} 0.99 \\ 0.3 \\ -1.0 \end{pmatrix} \text{ and } y(3) = \begin{pmatrix} 0.86 \\ 0.18 \end{pmatrix}$$

$$\text{at } t=4, x(4) = \begin{pmatrix} 2 \\ -1 \end{pmatrix}, h(4) = \begin{pmatrix} 0.71 \\ 0.1 \\ 0.79 \end{pmatrix} \text{ and } y(4) = \begin{pmatrix} 0.55 \\ 0.68 \end{pmatrix}$$

The output is $(y(1), y(2), y(3), y(4))$ where

$$y(1) = \begin{pmatrix} 0.41 \\ 0.37 \end{pmatrix}, y(2) = \begin{pmatrix} 0.97 \\ 0.11 \end{pmatrix}, y(3) = \begin{pmatrix} 0.86 \\ 0.18 \end{pmatrix}, y(4) = \begin{pmatrix} 0.55 \\ 0.68 \end{pmatrix}$$

$$\text{At } t=1: X(1) = \begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}$$

$$H(1) = \tanh(X(1)U + Y(0)V + B)$$

$$\begin{aligned} &= \tanh\left(\begin{pmatrix} 1 & 2 \\ -1 & 0 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}\right)(2.0 \ 1.3 \ -1.0) + \begin{pmatrix} 0.2 & 0.2 & 0.2 \end{pmatrix} \\ &= \begin{pmatrix} 0.2 & 0.72 & -1.0 \\ 0.83 & -0.29 & 0.0 \end{pmatrix} \end{aligned}$$

$$Y(1) = \text{sigmoid}(H(1)V + C)$$

$$\begin{aligned} &= \text{sigmoid}\left(\begin{pmatrix} 0.2 & 0.72 & -1.0 \\ 0.83 & -0.29 & 0.0 \end{pmatrix}\begin{pmatrix} 2.0 \\ 1.5 \\ 0.1 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.31 \\ 0.90 \end{pmatrix} \end{aligned}$$

$$\text{At } t=2: X(2) = \begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}$$

$$H(2) = \tanh(X(2)U + Y(1)V + B)$$

$$\begin{aligned} &= \tanh\left(\begin{pmatrix} -1 & 1 \\ 2 & -1 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.31 \\ 0.90 \end{pmatrix}\right)(2.0 \ 1.3 \ -1.0) + \begin{pmatrix} 0.2 & 0.2 & 0.2 \end{pmatrix} \\ &= \begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix} \end{aligned}$$

$$Y(2) = \text{sigmoid}(H(2)V + C)$$

$$\begin{aligned} &= \text{sigmoid}\left(\begin{pmatrix} 0.98 & 0.21 & -0.98 \\ -0.46 & 0.98 & 0.94 \end{pmatrix}\begin{pmatrix} 2.0 \\ 1.5 \\ 0.1 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix} \end{aligned}$$

$$\text{At } t=3: X(3) = \begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}$$

$$H(3) = \tanh(X(3)U + Y(2)V + B)$$

$$\begin{aligned} &= \tanh\left(\begin{pmatrix} 0 & 3 \\ 3 & -1 \end{pmatrix}\begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \end{pmatrix} + \begin{pmatrix} 0.83 \\ 0.11 \end{pmatrix}\right)(2.0 \ 1.3 \ -1.0) + \begin{pmatrix} 0.2 & 0.2 & 0.2 \end{pmatrix} \\ &= \begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.00 & 0.94 & -0.99 \end{pmatrix} \end{aligned}$$

$$Y(3) = \text{sigmoid}(H(3)V + C)$$

$$\begin{aligned} &= \text{sigmoid}\left(\begin{pmatrix} 1.0 & 0.92 & -1.0 \\ -1.00 & 0.94 & -0.99 \end{pmatrix}\begin{pmatrix} 2.0 \\ 1.5 \\ 0.1 \end{pmatrix} + \begin{pmatrix} 0.1 \\ 0.1 \end{pmatrix}\right) \\ &= \begin{pmatrix} 0.63 \\ 0.04 \end{pmatrix} \end{aligned}$$

$$\text{Output (batch): } \mathbf{Y} = (\mathbf{Y}(1) \ \mathbf{Y}(2) \ \mathbf{Y}(3)) = \begin{pmatrix} 0.31 & 0.83 & 0.63 \\ 0.11 & 0.11 & 0.63 \end{pmatrix}$$

Outputs for inputs

$$x_1 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad x_2 = \begin{pmatrix} 0 \\ 3 \end{pmatrix}, \quad x_3 = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

$$y_1 = (-0.31, 0.83, 0.63)$$

$$y_2 = (0.9, 0.11, 0.04)$$

RNN with top-down recurrence: batch processing

Given P patterns $\{x_p\}_{p=1}^P$ where $x_p = (x_p(t))_{t=1}^T$,

$$X(t) = \begin{pmatrix} x_1(t)^T \\ x_2(t)^T \\ \vdots \\ x_P(t)^T \end{pmatrix}$$

Let $X(t)$, $Y(t)$, and $H(t)$ be batch input, output, and hidden output of the network at time t

Activation of the three-layer Jordan-type RNN is given by:

$$H(t) = \phi(X(t)U + Y(t-1)V + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Example 2

A recurrent neural network with top-down recurrence receives 2-dimensional input sequences and produce 1-dimensional output sequences. It has three hidden neurons and following weight matrices and biases:

$$\text{Weight matrix connecting input to hidden layer } U = \begin{pmatrix} -1.0 & 0.5 & 0.2 \\ 0.5 & 0.1 & -2.0 \\ 2.0 & 1.3 & -1.0 \end{pmatrix}$$

$$\text{Recurrence weight matrix connecting previous output to the hidden layer } W = \begin{pmatrix} 2.0 & 1.3 & -1.0 \end{pmatrix}$$

$$\text{Weight matrix connecting hidden layer to the output } V = \begin{pmatrix} 2.0 \\ 1.5 \end{pmatrix}$$

$$\text{Bias to the hidden layer } b = \begin{pmatrix} 0.2 \\ 0.2 \end{pmatrix} \text{ and bias to the output layer } c = 0.1.$$

Given the following two input sequences:

$$x_1 = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \quad x_2 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$$

Using batch processing, find output sequences. Assume outputs are initialized to zero at the beginning. Assume tanh and sigmoid activations for hidden and output layer, respectively.

$$x_1 = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \quad x_2 = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$$

Two sequences as batch of sequences:

$$X = \begin{pmatrix} 1 & 2 & -1 \\ -1 & 0 & 1 \end{pmatrix}$$

Three hidden neurons and one output neuron.

$$H(t) = \phi(X(t)U + Y(t-1)V + B)$$

$$Y(t) = \sigma(H(t)V + C)$$

Initially,

$$Y(0) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Let's consider vanilla-RNN with hidden recurrence.

Forward propagation equations:

$$h(t) = \tanh(U^T x(t) + W^T h(t-1) + b)$$

$$u(t) = V^T h(t) + c$$

The gradients propagate backward, starting from the end of the sequence from two directions:

- Top-down direction
- Reverse sequence direction

For classification,

$$y(t) = \text{softmax}(u(t))$$

For regression,

$$y(t) = u(t)$$

Learnable parameters

B: weight vector that transforms raw inputs to the hidden layer
W: recurrent weight vector connecting previous hidden-layer output to hidden input
V: weight vector of the output layer
b: bias connected to hidden layer
c: bias connected to the output layer

The gradient computation involves performing a forward propagation pass moving left to right through the unfolded graph, followed by a backpropagation pass moving right to left through the graph. The gradients are propagated from the final time point to the initial time points.

The runtime is $O(T)$ where T is the length of input sequence and cannot be reduced by parallelization because the forward propagation is inherently sequential. The back-propagation algorithm applied to the unrolled graph with $O(T)$ cost is called back-propagation through time (BPTT).

The network with recurrence between hidden units is thus very powerful but also expensive to train.

Transformers

Notable for its use of **attention** to model long-range dependencies in data



A sequence-to-sequence model

Model of choice in natural language processing (NLP)

Like LSTM, Transformer is an architecture for transforming one sequence into another one with the help of two parts (Encoder and Decoder)

But it differs from the existing sequence-to-sequence models because it does not imply any Recurrent Networks (GRU, LSTM, etc.).

- During training, layer outputs can be calculated in parallel, instead of a series like an RNN
- Attention-based models allow modeling of dependencies without regard to their distance in the input or output sequences

Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified** by learned weights and thus the output is usually formed as a weighted average.

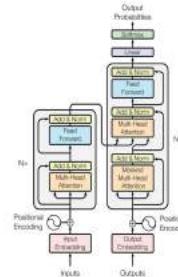
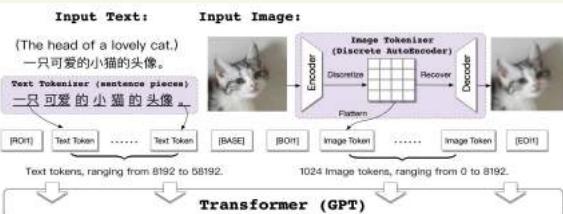


Figure 1: The Transformer - model architecture.



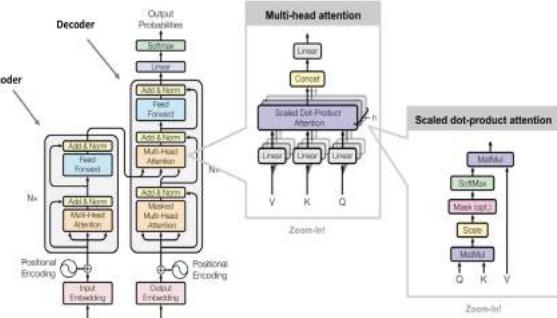
Attention is a mechanism that a model can learn to make predictions by selectively attending to a given set of data.

The amount of attention is **quantified** by learned weights and thus the output is usually formed as a weighted average.

Attention = WV



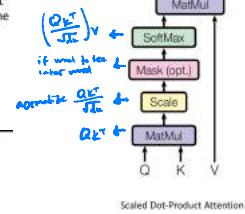
The model attends to image regions that are semantically relevant for classification



Self-attention = Scaled dot-product attention

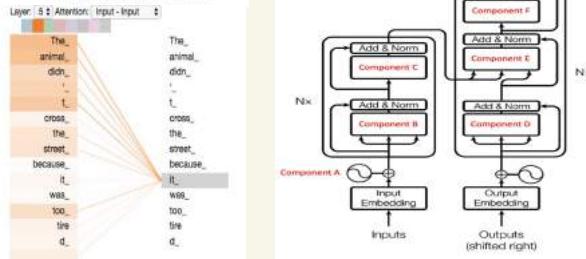
The output is a weighted sum of the values, where the weight assigned to each value is determined by the dot-product of the query with all the keys

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$



It is entirely built on the **self-attention** mechanisms without using sequence-aligned recurrent architecture

Self-attention is a type of attention mechanism where the model **makes prediction** for one part of a data sample using other parts of the observation about the same sample.



What does "it" in this sentence refer to? Is it referring to the street or to the animal?

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".

The encoder's inputs first flow through a self-attention layer – a layer that **helps the encoder look at other words in the input sentence** as it encodes a specific word

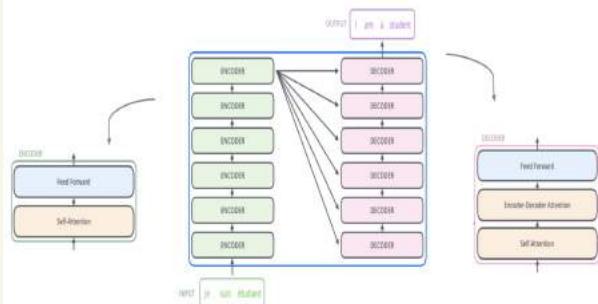
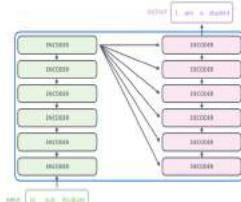
The outputs of the self-attention layer are fed to a **feed-forward neural network**.

The exact same feed-forward network is **independently applied** to each position (each word/token).

The decoder has both those layers, but between them is an attention layer that **helps the decoder focus on relevant parts of the input sentence**

The encoding component is a stack of encoders

The decoding component is a stack of decoders of the same number



Self-Attention in Detail

Matrix Calculation of Self-Attention

First Step

Create three vectors from each of the encoder's input vectors (in this case, the **embedding** of each word).



So for each word, we create a **Query** vector, a **Key** vector, and a **Value** vector.

These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

What are the "query", "key", and "value" vectors?

The names "query", "key" are inherited from the field of **information retrieval**

The dot product operation returns a measure of similarity between its inputs, so the weights $\frac{QK^T}{\sqrt{d_k}}$ depend on the relative similarities between the n -th query and all of the keys

The softmax function means that the key vectors "compete" with one another to contribute to the final result.

Second Step

Calculate a score for each word of the input sentence against a word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.

Third Step

Divide the scores by $\sqrt{d_k}$, the square root of the dimension of the key vectors

This leads to having more stable gradients (large similarities will cause softmax to saturate and give vanishing gradients)

Fourth Step

Softmax for normalization

$$z_1 = q_1 \cdot k_1 / \sqrt{d_k} = 112 / \sqrt{64} = 112/8 = 14, z_2 = q_1 \cdot k_2 / \sqrt{d_k} = 96 / \sqrt{64} = 96/8 = 12$$

$$\text{softmax}(z_1) = \exp(z_1) / \sum_{i=1}^2 (\exp(z_i)) = 0.88, \text{softmax}(z_2) = \exp(z_2) / \sum_{i=1}^2 (\exp(z_i)) = 0.12$$

Fifth Step

Multiply each value vector by the softmax score

Sixth Step

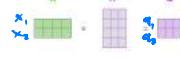
Sum up the weighted value vectors to get the output of the self-attention layer at this position (for the first word)

$$\text{Attention}(Q, K, V) = \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

First Step

Calculate the Query, Key, and Value matrices.

Pack our embeddings into a matrix X , and multiplying it by the weight matrices we've trained (W^Q, W^K, W^V)



Every row in the X matrix corresponds to a word in the input sentence.



Second Step

Calculate the outputs of the self-attention layer. SoftMax is **row-wise**.

Example

Assuming we have two sequences:
(1, 2, 3)
(4, 5, 6)

And the given W^Q, W^K, W^V matrices are respectively given as
 $\begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix}, \begin{pmatrix} 0.05 & 0.05 \\ 0.06 & 0.05 \\ 0.07 & 0.05 \end{pmatrix}, \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix}$

Compute the output of the scaled-dot product attention, $\text{Attention}(Q, K, V)$.

Step 1: Find Q, K, V

$$Q = XW^Q = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.01 & 0.03 \\ 0.02 & 0.02 \\ 0.03 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.14 & 0.10 \\ 0.32 & 0.28 \end{pmatrix}$$

$$K = XW^K = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.05 & 0.05 \\ 0.06 & 0.05 \\ 0.07 & 0.05 \end{pmatrix} = \begin{pmatrix} 0.38 & 0.30 \\ 0.92 & 0.75 \end{pmatrix}$$

$$V = XW^V = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \begin{pmatrix} 0.02 & 0.02 \\ 0.01 & 0.02 \\ 0.01 & 0.01 \end{pmatrix} = \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix}$$

Step 2: Find $\text{Attention}(Q, K, V)$

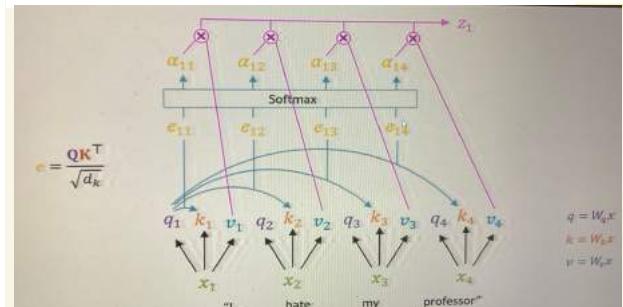
$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$\frac{e^{z_1}}{\sum_j e^{z_j}} = \frac{e^{0.0588}}{e^{0.0588} + e^{0.1441}} = 0.4787$$

Perform row-wise SoftMax

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \text{Softmax}\left(\frac{(0.14 \cdot 0.10)(0.38 \cdot 0.92)}{\sqrt{2}}\right)V = \text{Softmax}\left(\frac{0.0588 \cdot 0.1441}{0.1454 \cdot 0.3566}\right)V = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix}V$$

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \begin{pmatrix} 0.4787 & 0.5213 \\ 0.4474 & 0.5526 \end{pmatrix} \begin{pmatrix} 0.07 & 0.09 \\ 0.19 & 0.24 \end{pmatrix} = \begin{pmatrix} 0.1326 & 0.1682 \\ 0.1363 & 0.1729 \end{pmatrix}$$



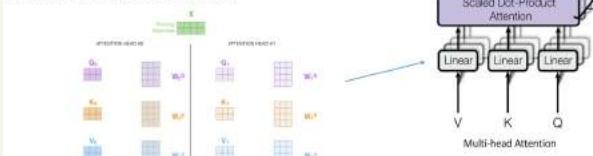
$$\begin{aligned} \text{Input} &: \text{Thinking} \quad \text{Machines} \\ \text{Embedding} &: x_1 \quad x_2 \\ \text{Queries} &: q_1 \quad q_2 \\ \text{Keys} &: k_1 \quad k_2 \\ \text{Values} &: v_1 \quad v_2 \\ \text{Score} &: q_1 \cdot k_1 = 112 \quad q_1 \cdot k_2 = 96 \\ \text{Divide by } \sqrt{d_k} &: 14 \quad 12 \\ \text{Softmax} &: 0.88 \quad 0.12 \\ \text{Softmax } X \text{ value} &: \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} \quad \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \\ \text{Sum} &: z_1 = 0.88v_1 + 0.12v_2 \\ \text{Attention}(Q, K, V) &= \text{SoftMax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \end{aligned}$$

Multi-Head Self-Attention

Multi-Head Self-Attention

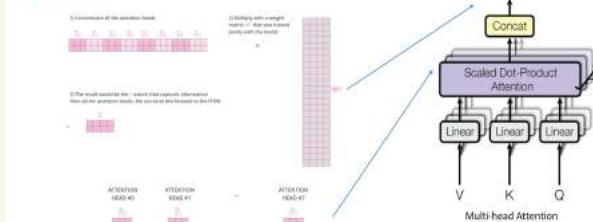
Rather than only computing the attention once, the multi-head mechanism runs through the scaled dot-product attention **multiple times in parallel**.

Due to different linear mappings, each head is presented with different versions of keys, queries, and values.



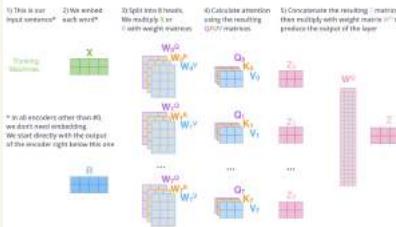
Multi-Head Self-Attention

The independent attention outputs are simply **concatenated** and linearly transformed into the expected dimensions.



Multi-Head Self-Attention

The big picture. Note that after the split each head can have a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.



Why?

"Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions."

SCALE UP MODEL

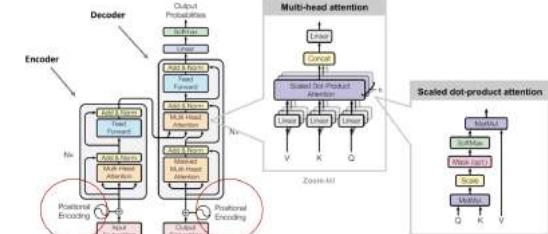
An intuitive example

Given a sentence:

"Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning."

Given "representation learning", the first head attends to "Deep learning," while the second head attends to the more general term "machine learning methods".

Positional Encoding



Self-attention layer works on sets of vectors and it **doesn't know the order** of the vectors it is processing

The positional encoding has the **same dimension** as the input embedding

Adds a vector to each input embedding to give information about the **relative or absolute position** of the tokens in the sequence

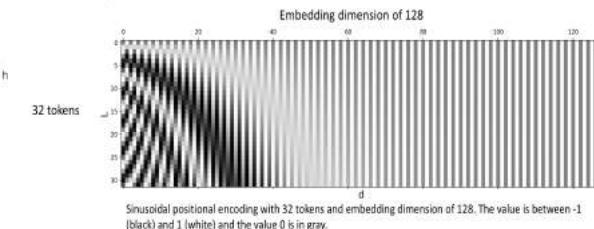
These vectors follow a specific pattern

What might this pattern look like?

Each row corresponds to the positional encoding of a vector.

The first row would be the vector we'd add to the embedding of the first word in an input sequence.

Each position is uniquely encoded and the encoding can deal with sequences longer than any sequence seen in the training time.



Sinusoidal positional encoding - interweaves the two signals (sine for even indices and cosine for odd indices)

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

where pos is the position and i is the dimension, $[0, \dots, d_{model}/2]$

Example: Positional Encoding

Example:

Given the following Sinusoidal positional encoding, calculate the $PE(pos=1)$ for the first five dimensions $[0, 1, 2, 3, 4]$. Assume $d_{model} = 512$

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Solution:

Given $pos = 1$ and $d_{model} = 512$

At dimension 0, $2i = 0$ thus $i = 0$, therefore $PE_{(pos, 2i)} = PE_{(1,0)} = \sin(1/10000^{0/512})$

At dimension 1, $2i + 1 = 1$ thus $i = 0$, therefore $PE_{(pos, 2i+1)} = PE_{(1,1)} = \cos(1/10000^{0/512})$

At dimension 2, $2i = 2$ thus $i = 1$, therefore $PE_{(pos, 2i)} = PE_{(1,2)} = \sin(1/10000^{2/512})$

At dimension 3, $2i + 1 = 3$ thus $i = 1$, therefore $PE_{(pos, 2i+1)} = PE_{(1,3)} = \cos(1/10000^{2/512})$

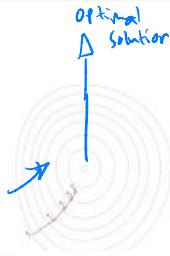
At dimension 4, $2i = 4$ thus $i = 2$, therefore $PE_{(pos, 2i)} = PE_{(1,4)} = \sin(1/10000^{4/512})$

Gradient descent (GD)

- Iteratively step in the direction of the negative gradient
- Gradient descent

$$w^{t+1} = w^t - \eta_t \frac{\partial f(w^t)}{\partial w}$$

New weight
Old weight
Learning rate
Gradient



Batch Gradient Descent

- Full sum is expensive when N is large

```
# Vanilla gradient descent
w = initialize_weights()
for t in range(num_steps):
    dw = compute_gradient(loss_fn, data, w)
    w -= learning_rate * dw
```

Stochastic Gradient Descent (SGD)

- Approximate sum using a minibatch of examples
- $32 / 64 / 128$ common minibatch size
- Additional hyperparameter on batch size

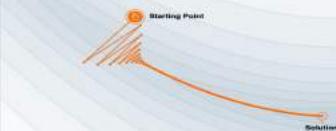
```
# Stochastic gradient descent
w = initialize_weights()
for t in range(num_steps):
    minibatch = sample_data(data, batch_size)
    dw = compute_gradient(loss_fn, minibatch, w)
    w -= learning_rate * dw
```

GD with Momentum

Deep neural networks have very complex error profiles. The method of momentum is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.

When the error function has the form of a shallow ravine leading to the optimum and steep walls on the side, stochastic gradient descent algorithm tends to oscillate near the optimum. This leads to very slow converging rates. This problem is typical in deep learning architecture.

Momentum is one method of speeding the convergence along a narrow ravine.



Momentum update is given by:

$$\begin{aligned} V &\leftarrow \gamma V - \alpha \nabla_w \\ W &\leftarrow W + V \end{aligned}$$

where V is known as the velocity term and has the same dimension as the weight vector W . The momentum parameter $\gamma \in [0, 1]$ indicates how many iterations the previous gradients are incorporated into the current update.

The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction.

Often, γ is initially set to 0.1 until the learning stabilizes and increased to 0.9 thereafter.

Learning rate

$$\text{New weight} \rightarrow w^{t+1} = w^t - \eta_t \frac{\partial f(w^t)}{\partial w}$$

New weight
Old weight
Learning rate
Gradient



Learning rate

The learning rate, often noted α or sometimes η , indicates at which pace the weights get updated. It can be fixed or adaptively changed.

The current most popular method is called Adam, which is a method that adapts the learning rate.

Adaptive learning rates

Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution.

While Adam optimizer is the most commonly used technique, others can also be useful.

Algorithms with adaptive learning rates:

- AdaGrad torch.optim.Adagrad()
- RMSprop torch.optim.RMSprop()
- Adam torch.optim.Adam()

Annealing

One way to adapting the learning rate is to use an annealing schedule: that is, to start with a large learning factor and then gradually reducing it.

A possible annealing schedule (t – the iteration count):

$$\alpha(t) = \frac{\alpha}{\varepsilon + t}$$

α and ε are two positive constants. Initial learning rate $\alpha(0) = \alpha/\varepsilon$ and $\alpha(\infty) = 0$.

AdaGrad

Adaptive learning rates with annealing usually works with convex cost functions.

Learning trajectory of a neural network minimizing non-convex cost function passes through many different structures and eventually arrive at a region locally convex.

AdaGrad algorithm individually adapts the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all historical squared values of the gradient. This improves the learning rates, especially in the convex regions of error function.

$$\begin{aligned} r &\leftarrow r + (\nabla_w)^2 \\ W &\leftarrow W - \frac{\alpha}{\varepsilon + \sqrt{r}} \cdot (\nabla_w) \end{aligned}$$

In other words, learning rate:

$$\delta = \frac{\alpha}{\varepsilon + \sqrt{r}}$$

α and ε are two parameters.

RMSprop

RMSprop improves upon AdaGrad algorithms uses an exponentially decaying average to discard the history from extreme past so that it can converge rapidly after finding a convex region.

$$\begin{aligned} r &\leftarrow \rho r + (1 - \rho)(\nabla_w)^2 \\ W &\leftarrow W - \frac{\alpha}{\sqrt{\varepsilon + r}} \cdot (\nabla_w) \end{aligned}$$

The decay constant ρ controls the length of the moving average of gradients. Default value = 0.9.

RMSprop has been shown to be an effective and practical optimization algorithm for deep neural networks.

Adam Optimizer

Adams optimizer combines RMSprop and momentum methods. Adam is generally regarded as fairly robust to hyperparameters and works well on many applications.

$$\begin{aligned} \text{Momentum term: } s &\leftarrow \rho_1 s + (1 - \rho_1)\nabla_w \\ \text{Learning rate term: } r &\leftarrow \rho_2 r + (1 - \rho_2)(\nabla_w)^2 \\ s &\leftarrow \frac{s}{1 - \rho_1} \\ r &\leftarrow \frac{r}{1 - \rho_2} \\ W &\leftarrow W - \frac{\alpha}{\sqrt{\varepsilon + r}} \cdot s \end{aligned}$$

Note that s adds the momentum and r contributes to the adaptive learning rate.

Suggested defaults: $\alpha = 0.001$, $\rho_1 = 0.9$, $\rho_2 = 0.999$, and $\varepsilon = 10^{-8}$

Vision Transformer (ViT)

- Do not have decoder

• Reshape the image $X \in \mathbb{R}^{H \times W \times C}$ into a sequence of flattened 2D patches $\mathbf{x} \in \mathbb{R}^{N \times (P^2 \cdot C)}$
 (H, W) is the resolution of the original image
 C is the number of channels
 (P, P) is the resolution of each image patch
 $N = HW/P^2$ is the resulting number of patches

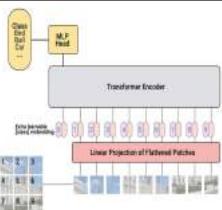


image \rightarrow word seq

Prepend a **learnable embedding** ($\mathbf{z}_0^0 = \mathbf{x}_{\text{class}}$) to the sequence of embedded patches

$$\mathbf{z}_0 = [\mathbf{x}_{\text{class}}; \mathbf{x}_1^1 \mathbf{E}; \mathbf{x}_2^2 \mathbf{E}; \dots; \mathbf{x}_P^P \mathbf{E}] + \mathbf{E}_{\text{pos}}$$

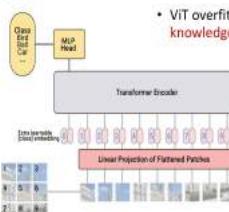
Patch embedding - Linearly embed each of them to D dimension with a trainable linear projection $\mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}$

Add learnable position embeddings $\mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D}$ to retain positional information

Feed the resulting sequence of vectors \mathbf{z}_0 to a standard Transformer encoder

$$\begin{aligned} \text{Transformer Encoder: } & \quad \mathbf{x}_l = \text{MSA}(\text{LN}(\mathbf{x}_{l-1})) + \mathbf{x}_{l-1}, \quad \mathbf{E} \in \mathbb{R}^{(P^2 \cdot C) \times D}, \quad \mathbf{E}_{\text{pos}} \in \mathbb{R}^{(N+1) \times D} \\ & \quad l = 1 \dots L \\ \text{Classification Head: } & \quad \mathbf{y} = \text{MLP}(\text{LN}(\mathbf{x}_L)) + \mathbf{x}_L \end{aligned}$$

The output of the additional [class] token is transformed into a class prediction via a small multi-layer perceptron (MLP) with tanh as non-linearity in the single hidden layer.

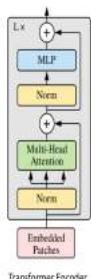


Transformer Encoder

- Consists of a multi-head self-attention module [MSA], followed by a 2-layer MLP (with **GELU**)
- LayerNorm (LN) is applied before MSA module and MLP, and a residual connection is applied after each module.

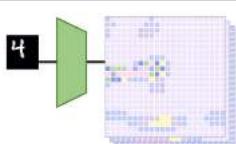
$$\mathbf{x}'_l = \text{MSA}(\text{LN}(\mathbf{x}_{l-1})) + \mathbf{x}_{l-1},$$

$$\mathbf{z}_l = \text{MLP}(\text{LN}(\mathbf{x}'_l)) + \mathbf{x}'_l,$$



ViT conducts global self-attention

- Relationships between a token and all other tokens are computed
- Quadratic complexity with respect to the number of tokens
- Not tractable for dense prediction or to represent a high-resolution image



An equivariant mapping is a mapping which preserves the algebraic structure of a transformation.

A translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping.

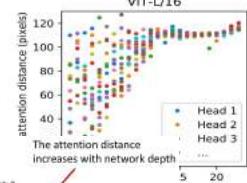
Examine the attention distance, analogous to receptive field in CNN

Compute the average distance in image space across which information is integrated, based on the attention weights.

Attention distance was computed for 128 example images by averaging the distance between the query pixel and all other pixels, weighted by the attention weight.

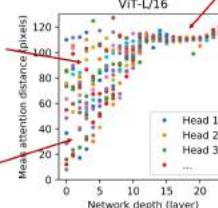
Each dot shows the mean attention distance across images for one of 16 heads at one layer. Image width is 224 pixels.

An example: if a pixel is 20 pixels away and the attention weight is 0.5 the distance is 10.



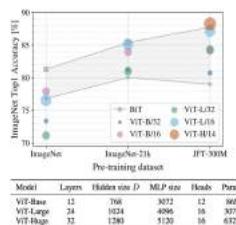
Some heads attend to most of the image already in the lowest layers, showing the capability of ViT in integrating information globally

Other attention heads have consistently small attention distances in the low layers



Performance of ViT

- ViT performs significantly worse than the CNN equivalent (BiT) when trained on ImageNet (1M images).
- However, on ImageNet-21k (14M images) performance is comparable, and on JFT (300M images), ViT outperforms BiT.
- ViT overfits the ImageNet task due to its lack of built-in knowledge about images

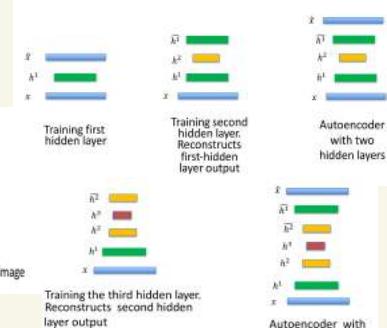


Deep stacked autoencoders

Deep autoencoders can be built by **stacking autoencoders** one after the other.

Training of deep autoencoders is done in a step-by-step fashion **one layer at a time**:

- After training the first level of denoising autoencoder, the resulting hidden representation is used to train a second level of the denoising encoder.
- The second level hidden representation can be used to train the third level of the encoders.
- This process is repeated and deep stacked autoencoder can be realized.



Inductive Bias

- ViT has much less image-specific inductive bias than CNNs

Inductive bias in CNN

- Locality
- Two-dimensional neighborhood structure
- Translation equivariance

ViT

- Only MLP layers are local and translationally equivariant. Self-attention is global
- Two dimensional neighborhood is used sparingly – i) only at the beginning where we cut image into patches, ii) learnable position embedding (spatial relations have to be learned from scratch)

An equivariant mapping is a mapping which preserves the algebraic structure of a transformation.

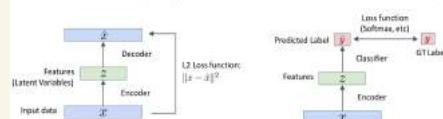
A translation equivariant mapping is a mapping which, when the input is translated, leads to a translated mapping.

Semi-Supervised Classification

- Many images, but few ground truth labels

start unsupervised
train autoencoder on many images

supervised fine-tuning
train classification network on labeled images



10. Autoencoders

Supervised Learning

Data: (x, y)
 x is data, y is label

Goal: Learn a function to map $x \rightarrow y$

Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.

Dimensionality Reduction (e.g. Principal Components Analysis)

Density Estimation

Unsupervised Learning

Data: x
Just data, no labels!

Goal: Learn some underlying hidden structure of the data

Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.

Feature Learning (e.g. autoencoders)

Autoencoders

Unsupervised method for learning feature vectors from raw data x , without any labels

Features should extract useful information (maybe object identities, properties, scene type, etc.) that we can use for downstream tasks

Originally: Linear + nonlinearity (sigmoid)
Later: Deep, fully-connected
Later: ReLU CNN

* Problem: How can we learn this feature transform from raw data?
Idea: Use the features to reconstruct the input data with a **decoder**
"Autoencoding" = encoding itself

An autoencoder is a neural network that is trained to attempt to copy its input to its output. It's hidden layer describes a **code** that represents the input.

The network consists of two parts: an **encoder** and a **decoder**:

Given an input x , the hidden-layer performs the encoding function $\mathbf{h} = \mathbf{E}(x)$ and the decoder φ produces the reconstruction $\hat{x} = \varphi(\mathbf{h})$.

If the autoencoder succeeds:
 $y = \varphi(\mathbf{h}) = \varphi(\mathbf{E}(x)) = x$

In order to be useful, autoencoders are designed to be unable to copy exactly and enable to copy only inputs that resembles the training data.

Since the model is forced to prioritize which aspects of the input should be copied, the hidden-layer often **learns useful properties of the data**.

Reverse mapping from the hidden layer to the output can be optionally constrained to be the same as the input to hidden-layer mapping. That is, if encoder weight matrix is W , the decoder weight matrix is W^T .

Hidden layer and output layer activation can be then written as

$$h = f(Wx + b)$$

$$\hat{x} = f(Wh + c)$$

f is usually a sigmoid.

Training autoencoders

The **cost function of reconstruction** can be measured by many ways, depending on the appropriate distributional assumptions on the inputs.

Learning of autoencoders is **unsupervised** as no specific targets are given.

Given a training set $\{\mathbf{x}_p\}_{p=1}^P$

The mean-square-error cost is usually used if the data is assumed to be continuous and Gaussian distributed:

$$J_{mse} = \frac{1}{P} \sum_{p=1}^P \|y_p - x_p\|^2$$

where y_p is the output for input x_p and $\|\cdot\|$ denotes the magnitude of the vector.

If the inputs are interpreted as bit vectors or vectors of bit probabilities, cross-entropy of the reconstruction can be used:

$$J_{cross\text{-}entropy}} = - \sum_{p=1}^P (x_p \log y_p + (1 - x_p) \log(1 - y_p))$$

Learning are done by using gradient descent:

$$\begin{aligned} W &\leftarrow W - \alpha \nabla_W J \\ b &\leftarrow b - \alpha \nabla_b J \\ c &\leftarrow c - \alpha \nabla_c J \end{aligned}$$

Example 1

Given input patterns:
 $(1, 0, 1, 0, 0), (0, 0, 1, 1, 0), (1, 1, 0, 1, 1)$ and $(0, 1, 1, 1, 0)$

Design an autoencoder with 3 hidden units.
Show the representations of inputs at the hidden layer upon convergence.

$H = f(XW + B)$
 $Y = f(HW^T + C)$
 $c = 0.1 (Y = 0.5)$

Gradient descent on entropy

$$J = - \sum_{p=1}^P \sum_{j=1}^3 (x_{pj} \log y_{pj} + (1 - x_{pj}) \log(1 - y_{pj}))$$

Bit errors = $\sum_{p=1}^P \sum_{j=1}^3 |0y_{pj} \neq x_{pj}|$

Denoising Autoencoders (DAE)

A **denoising autoencoder** (DAE) receives corrupted data points as input. It is trained to predict the original uncorrupted data points as its output.

The idea of DAE is that in order to do this, the hidden layer to **discover** more features it prevent it from simply learning the identity function. The autoencoder **attempts to reconstruct the input from a corrupt version of it**.

In other words, DAE attempts to encode the input (preserve the information about itself) and attempts to **undo the effect of corruption process applied to the input of the autoencoder**.

For training DAE:

- First, input data is corrupted to mimic the noise in the images: $x \rightarrow \tilde{x}$

\tilde{x} is the corrupted version of input data, each input x_i of input data is added with additive or multiplicative noise.

Additive noise:
 $\tilde{x}_i = x_i + \varepsilon$

where noise ε is Gaussian distributed: $\varepsilon \sim N(0, \sigma^2)$. And σ is the standard deviation that determines the S/N ratio. Usually used for continuous data.

Multiplicative noise:
 $\tilde{x}_i = ex_i$

where noise x could be Binomially distributed: $\varepsilon \sim \text{Binomial}(p)$. And p is the probability of ones and $1 - p$ is the probability of zeros (noise). Usually, used for binary data.

Autoencoders

Input dimension n and hidden dimension M :

- If $M < n$, **undercomplete autoencoders**
- If $M > n$, **overcomplete autoencoders**

Undercomplete autoencoders

In undercomplete autoencoders, the hidden-layer has a **lower dimension** than the input layer.

By learning to approximate an n -dimensional inputs with M ($< n$) number of hidden units, we obtain a **lower dimensional representation** of the input signals. The network reconstructs the input signals from the reduced-dimensional hidden representation.

Learning an undercomplete representation forces the autoencoder to capture the most salient features.

By limiting the number of hidden neurons, hidden structures of input data can be inferred from autoencoders. For example, correlations among input variables, learning principal components of data, etc.

Overcomplete autoencoders

In overcomplete autoencoders, the hidden-layer has a **higher dimension** than the dimension of the input.

In order to learn useful information from overcomplete autoencoders, it is necessary use **some constraints** on its characteristics.

Even when the hidden dimensions are large, one can still explore interesting structures of inputs by introducing other constraints such as **'sparsity'** of input data.

Regularizing autoencoders

Both undercomplete and overcomplete autoencoders fail to learn anything useful if the encoder and decoder are given too much capacity. Some form of constraints are needed in order to make them useful.

Deep autoencoders are only possible when some **constraints** are imposed on the cost function.

Regularized autoencoders incorporate a penalty to the cost function to learn interesting features from the input.

Regularized autoencoders provide the ability to train any autoencoder architecture successfully by choosing suitable code dimension and the capacity of the encoder and decoder.

With regularized autoencoders, one can use **larger model capacity** (for example, deeper autoencoders) and large code size.

Regularized autoencoders add an appropriate penalty function Ω to the cost function:

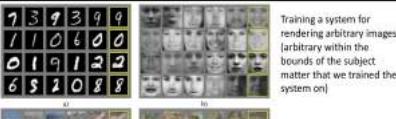
$$J = \frac{1}{P} + \beta \Omega(h)$$

where β is the penalty or regularization parameter. The penalty is usually **imposed on the hidden activations**.

A regularized autoencoder can be nonlinear and overcomplete but still learn something useful about the data distribution, even if the model capacity is great enough to learn trivial identity function.

The regularized loss function encourages the model to have other properties besides the ability to copy its input to its output.

Results from GAN



Example 1

Design a GAN to learn Gaussian distributed data with mean = 1.0 and standard deviation = 1.0. The generator receives uniformly distributed 1-dimensional inputs in the range [-1, +1].

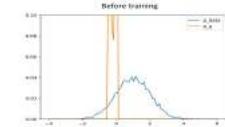
Discriminator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 10 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 10 (activation: Tanh)
- Output dimension = 1 (binary classification)

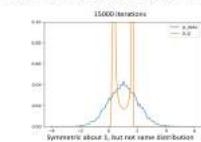
Generator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 5 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 5 (activation: Tanh)
- Output dimension = 1 (activation: None)

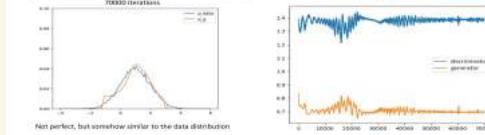
Histogram of 10000 randomly drawn points from $U[-1, 1]$



Histogram of 10000 randomly drawn points from $U[-1, 1]$

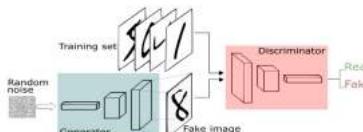


Histogram of 10000 randomly drawn points from $U[-1, 1]$



Example 2

Design a GAN to generate MNIST images from a uniformly distributed noise vector of 100 dimensions.



Example 2

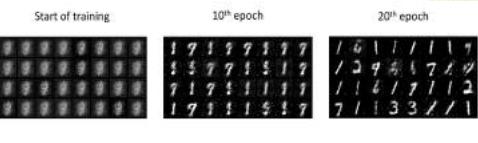
Generator:

- Input dimension = 100, drawn from a uniform distribution $U[-1, 1]$
- Hidden layers:
 - Number of hidden neurons = 256 (activation: ReLU)
 - Number of hidden neurons = 512 (activation: ReLU)
 - Number of hidden neurons = 1024 (activation: ReLU)
- Output dimension = 784 (activation: Tanh)

Discriminator:

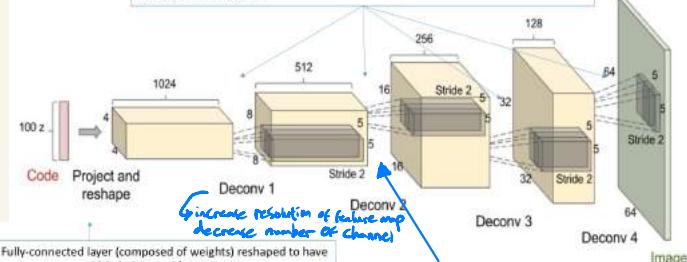
- Input dimension = 784
- Hidden layers:
 - Number of hidden neurons = 1024 (activation: ReLU)
 - Number of hidden neurons = 512 (activation: ReLU)
 - Number of hidden neurons = 256 (activation: ReLU)
- Output dimension = 1 (activation: Sigmoid, binary classification)

eg11.2.ipynb



Deep Convolutional GAN (DCGAN)

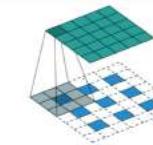
Most "deconv"s are batch normalized:
Normalize responses to have zero mean and unit variance over the entire mini-batch
Faster and more stable



We will go through the code in the tutorial

Fractionally-strided convolutions

Fractional strides involve: zeros are inserted between input units, which makes the kernel move around at a slower pace than with unit strides



<https://pytorch.org/docs/stable/generated/torch.nn.ConvTranspose2d.html>

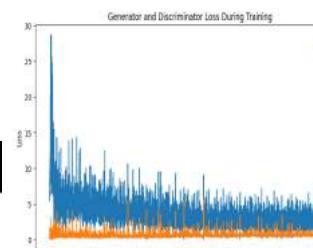
Up-sampling by fractional striding

Input 3x3, output = 5x5

stride = 1 convolution window = 3×3 (with respect to output)

Other tricks proposed by DCGAN

- Adam optimizer (adaptive moment estimation) = similar to SGD but with less parameter tuning
- Momentum = 0.5 (usually is 0.9 but training oscillated and was unstable)
- Low learning rate = 0.0002



Sanity check by walking on the manifold



Interpolating between a series of random points in z (z = the 100-d random numbers)
 $e.g.,$
 $z_{new} = m z_1 + (1 - m) z_2$

Note the smooth transition between each scene
 \leftarrow the window slowly appearing

This indicates that the space learned has smooth transitions!
 (and is not simply memorizing)

Long Short-term Memory (LSTM)

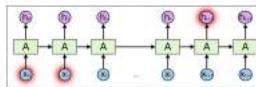
Long-term dependency

"The man who ate my pizza has purple hair"

Purple hair is for the man and not the pizza!
This is a long term dependency.

There are cases where we need even more context:

- To predict last word in "I grew up in France...long paragraph...I speak French"
- Using only recent information suggests that the last word is the name of a language. But more distant past indicates that it is French
- RNNs work upon the fact that the result of an information is dependent on its previous state or previous n time steps
- RNNs have difficulty in learning long range dependencies
- Gap between relevant information and where it is needed is large



Exploding and vanishing gradients in RNN

During gradient back-propagation learning of RNN, the gradient can end up being multiplied a large number of times (as many as the number of steps) by the weight matrix associated with the connections between the neurons of the recurrent hidden layer.

Note that each time the activations are forward propagated in time, the activations are multiplied by W and each time the gradients are back propagated, the gradients are multiplied by W^T .

$$\nabla_{\text{grad}} = W \text{diag}(1 - h^2(t-1)) \nabla_{\text{grad+1}} + V \nabla_{\text{grad}}$$

Gradient from reverse direction

If the weights in this matrix are small, the recursive derivative can lead to a situation called **vanishing gradients** where the gradient signal gets so small that learning either becomes very slow or stops working altogether.



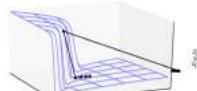
Contribution from the earlier steps becomes insignificant in the gradient.

Conversely, if the weights in this matrix are large, it can lead to a situation where the gradient signal is so large that it can cause learning to diverge. This is often referred to as **exploding gradients**.

Exploding gradient easily solved by clipping the gradients at a predefined threshold value.

Vanishing gradient is more concerning

Gradient Clipping



Due to long term dependencies, RNN tend to have gradients having very large or very small magnitudes. The large gradients resemble cliffs in the error landscape and when the gradient descent encounters the gradient updates can move the parameters away from true minimum.

A gradient clipping is employed to avoid the gradients become too large.

Commonly, two methods are used:

1. Clip the gradient g when it exceeds a threshold:

$$\text{if } \|g\| > v: \quad \|g\| \rightarrow v$$

$$\text{if } \|g\| > v: \quad \|g\| \rightarrow \frac{v}{\|g\|} v$$

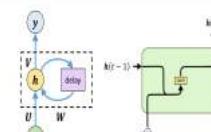
Basic RNN unit

The RNN cell (memory unit) is characterized by

$$h(t) = \phi(W^T x(t) + W h(t-1) + b)$$

where ϕ is the tanh activation function and RNN cell is referred to as tank units.

RNN build with simple tank units are also referred to as **vanilla RNN**.

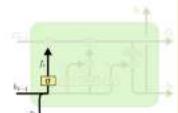


Forget gate

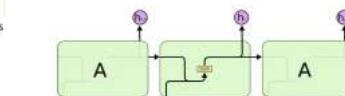
The forget gate can modulate the memory cell's self-recurrent connection, allowing the cell to remember or forget its previous state, as needed.

$$f(t) = \sigma(W_f^T x(t) + W_f h(t-1) + b_f)$$

Value of $f(t)$ determines if $c(t-1)$ is to be remembered or not.



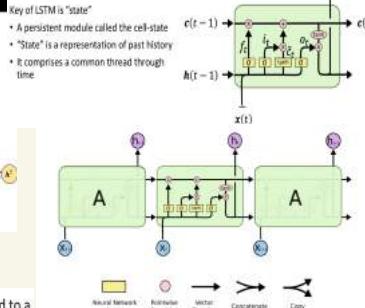
Basic RNN layer



Long short-term memory (LSTM) unit

Key of LSTM is "state"

- A persistent module called the cell-state
- "State" is a representation of past history
- It comprises a common thread through time



Cells are connected recurrently to each other - Replacing hidden units of ordinary recurrent networks

LSTMs provide a solution by incorporating memory units that allow the network to learn when to **forget previous hidden states** and when to **update hidden states** given new information.

Instead of having a single neural network layer, there are four, interacting in a very special way.

The key to LSTM is the **cell state** $c(t)$ - the horizontal line through the top of the diagram.

The long-term memory.

Like a conveyor belt that runs through entire chain with minor interactions

The LSTM has the ability to add and remove information to the cell states through gates.

Gates are a way to optionally let information through.

They are composed of sigmoid neural net layer and pointwise multiplication operations.

A value of zero means "let nothing through", while a value of one means "let everything through".

An LSTM has three of these gates to control cell state.

Input gate

The input gate can allow incoming signal to alter the state of the memory cell or block it. It decides what new information to store in the cell stage.

This has two parts:

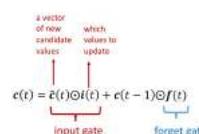
A sigmoid input gate layer decides which values to update;

A tanh layer creates a vector of new candidate values $\tilde{c}(t)$ that could be added to the state.

$$i(t) = \sigma(W_i^T x(t) + W_i h(t-1) + b_i)$$

$$\tilde{c}(t) = \phi(W_c^T x(t) + W_c h(t-1) + b_c)$$

Cell state



Example: Language modeling

In the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in previous steps

Output gate

The output gate can allow the state of the memory cell to have an effect on other neurons or prevent it.

$$o(t) = \sigma(W_o^T x(t) + W_o h(t-1) + b_o)$$

$$h(t) = \phi(c(t)) \odot o(t)$$

LSTM unit

*test control
not calculation*

$$i(t) = \sigma(W_i^T x(t) + W_i h(t-1) + b_i)$$

$$f(t) = \sigma(W_f^T x(t) + W_f h(t-1) + b_f)$$

$$o(t) = \sigma(W_o^T x(t) + W_o h(t-1) + b_o)$$

$$\tilde{c}(t) = \phi(W_c^T x(t) + W_c h(t-1) + b_c)$$

$$c(t) = \tilde{c}(t) \odot i(t) + c(t-1) \odot f(t)$$

$$h(t) = \phi(c(t)) \odot o(t)$$

LSTM introduces a gated cell where the information flow can be controlled.

The most important component is the state unit $c(t)$ (for time step t and cell i) which has a linear self-loop.

The self-loop weight is controlled by a **forget gate**, which sets this weight to a value between 0 and 1 via a sigmoid unit.

Clever idea!

The self-loop inside the cell is able to produce paths where the gradient can flow for long duration and to make the weight on the time scale of integration can be changed dynamically based on the input sequence.

In this way, LSTM help preserve error terms that can be propagated through many layers and time steps.

Implementation

The constructor initializes the module and sets up the dropout layer with the given dropout probability.

`d_model` is the dimension of the embeddings (or the depth of the model).

`position` is a tensor containing integers from 0 to `max_len-1`, representing each position in the sequence.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        ...
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

The sinusoidal functions (sine for even indices and cosine for odd indices) are applied to the positions and the results are stored in `pe`. This creates a unique positional encoding for each position.

`max_len` is the maximum expected length of the sequences.

`div_term` is a scaling term used to adjust the rate of the sinusoidal functions.

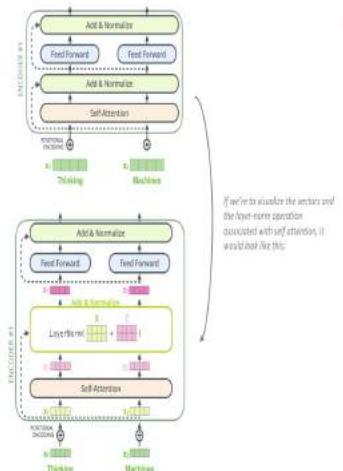
$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i+1/d_{model}})$$

The positional encoding (`self.pe`) corresponding to the length of the input sequence is added to the input tensor `x`.

This addition operation effectively combines the positional information with the embeddings of the tokens.

Transformer Encoder



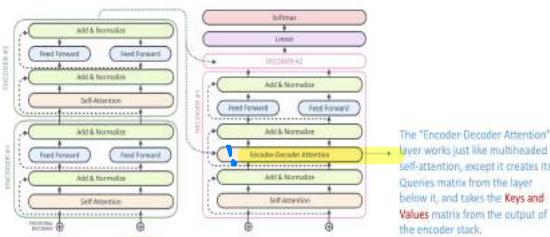
Encoder

- A stack of $N = 6$ identical layers.
- Each layer has a multi-head self-attention layer and a simple position-wise fully connected feed-forward network.
- The linear transformations are the same across different positions, they use different parameters from layer to layer.
- Each sub-layer adopts a residual connection and a layer normalization.

$$FFN(z) = \max(0, zW_1 + b_1)W_2 + b_2$$

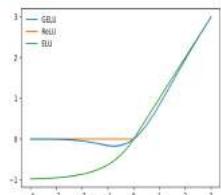
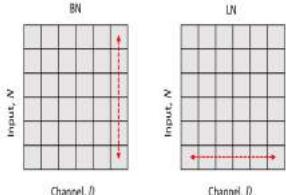
Transformer Decoder

The outputs of encoder go to the sub-layers of the decoder as well. If we're to think of a Transformer of two stacked encoders and decoders, it would look something like this:



How encoder and decoder work together

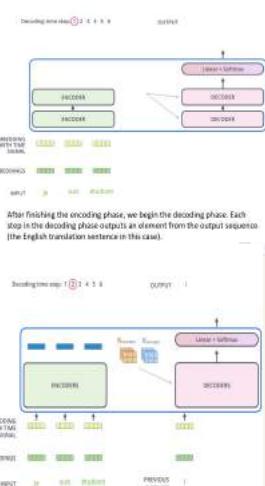
- The encoder starts by processing the input sequence.
- The output of the top encoder is then transformed into a set of attention vectors K and V .
- These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence.



How encoder and decoder work together

- The output of each step is fed to the bottom decoder in the next time step.
- Embed and add additional encoding to those decoder inputs. Process the inputs.
- Repeat the process until a special symbol is reached indicating the transformer decoder has completed its output.

Note:
In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to $-\infty$) before the softmax step in the self-attention calculation.



11. Generative Adversarial Networks

Why generative models?

- We've only seen **discriminative models** so far
 - Given an data x , predict a label y
 - Estimates $p(y|x)$

- Discriminative models have several key limitations**
 - Can't model $p(x)$, i.e., the probability of seeing a certain data
 - Thus, can't sample from $p(x)$, i.e., **can't generate new data**

- Generative models** (in general) cope with all of above
 - Can model $p(x)$
 - Can generate new data or images

Generative modeling

Density estimation – a core problem in unsupervised learning.



Image-to-Image Translation



Isha et al., "Image-to-Image Translation with Conditional Adversarial Networks", arXiv:1703.07004

Generative Adversarial Networks (GAN)

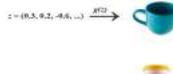


Gaussian

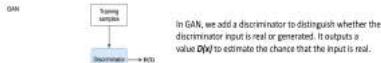
$$z = \mathcal{N}(0, 1)$$

or

$$z = \mathcal{U}(-1, 1)$$



GAN samples noise z using normal or uniform distribution and utilizes a deep network generator G to create an image $\hat{x} = G(z)$.



• Generative model G :

- Captures data distribution
- Fool $D(G(x))$
- Generate an image $G(z)$ such that $D(G(z))$ is wrong (i.e., $D(G(z)) = 1$)

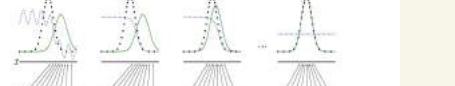
• Discriminative model D :

- Distinguishes between real and fake samples
- $D(x) = 1$ when x is a real image, and otherwise

--- data generating distribution

generative distribution

discriminative distribution



• D and G play the following two-player minimax game with value function $V(D, G)$

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_g(z)} [\log(1 - D(G(z)))]$$

Generator: input = random vector z
Output: $G(z)$ as synthetic image

Discriminator: input = generated/real images
Output: $D(x) = 1$ if real image

Example 1

Design a GAN to learn Gaussian distributed data with mean = 1.0 and standard deviation = 1.0. The generator receives uniformly distributed 1-dimensional inputs in the range [-1, +1].

Discriminator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 10 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 10 (activation: Tanh)
- Output dimension = 1 (binary classification)

Generator is a 4-layer DNN:

- Input dimension = 1
- Number of hidden neurons in hidden-layer 1 = 5 (activation: Tanh)
- Number of hidden neurons in hidden-layer 2 = 5 (activation: Tanh)
- Output dimension = 1 (activation: None)

eg11.1.ipynb

Histogram of 10000 randomly drawn points from $U(-1, 1)$:

Before training

Centred at 0

Std dev = 0.8

Histogram of 10000 randomly drawn points from $U(-1, 1)$:

After training

Centred at 0

Std dev = 0.8

Training Curves

Discriminator loss

Generator loss

Epochs

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 9000 10000

Loss iteration

1000 2000 3000 4000 5000 6000 7000 8000 900



1. (a) Give brief answers to the following. Each part carries 2 marks.
- (i) State the shape of the following tensor $[[[2, 0, 1], [[2, 5, 3]]]]$ written in Python.
 - (ii) Consider a trained logistic neuron. Would multiplying all its weights and the bias by a constant change its accuracy? State why?
 - (iii) State how you could use a linear neuron to learn a given nonlinear function.
 - (iv) State one advantage each for selecting mini-batch gradient descent method over batch gradient descent, and selecting mini-batch gradient descent method over stochastic gradient descent.
 - (v) To train a neuron layer, all the weights were initialized to 0.6. Is this good idea? Justify your answer.

You wish to train a convolutional neural network for classifying 12 different classes of flowers. You only have very limited training data, say 100 samples per class. Describe the steps to perform transfer learning to leverage a large-scale training dataset like ImageNet (which has 1000 classes).

- (b) Answer "TRUE" or "FALSE" to the following statements:
- (i) Layer normalization is usually used in Transformers.
 - (ii) Positional encoding helps to stabilize the training of Transformers.
 - (iii) Self-attention is performed in the decoder of Transformers.
 - (iv) Transformer model pays attention to a single most important word in a sentence.
 - (v) The feedforward network in the self-attention layer is not applied independently to each position.
 - (vi) Positional encoding has the same dimension as the input embedding.
- (6 marks)

- (c) Explain an advantage of attention-based models over recurrent-based ones.
(3 marks)
- (d) Describe a method to remedy mode collapse during training of Generative Adversarial Networks (GANs).
(3 marks)

- (iii) Describe a way to encourage an autoencoder to learn sparse hidden structure.
(3 marks)

Figure Q1(b) shows a dataset with each example belonging to one of the two classes, displayed in the space of its two features x_1 and x_2 . You are to design a discrete perceptron network with a single hidden layer to classify the examples.

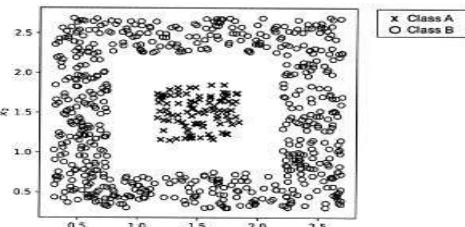


Figure Q1b

- (i) Draw a decision boundary separating the two classes to design the network.
(2 marks)
- (ii) State the number of neurons in the hidden layer.
(1 marks)
- (iii) Draw the network indicating the values of all weights and biases.
(8 marks)

1. (a) (i) (1, 2, 1, 3)
- (ii) Yes, because the activation function is a logistic one which does not give the same output when the inputs have been multiplied by the same constant. They need to be multiplied by varying ones according to the logarithmic graph for the accuracy to remain the same.
 - (iii) Change the nonlinear function into a linear one by substituting the variables by a variation of x (x_1, x_2, x_3).
 - (iv) Easier to fit in memory over batch gradient. It is more computationally efficient than stochastic.
 - (v) No, it can lead to the exploding gradient problem. Better initialization strategies could be used like random or Xavier initialization.
 - (vi) $(2 \times 5) + (5 \times 3) + 5 = 30$
 - (vii) 3-way over CV: Decreases training time – exclusive test set – good generalization. CV over 3-way: Reduces overfitting – better for lesser amount of data.

1. Select Source Task - You must select a related predictive modeling problem with an abundance of data where there is some relationship in the input data, output data, and/or concepts learned during the mapping from input to output data.
2. Develop Source Model - Next, you must develop a skillful model for this first task. The model must be better than a naive model to ensure that some feature learning has been performed. Augmentations can be performed on the dataset to artificially increase its size.
3. Reuse Model - The model fit on the source task can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the model, depending on the modeling technique used.
4. Tune Model - Optionally, the model may need to be adapted or refined on the input-output pair data available for the task of interest.

- (b) (i) TRUE
(ii) TRUE
(iii) FALSE
(iv) FALSE
(v) TRUE
(vi) FALSE
- (c) These mechanisms allow a task to focus on a set of elements of an input sequence, an intermediate sequence or a memory source.
- (d) To overcome this method, generate images and classify images in batches. The discriminator network should have extra layers or parameters to compute whether the
- 5
- If there are errors, please report using the form in bit.ly/SCSEPYPError

**22nd SCSE – Past Year Paper Solution (2021 – 2022 Semester 1)
CE/CZ 4042 – Neural Networks and Deep Learning**

batch is 'diverse' enough or not. Hence, if mode collapse occurs and the images generated aren't diverse enough, the discriminator will classify the images as fake. This forces the generator to generate diverse images

Sparse autoencoders are autoencoders with less hidden neurons typically used to learn features for another task such as classification. An autoencoder that has been regularized to be sparse must respond to unique statistical features of the dataset it has been trained on, rather than simply acting as an identity function. In this way, training to perform the copying task with a sparsity penalty can yield a model that has learned useful features as a byproduct.

- (ii) 4
- (iii) Let us initialize all biases to 0.
Using normal distribution initialization,
Weights connecting input layer to hidden layer:

1
If there are errors, please report using the form in bit.ly/SCSEPYPError

**22nd SCSE – Past Year Paper Solution (2021 – 2022 Semester 1)
CE/CZ 4042 – Neural Networks and Deep Learning**

$w \sim \text{Uniform Distribution } [- (6^{1/2}/(2+4)^{1/2}, + (6^{1/2}/(2+4)^{1/2})] = [-0.8660, +0.8660]$
 $w = [-0.8660, -0.6495, -0.433, -0.2165,$
 $0, 0.2165, 0.433, 0.6496]$

Weights connecting hidden layer to output layer:
 $w \sim \text{Uniform Distribution } [- (6^{1/2}/(4+1)^{1/2}, + (6^{1/2}/(4+1)^{1/2})] = [-1.095, +1.095]$
 $w = [-1.095,$
 $-0.5475,$
 $0,$
 $0.5475]$

Give brief answers to the following. Each part carries 2 marks.

- What is the total number of learnable parameters in a 2-layer network with 5 neurons in each layer and receiving 2-dimensional inputs.
- Write a tensor with the shape [2, 1, 3] and integer elements, as in Python.
- One wants to initialize all the weights in a 3-layer deep neural networks to 0.4. Is this a good idea? And why?
- You want to solve a classification problem with a training sample of 50 patterns. But the training loss was high. Then you decide to train with 1000 samples. Is this good approach? And why?
- You started training a neural network but the loss was neither decreasing or increasing. State two possible reasons.
- State two ways to handle local minima problem in gradient descent learning.

on No. 1 continues on Page 2

1

SC4001/CE4042/CZ4042

- State one advantage and one disadvantage of using a small batch size for training.
- State how you can fix underfitting in your neural network.

The output $\mathbf{y} \in \mathbb{R}^K$ of a linear neuron layer is given by $\mathbf{y} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$ where $\mathbf{x} \in \mathbb{R}^n$ is input, and \mathbf{W} and \mathbf{b} are the weight matrix and bias vector of the layer, respectively. Given a target vector $\mathbf{d} \in \mathbb{R}^K$, the square error loss J of the layer is given by $J = (\mathbf{d} - \mathbf{y})^T(\mathbf{d} - \mathbf{y})$.

- Write the expressions for $\nabla_{\mathbf{y}}J$, $\nabla_{\mathbf{W}}J$ and $\nabla_{\mathbf{b}}J$.

(5 marks)

- If L_2 -norm weight regularization penalty $\beta \|\mathbf{W}\|^2$ is added to the loss J , write the expressions for $\nabla_{\mathbf{W}}J_1$ and $\nabla_{\mathbf{b}}J_1$ where J_1 is the regularized loss. β is the penalty parameter.

(4 marks)

$$\begin{aligned} \text{(b) (i)} \quad \nabla_{\mathbf{y}}J &= -2(d - \mathbf{y}) \\ \nabla_{\mathbf{W}}J &= -2(d - \mathbf{y})\mathbf{x}^T \\ \nabla_{\mathbf{b}}J &= -2(d - \mathbf{y}) \end{aligned}$$

If there are errors, please report using:

Matrix/vector manipulation

You should be comfortable with these rules. They will come in handy when you want to simplify a expression before differentiating. All bold capitals are matrices, bold lowercase are vectors.

Rule	Comments
$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$	order is reversed, everything is transposed
$(\mathbf{a}^T \mathbf{b})^T = \mathbf{b}^T \mathbf{a}^T$	as above
$\mathbf{a}^T \mathbf{b} = \mathbf{b}^T \mathbf{a}$	(the result is a scalar, and the transpose of a scalar is itself)
$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$	multiplication is distributive
$(\mathbf{a} + \mathbf{b})^T = \mathbf{a}^T + \mathbf{b}^T$	as above, with vectors
$\mathbf{AB} \neq \mathbf{BA}$	multiplication is not commutative

Common vector derivatives

You should know these by heart. They are presented alongside similar-looking scalar derivatives to help memory. This doesn't mean matrix derivatives always look just like scalar ones. In these examples, b is constant scalar, and \mathbf{B} is a constant matrix.

Scalar derivative	Vector derivative
$f(x) \rightarrow \frac{df}{dx}$	$f(\mathbf{x}) \rightarrow \frac{df}{d\mathbf{x}}$
$bx \rightarrow b$	$x^T \mathbf{B} \rightarrow \mathbf{B}$
$bx \rightarrow b$	$\mathbf{x}^T b \rightarrow b$
$x^2 \rightarrow 2x$	$\mathbf{x}^T \mathbf{x} \rightarrow 2\mathbf{x}$
$bx^2 \rightarrow 2bx$	$\mathbf{x}^T \mathbf{B}x \rightarrow 2\mathbf{B}\mathbf{x}$

- Sinusoidal positional encoding uses sine and cosine functions of to encode the positions.

$$\begin{aligned} \text{(c) (i)} \quad \text{if } pos \text{ is even } PE_{pos,2i} &= \sin\left(\frac{pos}{10000^{2i}}\right), \\ \text{else if } pos \text{ is odd } PE_{pos,2i+1} &= \cos\left(\frac{pos}{10000^{2i}}\right) \end{aligned}$$

This is beneficial as it each position has a unique encoding value and since the sinusoidal function is continuous, models are able to generalize to longer sequences.

(ii) Dimension 0: -0.9589

If there are errors, please report using the form in bit.ly/SCSEPYPErrorForm

- Select the correct option (A, B, C or D) for each question.

- A. Both statements are TRUE.
- B. Statement I is TRUE, but statement II is FALSE.
- C. Statement I is FALSE, but statement II is TRUE.
- D. Both statements are FALSE.

- Statement I: The self-attention mechanism in Transformers allows each token in the input sequence to focus on different parts of the sequence when producing the output.
Statement II: In Transformers, the number of self-attention heads is always fixed at one for all models and applications.

B

(2 marks)

- Statement I: The Transformer model uses convolutional layers to capture local patterns within the sequence data.
Statement II: Layer normalization is a critical component in the Transformer's architecture, helping stabilize the activations throughout the network.

C

(2 marks)

- Statement I: The "query", "key", and "value" matrices in the self-attention mechanism of Transformers are all derived from the same initial input embeddings.
Statement II: In the multi-head self-attention mechanism, different heads can potentially learn to attend to different parts or aspects of the input sequence.

D

(2 marks)

- The Transformer architecture employs a mechanism known as "positional encoding" to account for the order of words or tokens in a sequence.

(5 marks)

- Explain how sinusoidal positional encoding is computed and justify why it might be beneficial over other potential positional encoding methods.

(5 marks)

- Based on the sinusoidal positional encoding, calculate the positional encoding values for dimension [0, 10] when the position of a word in the sequence is 5 (assuming the initial position is 0) and the dimension of the embeddings is 512. Provide your answers rounded to four decimal places.

(4 marks)

- Explain the concept of "mode collapse" in the context of GANs and discuss its implications on the quality of the generated data.

(4 marks)

24th SCSE Club – Past Year Paper Solution (2023 – 2024 Semester 1)

SC4001/CE4042/CZ4002 – Neural Networks and Deep Learning

Dimension 1: 0.2837
Dimension 2: -0.9939
Dimension 3: -0.1107
Dimension 4: -0.9982
Dimension 5: -0.0595
Dimension 6: -0.9750
Dimension 7: -0.2221
Dimension 8: -0.9277
Dimension 9: -0.3733
Dimension 10: -0.8600

Editor's note: This answer is in Radian Mode in my calculator. You can convert to Degree mode, but I think both answers are accepted as long as it's consistent. Mode collapse occurs in GANs when the generator starts producing a limited variety of outputs, often very similar or even identical to each other, despite variations in the input noise. This happens when the generator finds a particular type of output that consistently fools the discriminator and then overfits to these outputs.

The implications are a lack of diversity in the quality of the generated samples as all samples produced tend to be of a similar distribution. Additionally, outputs tend to lack realism and seem unrealistic compared to real data.

State whether each of the following statements is "TRUE" or "FALSE". Each subquestion carries one mark. (7 marks)

- (i) Gradient Descent (GD) learning always converges to the global minimum of the cost function.
- (ii) Stochastic Gradient Descent (SGD) learning generally finds better weights and biases than GD learning.
- (iii) A linear neuron can be trained to learn nonlinear mapping.
- (iv) Discrete perceptron learns a hyperplane as the decision boundary in the feature space.
- (v) Discrete perceptron learning is derived using GD learning.
- (vi) Weights of neurons are initialized to smaller values in order to operate in the linear region of sigmoid activation function.

1 No. 1 continues on Page 2

1

i) **FALSE.** GD learning (and even SGD learning) could converge to the **local minimum** of the cost function if it is trapped in a feature space 'trough' (i.e. a dip).

ii) **TRUE.** SGD learning is said to consider the randomness of individual data (GD learning could cancel out or 'suppress' some of this randomness) and is more likely to find a more optimal cost than GD learning, and hence can find better weights and biases than GD learning.

iii) **FALSE.** Functions learned by linear neurons are all **hyperplanes**, and due to the absence of the exponent-involved sigmoid function, cannot learn non-linear mapping.

iv) **TRUE.** A discrete perceptron, which uses a sigmoid activation function $\sigma(z)$, where z is the pre-synaptic input, returns 1 if $\sigma > 0.5$ and 0 if $\sigma \leq 0.5$. This is equivalent to 1 of $z > 0$ and 0 if $z \leq 0$. Since the pre-synaptic input z to a discrete perceptron is a hyperplane (i.e. $z = \vec{w} \cdot \vec{x} + b$), a discrete perceptron does learn a hyperplane as the decision boundary in feature space.

v) **FALSE.** Who are you to decide it could be done only with GD learning? Use SGD learning if you want, do what you please.

CE4042/CZ4042

vi) **TRUE.** This should be done to prevent gradient descent to be slow due to low gradients of both ends of the sigmoid function. Weights are initialized so that learning can proceed faster when z is within the linear region of the sigmoid activation function (gradient is larger, so

- (vii) As the batch size is increased, the time to update weights drops in mini-batch GD learning.

- (b) The softmax layer shown in Q1(b) has three neurons, receives 2-dimensional inputs $(x_1, x_2) \in \mathbb{R}^2$ and produces an output class label $y \in \{1, 2, 3\}$. Weights and biases to the neurons are given in the figure Q1b.

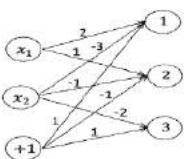


Figure Q1b

- (i) Write the weight vectors and biases of the neurons. (2 marks)
- (ii) Find the decision boundaries separating each pair of the classes. (6 marks)
- (iii) Plot the decision boundaries separating the three classes, clearly indicating the regions belonging to each class. (6 marks)
- (iv) Find the input that yields equal class probabilities. (4 marks)

Let us plot these three lines on Desmos in the next page.

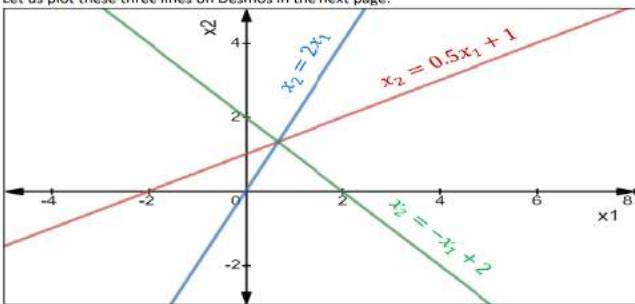
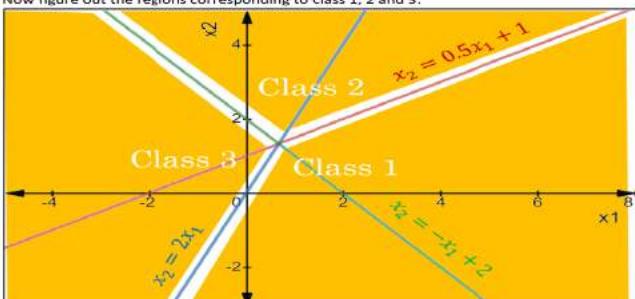


Figure 1: Decision boundaries

Now figure out the regions corresponding to Class 1, 2 and 3:



- iv) The input that yields equal class probabilities is the input (x_1, x_2) where the lines in the graph in 1(b)(iii) all intersect. That is, $(x_1, x_2) = (\frac{2}{3}, \frac{4}{3})$. To get this value, simply derive the intersection between any two of the three decision boundaries in 1(b)(ii):

$$x_2 = 0.5x_1 + 1 \text{ and } x_2 = 2x_1$$

$$0.5x_1 + 1 = 2x_1$$

$$x_1 = \frac{2}{3}$$

$$x_2 = 2x_1 = \frac{4}{3}$$

$$\therefore (x_1, x_2) = \left(\frac{2}{3}, \frac{4}{3}\right)$$

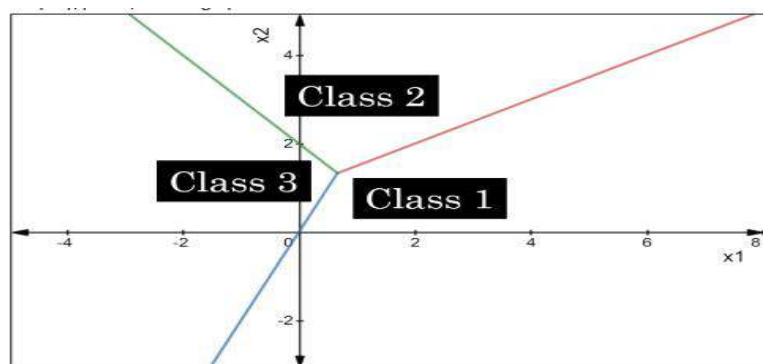


Figure 3: Simplified decision boundaries with class labels

3. (a) The following input I is applied to the input layer of a convolutional neural network (CNN).

$$I = \begin{pmatrix} -0.5 & -0.3 & 0.4 & -0.2 & -1.5 & 1.0 & -0.1 & 1.0 \\ 1.0 & 0.1 & 0.5 & 0.2 & 1.2 & -1.0 & 0.1 & 0.3 \\ -0.5 & -0.2 & 1.5 & 0.7 & 0.9 & 0.6 & -0.5 & -0.4 \\ 1.6 & -0.4 & 0.4 & -1 & -0.1 & 0.5 & 1.1 & 0.0 \end{pmatrix} \quad (3)$$

The input is processed by a convolution layer of neurons having weights $w = \begin{pmatrix} 1.0 & -0.2 \\ 0.2 & 0.5 \end{pmatrix}$ and rectified linear unit (ReLU) activation functions, and thereafter an average pooling layer having a pooling window of 2×2 size. The convolution is performed with stride = 2 and with 'VALID' padding. The bias connected to the convolution layer is 0.1.

Find the feature maps at the first convolution layer and pooling layer.

(10 marks)

- (b) A trained CNN is found to contain some dead neurons in its hidden layers. Neurons in these hidden layers use ReLU activation functions. Explain this phenomenon and provide a way to avoid dead neurons during training. (2 marks)

- (c) A convolutional neural network has five consecutive 3×3 convolutional layers with stride 1 and no pooling. How large is the receptive field (in terms of number of pixels) of a neuron in the fifth non-image layer of this network? (3 marks)

- (d) An autoencoder with a single hidden-layer is used to reconstruct 4-dimensional input patterns $x \in \mathbb{R}^4$. The hidden-layer has three neurons with activation functions $\phi(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$ and the output layer has four neurons with activation functions $\sigma(u) = \frac{1 - e^{-u}}{1 + e^{-u}}$. The weight matrix W connected to the hidden-layer, the bias vector b of the hidden-layer, and bias vector c of the output layer are given by

$$W = \begin{pmatrix} -2.0 & 1.2 & -0.8 \\ 0.5 & 1.5 & 2.2 \\ -2.2 & 3.2 & -1.2 \\ 2.0 & -1.0 & -3.6 \end{pmatrix}, b = \begin{pmatrix} 0.2 \\ -0.5 \\ 0.0 \end{pmatrix} \text{ and } c = \begin{pmatrix} 0.5 \\ 1.0 \\ -0.6 \\ 2 \end{pmatrix}$$

Question No. 3 continues on Page 5

4

TENTON: The Singapore Copyright Act applies to the use of this document. Nanyang Technological University Library
CE4042/CZ4042

For input patterns $x_1 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$ and $x_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$, find

- (i) the hidden layer activations. (3 marks)
- (ii) the reconstruction errors. (4 marks)
- (e) Undercomplete autoencoders are capable of learning hidden structures of data. Give the reasons. (3 marks)

b)

- i) The **generator** and the **discriminator**. The generator's goal is to maximize the cost function and hence generate images that can fool the discriminator while the discriminator's goal is to minimize the cost function and classify as accurately as possible which images are real and which images are fake.

- ii) Mode collapse is when the images generated by the **generator** network only fall under a restricted or limited set of categories instead of all of them (hence collapsing into a certain

- (b) The word "adversarial" in the acronym for GAN suggests a two-player game.

- (i) What are the two players, and what are their respective goals? (5 marks)
- (ii) Describe the phenomenon of mode collapse in GAN and suggest a method to overcome this issue. (5 marks)


mode of images). To overcome this method, generate images and classify images in batches. The discriminator network should have extra layers or parameters to compute whether the batch is 'diverse' enough or not. Hence, if mode collapse occurs and the images generated aren't diverse enough, the discriminator will classify the images as fake. This forces the generator to generate diverse images.

- (a) Given two inputs x and y , you are to train a neuron to approximate the following function ϕ when $0 \leq x, y \leq 1.0$.

$$\phi(x, y) = x + 2y^3 + xy - 0.5$$

- (i) Briefly state how you generate training data. (3 marks)
- (ii) State how you design the inputs to a linear neuron. (3 marks)
- (iii) Write the activation function if a perceptron is used. (4 marks)

- (b) Describe how the adversarial process between the generator and discriminator networks is implemented in the training of Generative Adversarial Networks (GAN).

- (a) Briefly state the following. Each part carries 2 marks.

- (i) The difference between gradient descent (GD) and stochastic gradient descent (SGD) learning algorithms.
 (ii) How the time to weight update varies with batch size in semi-batch SGD learning.
 (iii) How one could use a linear neuron to learn a given nonlinear equation.
 (iv) How a discrete perceptron is able to perform linear classification.
 (v) Two limitations when a logistic regression neuron is trained with gradient descent learning to perform classification.
 (vi) Two ways to initialize weights of a network to improve convergence.

(12 marks)

Reason to use padding and stride?

Select the correct option (A, B, C or D) for each question.

- A. Both statements are TRUE.
 B. Statement I is TRUE, but statement II is FALSE.
 C. Statement I is FALSE, but statement II is TRUE.
 D. Both statements are FALSE.

- (i) Statement I: Autoencoders are a supervised learning technique.
 Statement II: Autoencoder's output is exactly the same as the input.

(1 mark)

- (ii) Statement I: One way to implement undercomplete autoencoder is to constrain the number of nodes present in hidden layer(s) of the neural network.
 Statement II: To train a denoising encoder, we use a loss between the original input and the reconstruction from a noisy version of the input.

(1 mark)

- (iii) Statement I: Sparse autoencoders introduce information bottleneck by reducing the number of nodes at hidden layers.
 Statement II: With the sparsity constraint, we will observe more neuron outputs that are close to zero.

(1 mark)

- (ii) If we change this RNN to a Jordan-type RNN, and keep the same number of input dimensions, hidden and output neurons, what is the dimension of the top-down recurrence weight matrix W ?

(1 mark)

- (iii) Explain the reason of observing exploding gradients in RNN training. Describe a way to address this problem.

- (b) The statements below are all related to Transformers. Answer "TRUE" or "FALSE" to the following statements. Each part carries 1 mark.

- (i) Divide the dot product of the query vector with the key vector by the square root of the dimension of the key vectors leads to vanishing gradients.
 (ii) Multi-head attention alleviates the need for positional encoding.
 (iii) The encoder-decoder attention layer of each decoder layer accepts the query and value matrices obtained from the output of the encoder stack as input.
 (iv) In the same encoder layer, we apply the same feedforward network independently to each position.
 (v) Transformer uses positional encoding to help learn better attention.
 (vi) Positional encoding can either be pre-defined or made learnable.

- (d) Describe the changes that you need to make to turn an unconditional Generative Adversarial Network (GAN) to a conditional one that takes class labels. Explain one advantage of the conditional GAN in comparison to the unconditional GAN.

Briefly explain

- (i) why recurrent neural networks are difficult to train to learn long-term dependencies. (5 marks)
- (ii) how long short-term memory (LSTM) networks are able to learn long-term interactions. (5 marks)

Extras.

how about the feed forward?

ChatGPT

The feed-forward network within the Transformer architecture serves as a crucial component for learning complex, non-linear transformations on top of the self-attention mechanism. Here's why the feed-forward network is used in Transformers and its role in the overall architecture:

1. Non-Linearity and Feature Transformation:

- Purpose:** The feed-forward network introduces non-linearity to the Transformer model, enabling it to learn complex mappings from the input space to higher-level feature representations.
- Transformation:** It processes the output of the self-attention layer to generate more abstract and expressive representations that capture hierarchical patterns in the input data.

2. Layer-Wise Processing:

- Structure:** The feed-forward network is applied independently to each position in the sequence within the Transformer's multi-head attention mechanism.
- Sequential Computation:** This layer-wise processing allows the model to handle positional information and capture local context in addition to the global dependencies learned through self-attention.

3. Role of Activation Functions:

- Activation:** The feed-forward network typically consists of two linear transformations followed by a non-linear activation function (e.g., ReLU).
- Enhanced Expressivity:** Activation functions introduce non-linearities that enhance the model's expressive power, enabling it to approximate complex functions more effectively.

4. Dimensionality Reduction and Expansion:

- Dimensionality:** The feed-forward network often includes a dimensionality reduction step (e.g., projecting to a lower-dimensional space) followed by an expansion step.
- Bottleneck Effect:** This dimensionality reduction helps in learning more compact representations before expanding them back to the original dimension, creating a bottleneck effect that aids in feature learning.

how to calculate how many learnable parameter in transformer?

ChatGPT

Calculating the number of learnable parameters in a Transformer model involves considering the different components of the architecture, including embedding layers, positional encodings, self-attention layers, feed-forward layers, and output layers. Let's break down the calculation step-by-step:

Components of a Transformer Model:

1. Embedding Layers:

- Input Embeddings:** Suppose the input vocabulary size is V and the embedding dimension is d_{model} . The embedding layer consists of $V \times d_{\text{model}}$ parameters.

2. Positional Encodings:

- Positional Encoding Matrix:** If using learned positional encodings, each position in the sequence up to a certain maximum length N has an associated positional encoding vector of size d_{model} . This matrix has $N \times d_{\text{model}}$ parameters.

3. Self-Attention Layers:

- Query, Key, Value Projections:** Each self-attention layer typically involves three weight matrices: W_q, W_k, W_v . Each matrix has dimensions $d_{\text{model}} \times d_k$ or $d_{\text{model}} \times d_v$, depending on the desired key and value dimensions.
- Attention Weights:** Each attention head within a multi-head attention mechanism has $d_k \times d_v$ parameters.

4. Feed-Forward Layers:

- Linear Transformation Matrices:** Each feed-forward layer involves two weight matrices (input-to-hidden and hidden-to-output), each of size $d_{\text{model}} \times d_{\text{ff}}$, where d_{ff} is the feed-forward dimension.

5. Output Layer:

- Linear Projection:** The final linear layer maps the output of the Transformer to the desired output size (e.g., classification labels). It consists of $d_{\text{model}} \times V_{\text{out}}$ parameters, where V_{out} is the output vocabulary size.

The tutorial provides a simple walkthrough of the Vision Transformer. We hope you will be able to understand how it works by looking at the actual data flow during inference.

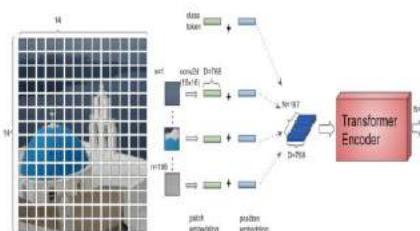


Figure 1: Vision Transformer inference pipeline.

1. Split Image into Patches
The input image is split into 14×14 vectors with dimension of 768 by Conv2d ($1 \times 1 \times 16$) with stride=(16,16).
2. Add Position Embedding
Learnable position embedding vectors are added to the patch embedding vectors and fed to the transformer encoder.
3. Transformer Encoder
The embedding vectors are encoded by the transformer encoder. The dimension of input and output vectors are the same. Details of the encoder are depicted in Fig. 2.
4. MLP (Classification) Head
The 0th output from the encoder is fed to the MLP head for classification to output the final classification results.

5. Parameterized Mapping:

- Learnable Parameters:** The feed-forward network consists of weights and biases that are learned during training.
- Adaptive Mapping:** These parameters allow the model to adaptively map input representations to higher-level features based on the task at hand.

6. Computational Efficiency:

- Parallel Processing:** Similar to self-attention, the feed-forward network operates in parallel across different positions in the sequence, enhancing computational efficiency.
- Scalability:** This parallelization enables Transformers to process sequences of varying lengths and scales efficiently.

7. Applications in Transformers:

- Interconnected Layers:** In Transformers, the feed-forward network is typically interleaved with self-attention layers in multiple transformer blocks (encoder and decoder).
- Enhanced Representations:** The combination of self-attention and feed-forward layers allows Transformers to learn rich, context-aware representations suitable for various sequence-based tasks.

In summary, the feed-forward network in Transformers plays a critical role in learning complex transformations and extracting high-level features from input sequences. By introducing non-linearities and parameterized mappings, it complements the self-attention mechanism to enable effective modeling of sequential data in natural language processing and other applications.



Let's consider a Transformer model with the following dimensions and configurations:

- Input Vocabulary Size (V): 10,000
- Embedding Dimension (d_{model}): 512
- Number of Self-Attention Heads (h): 8
- Feed-Forward Dimension (d_{ff}): 2048
- Output Vocabulary Size (V_{out}): 100

Calculation Steps:

1. Embedding Layer:

- Parameters: $10,000 \times 512 = 5,120,000$ parameters.

2. Positional Encodings:

- Assuming a maximum sequence length $N = 512$:
- Parameters: $512 \times 512 = 262,144$ parameters.

3. Self-Attention Layers:

- Query, Key, Value Projections: $512 \times 64 = 32,768$ parameters per head.
- Total for 8 heads: $8 \times 32,768 = 262,144$ parameters.
- Attention Weights: $64 \times 64 = 4,096$ parameters per head.
- Total for 8 heads: $8 \times 4,096 = 32,768$ parameters.

4. Feed-Forward Layers:

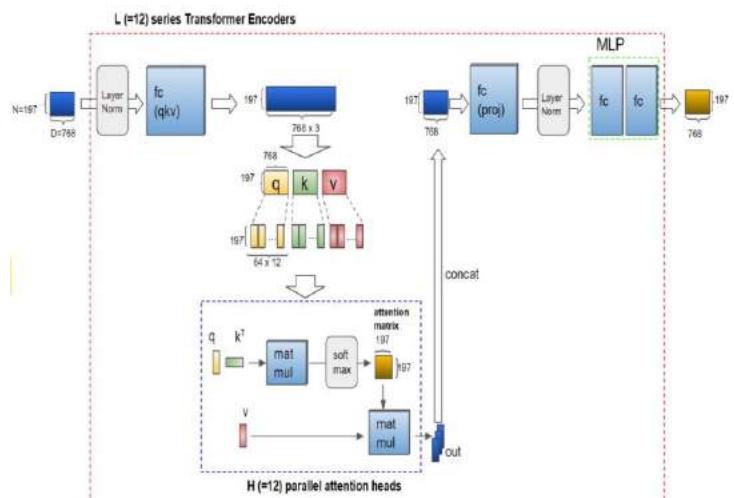
- Input-to-Hidden: $512 \times 2048 = 1,048,576$ parameters.
- Hidden-to-Output: $2048 \times 512 = 1,048,576$ parameters.

5. Output Layer:

- Parameters: $512 \times 100 = 51,200$ parameters.

Total Number of Parameters:

- Summing up all the parameters from the above components:
 $5,120,000 + 262,144 + 262,144 + 32,768 + 32,768 + 1,048,576 + 1,048,576 + 51,200 = 8,857,176$ parameters



Image=[1,3,224,224] resolution=16 Patch_emb=[1,196,768] Pos_emb=[1,197,768](196+1 class token)

Sparse Autoencoder Overview:

- Structure:** A sparse autoencoder is a type of autoencoder that includes a regularization term aimed at promoting sparsity in the activations of its hidden layer(s).
- Objective:** The primary objective of a sparse autoencoder is to learn a compact and efficient representation of the input data by inducing a small subset of active (non-zero) neurons in the hidden layer(s) for each input sample.

Encouraging Sparsity:

1. Regularization Term:

- In sparse autoencoders, sparsity is often enforced using a regularization term in the loss function.
- Common approaches include L1 regularization or Kullback-Leibler (KL) divergence regularization applied to the activations.

2. Activation Threshold:

- The idea is to encourage most of the neurons in the hidden layer(s) to remain inactive (close to zero) for a given input.
- Only a small fraction of neurons should become active (have non-zero activations) to capture essential features of the input.

3. Benefits of Sparsity:

- Improved Generalization:** Sparse representations tend to generalize better to new, unseen data by focusing on the most relevant features.
- Reduced Redundancy:** Sparse activations help in reducing redundancy and overfitting in the learned representations.

For matrix X :

$$\frac{\partial a^T X b}{\partial X} = ab^T$$

$$\frac{\partial a^T X^T b}{\partial X} = ba^T$$

$$\frac{\partial a^T X a}{\partial X} = \frac{\partial a^T X^T a}{\partial X} = aa^T$$

$$\frac{\partial b^T X^T X c}{\partial X} = X(bc^T + cb^T)$$

$$\frac{\partial X^T B X}{\partial X} = (B + B^T)X$$

$$\frac{\partial w^T X^T X w}{\partial w} = ((x^T x) + (x^T x)^T)w = (x^T x + x^T x)w = 2x^T x w$$

$$\frac{\partial (AX)}{\partial X} = A^T$$

$$\frac{\partial (X^T A)}{\partial X} = A$$

$$\frac{\partial (X^T X)}{\partial X} = 2X$$

$$\frac{\partial (X^T A X)}{\partial X} = AX + A^T X$$