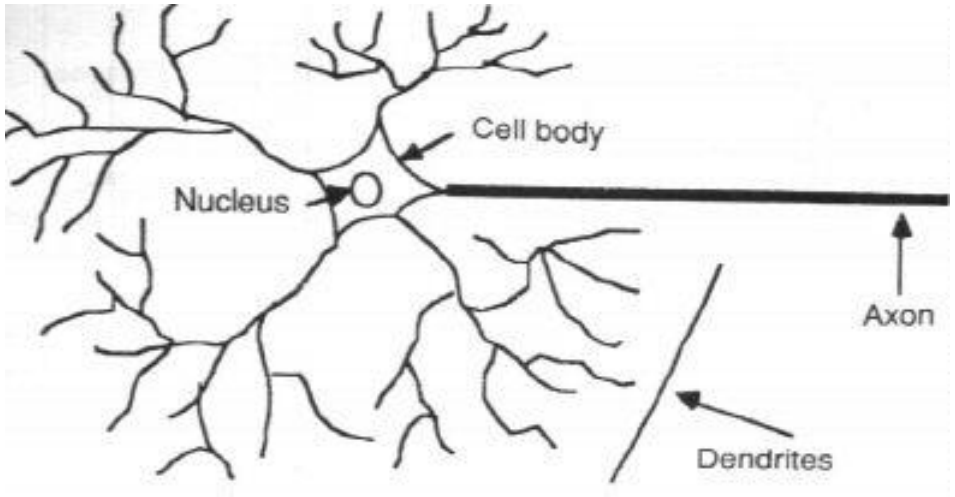
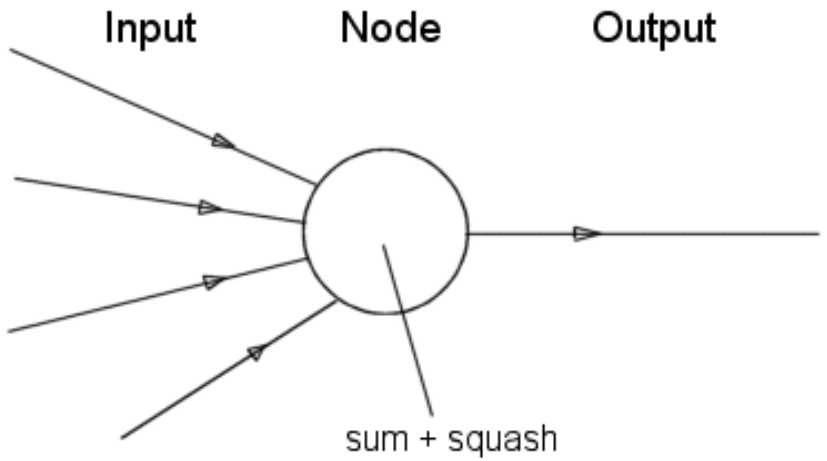
Neural Networks

Introduction

- Inspired by the human brain.
- Some NNs are models of biological neural networks
- Human brain contains a massively interconnected net of 10^{10}-10^{11} (10 billion) neurons (cortical cells)
  - Massive parallelism – large number of simple processing units
  - Connectionism – highly interconnected
  - Associative distributed memory
    - Pattern and strength of synaptic connections

Neuron

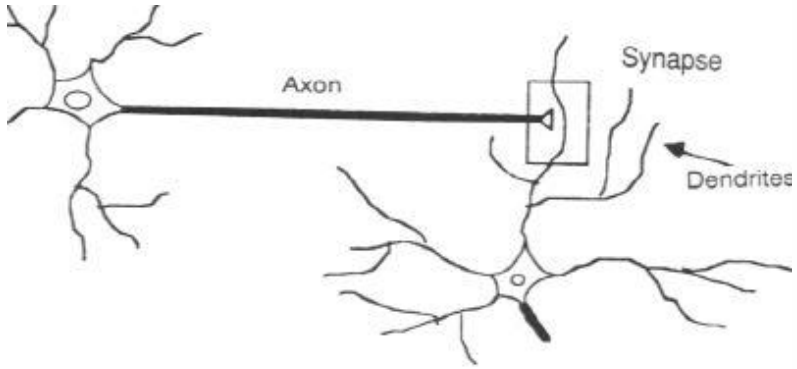


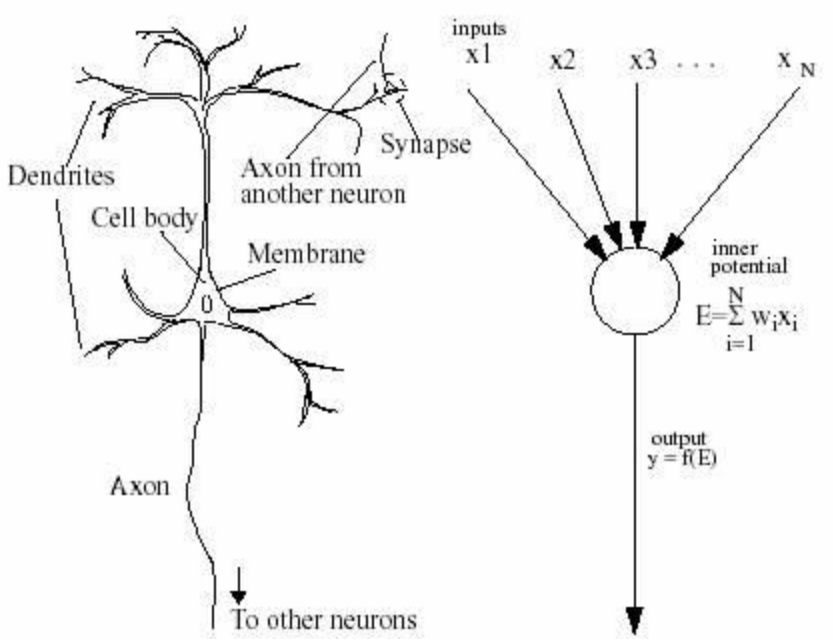


Neural Unit

ANNs

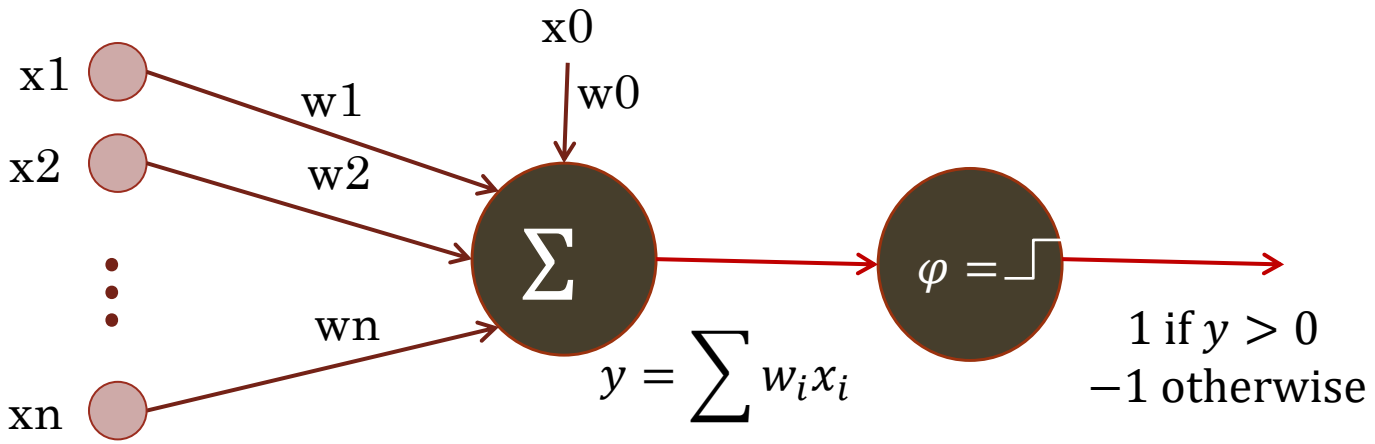
- ANNs incorporate the two fundamental components of biological neural nets:
  1. Nodes - Neurones
  2. Weights - Synapses





Perceptrons

- Basic unit in a neural network: Linear separator
  - N inputs, x1 ... xn
  - Weights for each input, w1 ... wn
  - A bias input x0 (constant) and associated weight w0
  - Weighted sum of inputs, $y = \sum_{i=0}^{n} w_i x_i$
  - A threshold function, i.e., 1 if y > 0, −1 if y <= 0



A Neural Network with a Single Neuron

- Consider the simple example of predicting house price given the size of the house
- In the case of linear regression, we fit a straight line: Predicts negative house price
- To prevent this, we define a parameterized function $h_\theta(x)$ with input x and parameter θ, which outputs house price y

$$h_\theta(x) = \max(wx + b, 0), \text{ where } \theta = (w, b) \in \mathbb{R}^2$$

- Here h(x) returns a single value: (wx+b) or zero, whichever is greater.
- In the context of neural networks, the function max{t; 0} is called a ***ReLU or rectified linear unit***

A Neural Network with a Single Neuron

When the input $x \in \mathbb{R}^d$ has multiple dimensions, a neural network with a single neuron can be written as

$$h_\theta(x) = \text{ReLU}(w^\top x + b), \text{ where } w \in \mathbb{R}^d, b \in \mathbb{R}, \text{ and } \theta = (w, b) \qquad (7.7)$$

Stacking Neurons

- In addition to the size of the house, suppose that you know the number of bedrooms, the zip code and the wealth of the neighborhood.
- Given these features
  - ***size, number of bedrooms, zip code, and wealth***
- We might then decide that the price of the house depends on the maximum family size it can accommodate.
- Suppose the ***family size*** is a function of the size of the *house* and *number of bedrooms*
- The ***zip code*** may provide additional information such as how *walkable* the neighborhood is
- Combining the ***zip code*** with the ***wealth of the neighborhood*** may predict the quality of the local neighbour hood

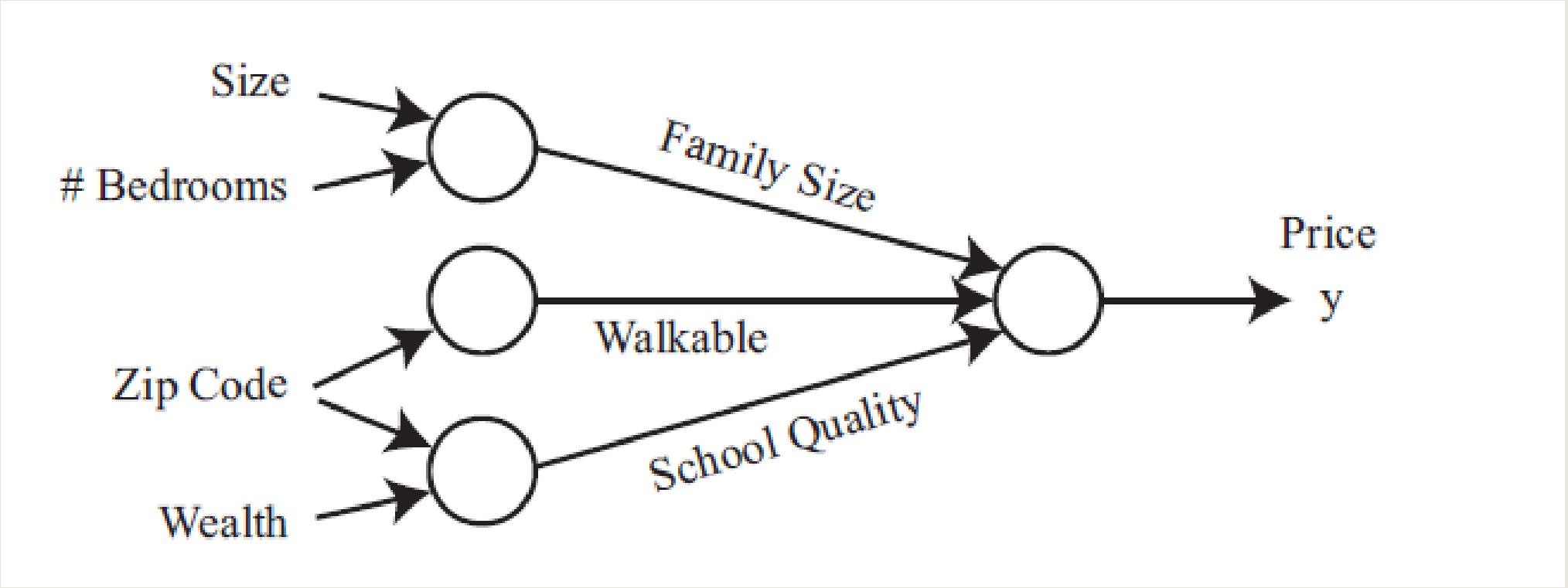
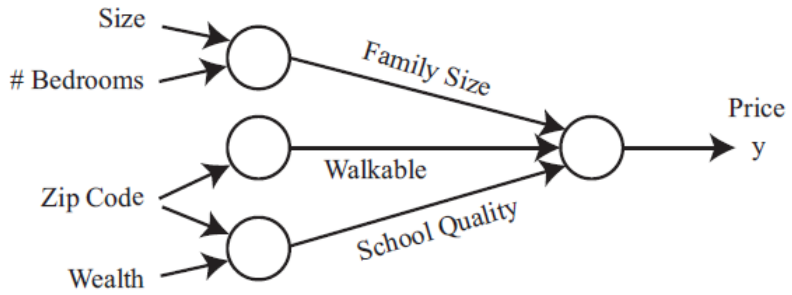


Diagram of a small neural network for predicting housing prices

A Neural Network with a Single Neuron

- Formally, the input to a neural network is a set of input features x1; x2; x3; x4.
- We denote the intermediate variables for “family size”, “walkable”, and “school quality” by a1; a2; a3 (these ai's are often referred to as “hidden units” or “hidden neurons”)

$$a_1 = \mathrm{ReLU}(\theta_1 x_1 + \theta_2 x_2 + \theta_3)$$
$$a_2 = \mathrm{ReLU}(\theta_4 x_3 + \theta_5)$$
$$a_3 = \mathrm{ReLU}(\theta_6 x_3 + \theta_7 x_4 + \theta_8)$$



$$h_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12}$$

Two-layer Fully-Connected Neural Networks

- We constructed the neural network in equation using a significant amount of prior knowledge belief about how the “family size”, “walkable”, and “Neighborhood quality” are determined by the inputs.
- We implicitly assumed that we know the family size is an important quantity to look at and that it can be determined by only the “size” and “# bedrooms”.
- ***Such a prior knowledge might not be available for other applications.***
- A simple way would be to write the intermediate variable a1 as a function of all x1; : : : ; x4

Two-layer Fully-Connected Neural Networks

$$a_1 = \mathrm{ReLU}(w_1^\top x + b_1), \text{ where } w_1 \in \mathbb{R}^4 \text{ and } b_1 \in \mathbb{R}$$
$$a_2 = \mathrm{ReLU}(w_2^\top x + b_2), \text{ where } w_2 \in \mathbb{R}^4 \text{ and } b_2 \in \mathbb{R}$$
$$a_3 = \mathrm{ReLU}(w_3^\top x + b_3), \text{ where } w_3 \in \mathbb{R}^4 \text{ and } b_3 \in \mathbb{R}$$

- Thus we have a so-called ***fully-connected neural network***

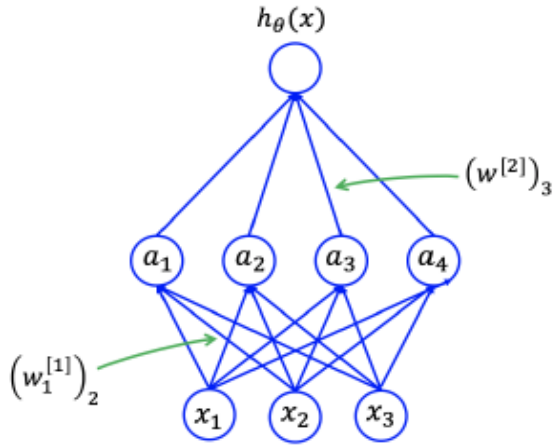


Figure 7.3: Diagram of a two-layer fully connected neural network. Each edge from node x_i to node a_j indicates that a_j depends on x_i. The edge from x_i to a_j is associated with the weight $(w_j^{[1]})_i$ which denotes the i-th coordinate of the vector $w_j^{[1]}$. The activation a_j can be computed by taking the ReLUof the weighted sum of x_i's with the weights being the weights associated with the incoming edges, that is, $a_j = \mathrm{ReLU}(\sum_{i=1}^{d}(w_j^{[1]})_i x_i)$.

Two-layer Fully-Connected Neural Networks

For full generality, a two-layer fully-connected neural network with m hidden units and d dimensional input $x \in \mathbb{R}^d$ is defined as

$$\forall j \in [1, ..., m], \quad z_j = {w_j^{[1]}}^\top x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \tag{7.10}$$

$$a_j = \text{ReLU}(z_j),$$

$$a = [a_1, \ldots, a_m]^\top \in \mathbb{R}^m$$

$$h_\theta(x) = {w^{[2]}}^\top a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R}, \tag{7.11}$$

Vectorization

We vectorize the two-layer fully-connected neural network as below. We define a weight matrix $W^{[1]}$ in $\mathbb{R}^{m\times d}$ as the concatenation of all the vectors $w_j^{[1]}$'s in the following way:

$$W^{[1]} = \begin{bmatrix} — & {w_1^{[1]}}^\top & — \\ — & {w_2^{[1]}}^\top & — \\ & \vdots & \\ — & {w_m^{[1]}}^\top & — \end{bmatrix} \in \mathbb{R}^{m\times d} \tag{7.12}$$

Now by the definition of matrix vector multiplication, we can write $z = [z_1, \ldots, z_m]^\top \in \mathbb{R}^m$ as

$$\underbrace{\begin{bmatrix} z_1 \\ \vdots \\ \vdots \\ z_m \end{bmatrix}}_{z\in\mathbb{R}^{m\times 1}} = \underbrace{\begin{bmatrix} — & {w_1^{[1]}}^\top & — \\ — & {w_2^{[1]}}^\top & — \\ & \vdots & \\ — & {w_m^{[1]}}^\top & — \end{bmatrix}}_{W^{[1]}\in\mathbb{R}^{m\times d}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}}_{x\in\mathbb{R}^{d\times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix}}_{b^{[1]}\in\mathbb{R}^{m\times 1}} \tag{7.13}$$

Or succinctly,

$$z = W^{[1]}x + b^{[1]} \tag{7.14}$$

Vectorization

$$a = \text{ReLU}(z) \tag{7.15}$$

Define $W^{[2]} = [w^{[2]\top}] \in \mathbb{R}^{1\times m}$ similarly. Then, the model in equation (7.11) can be summarized as

$$\begin{aligned} a &= \text{ReLU}(W^{[1]}x + b^{[1]}) \\ h_\theta(x) &= W^{[2]}a + b^{[2]} \end{aligned} \tag{7.16}$$

Here θ consists of $W^{[1]}, W^{[2]}$ (often referred to as the weight matrices) and $b^{[1]}, b^{[2]}$ (referred to as the biases). The collection of $W^{[1]}, b^{[1]}$ is referred to as the first layer, and $W^{[2]}, b^{[2]}$ the second layer. The activation a is referred to as the hidden layer. A two-layer neural network is also called one-hidden-layer neural network.

Multi-layer fully-connected neural networks

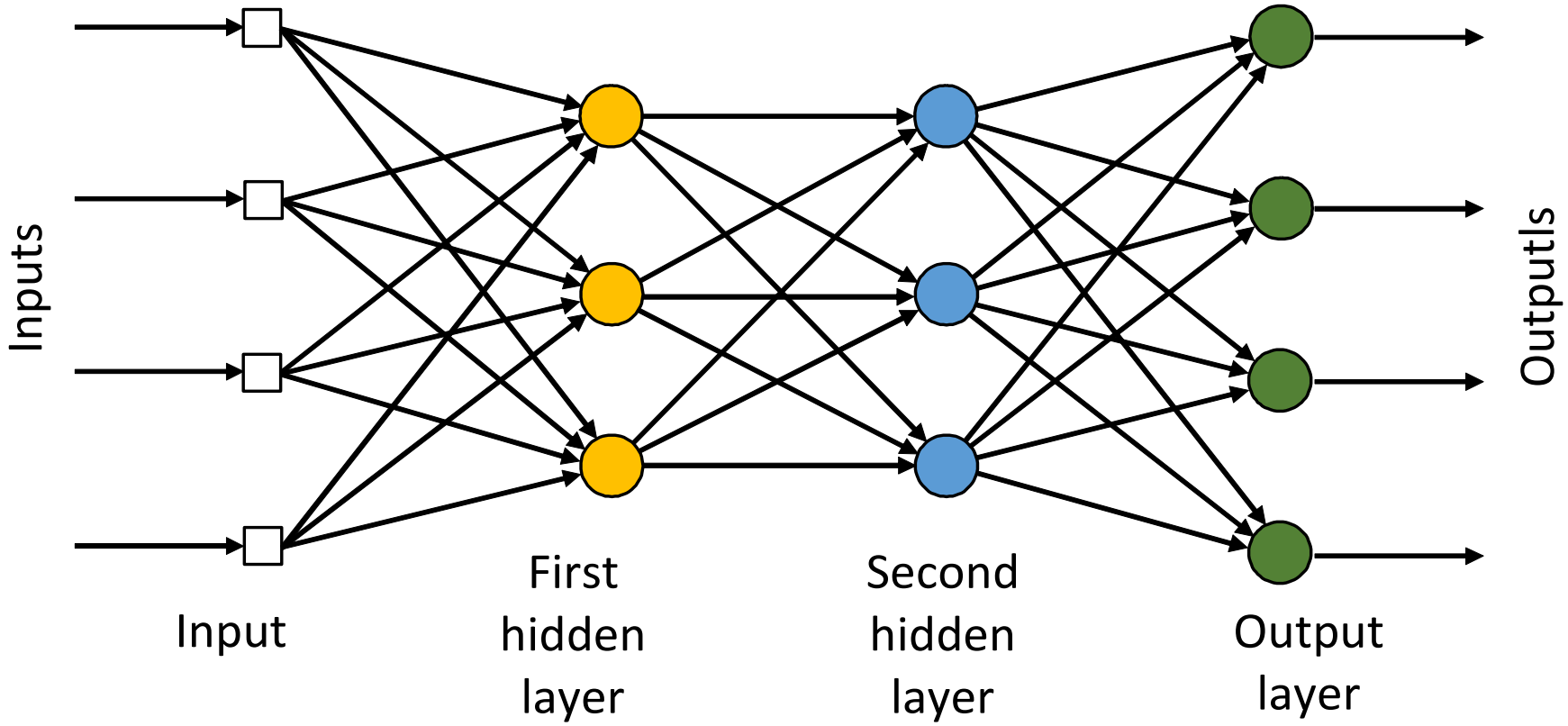
- Let r be the number of layers (weight matrices). Let W[1]; : : : ;W[r]; b[1]; : : : ; b[r] be the weight matrices and biases of all the layers. Then a multi-layer

$$
\begin{aligned}
a^{[1]} &= \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \\
a^{[2]} &= \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]}) \\
&\cdots \\
a^{[r-1]} &= \mathrm{ReLU}(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \\
h_\theta(x) &= W^{[r]}a^{[r-1]} + b^{[r]} \qquad (7.17)
\end{aligned}
$$

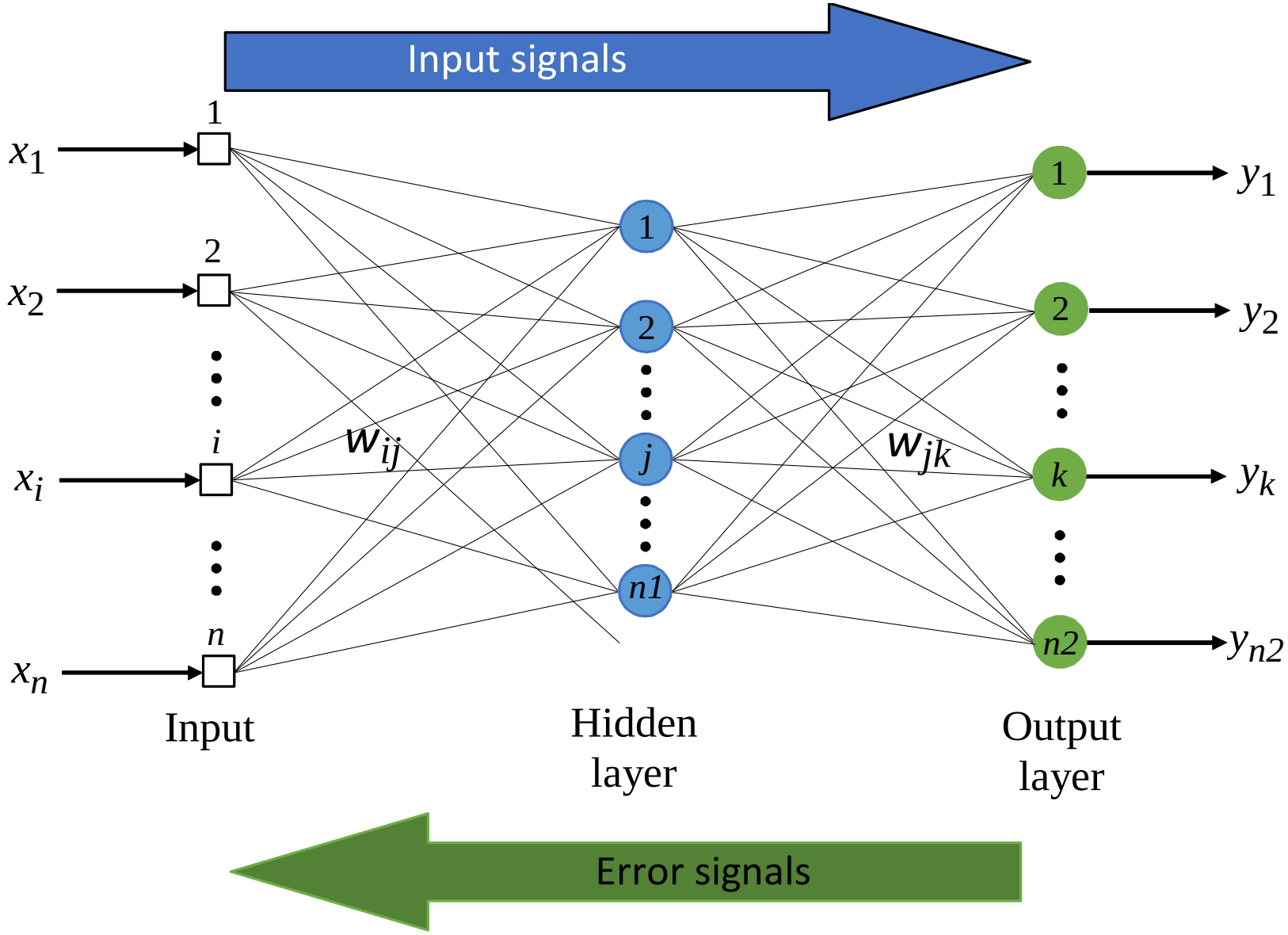
We note that the weight matrices and biases need to have compatible dimensions for the equations above to make sense. If $a^{[k]}$ has dimension m_k, then the weight matrix $W^{[k]}$ should be of dimension $m_k \times m_{k-1}$, and the bias $b^{[k]} \in \mathbb{R}^{m_k}$. Moreover, $W^{[1]} \in \mathbb{R}^{m_1 \times d}$ and $W^{[r]} \in \mathbb{R}^{1 \times m_{r-1}}$.

The total number of neurons in the network is $m_1 + \cdots + m_r$, and the total number of parameters in this network is $(d+1)m_1 + (m_1+1)m_2 + \cdots + (m_{r-1}+1)m_r$.

Multilayer Network



Two-layer back-propagation neural network



Backpropagation

- **Preliminary: chain rule**

We first recall the chain rule in calculus. Suppose the variable J depends on the variables $\theta_1, \ldots, \theta_p$ via the intermediate variable $g_1, \ldots, g_k$:

$$g_j = g_j(\theta_1, \ldots, \theta_p), \forall j \in \{1, \cdots, k\} \tag{7.27}$$

$$J = J(g_1, \ldots, g_k) \tag{7.28}$$

Here we overload the meaning of g_j's: they denote both the intermediate variables but also the functions used to compute the intermediate variables. Then, by the chain rule, we have that $\forall i$,

$$\frac{\partial J}{\partial \theta_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial g_j} \frac{\partial g_j}{\partial \theta_i} \tag{7.29}$$

Backpropagation: Two-layer neural networks

- Now we consider the two-layer neural network defined in equation (7.11). We compute the loss J by following sequence of operations

$$
\begin{aligned}
\forall j \in [1, ..., m], \quad z_j &= {w_j^{[1]}}^\top x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \\
a_j &= \mathrm{ReLU}(z_j), \\
a &= [a_1, \ldots, a_m]^\top \in \mathbb{R}^m \\
o &= {w^{[2]}}^\top a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R} \\
J &= \frac{1}{2}(y - o)^2
\end{aligned}
$$

Backpropagation: Two-layer neural networks

- By invoking chain rule with J as the output variable, o as intermediate variable, and (w[2])` as the input variable, we have

$$\frac{\partial J}{\partial (w^{[2]})_\ell} = \frac{\partial J}{\partial o}\frac{\partial o}{\partial (w^{[2]})_\ell}$$

$$= (o - y)\frac{\partial o}{\partial (w^{[2]})_\ell}$$

$$= (o - y)a_\ell$$

Backpropagation: Two-layer neural networks

It's more challenging to compute $\frac{\partial J}{\partial (w_j^{[1]})_\ell}$. Towards computing it, we first invoke the chain rule with J as the output variable, z_j as the intermediate variable, and $(w_j^{[1]})_\ell$ as the input variable.

$$\frac{\partial J}{\partial (w_j^{[1]})_\ell} = \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial (w_j^{[1]})_\ell}$$

$$= \frac{\partial J}{\partial z_j} \cdot x_\ell \qquad \text{(becaues } \frac{\partial z_j}{\partial (w_j^{[1]})_\ell} = x_\ell \text{.)}$$

Backpropagation: Two-layer neural networks

Thus, it suffices to compute the $\frac{\partial J}{\partial z_j}$. We invoke the chain rule with J as the output variable, a_j as the intermediate variable, and z_j as the input variable,

$$\begin{aligned}\frac{\partial J}{\partial z_j} &= \frac{\partial J}{\partial a_j}\frac{\partial a_j}{\partial z_j} \\ &= \frac{\partial J}{\partial a_j}\text{ReLU}'(z_j)\end{aligned}$$

Backpropagation: Two-layer neural networks

Now it suffices to compute $\frac{\partial J}{\partial a_j}$, and we invoke the chain rule with J as the output variable, o as the intermediate variable, and a_j as the input variable,

$$\begin{aligned}\frac{\partial J}{\partial a_j} &= \frac{\partial J}{\partial o}\frac{\partial o}{\partial a_j} \\ &= (o - y) \cdot (w^{[2]})_j\end{aligned}$$

Now combining the equations above, we obtain

$$\frac{\partial J}{\partial (w_j^{[1]})_\ell} = (o - y) \cdot (w^{[2]})_j \mathrm{ReLU}'(z_j) x_\ell$$

Algorithm 3 Backpropagation for two-layer neural networks

1: Compute the values of $z_1, \ldots, z_m$, $a_1, \ldots, a_m$ and o as in the definition of neural network (equation (7.34)).
2: Compute $\frac{\partial J}{\partial o} = (o - y)$.
3: Compute $\frac{\partial J}{\partial z_j}$ for $j = 1, \ldots, m$ by

$$\frac{\partial J}{\partial z_j} = \frac{\partial J}{\partial o}\frac{\partial o}{\partial a_j}\frac{\partial a_j}{\partial z_j} = \frac{\partial J}{\partial o} \cdot (w^{[2]})_j \cdot \mathrm{ReLU}'(z_j) \tag{7.35}$$

4: Compute $\frac{\partial J}{\partial (w_j^{[1]})_\ell}$, $\frac{\partial J}{\partial b_j^{[1]}}$, $\frac{\partial J}{\partial (w^{[2]})_j}$, and $\frac{\partial J}{\partial b^{[2]}}$ by

$$\frac{\partial J}{\partial (w_j^{[1]})_\ell} = \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial (w_j^{[1]})_\ell} = \frac{\partial J}{\partial z_j} \cdot x_\ell$$

$$\frac{\partial J}{\partial b_j^{[1]}} = \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial b_j^{[1]}} = \frac{\partial J}{\partial z_j}$$

$$\frac{\partial J}{\partial (w^{[2]})_j} = \frac{\partial J}{\partial o}\frac{\partial o}{\partial (w^{[2]})_j} = \frac{\partial J}{\partial o} \cdot a_j$$

$$\frac{\partial J}{\partial b^{[2]}} = \frac{\partial J}{\partial o}\frac{\partial o}{\partial b^{[2]}} = \frac{\partial J}{\partial o}$$

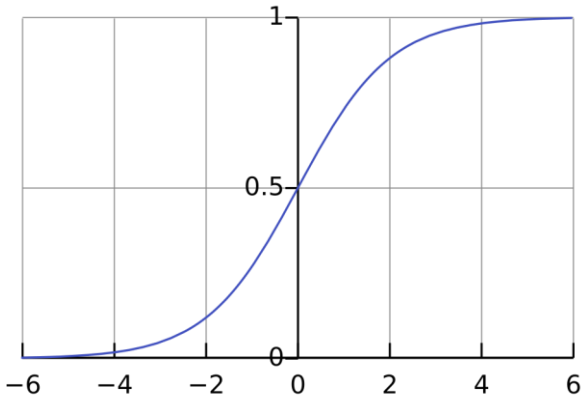
Activation functions

- An **activation function** in deep learning is a mathematical function applied to the output of each neuron in a neural network.
- It introduces **non-linearity** to the model, enabling it to learn complex patterns and relationships in data.
- Without activation functions, a neural network would behave like a simple linear regression model, limiting its ability to solve complex problems.
  - **Non-linearity** – Enables the network to learn non-linear mappings.
  - **Feature Extraction** – Helps in detecting and representing patterns in the input.
  - **Gradient Flow** – Ensures effective backpropagation by controlling gradient values.
  - **Computational Efficiency** – Determines the speed and stability of training.
  - **Vanishing/Exploding Gradient Avoidance** – Helps in stable learning by addressing gradient issues.

Activation functions

- Sigmoid Activation Function:

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



- Outputs values between **0 and 1**.
- Used for **binary classification** problems.
- Suffers from **vanishing gradient problem** (gradients become very small for large or small inputs).

Activation functions

- Rectified Linear Unit (ReLU:
  - The ReLU function simply "turns off" negative inputs, and passes positive inputs unchanged
- Introduces sparsity by setting negative values to **zero**.
- Faster and computationally efficient.
- Suffers from the **dying ReLU problem** (neurons can become inactive if they output zero for all inputs).

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

Activation functions

- TanH (Hyperbolic Tangent) Activation Function:

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

- Outputs values between **-1 and 1**.
- Centered at zero, which helps in faster learning compared to the sigmoid function.
- Still suffers from the **vanishing gradient problem** for large values of x.

Activation functions

- EReLU (Exponential Linear Unit) Activation Function:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha(e^x - 1), & \text{otherwise} \end{cases}$$

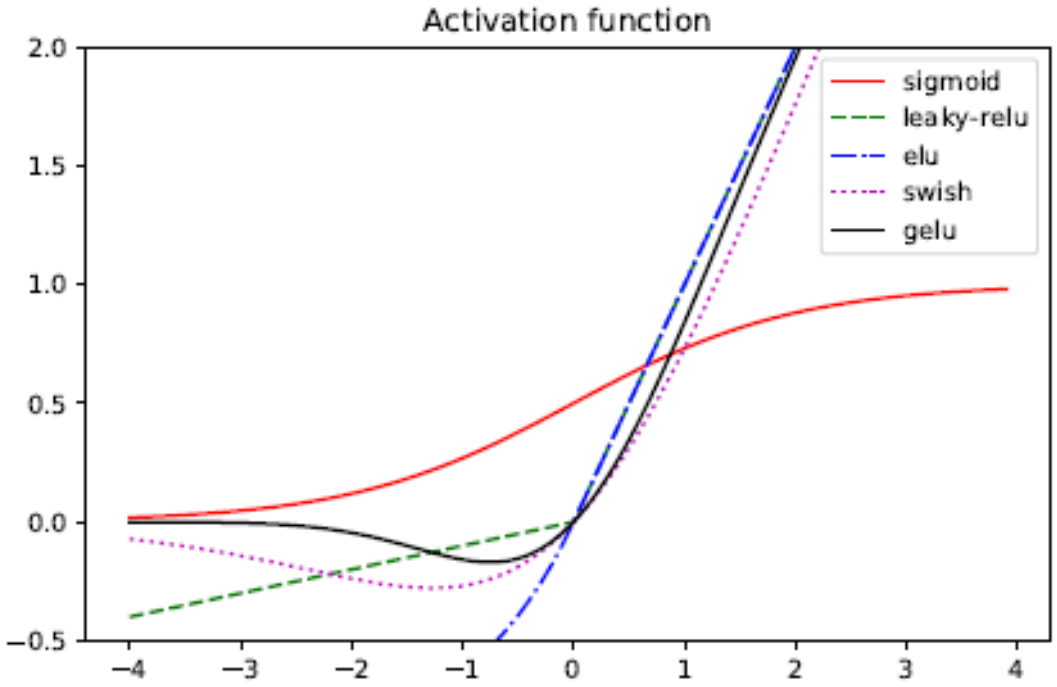
- where α is a small positive constant.
- Avoids dying neurons by allowing small negative values.
- Improves gradient propagation.
- More robust compared to standard ReLU.

Activation functions

- Swish Activation Function

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

- Smooth and non-monotonic.
- Helps improve training performance and accuracy.



Activation functions

- The **Softmax activation function** is used in multi-class classification problems to convert raw model outputs (logits) into **normalized probability distributions** over multiple classes.
- It ensures that the sum of all output values equals **1**, making it interpretable as a probability distribution.
- Given an input vector of logits = $[z_1, z_2, \ldots, z_C]$, where C is the number of classes, the **Softmax function** is defined as:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

- for each class i in C.
- z_i represents the logit (raw score) for class i
- e^z exponentiates the logits to ensure all values are **positive**
- The denominator normalizes the values so that their sum equals **1**

Activation functions

- Probability Distribution: $\sum_{i=1}^{C} \sigma(z_i) = 1$

- This ensures that the output can be interpreted as class probabilities
- Consider a classification problem with three classes. Given logits: $z = [2.0, 1.0, 0.1]$

- We compute:
  1. Compute exponentials: $e^{2.0} = 7.389, \quad e^{1.0} = 2.718, \quad e^{0.1} = 1.105$
  2. Compute sum: $7.389 + 2.718 + 1.105 = 11.212$
  3. Compute softmax probabilities:

$$\sigma(z_1) = \frac{7.389}{11.212} \approx 0.659$$

$$\sigma(z_2) = \frac{2.718}{11.212} \approx 0.242$$

$$\sigma(z_3) = \frac{1.105}{11.212} \approx 0.099$$

Binary Cross Entropy (BCE)

- Cross-entropy is a widely used loss function in classification problems. It measures the dissimilarity between two probability distributions: the predicted output and the true label distribution.
- Binary Cross Entropy (also called Log Loss) is used for **binary classification** problems, where the target labels belong to one of two classes (0 or 1).
- For a single data point with true label y and predicted probability ŷ, the Binary Cross Entropy loss is given by:

$$L = -\left[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\right]$$

- For a dataset with N samples, the total loss is:

$$L = -\frac{1}{N} \sum_{i=1}^{N} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)\right]$$

Categorical Cross Entropy (CCE)

- Categorical Cross Entropy is used for **multi-class classification** problems, where each sample belongs to one of C classes.
- For a single data point with **one-hot encoded** true label y and predicted probability vector y^, the loss is:

$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

- For a dataset with N samples:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

Binary Cross Entropy (BCE)

- Suppose we have a **binary classification** problem, and a single data point has:
- True label: y=1
- Predicted probability: y^=0.9
- The loss is:

$$L = -[1 \cdot \log(0.9) + (1 - 1) \cdot \log(1 - 0.9)]$$

$$L = -\log(0.9) \approx 0.105$$

- If y^=0.1 (bad prediction),

$$L = -\log(0.1) \approx 2.302$$

Categorical Cross Entropy (CCE)

- For a **3-class classification problem** (e.g., dog, cat, bird), suppose we have:
- True label: **Cat** (one-hot encoded as y=[0,1,0])
- Predicted probabilities: y^=[0.2,0.7,0.1]
- The loss is:

$$L = -[0 \cdot \log(0.2) + 1 \cdot \log(0.7) + 0 \cdot \log(0.1)]$$

$$L = -\log(0.7) \approx 0.357$$

- If the model predicted wrong (y^=[0.7,0.2,0.1),

$$L = -\log(0.2) \approx 1.609$$

Regularization

- We defined regularization as

"Any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error"

- Many regularization approaches are based on limiting the capacity of models, such as neural networks, linear regression, or logistic regression, by adding a **parameter norm penalty $\Omega(\theta)$** to the objective function J .

- We denote the regularized objective function by $\tilde{J}$:

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

- where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term, Ω, relative to the standard objective function J. Setting α to 0 results in no regularization. Larger values of α correspond to more regularization.

L2 Parameter Regularization

- The *L2* parameter norm penalty commonly known as **weight decay**.
- This regularization strategy drives the weights closer to the origin by adding a regularization $\boldsymbol{term\ \Omega(\theta) = \frac{1}{2}\ |w|^2}$ to the objective function.
- In other academic communities, *L2* regularization is also known as **ridge regression** or **Tikhonov regularization**.

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- with the corresponding parameter gradient:

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

L2 Parameter Regularization

- To take a single gradient step to update the weights, we perform this update:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon \left(\alpha \boldsymbol{w} + \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \right)$$

- Written another way, the update is:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \epsilon \nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

- We can see that the addition of the weight decay term has modified the learning rule to multiplicatively shrink the weight vector by a constant factor on each step, just before performing the usual gradient update.
- Parameters that contribute to the objective functions are preserved while the weight vector of unimportant features are decayed away

L1 Regularization

- There are other ways to penalize the size of the model parameters
- $L1$ regularization on the model parameter w is defined as:

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|$$

- that is, as the sum of absolute values of the individual parameters $L1$ weight decay controls the strength of the regularization by scaling the penalty Ω using a positive hyperparameter α.
- Thus, the regularized objective function

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

L1 Regularization

- with the corresponding gradient

$$\nabla_{\boldsymbol{w}} \tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{w})$$

- In comparison to $L2$ regularization, $L1$ regularization results in a solution that is more **sparse**.
- Sparsity in this context refers to the fact that some parameters have an optimal value of zero.
- The sparsity property induced by $L1$ regularization has been used extensively as a **feature selection** mechanism.
- Feature selection simplifies a machine learning problem by choosing which subset of the available features should be used.

Dataset Augmentation

- The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set.
- This approach is easiest for classification. A classifier needs to take a complicated, high dimensional input x and summarize it with a single category identity y.
- This means that the main task facing a classifier is to be invariant to a wide variety of transformations. We can generate new (x, y) pairs easily just by transforming the x inputs in our training set.
- One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between ‘b’ and ‘d’ and the difference between ‘6’ and ‘9’, so horizontal flips and 180◦ rotations are not appropriate ways of augmenting datasets for these tasks.

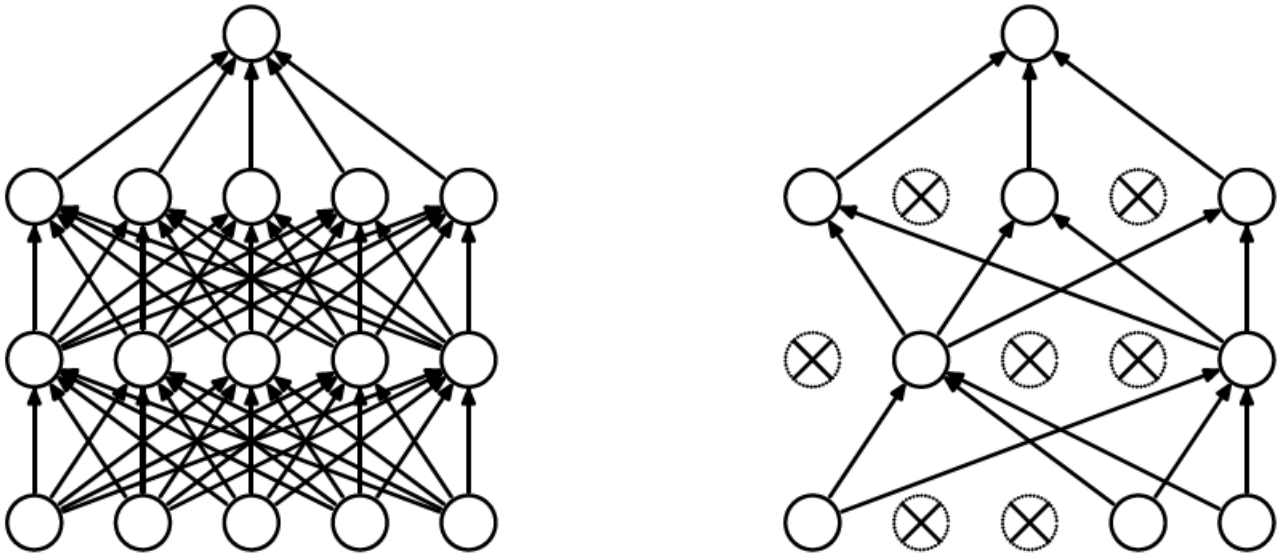
Dataset Augmentation

- Injecting noise in the input to a neural network (Sietsma and Dow, 1991) can also be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input.
- When comparing machine learning benchmark results, it is important to take the effect of dataset augmentation into account. Often, hand-designed dataset augmentation schemes can dramatically reduce the generalization error of a machine learning technique.
- To compare the performance of one machine learning algorithm to another, it is necessary to perform controlled experiments. When comparing machine learning algorithm A and machine learning algorithm B, it is necessary to make sure that both algorithms were evaluated using the same hand-designed dataset augmentation schemes.

Dropout

- Suppose that we randomly (on a per-example basis) turn off all the outgoing connections from each neuron with probability p, as illustrated in Figure. This technique is known as dropout
- Intuitively, the reason dropout works well is that it prevents complex co-adaptation of the hidden units.
- In other words, each unit must learn to perform well even if some of the other units are missing at random.
- This prevents the units from earning complex but fragile dependencies on each other

Dropout

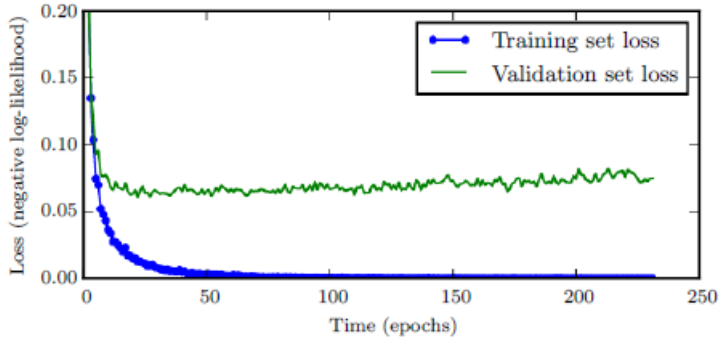


(a) (b)

Figure 13.18: Illustration of dropout. (a) A standard neural net with 2 hidden layers. (b) An example of a thinned net produced by applying dropout with $p_0 = 0.5$. Units that have been dropped out are marked with an x. From Figure 1 of [Sri+14]. Used with kind permission of Geoff Hinton.

Early Stopping

- When training large models with sufficient representational capacity to overfit the task, we often observe that training error decreases steadily over time, but validation set error begins to rise again.
- This means we can obtain a model with better validation set error (and thus, hopefully better test set error) by returning to the parameter setting at the point in time with the lowest validation set error.
- Every time the error on the validation set improves, we store a copy of the model parameters. When the training algorithm terminates, we return these parameters, rather than the latest parameters algorithm terminates when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. This strategy is known as **early stopping**.
- It is probably the most commonly used form of regularization in deep learning



Solved example: MLP

- **Consider the following architecture:**
  - **4 Input Neurons**
  - **1 Hidden Layer with 4 Neurons** (Sigmoid Activation)
  - **1 Output Neuron** (Sigmoid Activation)
  - **Binary Cross-Entropy (BCE) Loss Function**
- **Inputs:**
  - x=[0.1,0.2,0.3,0.4]
  - y=1
- Perform one single forward pass and back propagation

Solved example

Weights and Biases:

$$W^{(1)} = \begin{bmatrix} 0.2 & 0.4 & 0.5 & 0.7 \\ 0.3 & 0.1 & 0.6 & 0.8 \\ 0.9 & 0.2 & 0.3 & 0.5 \\ 0.5 & 0.7 & 0.2 & 0.3 \end{bmatrix}$$

$$b^{(1)} = [0.1, 0.2, 0.3, 0.4]$$

$$W^{(2)} = [0.3, 0.5, 0.7, 0.9]$$

$$b^{(2)} = 0.2$$

Solved example

Pre-activation:

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$$z^{(1)} = \begin{bmatrix} (0.2 \times 0.1) + (0.4 \times 0.2) + (0.5 \times 0.3) + (0.7 \times 0.4) + 0.1 \\ (0.3 \times 0.1) + (0.1 \times 0.2) + (0.6 \times 0.3) + (0.8 \times 0.4) + 0.2 \\ (0.9 \times 0.1) + (0.2 \times 0.2) + (0.3 \times 0.3) + (0.5 \times 0.4) + 0.3 \\ (0.5 \times 0.1) + (0.7 \times 0.2) + (0.2 \times 0.3) + (0.3 \times 0.4) + 0.4 \end{bmatrix}$$

$$z^{(1)} = \begin{bmatrix} 0.2 \\ 0.49 \\ 0.68 \\ 0.65 \end{bmatrix}$$

Solved example

- Applying Sigmoid activation:

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

$$a^{(1)} = \begin{bmatrix} 0.5498 \\ 0.6201 \\ 0.6637 \\ 0.6570 \end{bmatrix}$$

- Compute Output Layer Activation:

$$z^{(2)} = W^{(2)} a^{(1)} + b^{(2)}$$

$$z^{(2)} = (0.3 \times 0.5498) + (0.5 \times 0.6201) + (0.7 \times 0.6637) + (0.9 \times 0.6570) + 0.2$$

$$z^{(2)} = 1.448$$

- Apply **sigmoid activation**:

$$\hat{y} = \sigma(1.448) = \frac{1}{1 + e^{-1.448}} = 0.8097$$

Solved example

- The BCE loss function is: $L = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$

- Since, y=1:

$$L = -\log(0.8097)$$

$$L = 0.2108$$