Intro to Deep Learning

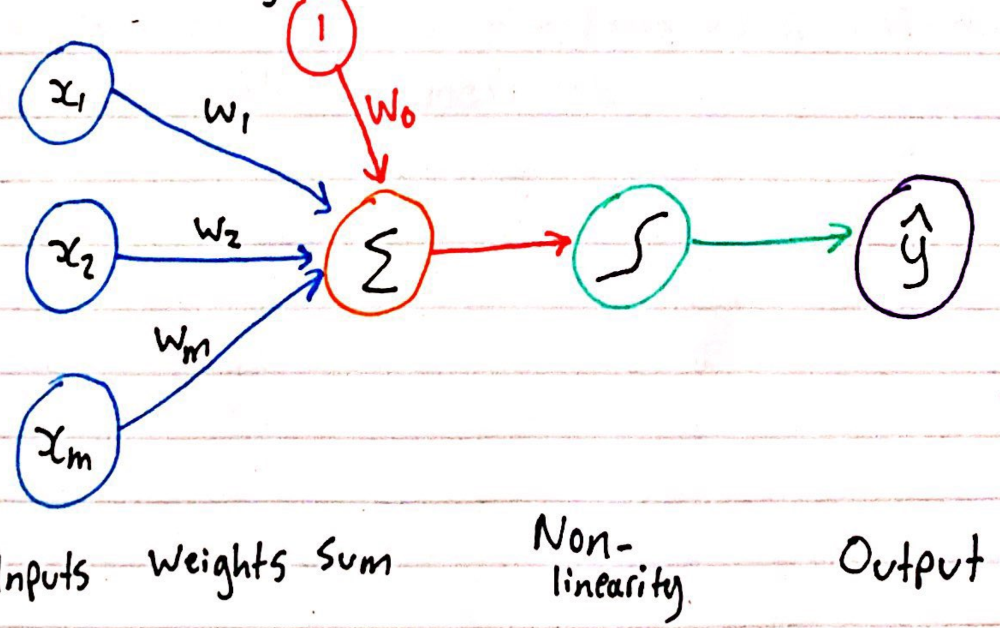
Artificial Intelligence: • Any technique that enables computers to • mimic human behavior

Machine Learning: Ability to learn without explicitly being programmed

Deep Learning: Extract patterns from data using neural networks

Perceptron / Neuron: the structural building block of deep learning

↳ Forward Propagation:



output

bias term

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

Non-linearity activation function

linear combination of inputs

$\longrightarrow \hat{y} = g(w_0 + X^T W)$ where $X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}, W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

Common Activation Functions

Sigmoid Function:

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

$$g'(z) = g(z)(1-g(z))$$

Hyperbolic Tangent:

$$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

$$g'(z) = 1-g(z)^2$$

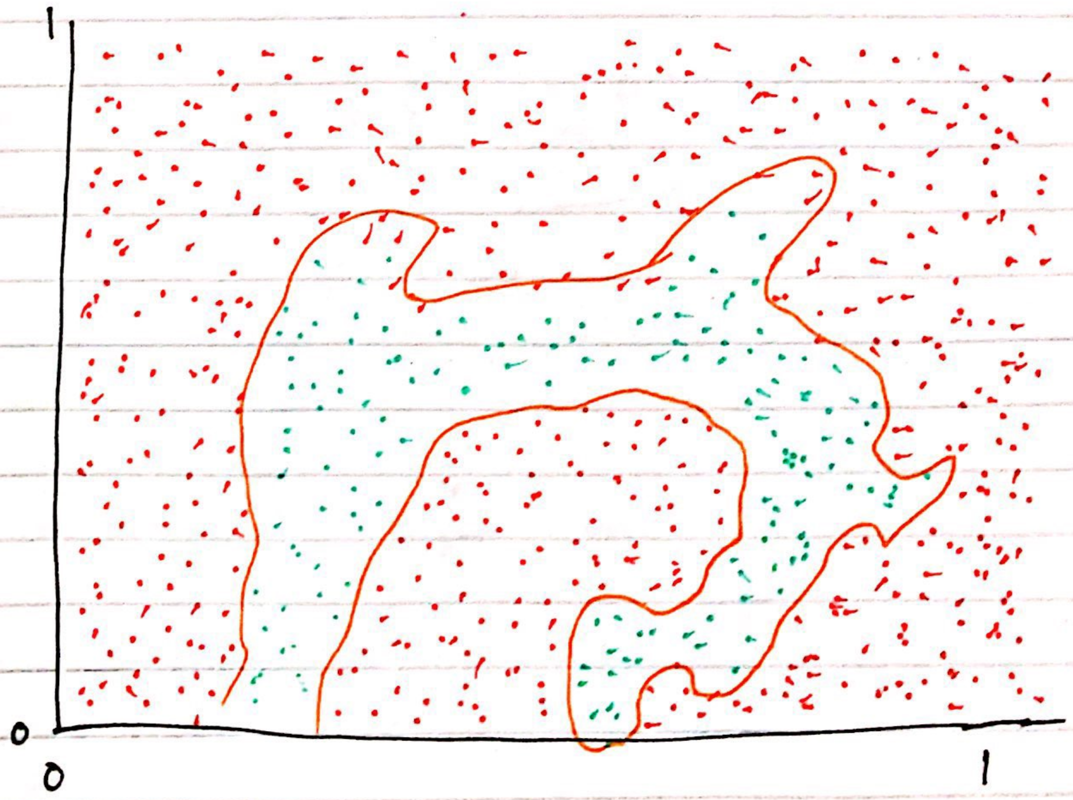
Rectified Linear Unit (ReLU):

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z>0 \\ 0, & o/w \end{cases}$$

Importance of Activation Functions:

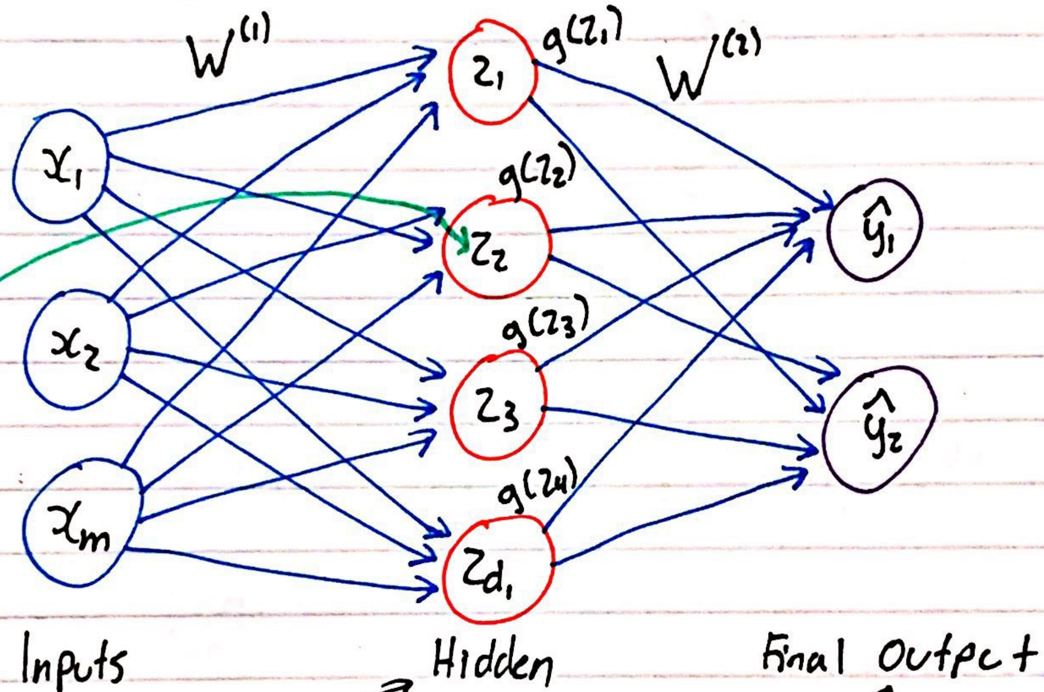
The purpose of activation functions is to introduce non-linearities into the network



Non-linearities allow us to approximate arbitrarily complex functions

Building Neural Networks with Perceptrons

Single Layer Neural Network:



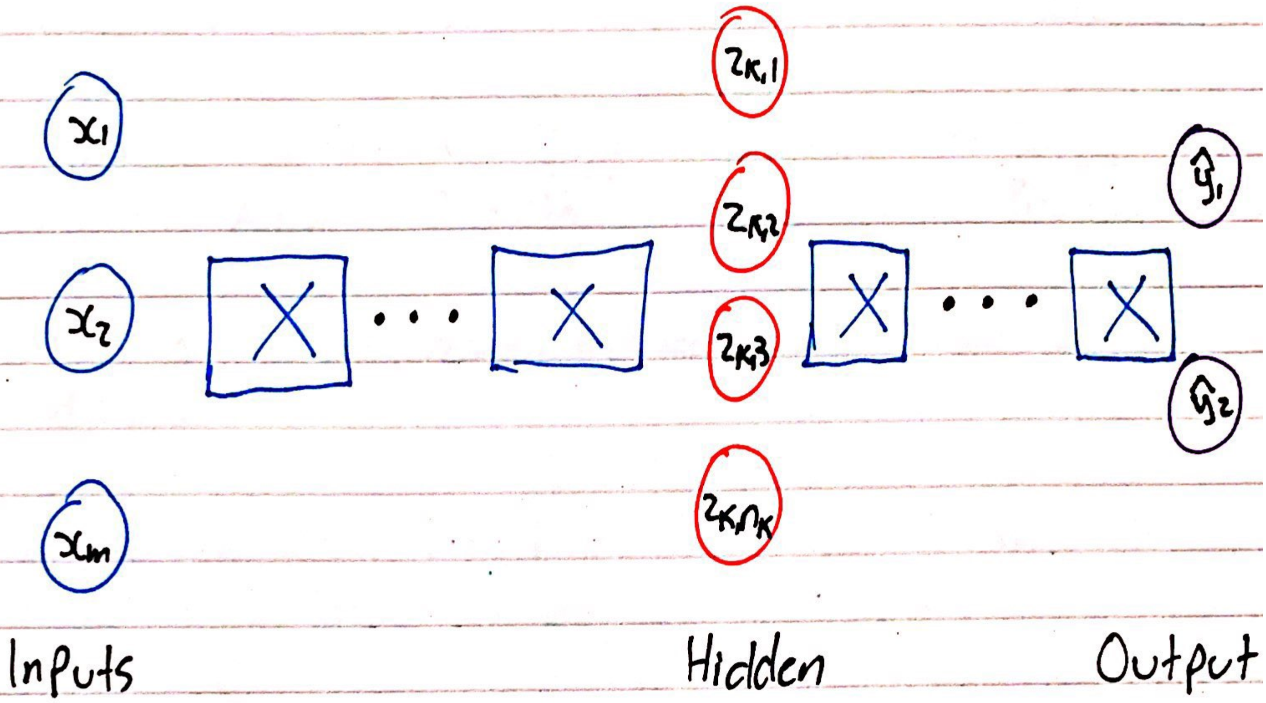
$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j w_{j,i}^{(1)}$$

$$\hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j w_{j,i}^{(2)}\right)$$

$$z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j w_{j,2}^{(1)}$$

$$= w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)}$$

Deep Neural Networks:



Hidden Layer: $z_{K,i} = w_{0,i}^{(K)} + \sum_{j=1}^{n_{K-1}} g(z_{K-1,j})\, w_{j,i}^{(K)}$

Quantifying Loss:

The loss of our network measures the cost incurred from incorrect predictions

$$\mathcal{L}(\underbrace{f(x^{(i)};W)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Empirical Loss: measures the total loss over our entire dataset

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(\underbrace{f(x^{(i)};W)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

Note: It is the average loss amongst all individual losses

Binary Cross Entropy Loss: can be used with models that output a probability between 0 and 1

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log(\underbrace{f(x^{(i)};W)}_{\text{Predicted}}) + (1-\underbrace{y^{(i)}}_{\text{Actual}}) \log(1-\underbrace{f(x^{(i)};W)}_{\text{Predicted}})$$

↳ This is the difference between actual and predicted distributions

Mean Squared Error Loss: can be used with regression models that output continuous real numbers

$$J(W) = \frac{1}{n}\sum_{i=1}^{n} (\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f(x^{(i)};W)}_{\text{Predicted}})^2$$

Training Neural Networks

Goal: we want to find the network weights that achieve the lowest loss

$$W^* = \underset{W}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(f(x^{(i)};W), y^{(i)})$$

$$= \underset{W}{\operatorname{argmin}} \underbrace{J(W)}_{\text{Total Loss Function}}$$

$W = \{W^{(0)}, W^{(1)}, \dots\}$

=> Try to find weights that minimize $J(W)$

Scanned with CamScanner

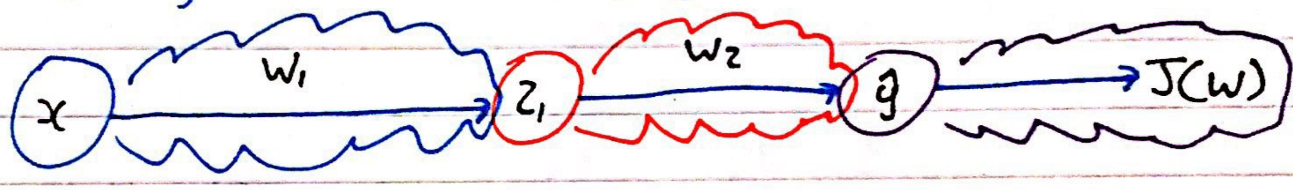
Gradient Descent

Algorithm

1. Initialize weights randomly $\sim N(0, \sigma^2)$
2. Loop until convergence (local minimum):
3.   Compute gradient $\frac{\partial J(W)}{\partial W}$
4.   Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

↳ Learning rate

Computing Gradients: Back propagation



$$\frac{\partial J(W)}{\partial W_1} = \underline{\frac{\partial J(W)}{\partial \hat{y}}} * \underline{\frac{\partial \hat{y}}{\partial z_1}} * \underline{\frac{\partial z_1}{\partial W_1}}$$

Note: ML languages can do this for you

Goal: Pick a learning rate that isn't too small and get stuck at a false local minima but also don't pick an overly aggressive learning rate that causes divergence

↳ Pick a learning rate that adapts to the landscape

Examples: SGD, Adam, Adadelta, Adagrad, RMSProp

Neural Networks in Practice: Mini-batchies

Note: Calculate gradient descent for each data point within a dataset is computationally intensive
↳ Instead we can choose a single point and compute gradient descent relative to that point

However, this is very noisy (Stochastic)

Middle ground: Pick a batch of data points

Modified Step 4: $\frac{\partial J(W)}{\partial W} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$

This allows us to converge at a more optimal rate

Neural Networks in Practice: Overfitting

We want a middle ground between underfitting and overfitting
↳ Our model continues to generalize when it sees new data

Regularization: technique that constrains our optimization problem to discourage complex models

↳ Dropout: During training, randomly set some activations to 0. Typically drop 50% of activation layer. Forces network not to rely on any one node (in hidden layers)

→ Early Stopping: Stop training before we have a chance to overfit