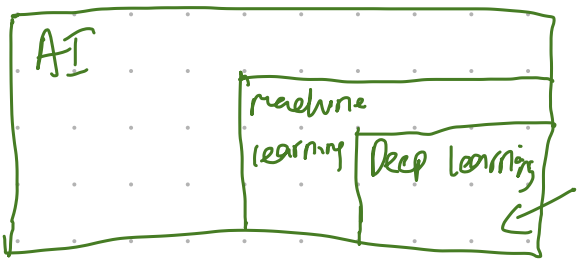
1

History
2020 → Photorealism
2025 → Voice Cloning → class demo.



Lab 1: Music Generation using neural networks
Lab 2: Facial detection systems
→ Fix
Lab 3: Finetune LLMs

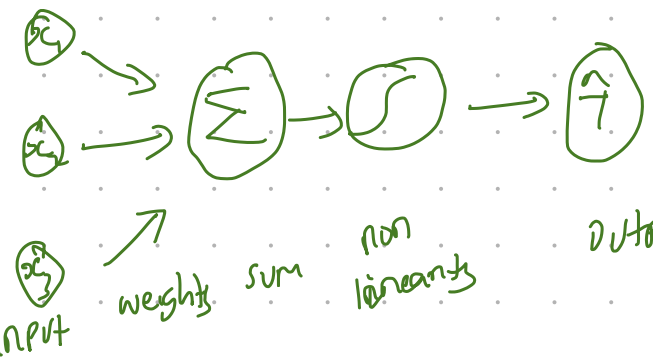
Why deep learning

→ Learn features directly from data
Faces: ① Lines + Edges
② Nose, ~~Ear~~ etc
③ Facial ~~Structure~~
} Bottom up

① Can this be probabilistic
→ Diffusion??
→ Mix & match features.

1 Perceptron

→ Single neuron. output



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

Linear combination of output

Non linear activation function

Bias.

(IT) Why Transpose?

$$\hat{y} = g(w_0 + x^T w)$$

where $x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$ and $w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$

II Common Activation Functions

→ They introduce non linearity (most real world data is non linear)

(I) Sigmoid $g(z) = \frac{1}{1+e^{-z}}$ → good for probability

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

tf.math.sigmoid(x) torch.sigmoid(x)

(II) ReLU $g(z) = \max(0, z)$, $g'(z) = \begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$

tf.nn.relu(x) torch.nn.ReLU(x)

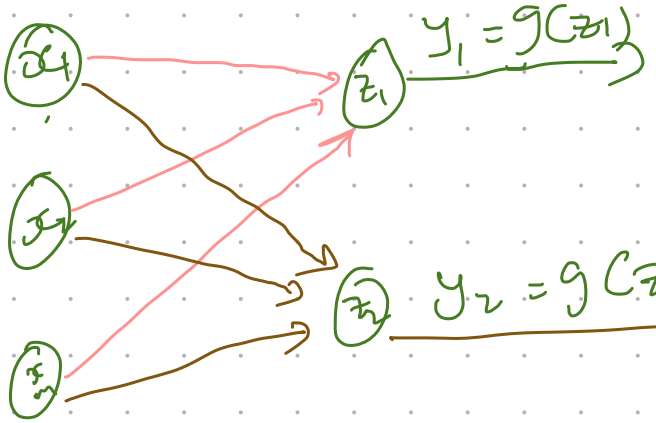
(III) Hyperbolic tangent

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

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

tf.math.tanh(x) torch.tanh(x)

III Multi-Output Perceptron



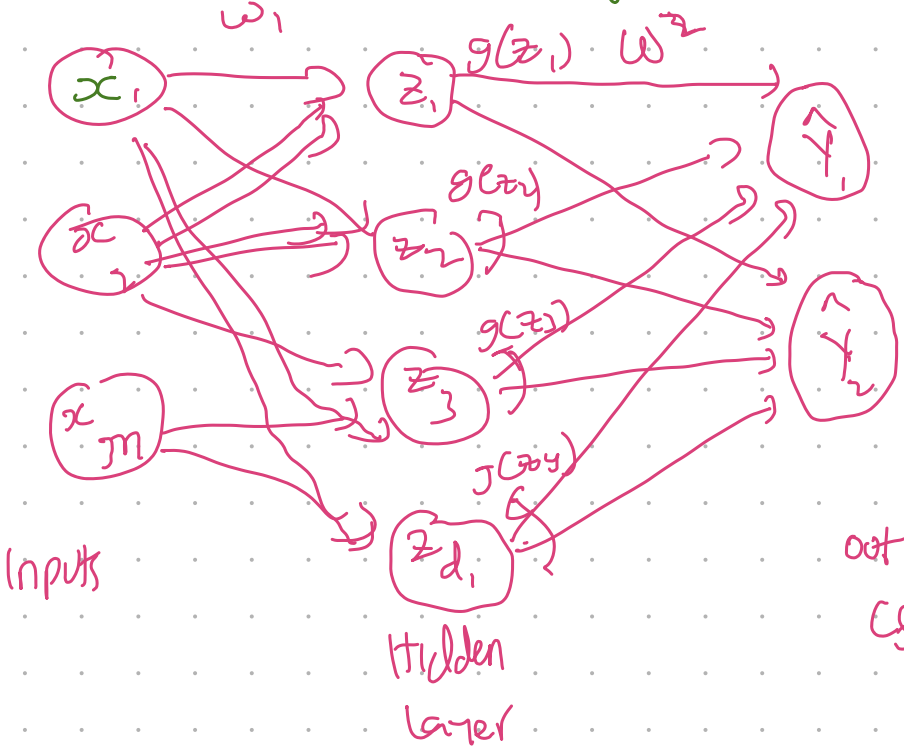
All inputs densely connected to all outputs hence called Dense layers

$$z_i = w_{0,i} + \sum_{j=1}^{m} w_j x_{j,i}$$

[Review Code: M.n 3]

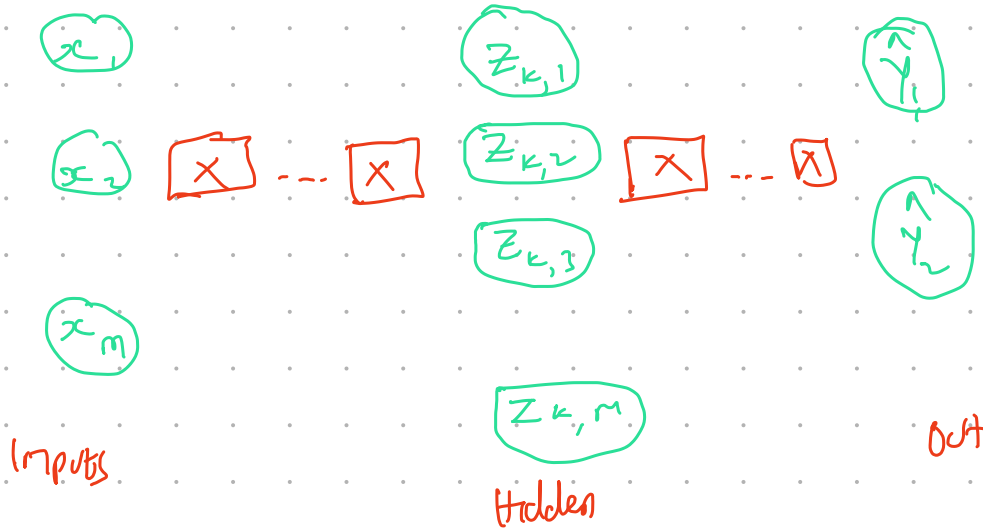
IV | Single Layer Neural Network

-> contains hidden layer



Deep Neural Networks

output of a Hierarchical linear, non linear combination



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

Empirical Loss

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

$$L\left(f(x^i; W), y^i\right)$$

$f(x^i; W)$: Predicted; y^i: Actual

Empirical Loss => total loss over entire dataset.
(cost function / objective function)

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} L\left(f(x^i; W), y^i\right)$$

$f(x^i; W)$: predicted; y^i: actual

Average of every loss in the dataset. (for every point)

Binary Cross Entropy loss

maps to 0, 1, if 0.6 and 1, then no loss

$$\begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \end{bmatrix} \overset{\text{correct}}{\underset{\sim}{\propto}} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

[code]

Mean Squared Error Loss

$$= \frac{1}{n} \sum_{i=1}^{n} (\text{Actual} - \text{Predicted})^2$$

[code]

TRAINING NEURAL NETWORKS

Loss optimization.

→ Improve the network
→ Build models that minimize the loss on a dataset
→ Find weights (W) that minimize loss (L) over dataset
→ Loss function is a function of weights.

How to

- Start at random point.
→ compute gradient g → $g = \frac{\partial J(W)}{\partial W}$
→ says which way is up. (locally)
→ Take a small step towards opposite direction
→ find local minimum

Gradient Descent Pseudocode

1. Initialize weights randomly $\sim N(0, \sigma^2)$
2. Loop until convergence:
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.

Gradient Computation is back propagation $\left(\frac{\partial J(W)}{\partial W}\right)$

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}$$

Repeat for every weight in network using gradients from later layers.

* Tells us whether if we increase our weight, the loss will decrease.

(Just application of chain rule → in theory)

In practice:

Training Neural network is easy

Loss Function Optimization

① difficult.

② Uses Gradient descent.

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

η → learning rate. } How quickly we move

small → gets stuck

high → unstable; diverge.

How to get learning rate

① Try random learning rates and find one that just work

② Design adaptive learning rate that adapts as a function of gradient.

Multiple options: SGD, Adam etc

Gradient Descent vs Stochastic Gradient Descent

Gradient Descent: Computing gradient at every point is very expensive

Stochastic Gradient Descent

1. Initialize weights randomly $\sim N(0, \sigma^2)$
2. Loop until convergence:
   3. Pick a single data point (expensive vs. noisy)
   4. Compute Gradient, $\frac{\partial J_i W}{\partial W}$
   5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. return weights.

Above but with mini batches

→ Pick a batch of B points
→ Smoother convergence
→ Larger learning rates
  1. Possible to trust the gradients
  2. Possible to parallelize

Over-Fitting

Training data is a loose fit, we want it to generalize

Underfitting → model does not fully learn the data

overfitting ⇒ Over-complex, not general. Too many parameters

Just memorizes all the training set.

Regularization

→ Discourages complex model from over-learning/memory.

How

① Dropout

→ Set activations of some neurons to 0.

→ Forces network to not rely on any one node.

→ Forces to rely on multiple parameters

(11) Early Stopping

→ Stop training before it overfits