What is Neural Network

x size → ◯ Neural → Y price

Supervised Learning with Neural network

└→ train with data → has input and output that correct

Standard Neural Network → data with simple input output

Convolution Neural Network → image

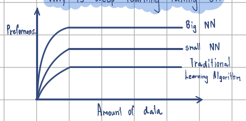
Recurrent Neural Network → audio → text, translator

Custom / Hybrid Neural Network → mix

Structured Data → with columns or rows (sorted data)

Unstructured Data → Audio, Image, Text

Why is deep learning taking off



Data → more training data

Computation → fast training (better hardware)

Algorithms → performance training

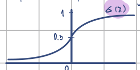
Binary Classification

e.g. Input → 💡 → output → 2 type by using 0 or 1

Logistic Regression

Input → $\vec{x}$, output → $\hat{y} = P(y = 1 \mid x) = \sigma(w^T x + b)$

- want $\hat{y}^{(i)} \approx y^{(i)}$
- $0 \le \hat{y} \le 1$
- but value can negative or > 1 — linear function
- $\sigma(z)$ — sigmoid function



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

cost function $\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$$

If $y = 1$; $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ → want $\log \hat{y}$ large

If $y = 0$; $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ → want $\log(1-\hat{y})$ large (small $\hat{y}$)

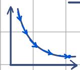
Cost function → $J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

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

Gradient Descent

→ find best value of w, b to minimize $J(\vec{w}, b)$ (→ Global minima)

$$w = w - \alpha \frac{dJ(\vec{w}, b)}{dw} \quad \Big| \quad b = b - \alpha \frac{dJ(\vec{w}, b)}{db}$$



Computation Graph

→ explain why it organized compute gradients or compute derivatives

Derivatives with Computation Graphs

$a = 5$

$b = 3$

$c = 2$

$u = bc$ (6) →	$v = a + u$ (11) →	$J = 3v$ (33)
$\frac{dJ}{du} = \frac{dv}{du} = 3$	$\frac{dJ}{dv} = \frac{dv}{dv} = 3$	$\frac{dJ}{dv} = 3$

$\frac{dJ}{da}(3a + 3u) = 3$

$\frac{d\ \text{FinalOutputVariable}}{d\ \text{Var}}$ → d var

└→ change value b to treat bc a little bit in order to minimize

$\frac{dJ}{db} = \frac{d}{db}\, 3a + 3bc$ or $\frac{dJ}{du} * \frac{du}{db}$

$= 3 * 2 = 6$

$$\frac{dJ}{du} = 3a + 3u = 3 = \frac{dJ}{dv} * \frac{dv}{du}$$

($\frac{dJ}{dv}$ = 3, $\frac{dv}{du}$ = 1)

Logistic Regression Gradient Descent (using for update parameter)

$$z = w_1 x_1 + w_2 x_2 + b \rightarrow \hat{y} = a = \sigma(z) \rightarrow \mathcal{L}(a, y)$$

$dz = \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}(a,y)}{dz}$

$= a - y$

$= \frac{d\mathcal{L}}{da} \cdot \frac{da}{dz}$ ($\frac{da}{dz} = a(1-a)$)

$da = \frac{d\mathcal{L}(a,y)}{da} = \frac{-y}{a} + \frac{1-y}{1-a}$

$\frac{\partial \mathcal{L}}{\partial w} = dw = x_1 * dz$ $\qquad \frac{\partial \mathcal{L}}{\partial w_2} = d_{w_2} = x_2 * d_z$

$\frac{\partial \mathcal{L}}{\partial b} = dz$

Gradient Descent on m examples

$$J(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y)$$

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

$$\frac{\partial}{\partial w_1} J(w, b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})$$

→ $dw_1^{(i)} - (x^{(i)}, y^{(i)})$

coding $J = 0,\ dw_1 = 0,\ dw_2 = 0,\ db = 0$

for $i = 1$ to m

$z^{(i)} = w^T x^{(i)} + b$

$a^{(i)} = \sigma(z^{(i)})$

calculate loss $J^{(i)} = -[y^{(i)} \log a^{(i)} + (1 - y^{(i)})\log(1 - a^{(i)})]$

$dz^{(i)} = a^{(i)} - y^{(i)}$

plus value gradient $dw_1 += x_1^{(i)} \cdot dz^{(i)}$

$dw_2 += x_2^{(i)} \cdot dz^{(i)}$

$db += dz^{(i)}$

Average Gradients

$J /= m,\ dw_1 /= m,\ dw_2 /= m,\ db /= m$

$w_1 = w_1 - \alpha dw_1 \ \big|\ w_2 = w_2 - \alpha dw_2$

$b = b - \alpha db$

Vectorization (z = np.dot(w, x) + b)

→ multiply same time (CPU, GPU)
→ SIMD - single instruction multiple data

examples

$v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \rightarrow u = \begin{bmatrix} e^{v_1} \\ \vdots \\ e^{v_n} \end{bmatrix}$

u = np.exp(V) | another function: np.log(), np.abs(), np.maximum

with for loop:

```
u = np.zeros((n,1))
for i in range(n):
    u[i] = math.exp(V[i])
```

Vectorizing Logistic Regression

$z^{(1)} = w^T x^{(1)} + b \qquad z^{(2)} = w^T x^{(2)} + b$

$a^{(1)} = \sigma(z^{(1)}) \qquad a^{(2)} = \sigma(z^{(2)})$

$Z = [z^{(1)}\ z^{(2)} \dots z^{(m)}] = w^T X + [b_1\ b_2 \dots b_m]$, with $X = [x^{(1)}\ x^{(2)} \dots x^{(m)}]$

call it Broadcasting $= w^T x^1 + b \quad w^T x^2 + b \ \dots \ w^T x^m + b$

Z = np.dot(w.T, X) + b (→ 1×1 matrix)
$A = [a^{(1)}\ a^{(2)} \dots a^{(m)}] = \sigma(Z)$

Vectorizing Logistic Regression's Gradient Computation

$dz^{(1)} = a^{(1)} - y^{(1)}, \dots, dz^{(m)} = a^{(m)} - y^{(m)}$

$dZ = [dz^{(1)} \dots dz^{(m)}]$

$A = [a^{(1)} \dots a^{(m)}] \qquad Y = [y^{(1)} \dots y^{(m)}]$

$dZ = A - Y = [a^{(1)} - y^{(1)} \dots a^{(m)} - y^{(m)}]$

$dw = 0$ (just matrix transpose) | $db = 0$

$dw \mathrel{+}= x^{(1)} dz^{(1)}$ | $db \mathrel{+}= dz^{(1)}$

$db = \frac{1}{m} \sum_{i=1}^{m} (dz^{(i)}) = \frac{1}{m}$ np.sum(dZ)

$dw = \frac{1}{m} X dZ^T = \frac{1}{m} [x^{(1)} \dots x^{(m)}] \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix} = \frac{1}{m} [x^{(1)} dz^{(1)} + \dots + x^{(m)} dz^{(m)}]$

$\vdots \qquad \vdots$

$dw \mathrel{/}= m \qquad db \mathrel{/}= m$

Broadcasting in Python ⇒ made code run faster

A = np.array [3 rows 4 columns]

cal = A.sum(axis = 0) → sum of num in any columns (axis 0 = down, axis 1 = across)

percentage = 100 * A / (cal.reshape(1, 4)) → get percent for value in array rows by rows

example

$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + 100 \rightarrow \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix} = \begin{bmatrix} 101 \\ 102 \\ 103 \\ 104 \end{bmatrix}$

(m, n): $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + [100\ 200\ 300]$ (1, n) → (m, n) $\begin{bmatrix} 100 & 200 & 300 \\ 100 & 200 & 300 \end{bmatrix} = \begin{bmatrix} 101 & 202 & 303 \\ 104 & 205 & 306 \end{bmatrix}$

(m, n): $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 100 \\ 200 \end{bmatrix}$ (m, 1) → (m, n) $\begin{bmatrix} 100 & 100 & 100 \\ 200 & 200 & 200 \end{bmatrix} = \begin{bmatrix} 101 & 102 & 103 \\ 204 & 205 & 206 \end{bmatrix}$

A note python / numpy vectors

a = np.random.randn(5); print(a.T) → not recommend, a Transpose

print(np.dot(a, a.T)) → single value

a = np.random.randn(5,1); → recommend

print(np.dot(a, a.T)) → outer product vector

assert(a.shape == (5,1)) → check condition

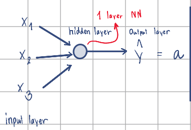
Explain of logistic regression cost function

$\hat{y} = \sigma(w^T x + b) \ ; \ \sigma(z) = \frac{1}{1 + e^{-z}}$

$\hat{y} = p(y = 1 \mid x)$

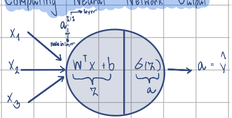
if $y = 1$: $p(y \mid x) = \hat{y}$
$y = 0$: $p(y \mid x) = 1 - \hat{y}$ } $p(y \mid x)$

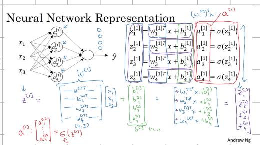
Neural Network Overview

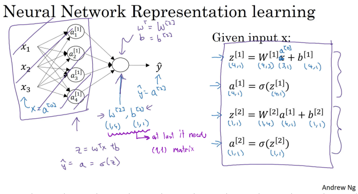


x, w, b | $z = w^T x + b \rightarrow a = \sigma(z) \rightarrow \mathcal{L}(a, y)$ → lost function

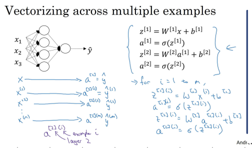
Computing Neural Network Output

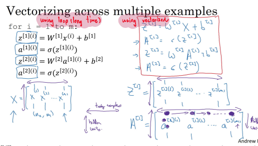






Vectorize Across Multiples Examples

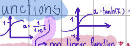




Activations functions

$a^{[1]} = \sigma(z^{[1]})$ → sigmoid function

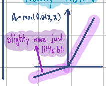
can be another → $g(z^{[1]})$ → non linear function ⇒ always better sigmoid function because it has value between 1 to -1. using sigmoid can has mean = 0 but non linear just made value near to 0



but Output layer can only be Sigmoid because output has value between 0-1

in 2 of activation has bad thing, if value really low or high the value of it is slope close to 0 another option is

Rectified Linear Unit (ReLU) | leaky ReLU



Why Non-linear Activation function

→ ReLU, sigmoid, tanh → Yeah It straight but it change graph, not straight by normally

e.g. decision boundary

⇒ using for learn image, sound, language

Derivatives of activation functions

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

slope = $\frac{d}{dz} g(z) = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) = g(z)(1-g(z))$

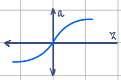
$z = 10, g(z) \approx 1$ $\quad \frac{d}{dz} g(z) \approx (1)(1-1) \approx 0$

$z = -10, g(z) \approx 0$ $\quad \frac{d}{dz} g(z) \approx (0)(1-0) \approx 0$

$z = 0, g(z) \approx \frac{1}{2}$ $\quad \frac{d}{dz} g(z) \approx \frac{1}{2}(1-\frac{1}{2}) = \frac{1}{4}$

Tanh activation function

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



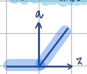
$\frac{d}{dz} g(z) = 1 - (\tanh(z))^2$

$z = 10, \tanh(z) \approx 1, g'(z) \approx 0$

$z = -10, \tanh(z) \approx -1, g'(z) \approx 0$

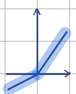
$z = 0, \tanh(z) = 0, g'(z) = 1$

ReLU and leaky ReLU



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

$g'(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \text{ (or 1 if } z \geq 0) \\ \text{undefind} & \text{if } z = 0 \end{cases}$



$g(z) = \max(0.01z, z)$

$g'(z) = \begin{cases} 0.01 & \text{if } z < 0 \\ 1 & \text{if } z > 0 \end{cases}$

Gradient Descent for Neural networks

Parameters ⇒ $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ $\quad (n^{[1]}, n^{[0]}) \ (n^{[1]}, 1) \ (n^{[2]}, n^{[1]}) \ (n^{[2]}, 1)$ $\quad n_x = n^{[0]}, n^{[1]}, n^{[2]} = 1$

Cost function ⇒ $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}, y)$

Gradient Descent ⇒ loops { compute predicts $(\hat{y}^{(i)}, i = 1 \dots m)$

$dW^{[1]} = \frac{dJ}{dW^{[1]}}$; $db^{[1]} = \frac{dJ}{db^{[1]}}$

$W^{[1]} = W^{[1]} - \alpha \, dW^{[1]}$

$b^{[1]} = b^{[1]} - \alpha \, db^{[1]}$ }

Formulas for computing derivatives

Forward Propagation

$Z^{[1]} = W^{[1]}X + b^{[1]}$

$A^{[1]} = g^{[1]}(Z^{[1]})$ → activation function

$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$

$A^{[2]} = g^{[2]}(Z^{[2]}) = \sigma(Z^{[2]})$ → sigmoid function (for output)

Backpropagation (calculate gradient descent)

$dZ^{[2]} = A^{[2]} - Y$ $\quad Y = [Y^{[1]} \ Y^{[2]} \dots Y^{[m]}]$ (how different value between prediction and target)

$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$

$db^{[2]} = \frac{1}{m}\sum dZ^{[2]}$ → np.sum(dZ[2], axis 1, keepdims = True) → python always output like this $(n^{[2]}, 1)$

$dZ^{[1]} = (W^{[2]T} dZ^{[2]}) * g'^{[1]}(Z^{[1]})$ $\quad (n^{[1]}, m)$; * → product; g' → derivatives of activation function

$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$

$db^{[1]} = \frac{1}{m}\sum dZ^{[1]}$ → np.sum(dZ[1], axis = 1, keepdims = True)

Random Initialization

what happen if you initialize weight to zero ⇒ model still same | same function → both hidden unit are still computing exactly the same function

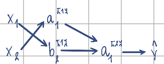
What to do

$W^{[1]}$ = np.random.radn((2,2)) * 0.01 → made weight small

$b^{[1]}$ = np.zero((2,1))

$W^{[2]} = \dots, b^{[2]} = 0$

weight too large → function ⇒ when compute activation if value large after compute all value can be flatout



Deep L-layer Network

shallow ⇒ 1 NN, **deep** ⇒ multiples NN

deep neural network notation

x_1, x_2, x_3 → 5 Neuron → 5 Neuron → 3 Neuron → 1 Neuron → $\hat{y}$

(x_1, x_2, x_3 → call it 'x' or $a^{[0]}$)

$L = 4$ (# layers) $\quad n^{[1]} = 5, n^{[2]} = 5, n^{[3]} = 3, n^{[4]} = n^{[L]} = 1$ (n capital L)

$n^{[l]}$ = # units in layer l $\quad n^{[0]} = n_x = 3$

$a^{[l]}$ = activations in layer l

$a^{[l]} = g^{[l]}(z^{[l]})$, $W^{[l]}$ & $b^{[l]}$ = weights for $z^{[l]}$

Forward Propagation in a deep learning

$z^{[1]} = W^{[1]} x + b^{[1]}$ ($x = a^{[0]}$) $\quad a^{[1]} = g^{[1]}(z^{[1]})$ $\quad z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$ $\quad a^{[2]} = g^{[2]}(z^{[2]})$

output layer $z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$

$a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$

vectorize $Z^{[1]} = W^{[1]} X + b^{[1]}$ ($X = A^{[0]}$)

$A^{[1]} = g^{[1]}(Z^{[1]})$ $\quad Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$ $\quad A^{[2]} = g^{[2]}(Z^{[2]})$

$\hat{Y} = g(Z^{[4]}) = A^{[4]}$

Getting your matrix dimensions right

parameter $W^{[1]}, b^{[1]}$

$Z^{[1]} = W^{[1]} \cdot X + b^{[1]}$

$(3,1) \quad (3,2) \quad (2,1) \quad (3,1)$

$W^{[1]} = (n^{[1]}, n^{[1-1]})$

$Z^{[1]} = W^{[1]} \cdot A^{[1]} + b^{[1]}$

$(5,1) \quad (5,3) \quad (3,1) \quad (5,1)$

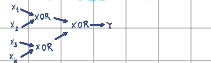
Why deep representations?

circuit theory and deep learning

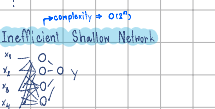
There are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute

only 1 or 2 hidden layers

Efficient Deep Network $y = x_1 \text{ XOR } x_2 \text{ XOR } \dots x_n \text{ XOR}$ → complexity → $O(\log n)$



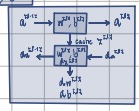
Inefficient Shallow Network → complexity → $O(2^n)$



Building Blocks of a Deep neural network and propagation

Forward and backward functions

layer l: $W^{[l]}, b^{[l]}$



1) Forward Input $a^{[l-1]}$, output $a^{[l]}$

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$ (cache $z^{[l]}$); $a^{[l]} = g^{[l]}(z^{[l]})$ (cache $z^{[l]}$)

2) Backward Input $da^{[l]}$, output $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$

$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]}$

$db^{[l]} = dz^{[l]}$

$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$

$dz^{[l-1]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$

Vectorize version

$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$

$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$

$db^{[l]} = \frac{1}{m} \text{np.sum}(dZ^{[l]}, \text{axis}=1, \text{keepdims}=\text{True})$

$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$

Summary

$X \to \text{ReLU} \to \text{ReLU} \to \text{sigmoid} \to \hat{y} \to L(\hat{y}, y)$ (compute loss)

$X \to Z^{[1]} \to Z^{[2]} \to Z^{[3]}$

backward iterations: $da^{[3]} = -\frac{y}{a^{[3]}} + \frac{(1-y)}{(1-a^{[3]})}$ → $da^{[2]}, da^{[1]}$ → $dW^{[3]}, db^{[3]}$ → $dW^{[2]}, db^{[2]}$ → $dW^{[1]}, db^{[1]}$

Parameters vs Hyperparameters

Hyperparameter ⇒ e.g. learning rate, # iterations, # hidden layer, ...
- you need to tell learning algorithm
- controls parameters

Parameters ⇒ $W^{[1]}, b^{[1]}, \dots$