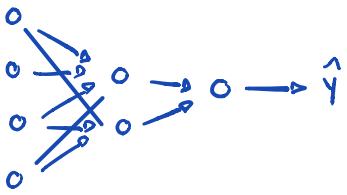
Charlie Mackie
May 22nd 2020

Neural Networks and Deep Learning

Basic Neural Architecture



(*) if all of the nodes are connected between two layers [l-1] and [l] then these layers are 'Deeply Connected'

Supervised Learning:

Unstructured Data
- Sequences (RNNs): audio → transcript
- Images (CNNS / ConvNets): object detection

Structured
- Structured Data (Regression / ANNs): House $ given ___

Unsupervised Learning:

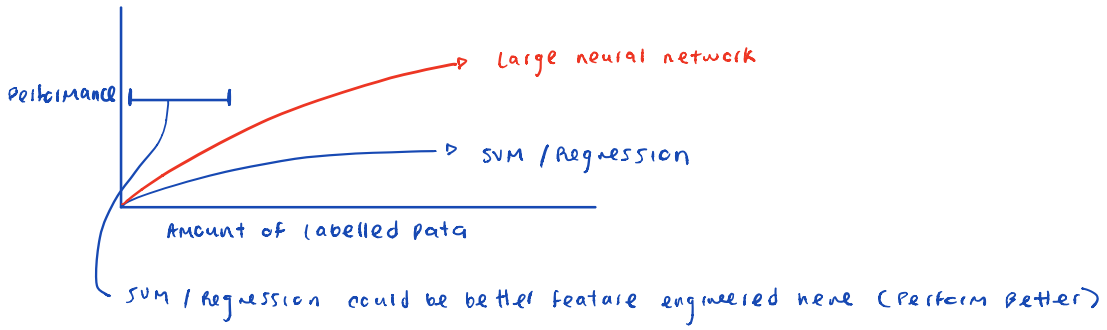
given a dataset without any explicit instructions, find structure in data

Clustering: look for training data that are similar & Group them
Anomaly detection: flag outliers in Data (ex: credit card fraud)
Association: similarity metrics in Data

Reinforcement Learning:

Iterative process of rewarding an agent (network) for good behaviour. Agent relies on past data and exploration of new tactics.

General Network Performance:



(*) with lots of data available, we use Neural Nets

Current State/Progress of NN's:

1. Data: always getting more/better data
2. Computation: more powerful compute = more tuning (iterative process)
3. Algorithms: ex sigmoid → ReLu

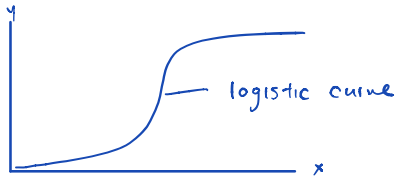
Logistic Regression as a Neural Network

Training Data: 'm' = m train

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix} \; n_x \qquad (n_x, m)$$

$$Y = \begin{bmatrix} y^{(1)}, y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \qquad (1, m)$$

Logistic Regression (Binary Classification)



(*) How can we fit this curve to data to classify it as either 1 or 0?

Training

$\theta = [w, b] = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}$ — θ_1: bias; $\theta_2 \ldots \theta_n$: weights

we want to multiply all X by a weight W and add a bias b.

↳ use $(w^T x)$

(*) use $\hat{y} = \sigma(w^T x + b)$ where sigmoid $= \frac{1}{1 + e^{-x}}$

- $w^T x$: weight · x
- $+ b$: bias
- σ: sigmoid

- if x is small, sigmoid is → 0
- x is large, sigmoid → 1

↳ This will determine if y is 1 or 0 which we can check with the training labels

Loss function

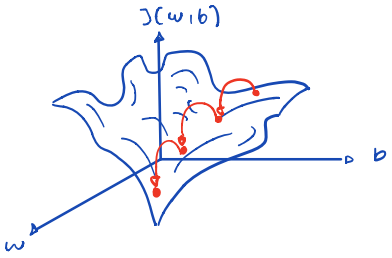
$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ (Squared Error)

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

} loss function examples

↓ sum the loss function over all data

Cost Function = $J(w,b) = \frac{1}{m}\sum_i \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$



(*) we want to iterate through our training data and 'minimize' our cost function

Repeat: $w := w - \alpha \frac{\partial J(w,b)}{\partial w}$ ← learning rate ex: 0.01

(*) update the weights until a 'global optimum' is reached

$\frac{\partial (J(w,b))}{\partial w}$ = slope of $J(w,b)$ in the direction of w

Computation Graph → how derivatives are calculated given select equations & values

ex

a → $v = bc + a$ → $J = 3v$

b → $u = bc$ → $v = bc + a$

c → $u = bc$

(*) applied to a our Neural Network

Our Goal is to find dw_1 & dw_2 so we can optimize

Parameters: x_1, w_1, x_2, w_2, b — Intermediate values — Loss of Example

$z = w_1x_1 + w_2x_2 + b$ → $a = \sigma(z)$ → $\mathcal{L}(a,y)$

$\frac{\partial \mathcal{L}(a,y)}{\partial z} = \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z}$ $\quad \frac{\partial \mathcal{L}(a,y)}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a}$

$= a(1-a)$

$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w_1}$

Gradient Descent (*) with a basic $o \to \sigma \to \hat{y}$ structure

minimizing functions: $J(w,b) = \frac{1}{m} \sum_i \mathcal{L}(a^{(i)}, y)$

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

```
for m in training examples:
    z(i)  = w^T . x(i) + b
    a(i)  = sigmoid(z(i))
    J    += -[-y(i) log(a(i)) + (1-y(i))(log(1 - a(i)))]   (loss function)
    dz(i) = a(i) - y(i)
    dw(i) += x(i) . dz
    db(i) += dz(i)
J /= m
dw(i) /= m
```

Vectorization (for loops have time/space complexity ... use vectors)

ex `z = np.dot(w.T, x) + b`

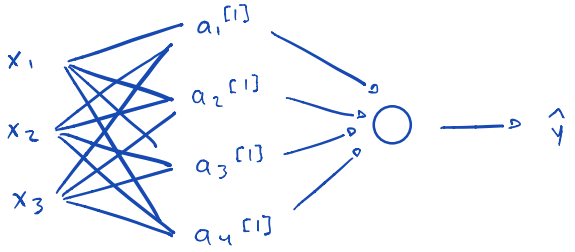
```
Z  = np.dot(w.T, X) + B
A  = σ(Z)
dZ = A - Y
dw = 1/m X dZ^T
db = np.sum(dZ)
```

Summary of a LR Neural Network:

- Model structure / Data
- Initialize parameters (w, b)
- Loop over: current loss
  current Grad
  update parameters

Shallow Neural Network

A Shallow Neural Network has multiple layers of Nodes:



Compute forward output of Shallow Neural Network:
(vectorized implementation)

① $z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

$(4,1)$ $(4,3)$ $(3,1)$ $(4,1)$

* weights for inputs → L1
* inputs

② $a^{[1]} = \sigma(z^{[1]})$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \frac{1}{1+e^{-x}} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}$$

③ $z^{[2]} = W^{[2]} \cdot a^{[1]} + b^{[2]}$

$$\underset{(1,1)}{[z]} = \underset{(1,4)}{[w_1\ w_2\ w_3\ w_4]} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} + [b]$$

* weight matrix dims are used to achieve desired output size

④ $a^{[2]} = \sigma(z^{[23]}) = \hat{y}$

$$\underset{(1,1)}{[a]} = \sigma \underset{(1,1)}{[z]} = \hat{y}$$

* we just vectorized across all weights / bias, also vectorize across the number of examples.

For $i = 1$ to m (training examples):

$$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$$
$$a^{[1](i)} = \sigma(z^{[1](i)})$$
$$z^{[2](i)} = W^{[2]} a^{[1](i)} + b^{[2]}$$
$$a^{[2](i)} = \sigma(z^{[2](i)})$$

* Notice how these params (W, b) are consistent across Tx, changed with GD.

* This is vectorized (NO more 'For' Loop)

$$\underset{([L+1], m)}{z^{[1]}} = W^{[1]} \cdot \underset{(n_x, m)}{X} + b^{[1]}$$

* (hidden units, Tx)
* (inputs, Training examples)

Now ...

$$z^{[1]} = w^T \cdot X + b \quad , \quad a^{[1]} = \sigma(z^{[1]})$$

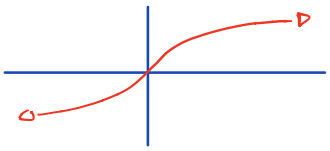
$$\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix} \qquad \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \dots & a^{[1](m)} \\ \downarrow & \downarrow & & \downarrow \end{bmatrix}$$

Activation Function - at this point, we can try to use some better activation functions

activation functions add complexity ... we need them to be non-linear for hidden layers to be effective

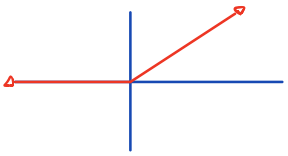
① **tanH**

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



② **ReLu**

$$\text{Relu}(z) = \max(0, z)$$



Tends to learn faster because of the gradient of ReLu.

↳ In tanH or sigmoid, gradient get very small given large values

Forward Propagation through our shallow NN

Forward Prop:

$$z^{[1]} = w^{[1]} X + b^{[1]}$$
$$a^{[1]} = g^{[1]}(z^{[1]})$$
$$z^{[2]} = w^{[2]} a^{[1]} + b^{[2]}$$
$$a^{[2]} = g^{[2]}(z^{[2]})$$

* $g^{(x)}$ = activation function chosen for that layer

Backward Prop:

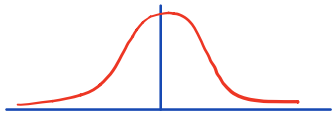
$$dz^{[2]} = A^{[2]} - Y$$
$$dw^{[2]} = \left(\frac{1}{m}\right) dz^{[2]} A^{[1]T}$$
$$db^{[2]} = \left(\frac{1}{m}\right) \text{np.sum}(dz^{[2]}, \text{axis}=1, \text{keepdims})$$
$$dz^{[1]} = (w^{[2]T} \cdot dz^{[2]} * g^{[1]'} \cdot z^{[1]})$$
$$dw^{[1]} = \left(\frac{1}{m}\right) dz^{[1]} \cdot X^T$$
$$db^{[1]} = \left(\frac{1}{m}\right) \text{np.sum}(dz^{[1]})$$

Random Initialization (w, b)

* when we initialize with zeros the activation gradients are low which impedes learning speed

* $w^{[1]} = np.random.randn((2,2)) * 0.01$
  $b^{[1]} = np.zeros((2,1))$
  → keeps derivatives high and learning fast

np.random.randn, returns numbers from a normal distribution.



* high chance of numbers close to 0

Deep Neural Network → every hidden layer serves a purpose, deeper layers compute more complex features

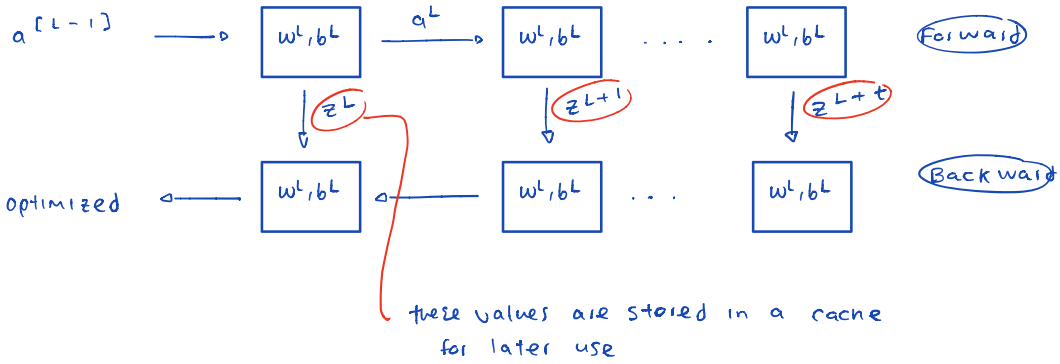
Matrix Dimensions: (# of hidden units layer L)

$w^{[L]} = (n^{[L]}, n^{[L-1]})$
$b^{[L]} = (n^{[L]}, 1)$

→ vectorized across 'm' examples:

$z^{[1]} = w^{[1]} X + b^{[1]}$
$(n^{[1]}, m) \quad (n^{[1]}, n^{[0]}) \quad (n^{[0]}, m)$

Process Visualization (for coding purposes)



⊛ [w, b] are parameters, LR α, num iterations, hidden units/layers
choice of activation are Hyperparameters