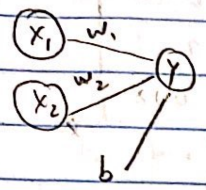
1.



$x_1 = 2$ $\quad w_1 = 0.2$ $\quad b = 0.1$

$x_2 = 1$ $\quad w_2 = 0.3$

Sigmoid activation function

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

Binary cross entropy

$$E = \frac{1}{N}\sum_i -\left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right)$$

$$E'(y_i) = -\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i}$$

(a) Predicted output $= \varphi\left(\sum_i^N x_i w_i + b\right)$

$$= \varphi(x_1 w_1 + x_2 w_2 + b)$$
$$= \varphi((2)(0.2) + 1(0.3) + 0.1)$$
$$= \varphi(0.8)$$
$$= \frac{1}{1+e^{-0.8}}$$
$$= 0.68997$$

(b) $E = -(1)\log(0.68997) + (1-1)(\log(1-0.68997))$

$$= 0.53539$$

(c) Bias allows for shifting of the baseline in the perceptron such as shifting the activation function to be zero at zero $\varphi(0) = 0$

2. $M = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ $\quad K = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}$

4x4 $\quad$ 2x2

stride = 1

padding = 'valid'

assume $b = 0$

assume linear activation function

(a) Output of the convolutional layer

# of neuron: $\left(\frac{4-2}{1}+1\right) \times \left(\frac{4-2}{1}+1\right) \times 1 = 3 \times 3$

$$O = \begin{bmatrix} O_{00} & O_{01} & O_{02} \\ O_{10} & O_{11} & O_{12} \\ O_{20} & O_{21} & O_{22} \end{bmatrix}$$

$O_{00} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 1$ $\quad O_{10} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 0$

$O_{01} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 1$ $\quad O_{11} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 1$

$O_{02} = \begin{bmatrix} 0 & 2 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 1$ $\quad O_{12} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 3$

$$= \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$$

$O_{20} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 0$ $\quad O_{22} = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 1$

$O_{21} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} = 0$

(b) $O = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 2 \\ 0 & 2 & 6 \\ 0 & 0 & 2 \end{bmatrix}$ #

(c) $O = \left[\begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix} \right]$ #

3. 1st convolutional layer

input = $21 \times 21 \times 3$ stride = 2

kernel = $5 \times 5 \times 6$ padding = 'valid'

Dimension of output neurons $= \left(\frac{21-5}{2}+1\right) \times \left(\frac{21-5}{2}+1\right) \times 6$

$= 9 \times 9 \times 6$

Number of trainable parameters $= (5 \times 5 \times 3 + 1) \times 6$

$= 456$

Max pooling layer

input $= 9 \times 9 \times 6$ stride = 2

pool size $= 3 \times 3$ padding = 'valid'

Dimension of output neurons $= \left(\frac{9-3}{2}+1\right) \times \left(\frac{9-3}{2}+1\right) \times 6$

$= 4 \times 4 \times 6$

Number of trainable parameters = 0

2nd convolutional layer

input = $4 \times 4 \times 6$ stride = 1

kernel = $3 \times 3 \times 10$ padding = 'same'

Dimension of output neurons $= \left(\frac{4+2-3}{1}+1\right) \times \left(\frac{4+2-3}{1}+1\right) \times 10$

$= 4 \times 4 \times 10$

Number of trainable parameters : $(3 \times 3 \times 6 + 1) \times 10$

$= 550$

Flatten layer

input $= 4 \times 4 \times 10$

Dimension of output neurons : 160×1

Number of trainable parameters : 0

Dense layer

input : 160×1

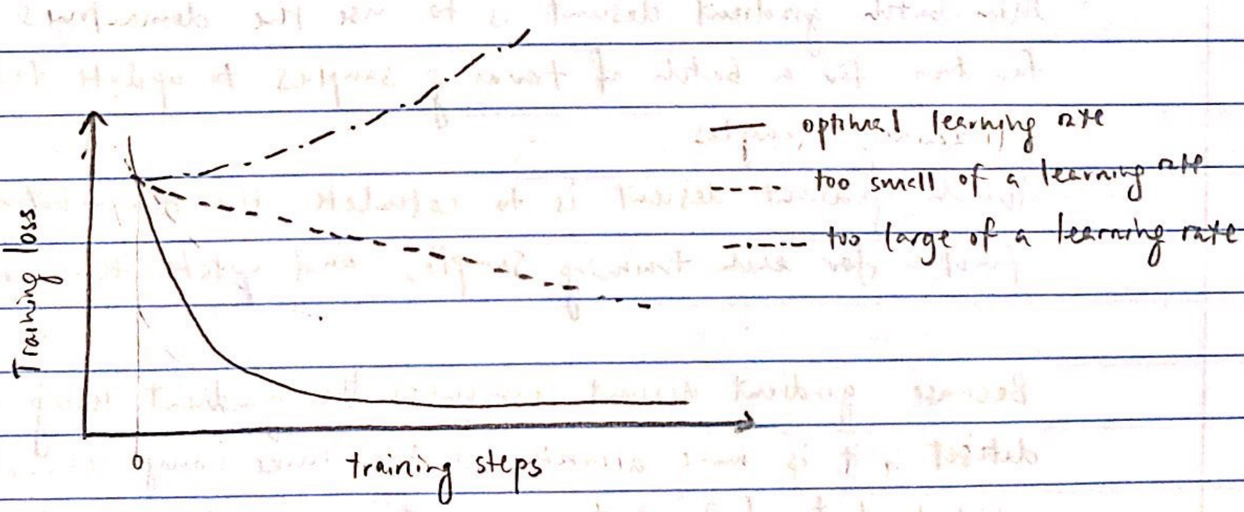
Dimension of output neuron $= 10 \times 1$

Number of trainable parameters $= 160 \times 10$

$= 1600$

4.

(a)



An optimal learning rate converges fairly quickly

Too small of a learning rate takes too long to converge

Too large of a learning rate might not be able to converge and diverge

(b) When performing stochastic gradient descent, even though the amount of time it takes to compute a step is small, the number of steps taken to reach convergence can be high that leads to high training cost.

This is because the loss function in stochastic gradient descent changes at each iteration by only taking one data point. Hence it would take a lot more steps to find the right weights to minimize the loss function for the whole dataset. Frequent update after each step can be computationally expensive as the whole network needs to do a forward pass before doing another backpropagation to do another step of stochastic gradient descent. Hence, this may increase the training cost.

A way to partly alleviate this problem is to use mini-batch gradient descent. Mini-batch gradient descent attempts to find a middle ground between gradient descent and stochastic gradient descent by computing the derivatives of a batch of data points instead of one data point. This allows the loss function to be better aligned with the global loss function.

(c) - Stochastic gradient descent is to use the derivative of the loss function for one training sample to update the weights for all training samples.
- Mini-batch gradient descent is to use the derivatives of the loss function for a batch of training samples to update the weights for all training samples.
- (Batch) gradient descent is to calculate the derivatives of the loss function for each training samples and update the weights respectively

Because gradient descent computes the gradient using the whole dataset, it is more accurate but also more computationally expensive, which leads to higher training cost. Meanwhile stochastic gradient descent computes the gradient using a single sample, it is less accurate than gradient descent but is more computationally inexpensive in computing the gradient. Because it only computes gradient of a

Scanned with CamScanner

single sample, it takes more steps to find a gradient that works for the whole dataset. Mini-batch gradient descent attempts to maximize the potential of both gradient descent and stochastic gradient descent by computing the gradient of a batch of samples. This allows for a better approximation of the global loss function but does not require to compute gradient of the whole dataset. Hence, mini-batch gradient descent is widely used. Parallel processing also makes mini batch gradient descent attractive as the cost to compute a batch of training samples can be the same as the cost to compute one training sample.

(d) To alleviate the issue of vanishing gradients when the networks get very deep, GoogLeNet uses auxillary classifiers in different layers in the network to inject the loss function at intermediate layers instead of just the loss function in the final layer. Meanwhile Resnet uses skip connections to allow gradient from the previous layers to be passed on without being diminished by local gradients.

5. (1) INPUT CONV1 OUTPUT learning rate $= \alpha$

x_i $\quad c_i(X)$ $\quad o_i(X)$ loss function $E = \Sigma(y_i - \hat{y}_i)^2$

w_i^c $\quad w_i^f$ $\quad \frac{dE}{dy} = -2(y_i - \hat{y}_i)$

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

$\frac{d\varphi}{dx} = \frac{e^{-x}}{(1+e^{-x})^2}$

Given a labeled datapoint (x, y)

(a) The update of the weight i of the output node layer

$$w_i^f = w_i^f - \alpha \frac{\partial E}{\partial w_i^f}$$

$$\frac{\partial E}{\partial w_i^f} = \frac{\partial E}{\partial O_i(x)} \frac{\partial O_i(x)}{\partial n_i^f(x)} \frac{\partial n_i^f(x)}{\partial w_i^f}$$

$$= -2(y_i - o_i(x)) \cdot \frac{e^{-c_i(x) w_i^f}}{(1+e^{-c_i(x)\cdot w_i^f})^2} \cdot c_i(x)$$

#

(b) The update to the bias of the output node layer

$$b_i^f = b_i^f - \alpha \frac{\partial E}{\partial b_i^f}$$

$$\frac{\partial E}{\partial b_i^f} = \frac{\partial E}{\partial O_i(x)} \frac{\partial O_i(x)}{\partial n_i^f(x)} \frac{\partial n_i^f(x)}{\partial b_i^f}$$

$$= -2(y_i - o_i(x)) \cdot \frac{e^{-b_i^f}}{(1+e^{-b_i^f})^2} \cdot 1$$

#

(c) The update to the weights w_0^c, w_1^c of the convolutional layer

$$w_i^c = w_i^c - \alpha \frac{\partial E}{\partial w_i^c}$$

$$\delta_{o_i} = \frac{\partial E}{\partial o_i(x)} \frac{\partial o_i(x)}{\partial n_i^f(x)} \frac{\partial n_i^f(x)}{\partial w_i^f}$$

$$= -2(y - o_i(x)) \cdot \frac{e^{-c_i(x) w_i^f}}{(1+e^{-c_i(x) w_i^f})^2} \, c_i(x)$$

$$\frac{\partial E}{\partial w_i^c} = \delta_{o_i} \frac{\partial c_i(x)}{\partial n_i^c(x)} \frac{\partial n_i^c(x)}{\partial w_i^c}$$

$$= \delta_{o_i} \frac{e^{-x_i \otimes w_i^c}}{(1+e^{-x_i \otimes w_i^c})^2} x_i$$

$$= -2(y - o_i(x)) \cdot \frac{e^{-c_i(x) \cdot w_i^f}}{(1+e^{-c_i(x)\cdot w_i^f})^2} \cdot c_i(x) \frac{e^{-x_i \otimes w_i^c}}{(1+e^{-x \otimes w_i^c})^2} \cdot x_i$$

#

(d) To update the bias of the convolutional layer

$$b_i^c = b_i^c - \alpha \frac{\partial E}{\partial b_i^c}$$

$$\frac{\partial E}{\partial b_i^c} = \delta_{o_i} \frac{\partial C_i(x)}{\partial n_i^c(x)} \frac{\partial n_i^c(x)}{\partial b_i^c}$$

$$= \delta_{o_i} \frac{e^{-b_i^c}}{(1+e^{-b_i^c})^2} \cdot 1$$

$$= -2(y - O_i(x)) \cdot \frac{e^{-C_i(x) \cdot w_i^f}}{(1+e^{-C_i(x) \cdot w_i^f})^2} \cdot C_i(x) \cdot \frac{e^{-b_i^c}}{(1+e^{-b_i^c})^2} \cdot 1$$

\#