Neural Networks for Classification

CS412, 2021 Spring

Liu Zuozhu, ZJUI, Mar 2021

Announcement

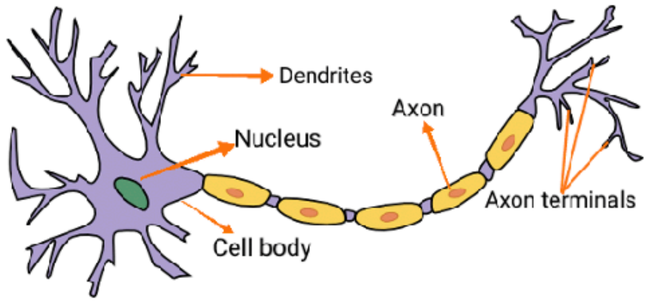
- The deadline for the first homework is extended to 23:59pm, 12 Mar
- Course Projects will be announced next week, please team up asap.

Neural Networks

- **Artificial Neuron**
- Multilayer Perceptron
- Backpropagation
- Training a Neural Network
- Gradient Descent
- Stochastic Gradient Descent

ANN

- Encouragements from the past
  - Biological neural networks -> Deep neural networks



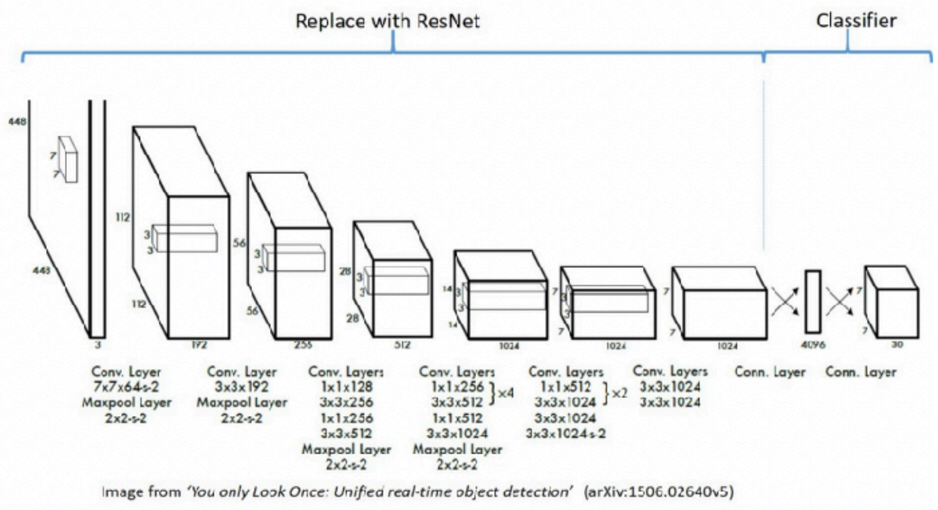
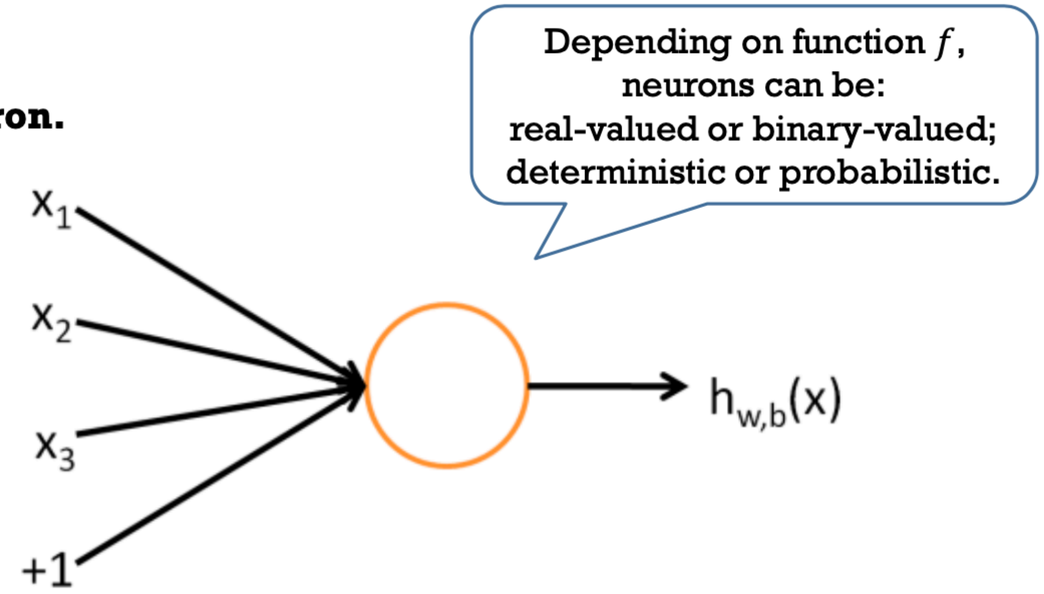


Image from *'You only Look Once: Unified real-time object detection'* (arXiv:1506.02640v5)

Perceptron

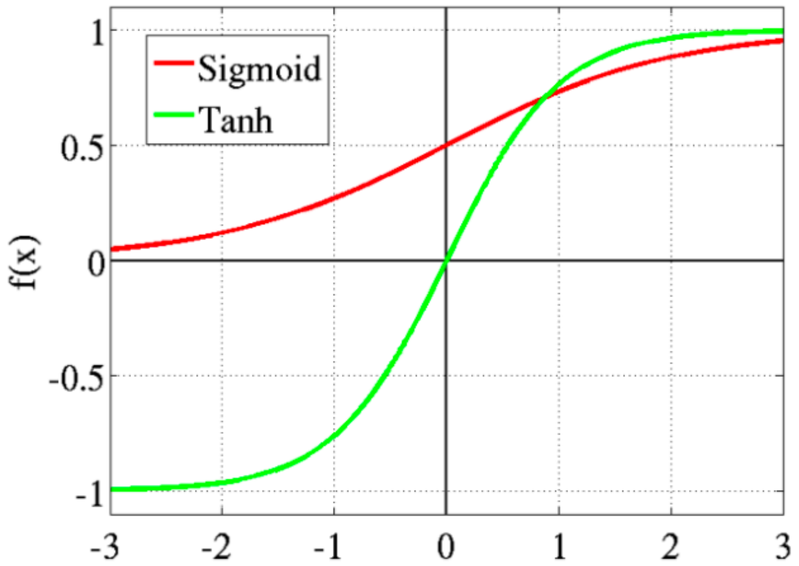
Perceptron.



$$h_{w,b}(x) = f(w^\top x) = f\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

Compare with linear / logistic regression?

Perceptron - Activation Function



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

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

$$h_\theta(x) = g(\theta^T x)$$

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

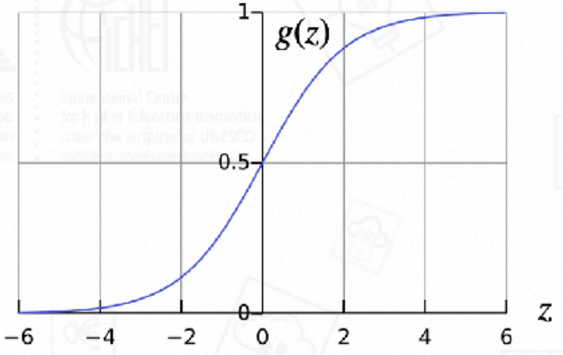
Logistic Function

Sigmoid Function

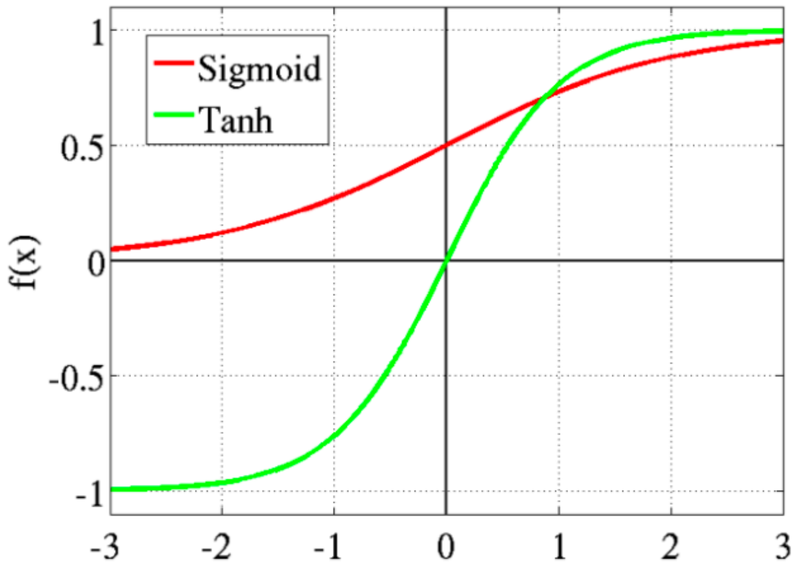
$$h_{w,b}(x) = f(w^\top x) = f\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

$$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$$

Learn θ

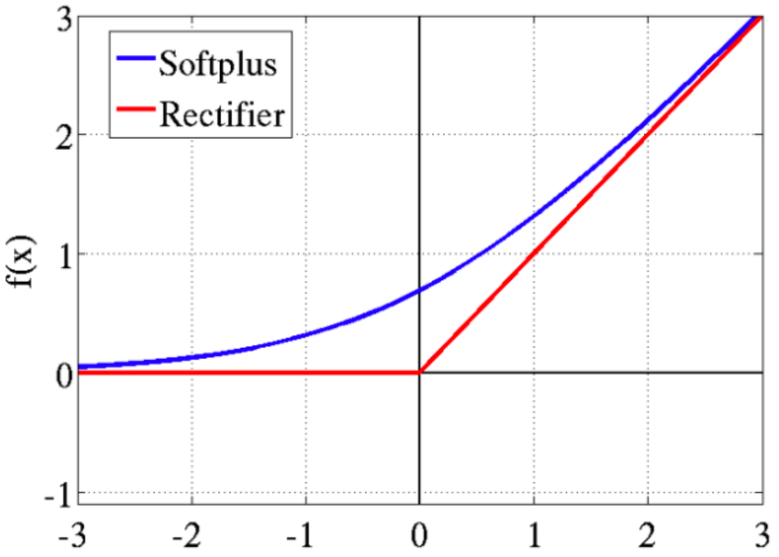


Perceptron - Activation Function



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

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



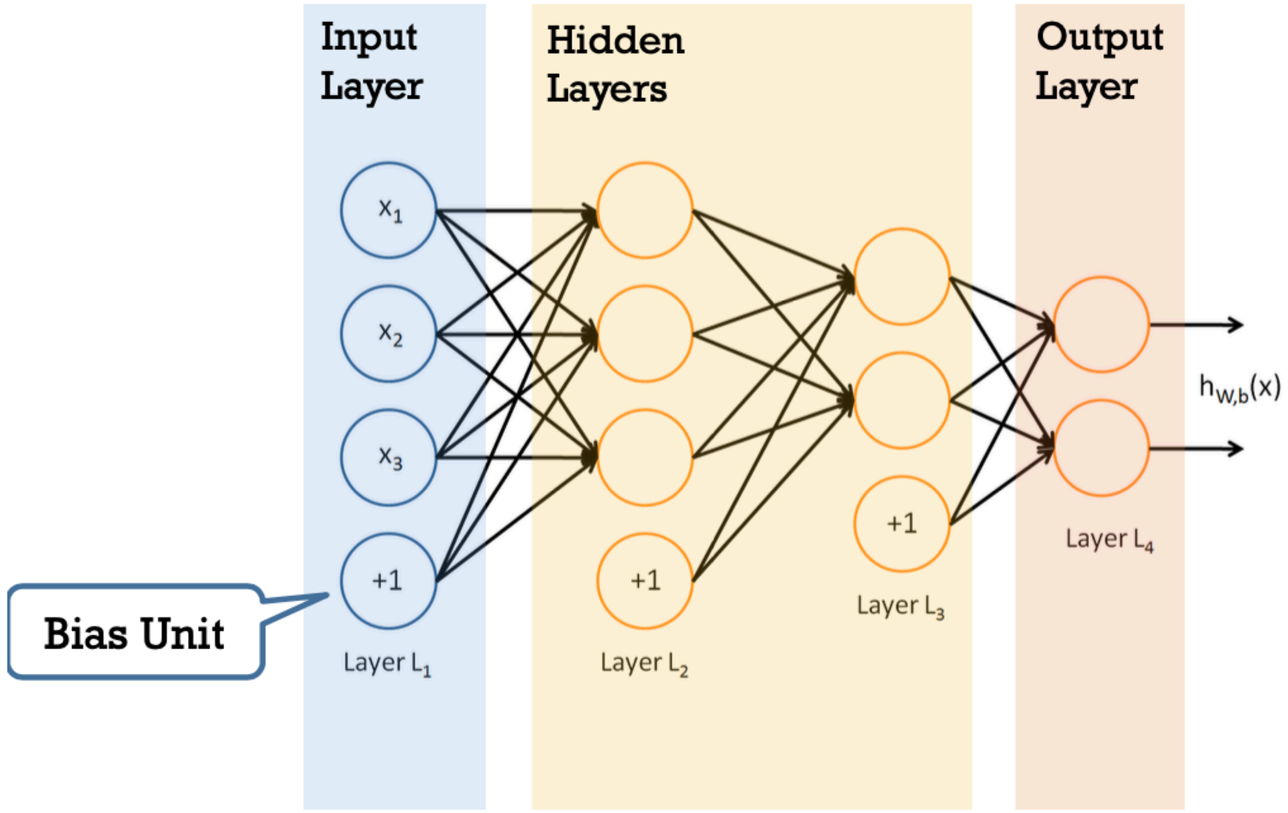
softplus $f(z) = \ln(1 + e^{-z})$

rectified linear unit (ReLU) $f(z) = \max(0, z)$

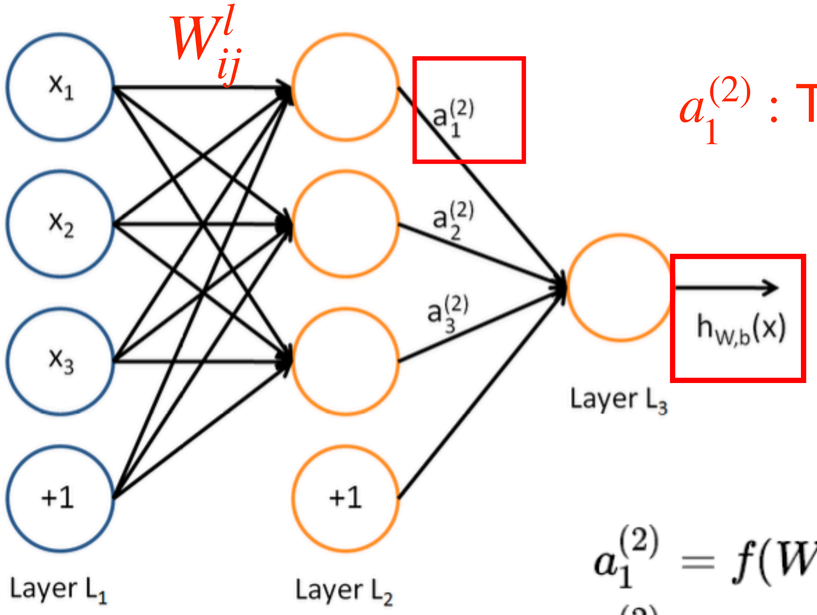
Neural Networks

- Artificial Neuron
- **Multilayer Perceptron**
- Backpropagation
- Training a Neural Network
- Gradient Descent
- Stochastic Gradient Descent

Multi-layer Perceptron



Computation in MLP

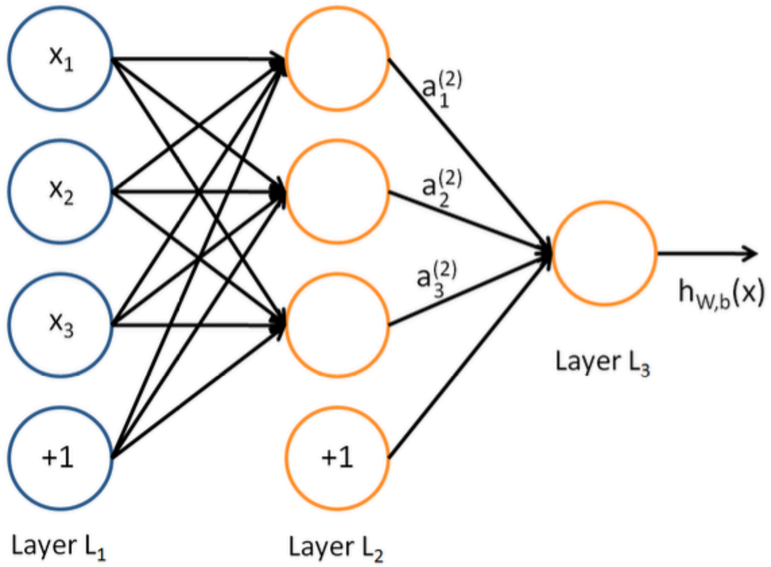


$a_1^{(2)}$: The 1st neuron at the 2-nd layer

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$
$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$
$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

Forward Propagation in MLP

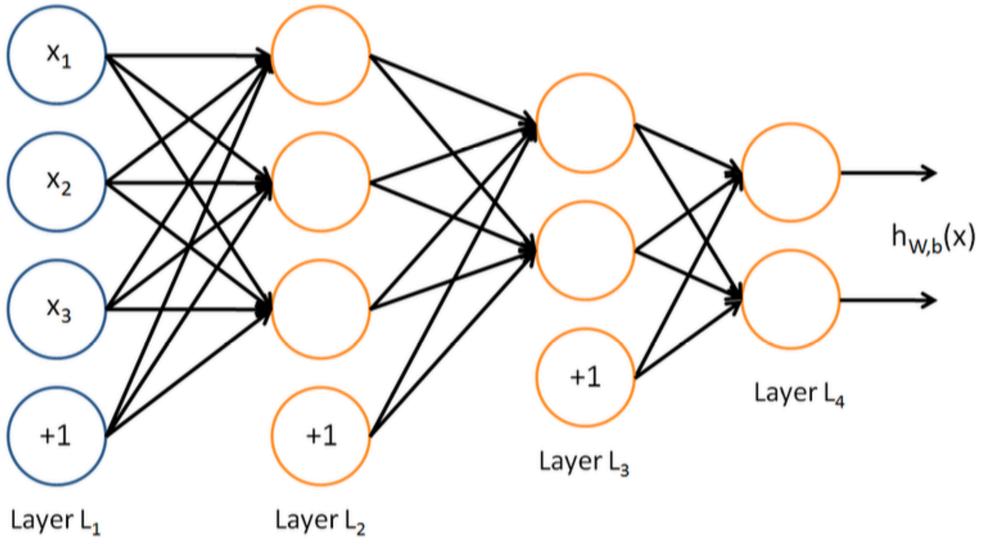


Vectorized Computeation

Forward Propagation.

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

Forward Propagation in General MLP



Feed-Forward Neural Network.

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)})$$

Activation

Loss Functions

For binary neurons, other loss functions are used.

Error for one data point

$$J(W, b; x, y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$

Linear Regression: $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_i^2$

CrossEntropy(P, Q) $= -\sum_{i=1}^{m}[y^{(i)}\log h_\theta(x^{(i)}) + (1 - y^{(i)})\log(1 - h_\theta(x^{(i)})]$

Cross Entropy

Suppose we have a random variable X, X takes n realizations $x_1, x_2, \ldots x_n$,

P(X) is the true distributions of X, Q(X) is our estimated distribution of X,

Cross Entropy : Measures the difference between Q and P

$$\text{CrossEntropy(P, Q)} = H(P, Q) = \sum_i p(x_i) \log \frac{1}{q(x_i)}$$

$$H(P, Q) = \mathbb{E}_{p(x) \in P}[-\log q(x)] = \mathbb{E}_p[-\log q]$$

Properties of KL-divergence:

$$KL(P||Q) \geq 0$$

$$KL(P||Q) = 0 \text{ iff } Q = P$$

Kullback–Leibler divergence

A measure of how one probability distribution is different from a second, reference probability distribution

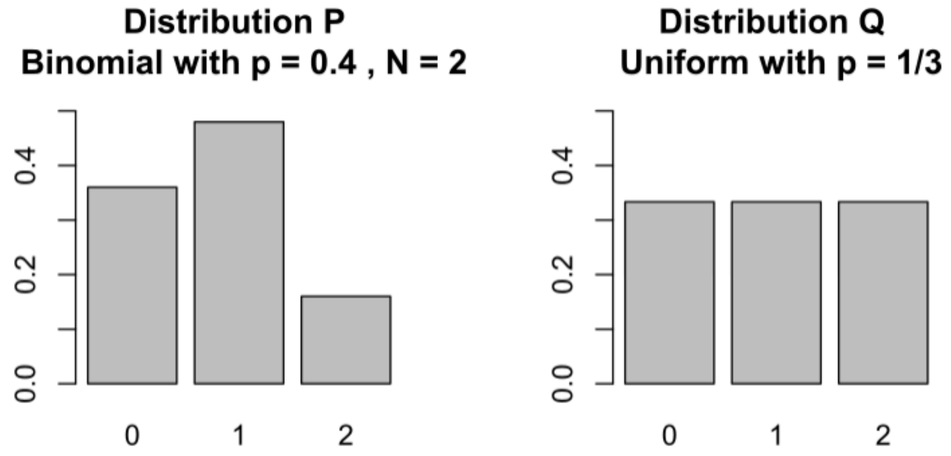
$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{x \in \mathcal{X}} P(x) \log\left(\frac{P(x)}{Q(x)}\right).$$

which is equivalent to

$$D_{\mathrm{KL}}(P \parallel Q) = -\sum_{x \in \mathcal{X}} P(x) \log\left(\frac{Q(x)}{P(x)}\right)$$

Kullback–Leibler divergence

KL is asymmetric



x	0	1	2
Distribution *P*(x)	$9/25$	$12/25$	$4/25$
Distribution *Q*(x)	$1/3$	$1/3$	$1/3$

$$
\begin{aligned}
D_{\mathrm{KL}}(P \parallel Q) &= \sum_{x \in \mathcal{X}} P(x) \ln\left(\frac{P(x)}{Q(x)}\right) \\
&= \frac{9}{25} \ln\left(\frac{9/25}{1/3}\right) + \frac{12}{25} \ln\left(\frac{12/25}{1/3}\right) + \frac{4}{25} \ln\left(\frac{4/25}{1/3}\right) \\
&= \frac{1}{25}\left(32\ln(2) + 55\ln(3) - 50\ln(5)\right) \approx 0.0852996
\end{aligned}
$$

$$
\begin{aligned}
D_{\mathrm{KL}}(Q \parallel P) &= \sum_{x \in \mathcal{X}} Q(x) \ln\left(\frac{Q(x)}{P(x)}\right) \\
&= \frac{1}{3} \ln\left(\frac{1/3}{9/25}\right) + \frac{1}{3} \ln\left(\frac{1/3}{12/25}\right) + \frac{1}{3} \ln\left(\frac{1/3}{4/25}\right) \\
&= \frac{1}{3}\left(-4\ln(2) - 6\ln(3) + 6\ln(5)\right) \approx 0.097455
\end{aligned}
$$

Cross Entropy for LogReg

The prediction label Y is a random variable , Y takes 2 realizations {0, 1},

P(Y) is the true target distributions of Y, Q(Y) is our estimated distribution of Y,

Model: $y = h_\theta(x) = \frac{1}{1 + e^{-\theta^\top x}}$

CrossEntropy(P, Q) $= -\sum_i p(y^{(i)}) \log q(y^{(i)}) = -\sum_i [p(y^{(i)} = 1) \log q(y^{(i)} = 1) + p(y^{(i)} = 0) \log q(y^{(i)} = 0)]$

$$P(y = 1; \theta, x) = h_\theta(x),\ P(y = 0; \theta, x) = 1 - h_\theta(x)$$

CrossEntropy(P, Q) $= -\sum_i [y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})]$

Cross Entropy for LogReg

$$\text{CrossEntropy(P, Q)} = -\sum_i [y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})]$$

Cross Entropy : Measures the difference between Q and P

Our Goal: minimize the difference between our prediction and the true target

min CrossEntropy(P, Q)

Loss Functions

For binary neurons, other loss functions are used.

Error for one data point

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$

Other losses?

Cross-entropy, negative lld,

Minmax games etc…

Error for training set + Weight decay regularization

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

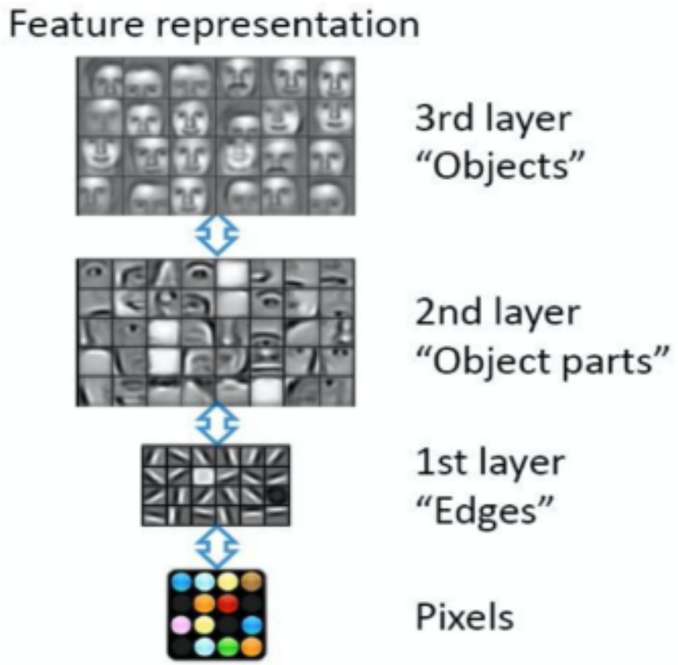
$$= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\left\|h_{W,b}(x^{(i)}) - y^{(i)}\right\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$

λ weight decay parameter

Why Multi-layer

Face Recognition:

- Deep Network can build up increasingly higher levels of abstraction
- Lines, parts, regions

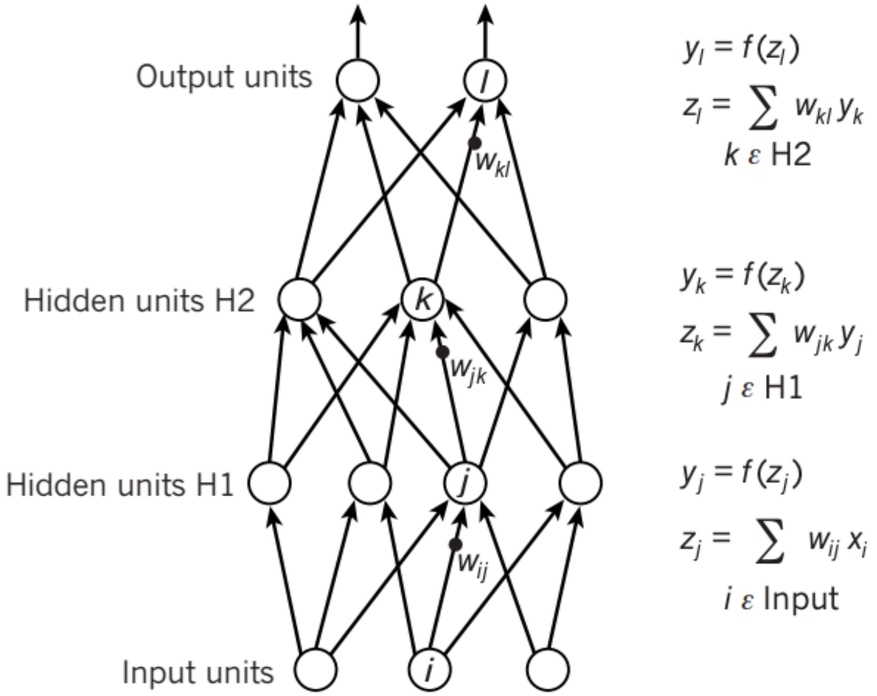


Example from Honglak Lee (NIPS 2010)

Neural Networks

- Artificial Neuron
- Multilayer Perceptron
- **Backpropagation**
- Training a Neural Network
- Gradient Descent
- Stochastic Gradient Descent

How to Update the Parameters in NN



$$y_l = f(z_l)$$
$$z_l = \sum_{k \,\varepsilon\, \text{H2}} w_{kl}\, y_k$$

$$y_k = f(z_k)$$
$$z_k = \sum_{j \,\varepsilon\, \text{H1}} w_{jk}\, y_j$$

$$y_j = f(z_j)$$
$$z_j = \sum_{i \,\varepsilon\, \text{Input}} w_{ij}\, x_i$$

Forward propagation

Inference/Compute loss

$$\frac{\partial E}{\partial y_l} = y_l - t_l$$
$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial y_k} = \sum_{l \,\varepsilon\, \text{out}} w_{kl}\frac{\partial E}{\partial z_l}$$

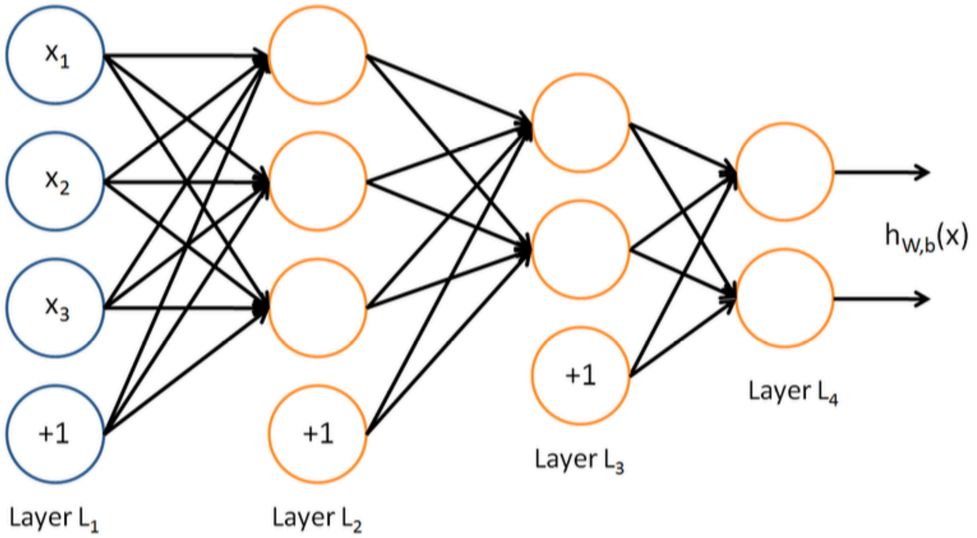
$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \,\varepsilon\, \text{H2}} w_{jk}\frac{\partial E}{\partial z_k}$$
$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$$

Backward propagation

GD for Parameter Update

Gradient Descent and Chain Rule



Feed-Forward Neural Network.

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)})$$
$$h(W,b) = f(z^{(l+1)}) = f(W^{(l)}x + b^{(l)})$$

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$

Gradient Descent:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b)$$

Chain Rule

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$

Gradient Descent:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b)$$

Chain Rule:

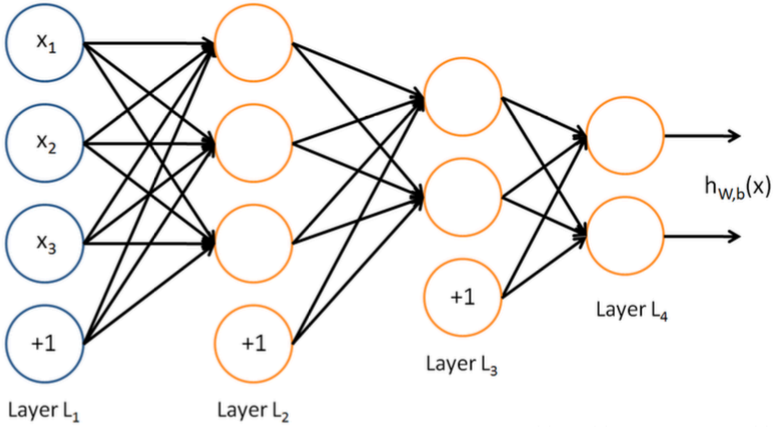
$$\frac{\partial J(W,b)}{\partial W^{(l)}} = \frac{\partial J(W,b)}{\partial z^{(l+1)}} * \frac{\partial z^{(l+1)}}{\partial W^{(l)}}$$

$$\frac{\partial J(W,b)}{\partial b^{(l)}} = \frac{\partial J(W,b)}{\partial z^{(l+1)}} * \frac{\partial z^{(l+1)}}{\partial b^{(l)}}$$

Chain Rule

$$h(W,b) = a^{(4)} = f(z^{(4)}) = f(W^{(3)}a^{(3)} + b^{(3)})$$

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$



For each neuron i at the 4-th layer :

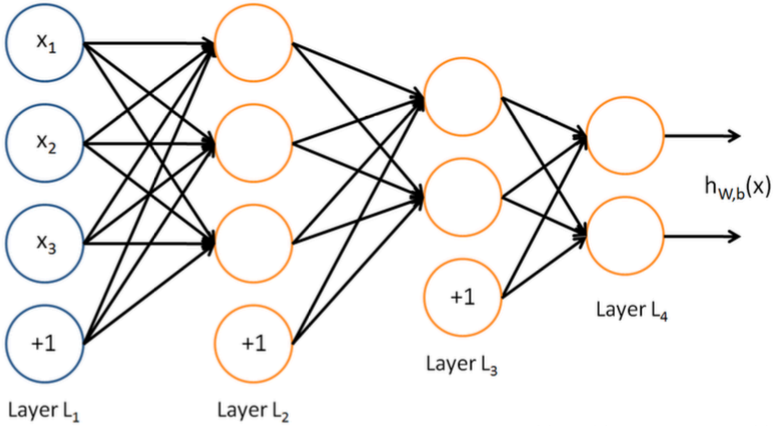
$\delta_i^{(4)}$: error of node i at layer 4

$$\delta_i^{(4)} = \frac{\partial J(W,b)}{\partial z_i^{(4)}} = \frac{\partial \frac{1}{2}\|h_{W,b}(x) - y\|^2}{\partial z_i^{(4)}} = (a_i^{(4)} - y_i) * f'(z_i^{(4)})$$

Chain Rule

$$h(W,b) = a^{(4)} = f(z^{(4)}) = f(W^{(3)}a^{(3)} + b^{(3)})$$

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$



$$\frac{\partial J(W,b)}{\partial W^{(l)}} = \frac{\partial J(W,b)}{\partial z^{(l+1)}} * \frac{\partial z^{(l+1)}}{\partial W^{(l)}}$$

$$z^{(4)} = W^{(3)}a^{(3)} + b^{(3)}$$

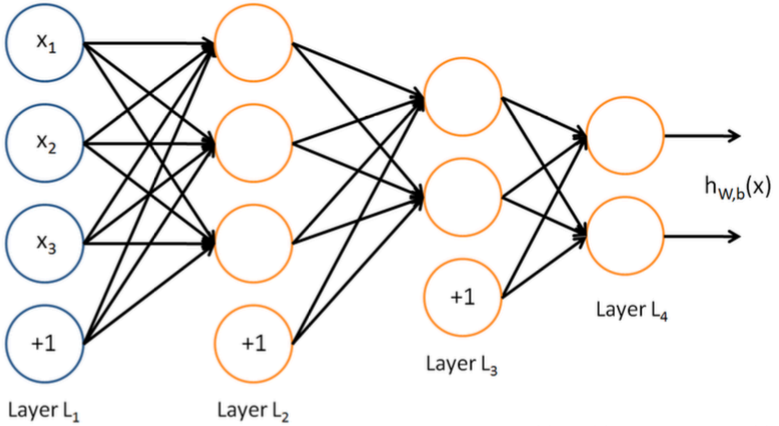
$$z_i^{(4)} = \sum_j W_{ji}^{(3)} a_{ji}^{(3)} + b^{(3)}$$

$$\frac{\partial z_i^{(4)}}{\partial W_{ji}^{(3)}} = a_j^{(3)}$$

Chain Rule

$$h(W,b) = a^{(4)} = f(z^{(4)}) = f(W^{(3)}a^{(3)} + b^{(3)})$$

$$J(W,b;x,y) = \frac{1}{2}\|h_{W,b}(x) - y\|^2$$



$$\frac{\partial J(W,b)}{\partial W^{(l)}} = \frac{\partial J(W,b)}{\partial z^{(l+1)}} * \frac{\partial z^{(l+1)}}{\partial W^{(l)}}$$

$$z^{(4)} = W^{(3)}a^{(3)} + b^{(3)}$$

$$z_i^{(4)} = \sum_j W_{ji}^{(3)} a_{ji}^{(3)} + b^{(3)}$$

$$\frac{\partial z_i^{(4)}}{\partial W_{ji}^{(3)}} = a_j^{(3)}$$

Chain Rule

$$\frac{\partial J(W,b)}{\partial W^{(l)}} = \frac{\partial J(W,b)}{\partial z^{(l+1)}} * \frac{\partial z^{(l+1)}}{\partial W^{(l)}}$$

$$\delta_i^{(4)} = \frac{\partial J(W,b)}{\partial z_i^{(4)}} = \frac{\partial \frac{1}{2}\|h_{W,b}(x) - y\|^2}{\partial z_i^{(4)}} = (a_i^{(4)} - y_i) * f'(z_i^{(4)}) \qquad \frac{\partial z_i^{(4)}}{\partial W_{ji}^{(3)}} = a_j^{(3)}$$

$$\frac{\partial J(W,b)}{\partial W_{ji}^{(3)}} = a_j^{(3)} * \delta_i^{(4)} \qquad \frac{\partial J(W,b)}{\partial b^{(3)}} = \delta_i^{(4)}$$

Chain Rule - Continued

$$h(W,b) = a^{(4)} = f(z^{(4)}) = f(W^{(3)}a^{(3)} + b^{(3)})$$

$$a^{(3)} = f(z^{(3)}) = f(W^{(2)}a^{(2)} + b^{(2)})$$

Vectorization / Element-wise product

$$\delta^{(3)} = \frac{\partial J(W,b)}{\partial z^{(3)}} = {W^{(3)}}^{\top}\delta^{(4)} .* f'(z^{(3)})$$

No error at the first layer

How about layer 2?

$$\delta^{(2)} = \frac{\partial J(W,b)}{\partial z^{(2)}} = {W^{(2)}}^{\top}\delta^{(3)} .* f'(z^{(2)})$$

Chain Rule in MLP

For each neuron i at the l-th layer :

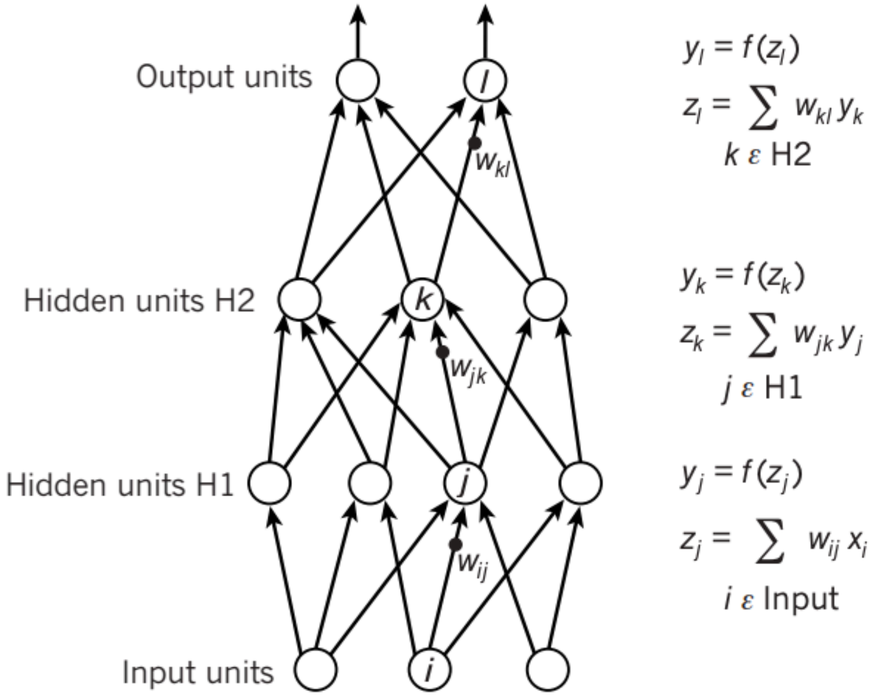
$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

Compute the derivatives:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}.$$

How to Update the Parameters in NN



$$y_l = f(z_l)$$
$$z_l = \sum_{k \,\varepsilon\, \text{H2}} w_{kl}\, y_k$$

$$y_k = f(z_k)$$
$$z_k = \sum_{j \,\varepsilon\, \text{H1}} w_{jk}\, y_j$$

$$y_j = f(z_j)$$
$$z_j = \sum_{i \,\varepsilon\, \text{Input}} w_{ij}\, x_i$$

Forward propagation

Inference/Compute loss

$$\frac{\partial E}{\partial y_l} = y_l - t_l$$
$$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}$$

$$\frac{\partial E}{\partial y_k} = \sum_{l \,\varepsilon\, \text{out}} w_{kl}\frac{\partial E}{\partial z_l}$$

$$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$$

$$\frac{\partial E}{\partial y_j} = \sum_{k \,\varepsilon\, \text{H2}} w_{jk}\frac{\partial E}{\partial z_k}$$
$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$$

Backward propagation

GD for Parameter Update

Backprop Algorithm

1: Perform a feedforward pass, computing the activations for layers $L_2, L_3 \ldots L_l$

Backprop Algorithm

1: Perform a feedforward pass, computing the activations for layers $L_2, L_3 \ldots L_l$

2: Compute the “error” for layer $L_l, L_{l-1}, \ldots L_2$

$$\delta^{(l)} = (a_i^{(l)} - y_i) .* f'(z^{(l)})$$

$$\delta^{(k)} = W^{(k)\top}\delta^{(k+1)} .* f'(z^{(k)}), \text{ for } k \in \{l-1, l-2, ...2\}$$

Backprop Algorithm

1: Perform a feedforward pass, computing the activations for layers $L_2, L_3 \ldots L_l$

2: Compute the "error" for layer $L_l, L_{l-1}, \ldots L_2$

$$\delta^{(l)} = (a_i^{(l)} - y_i) .* f'(z^{(l)})$$

$$\delta^{(k)} = W^{(k)\top} \delta^{(k+1)} .* f'(z^{(k)}), \text{ for } k \in \{l-1, l-2, \ldots 2\}$$

3: Compute the derivatives

$$\frac{\partial J(W,b)}{\partial W_{ji}^{(k)}} = a_j^{(k)} * \delta_i^{(k+1)}, k \in \{k = l-1, l-2, \ldots 1\}, a^{(1)} = x$$

$$\frac{\partial J(W,b)}{\partial b^{(k)}} = \delta^{(k+1)}, k \in \{k = l-1, l-2, \ldots 1\}$$

Backprop Algorithm

1: Perform a feedforward pass, computing the activations for layers $L_2, L_3 \ldots L_l$

2: Compute the "error" for layer $L_l, L_{l-1}, \ldots L_2$

$$\delta^{(l)} = (a_i^{(l)} - y_i) .* f'(z^{(l)})$$

$$\delta^{(k)} = W^{(k)\top}\delta^{(k+1)} .* f'(z^{(k)}), \text{ for } k \in \{l-1, l-2, \ldots 2\}$$

3: Compute the derivatives, matrix-vector form

$$\frac{\partial J(W,b)}{\partial W^{(k)}} = \delta^{(k+1)} a^{(k)\top}, k \in \{k = l-1, l-2, \ldots 1\}$$

$$\frac{\partial J(W,b)}{\partial b^{(k)}} = \delta^{(k+1)}, k \in \{k = l-1, l-2, \ldots 1\}$$

Gradient Descent with Backpropagation

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.

Gradient Descent with Backpropagation

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.

2. For $i = 1$ to m,

    1. Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.
    2. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.
    3. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.

m samples

Gradient Descent with Backpropagation

1. Set $\Delta W^{(l)} := 0$, $\Delta b^{(l)} := 0$ (matrix/vector of zeros) for all l.

2. For $i = 1$ to m,

    1. Use backpropagation to compute $\nabla_{W^{(l)}} J(W, b; x, y)$ and $\nabla_{b^{(l)}} J(W, b; x, y)$.
    2. Set $\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} J(W, b; x, y)$.
    3. Set $\Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} J(W, b; x, y)$.

3. Update the parameters:

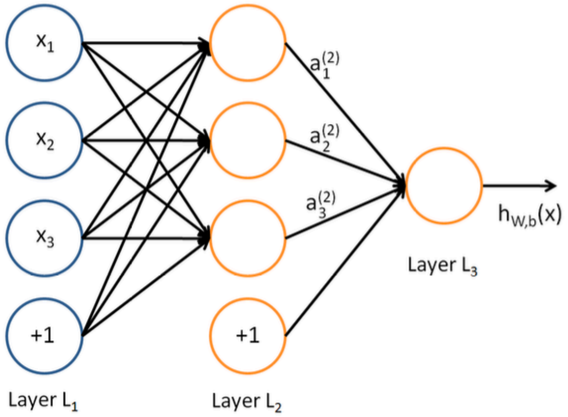
$$W^{(l)} = W^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta W^{(l)} \right) + \lambda W^{(l)} \right]$$

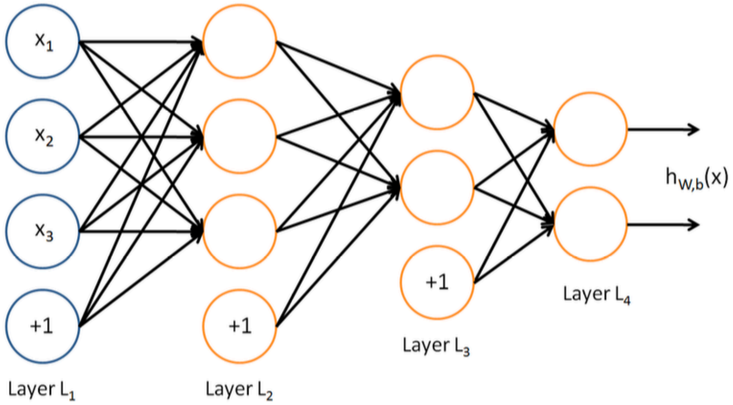
$$b^{(l)} = b^{(l)} - \alpha \left[\frac{1}{m} \Delta b^{(l)} \right]$$

Neural Networks

- Artificial Neuron
- Multilayer Perceptron
- Backpropagation
- **Training a Neural Network**
- Gradient Descent
- Stochastic Gradient Descent

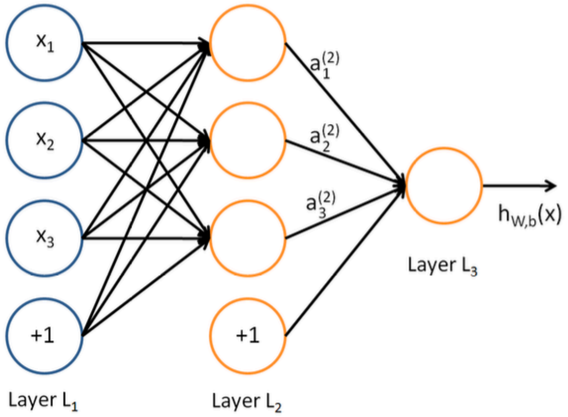
Neural Network: Architecture

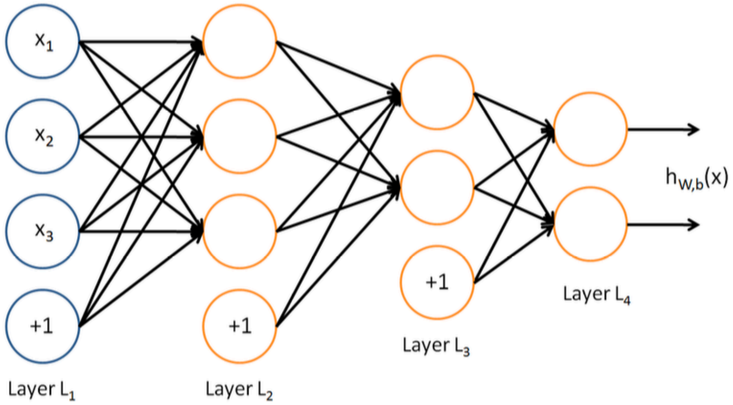




- Choose the number of hidden layers, (e.g.., 1 hidden layer, > 1 hidden layers)

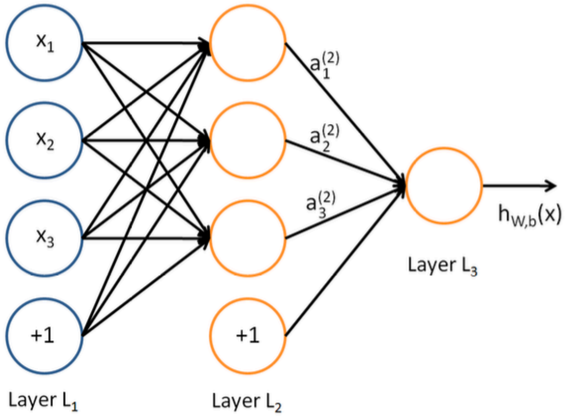
Neural Network: Architecture

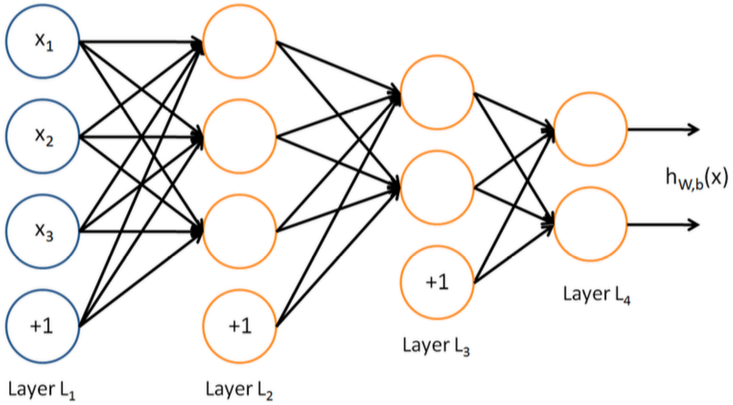




- Choose the number of hidden layers, (e.g.., 1 hidden layer, > 1 hidden layers)
- Choose the number of hidden units (10, 128, 512 …, usually the more the better, but might overfitting)

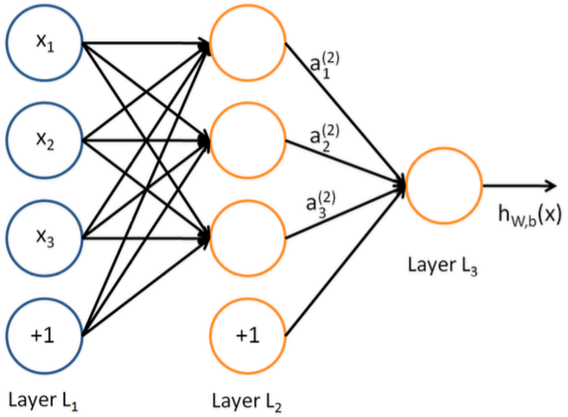
Neural Network: Task and Loss

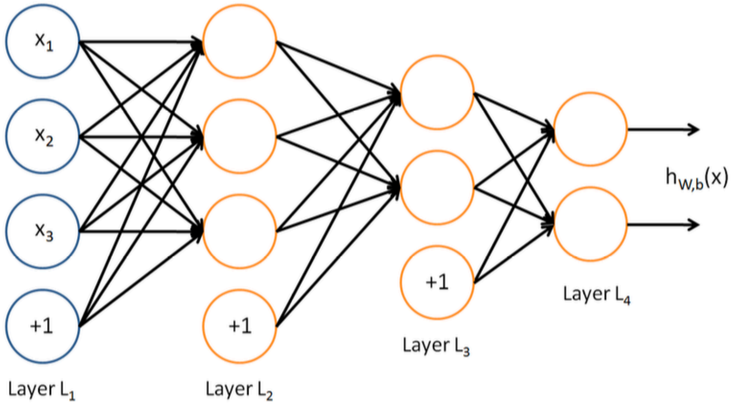




- Classification task, output units = number of classes
- Use cross entropy loss for binary or multi-class classification
- Add regularization to avoid overfitting
- Similar to logistic regression loss

Neural Network: Task and Loss





- Regression task, output units = desired output dimension
- Reconstruct an image, output units = input image size
- Reconstruction loss + regularization
- Similar to linear regression loss

Training a NN

1. Randomly initialize the weight and bias parameters
2. Forward Pass: forward with the NN to get $h_{W,b}(x^{(i)})$ for every sample $x^{(i)}$
3. Compute Loss: compute the loss $J(W, b)$ with predictions $h_{W,b}(x^{(i)})$ and true targets $y^{(i)}$

Training a NN

1. Randomly initialize the weight and bias parameters
2. Forward Pass: forward with the NN to get $h_{W,b}(x^{(i)})$ for every sample $x^{(i)}$
3. Compute Loss: compute the loss $J(W, b)$ with predictions $h_{W,b}(x^{(i)})$ and true targets $y^{(i)}$
4. Backprop: Implement backprop to compute derivatives

$$\frac{\partial J(W, b)}{\partial W^{(k)}}, \frac{\partial J(W, b)}{\partial b^{(k)}}$$

Training a NN

1. Randomly initialize the weight and bias parameters
2. Forward Pass: forward with the NN to get $h_{W,b}(x^{(i)})$ for every sample $x^{(i)}$
3. Compute Loss: compute the loss $J(W, b)$ with predictions $h_{W,b}(x^{(i)})$ and true targets $y^{(i)}$
4. Backprop: Implement backprop to compute derivatives

$$\frac{\partial J(W,b)}{\partial W^{(k)}}, \frac{\partial J(W,b)}{\partial b^{(k)}}$$

5. Gradient Checking: check if gradient is correct (automatically gradient computation with modern frameworks such as pytorch/tensorflow)
6. Gradient Descent: Use stochastic gradient descent or advanced method (Adam, RMSprop) to learn the optimal parameters

Neural Networks

- Artificial Neuron
- Multilayer Perceptron
- Backpropagation
- Training a Neural Network
- **Gradient Descent**
- Stochastic Gradient Descent

Intuition of Gradient Descent

Hypothesis/Model: $h_\theta(x) = \theta_0 + \theta_1 x$

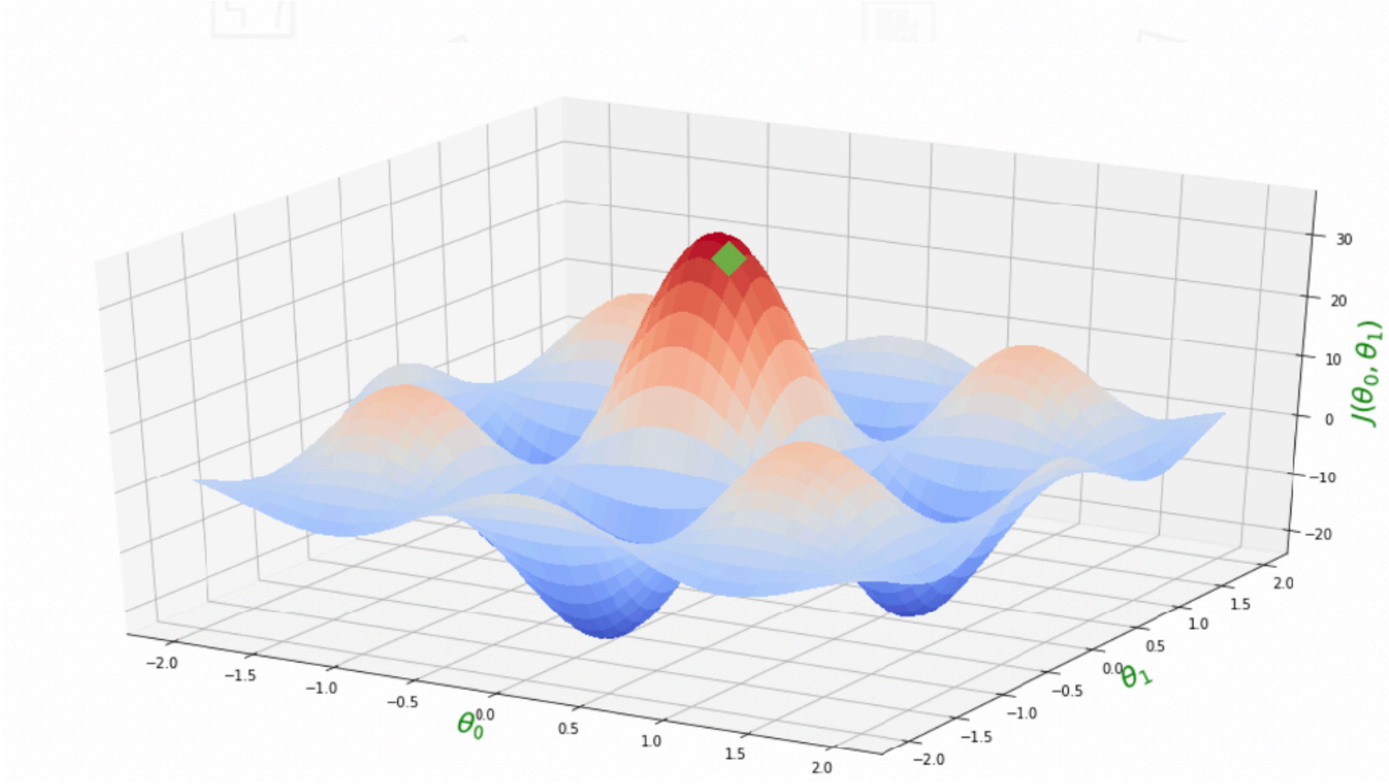
Parameters: $\theta = \{\theta_0, \theta_1\}$

Cost/Loss/Objective Function: $J(\theta) = \min_\theta \frac{1}{2N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})^2$

Methods:

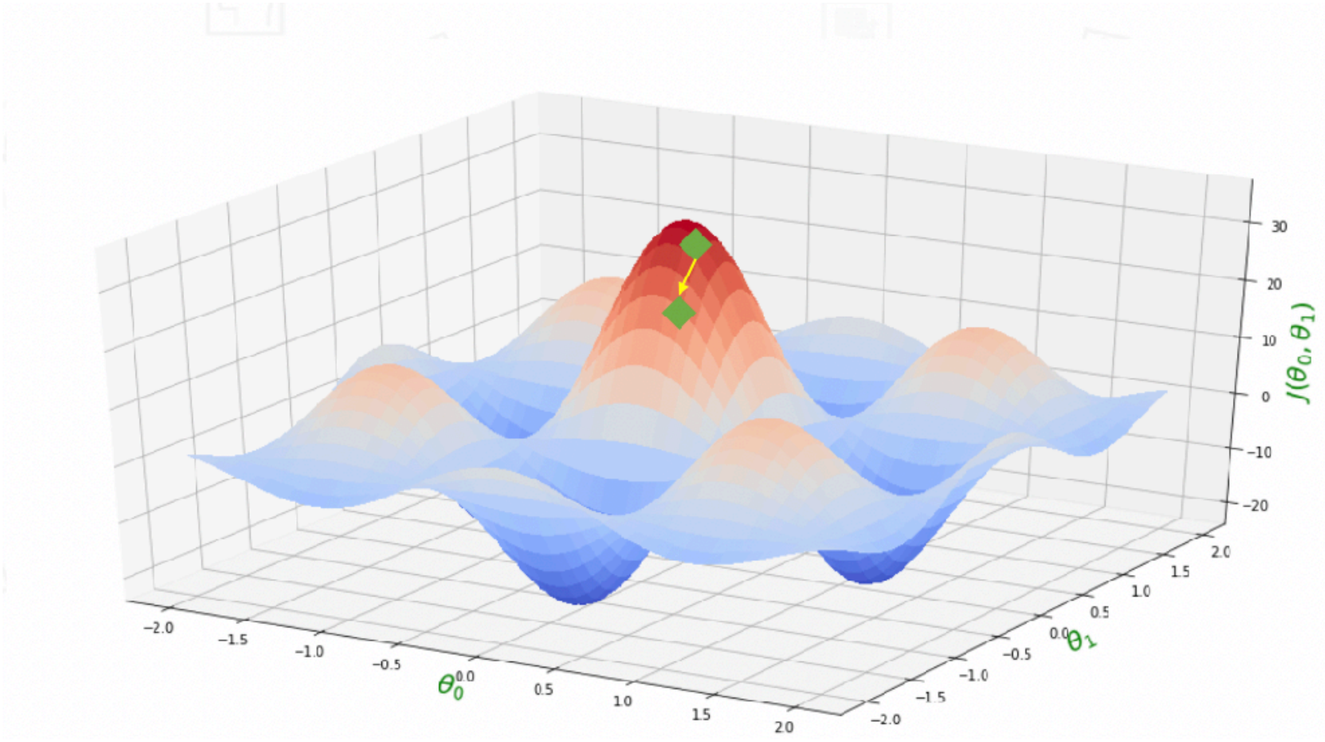
- Start with some initial guess, e.g, $\theta_0 = 0$, $\theta_1 = 0$
- Keep updating them until we reach a local optimum of the loss

Visualization



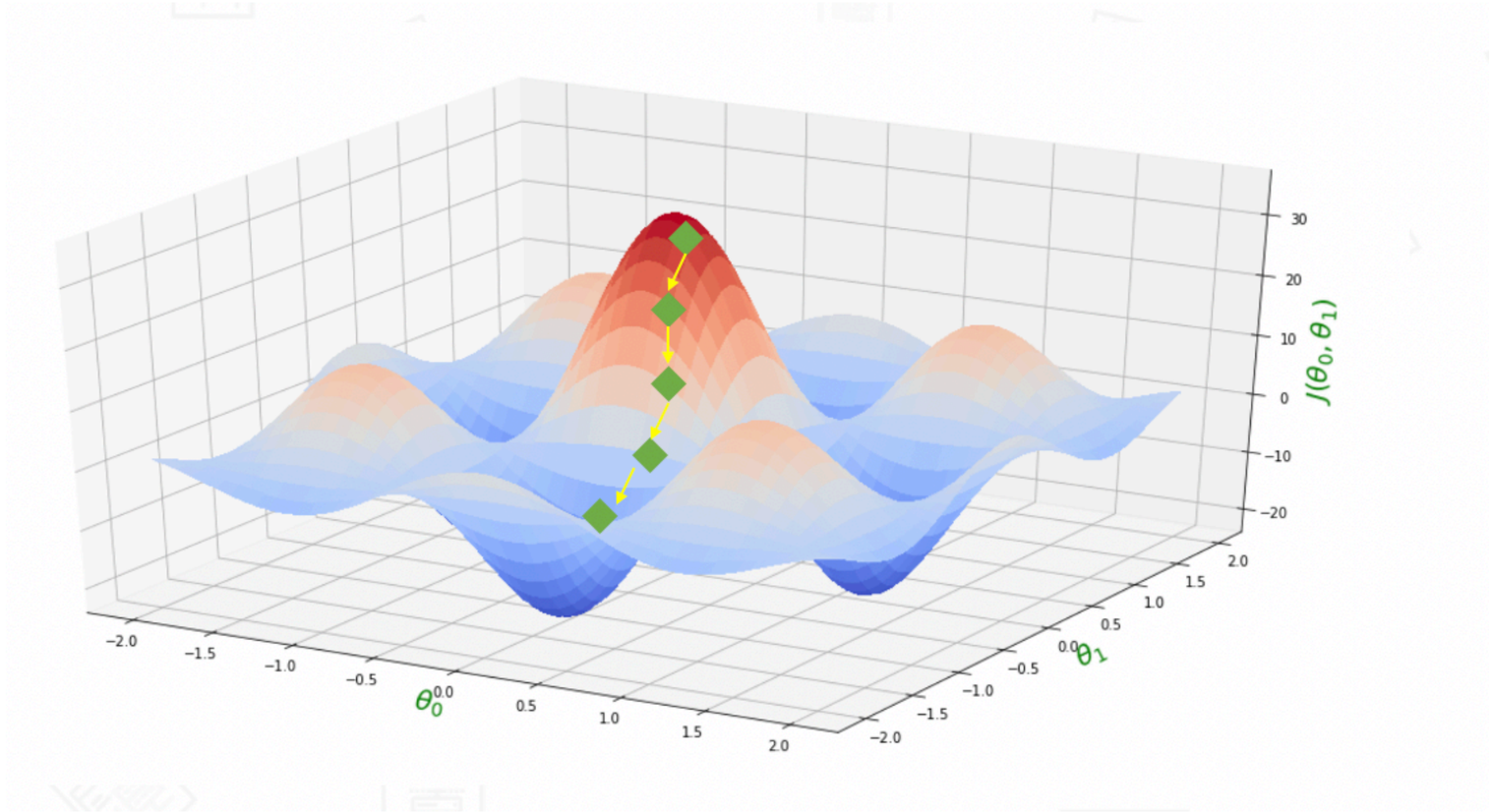
Initial guess: (0, 0)

Visualization



Small update
down the hill

Visualization



Mathematical Formulation

$$J(\theta) = \min_\theta \frac{1}{2N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})^2 \qquad \theta = \{\theta_0, \theta_1\}$$

Repeat Until Convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1); \text{ for } j = 0, 1$$

}

Gradient Descent Algorithm

Goal: $min\ J(\theta)$

Learning rate, 0.00001, 0.1, 1

Repeat Until Convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

Simultaneously update all θ_j

}

Partial derivative

Use Gradient Descent Correctly

Repeat Until Convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

}

Simultaneously update all θ_j

Correct:

$$\text{temp}_0 \leftarrow \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1);$$

$$\text{temp}_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1);$$

$$\theta_0 \leftarrow \text{temp}_0;$$

$$\theta_1 \leftarrow \text{temp}_1;$$

Wrong X:

$$\text{temp}_0 \leftarrow \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1);$$

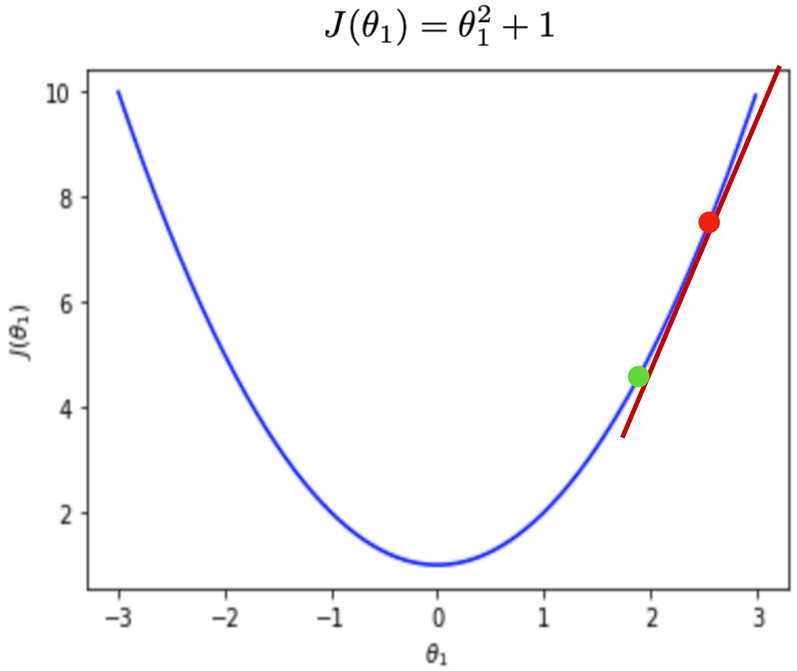
$$\theta_0 \leftarrow \text{temp}_0;$$

$$\text{temp}_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1);$$

$$\theta_1 \leftarrow \text{temp}_1;$$

Understand GD with Example

Gradients: Direction and rate of fastest increase



$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1);$$

$$\frac{\partial J(\theta_1)}{\theta_1} = 2 * \theta_1$$

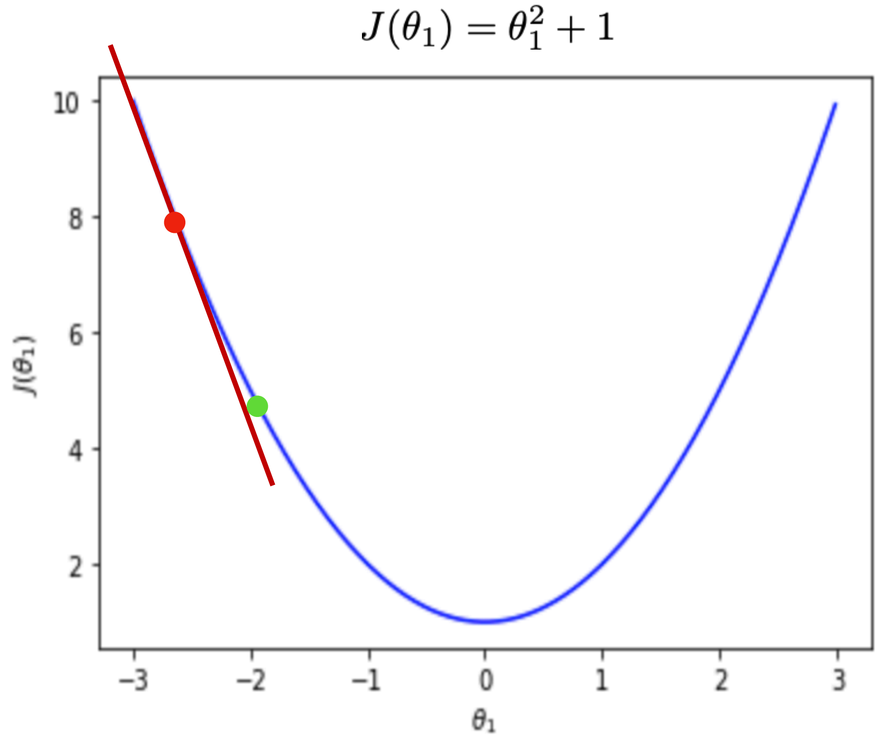
Guess: $\theta_1 = 2.5$

Set $\alpha = 0.1$, calculate new θ

$$\theta_1^{new} \leftarrow \theta_1 - \alpha * 2 * \theta_1 = 2$$

$J(\theta_1)$ will be decreased.

Understand GD with Example



$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1);$$

$$\frac{\partial J(\theta_1)}{\theta_1} = 2 * \theta_1$$

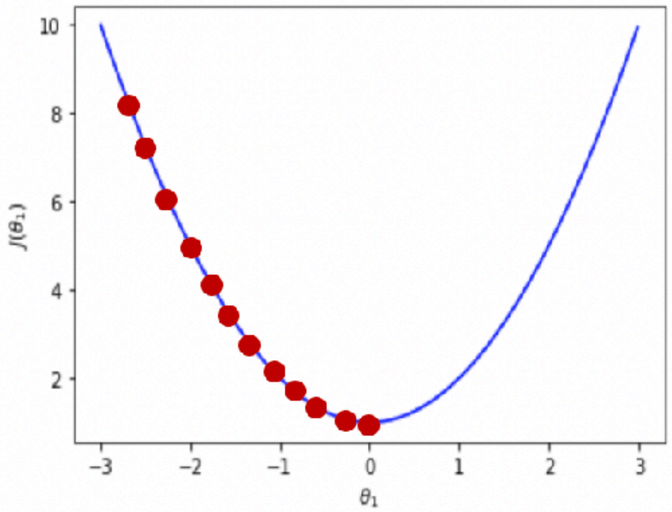
Guess: $\theta_1 = -2.5$

Set $\alpha = 0.1$, calculate new θ

$$\theta_1^{new} \leftarrow \theta_1 - \alpha * 2 * \theta_1 = -2$$

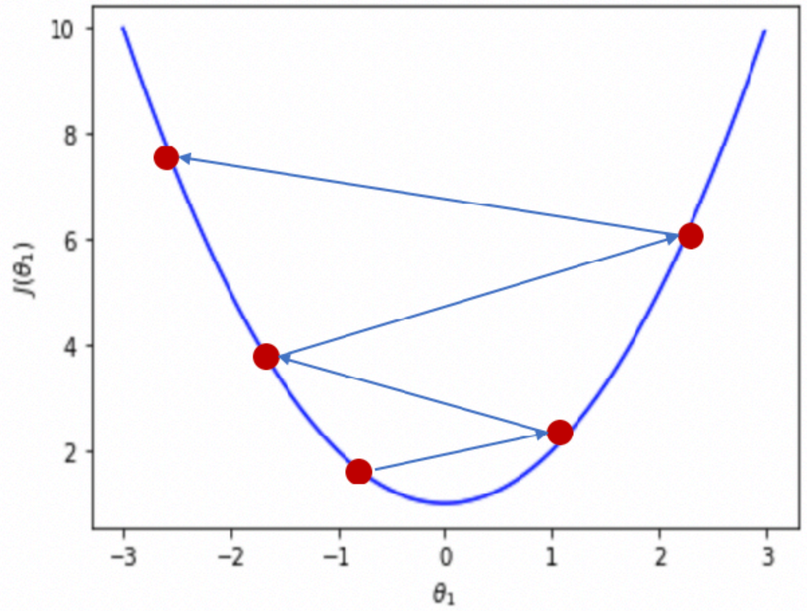
$J(\theta_1)$ will be decreased.

Discussion of GD: Learning Rate



When α is small,
the convergence will be slow

Discussion of GD: Learning Rate

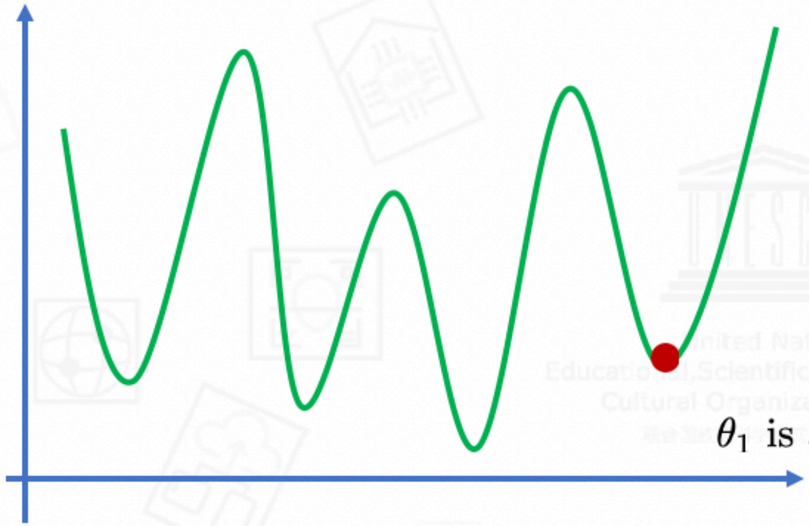


When α is large,
it will converge faster
but it may fail to converge
or even diverge

$$\alpha = \{0.1, 0.01, 0.001, 0.0001, ...\}$$

The more complex the model is ,
the smaller learning rate

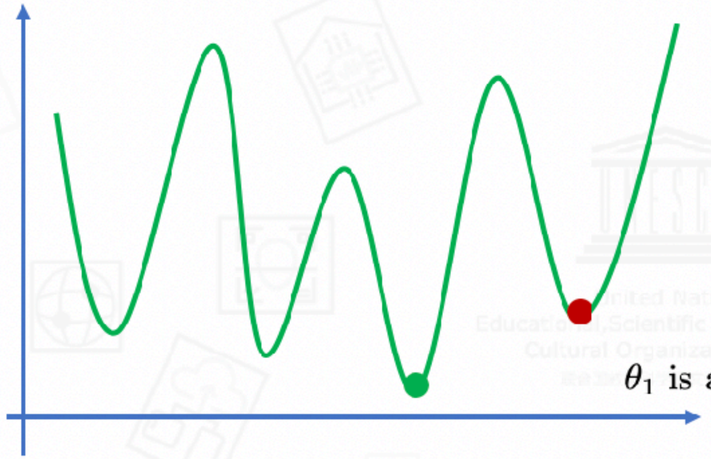
Discussion of GD: Local Minimum



$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1) = \theta_1 - \alpha * 0 = \theta_1$$

When the the cost function reaches a local minimum,
the update procedure will stop automatically as the gradients become 0

Discussion of GD: Local Minimum

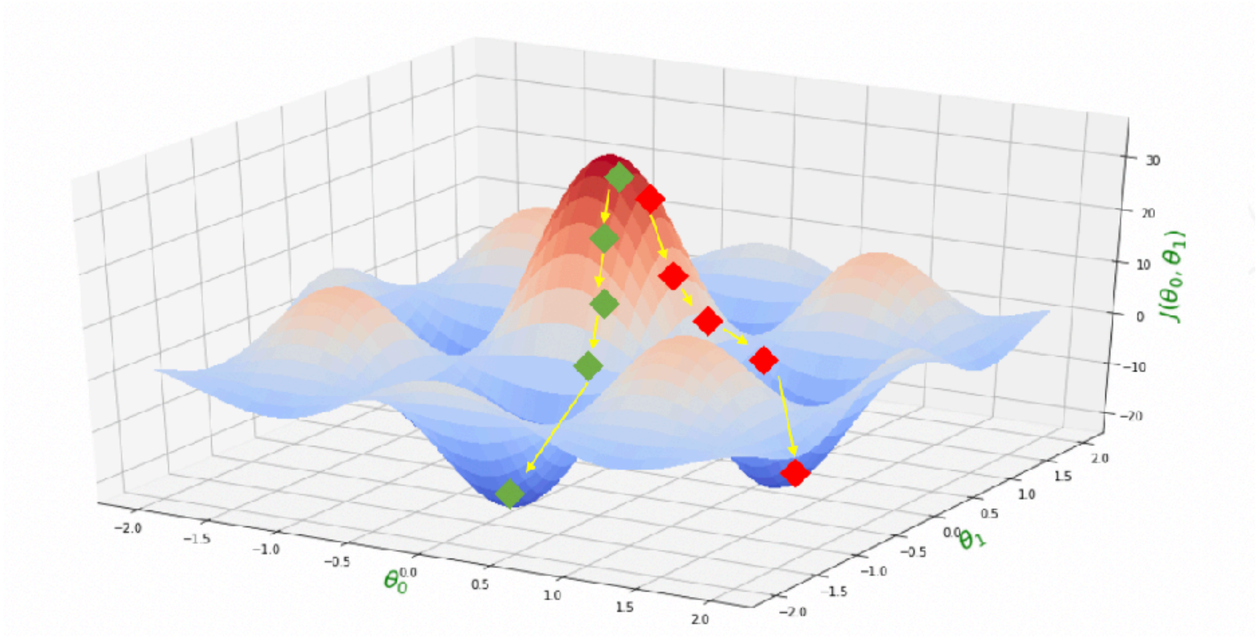


$$\theta_1 \leftarrow \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1) = \theta_1 - \alpha * 0 = \theta_1$$

The cost function might stuck at a local minimum,
and cannot reach a global minimum

Discussion of GD: Initialization

Sensitive to Initial Value



- GD is also sensitive to initialization, different initialization might lead to different solutions
- But different local minimums might give similar inference results

GD for Linear Regression

Traing Set: $D = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), (x^{(3)}, y^{(3)}), \ldots (x^{(N)}, y^{(N)})\}$

$$J(\theta) = \min_\theta \frac{1}{2N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})^2 \qquad h_\theta(x) = \theta_0 + \theta_1 x$$

Compute Gradients:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$$

Repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$$

}

GD for Logistic Regression

Final Form: $J(\theta) = \frac{1}{m}[\sum_{i=1}^{m} - y^{(i)} \log(h_\theta(x^{(i)})) - (1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))]$

Goal: min $J(\theta)$

Linear Reg: $h_\theta(x) = \theta^\top x$

Logistic Reg: $h_\theta(x) = \frac{1}{1 + e^{-\theta^\top x}}$

Repeat Until Convergence {

$$\theta_j \leftarrow \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Simultaneously update all θ_j

}

Gradients for LR

$$
\begin{aligned}
\ell(\theta) &= \log L(\theta) \\
&= \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)}))
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} \ell(\theta) &= \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\
&= \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) g(\theta^T x)(1 - g(\theta^T x)) \frac{\partial}{\partial \theta_j} \theta^T x \\
&= \left(y(1 - g(\theta^T x)) - (1 - y) g(\theta^T x) \right) x_j \\
&= (y - h_\theta(x))\, x_j
\end{aligned}
$$

Batch GD v.s. Stochastic Mini-Batch GD

Batch GD: Compute the gradient based on the whole dataset (N samples)

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$$

What if N is huge (1 million +) and each data sample is very complicated (image or text)

- Time-consuming to evaluate the batch gradient
- Memory might be overloaded

Batch GD v.s. Stochastic Mini-Batch GD

Batch GD: Compute the gradient based on the whole dataset (N samples)

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{N} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$$

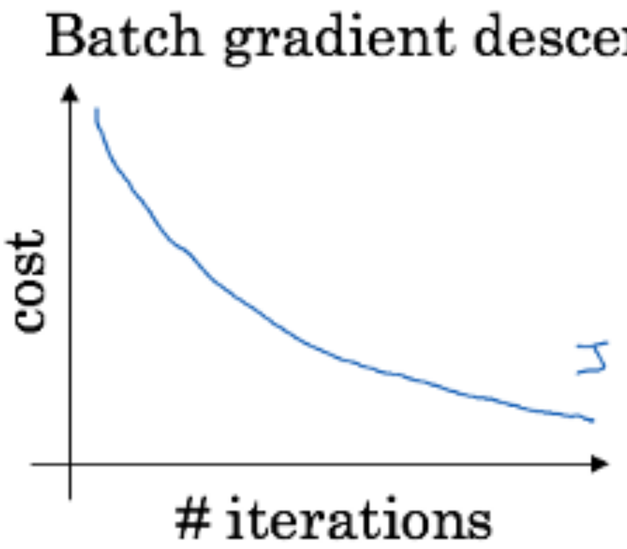
Stochastic Mini-batch GD: N = k * M, we partition N samples into k mini-batches, each with M samples, update k times for 1 epoch, and multiple epochs until convergence. E.g., for 1 epoch:

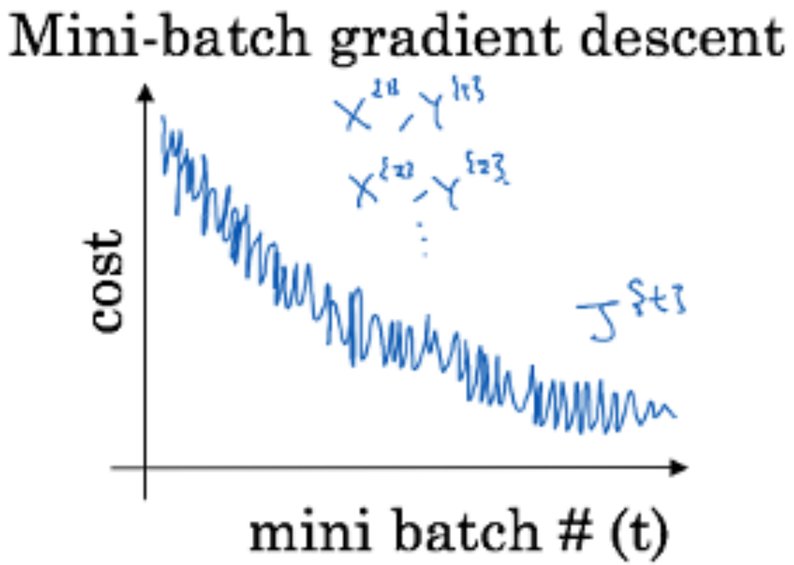
Repeat for m =1 to k until convergence {

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{M} \sum_{i=1}^{M} (h_\theta(x^{(i)}) - y^{(i)}) * x^{(i)}$$

}

Stochastic Gradient Descent





- Batch Gradient Descent: Evaluate over 1000, 000 image input, compute gradients
- Mini-Batch Gradient Descent: Evaluate over 100 images (1 batch), compute gradient, evaluate over another 100 images ….
- Learning process fluctuate, but work more efficiently in practice

Issues with SGD

Goal: $min\ J(\theta)$

Repeat Until Convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$

Simultaneously update all θ_j

}

- Need hyper-parameter search for find a good learning rate α
- Convergence is not very fast in practice or not converge to a good optimal

Advanced Gradient Descent

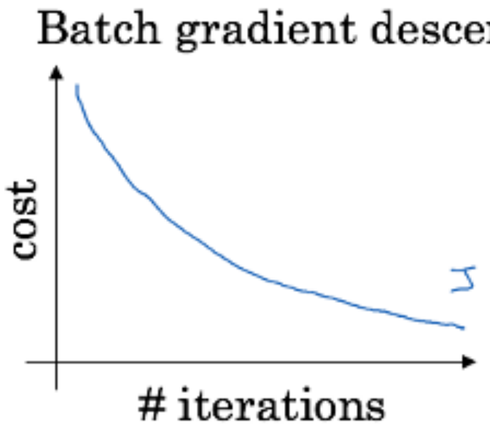
- RMSprop
- SGD+Momentum
- Adam
- …

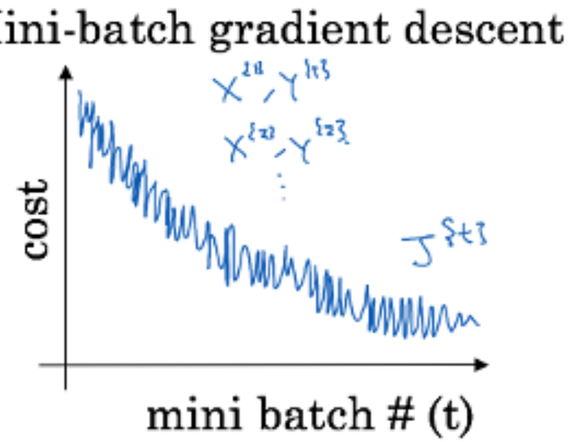
- Still need to manually choose α
- Converge very nicely in practice
- No time-consuming computation or estimation of hessian
- Used a lot in current machine learning community

Neural Networks

- Artificial Neuron
- Multilayer Perceptron
- Backpropagation
- Training a Neural Network
- Gradient Descent
- **Stochastic Gradient Descent**

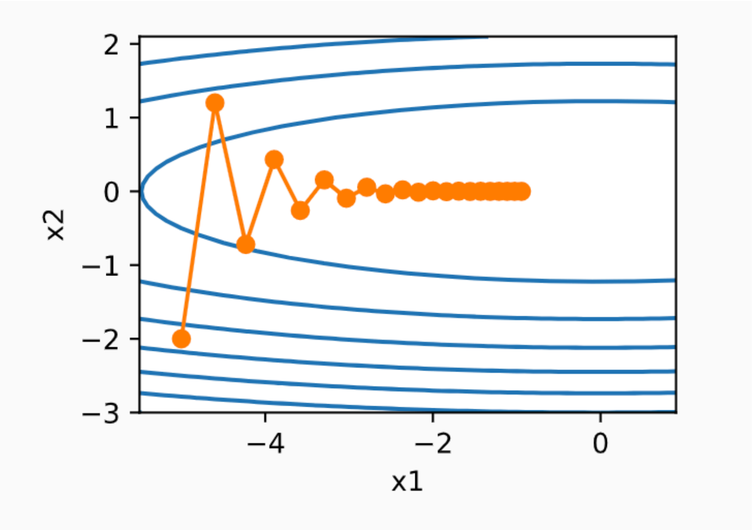
Stochastic Gradient Descent - Issues





- Choose learning rate
- Same learning rate works for all parameter updates
- Bad local optimum
- Convergence not fast

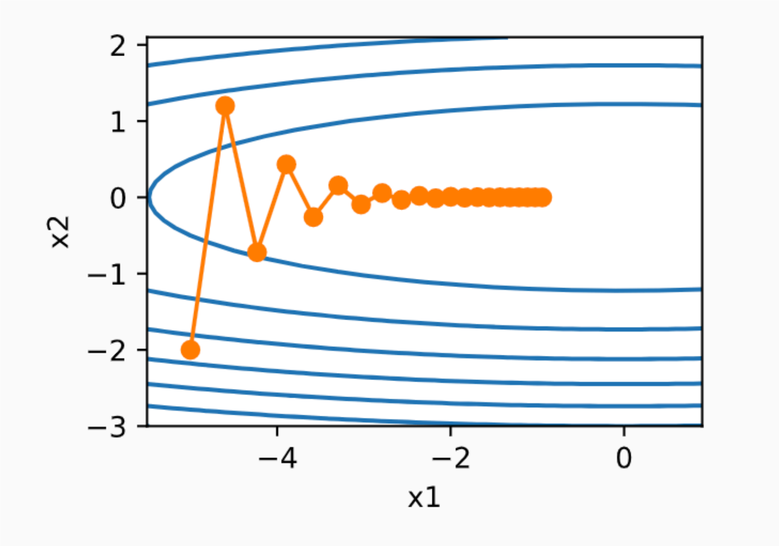
SGD Issues

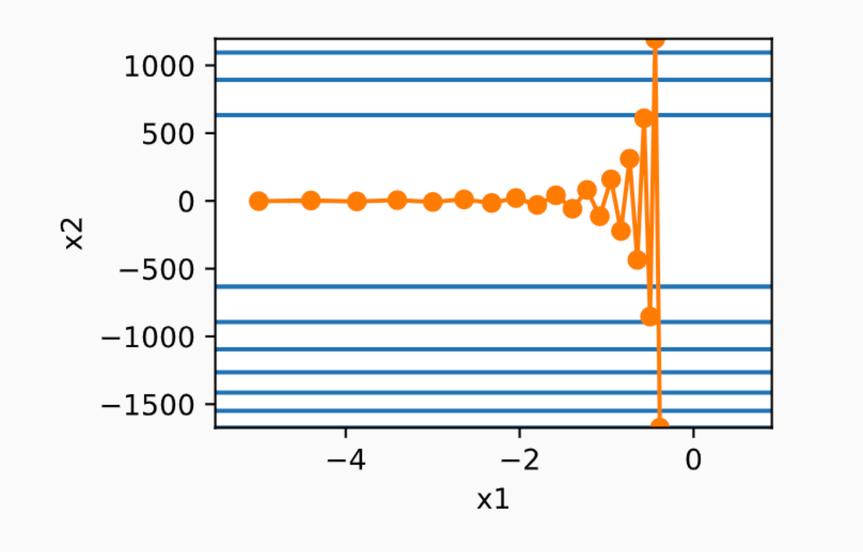


$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$$

- Gradient in x2 is much larger and changes more rapidly than x1
- **ill-conditioned optimization:** there are some directions where progress is much slower than in others

SGD Issues





- A large learning rate may cause x2 diverge

SGD with Momentum

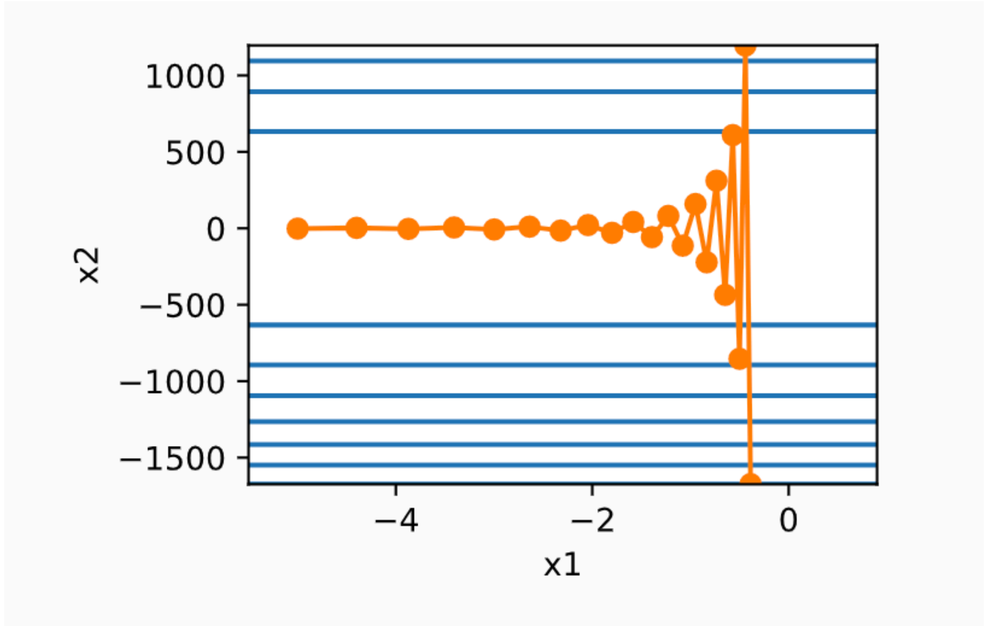
$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1},$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.$$

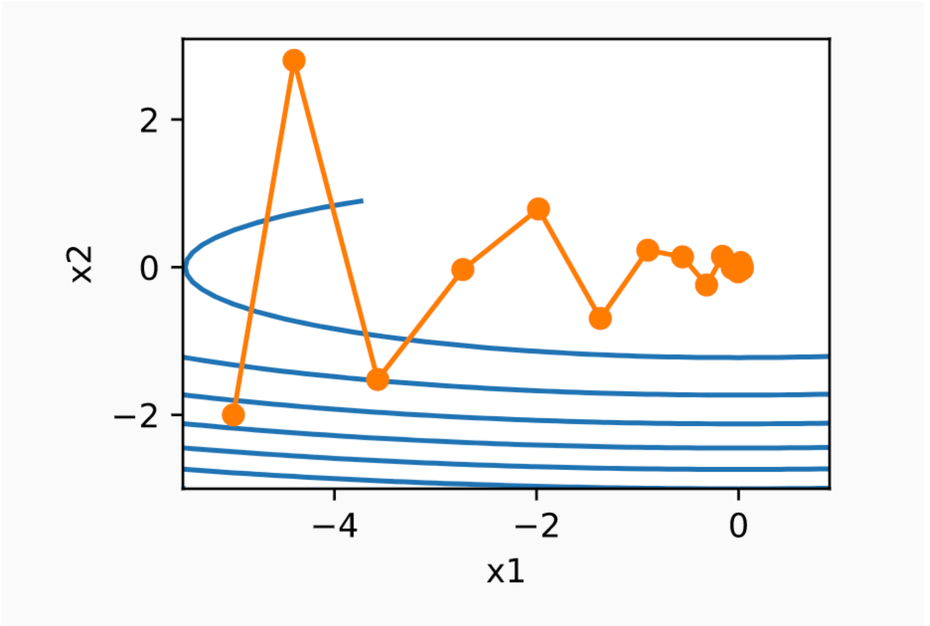
Intuition: averaging gradients over the past

$$\mathbf{v}_t = \beta^2 \mathbf{v}_{t-2} + \beta \mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \dots, = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}$$

Intuition: Exponential moving average over subsequent gradients to get more stable directions of descent

SGD with Momentum





A large learning rate may cause x2 diverge

With momentum: same large learning rate but converges very well

SGD with Momentum

$$\mathbf{v}_t \leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1},$$
$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.$$

Exponential moving average:

$$y_t = \gamma y_{t-1} + (1-\gamma) x_t.$$

$$v_t \leftarrow \gamma v_{t-1} + (1-\gamma)\left(\frac{\eta_t}{1-\gamma} \mathbf{g}_t\right)$$

$$\lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = \exp(-1) \approx 0.3679.$$

$$\begin{aligned} y_t &= (1-\gamma) x_t + \gamma y_{t-1} \\ &= (1-\gamma) x_t + (1-\gamma) \cdot \gamma x_{t-1} + \gamma^2 y_{t-2} \\ &= (1-\gamma) x_t + (1-\gamma) \cdot \gamma x_{t-1} + (1-\gamma) \cdot \gamma^2 x_{t-2} + \gamma^3 y_{t-3} \\ &\dots \end{aligned}$$

Intuition: Exponential moving average over subsequent gradients to get more stable directions of descent

AdaGrad

$$\begin{aligned}
\mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\
\mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\
\mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.
\end{aligned}$$

Intuition 1: Automatically scaling the learning rate for each parameter

Intuition 2: For sparse feature, the adjustment (accumulated denominator) will be smaller; for frequently common features, the adjustment is significantly larger

Effective optimization in areas such as computational advertising/ language where sparse features exist

AdaGrad

$$\begin{aligned}
\mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\
\mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\
\mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.
\end{aligned}$$

Issue: the learning rate will decrease too fast as we are always accumulating over all the previous gradients.

For non-convex problem such as deep learning, might cause bad convergence

RMSProp

$$\begin{aligned}
\mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\
\mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\
\mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.
\end{aligned}$$

$$\begin{aligned}
\mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2, \\
\mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.
\end{aligned}$$

$$\begin{aligned}
\mathbf{s}_t &= (1-\gamma)\mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\
&= (1-\gamma)\left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots,\right)
\end{aligned}$$

Intuition: Moving average over squared gradients, better scaling

Adam

$\beta_1 = 0.9$ and $\beta_2 = 0.999$

$$\mathbf{v}_t \leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1)\mathbf{g}_t,$$
$$\mathbf{s}_t \leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2.$$

Intuition: Exponential Moving Average of both gradients and squared gradients

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}.$$

Intuition: Renormalize to remove small bias

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}.$$

Intuition: Use epsilon to avoid 0 divisor, such as 1e-8

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'.$$

Beyond SGD

- Natural gradients
- Conjugate gradient methods
- Information geometry
- Use Hessian matrix, Fisher information etc