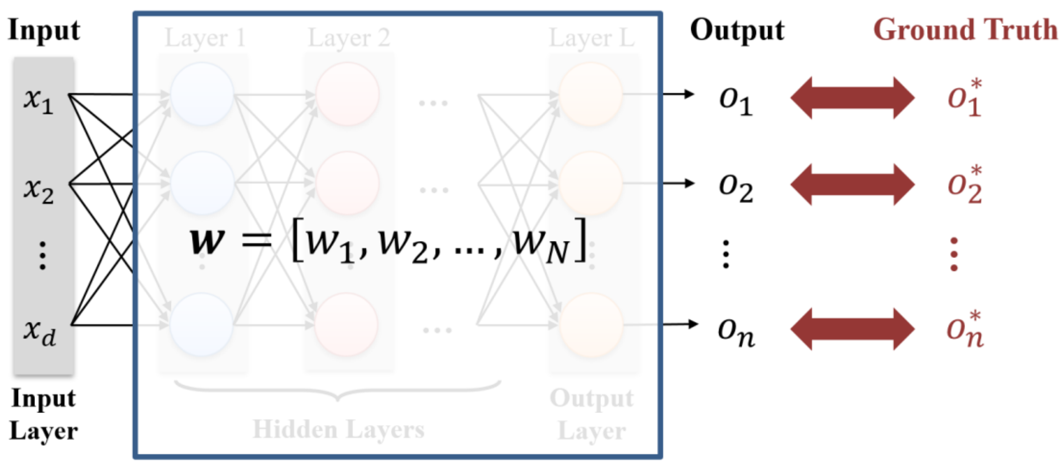
Machine Learning & Intelligence for Electrical Engineers

Neural Network Optimization

Korea University

Error/Loss Function

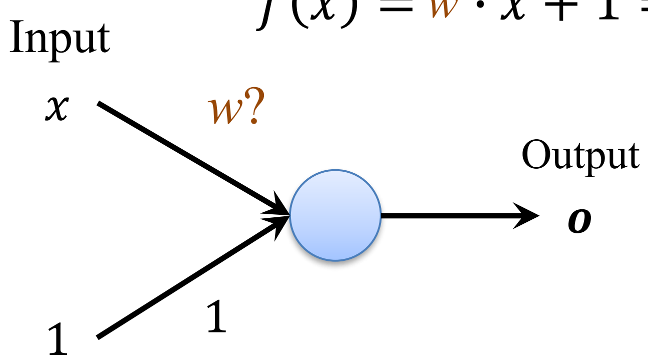
- Given a dataset $\mathcal{D}$ and a neural network f, the objective of the learning/training procedure is to *minimize* the **error/loss** function



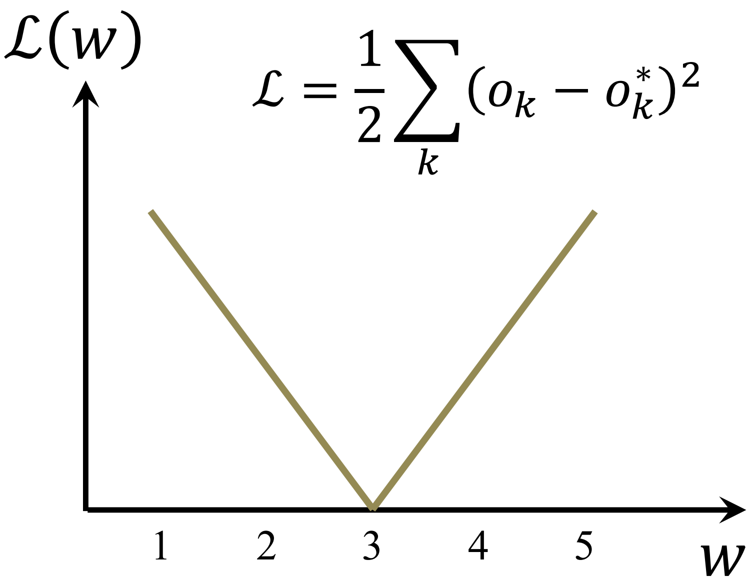
$$\boldsymbol{w}^* = \underset{\boldsymbol{w}}{\operatorname{argmin}} \mathcal{L}(\boldsymbol{x}, \boldsymbol{o}; \boldsymbol{w})$$

What is Error/Loss Function?

$$f(x) = w \cdot x + 1 = \boldsymbol{o}$$



$$\mathcal{L} = \frac{1}{2}\sum_k (o_k - o_k^*)^2$$

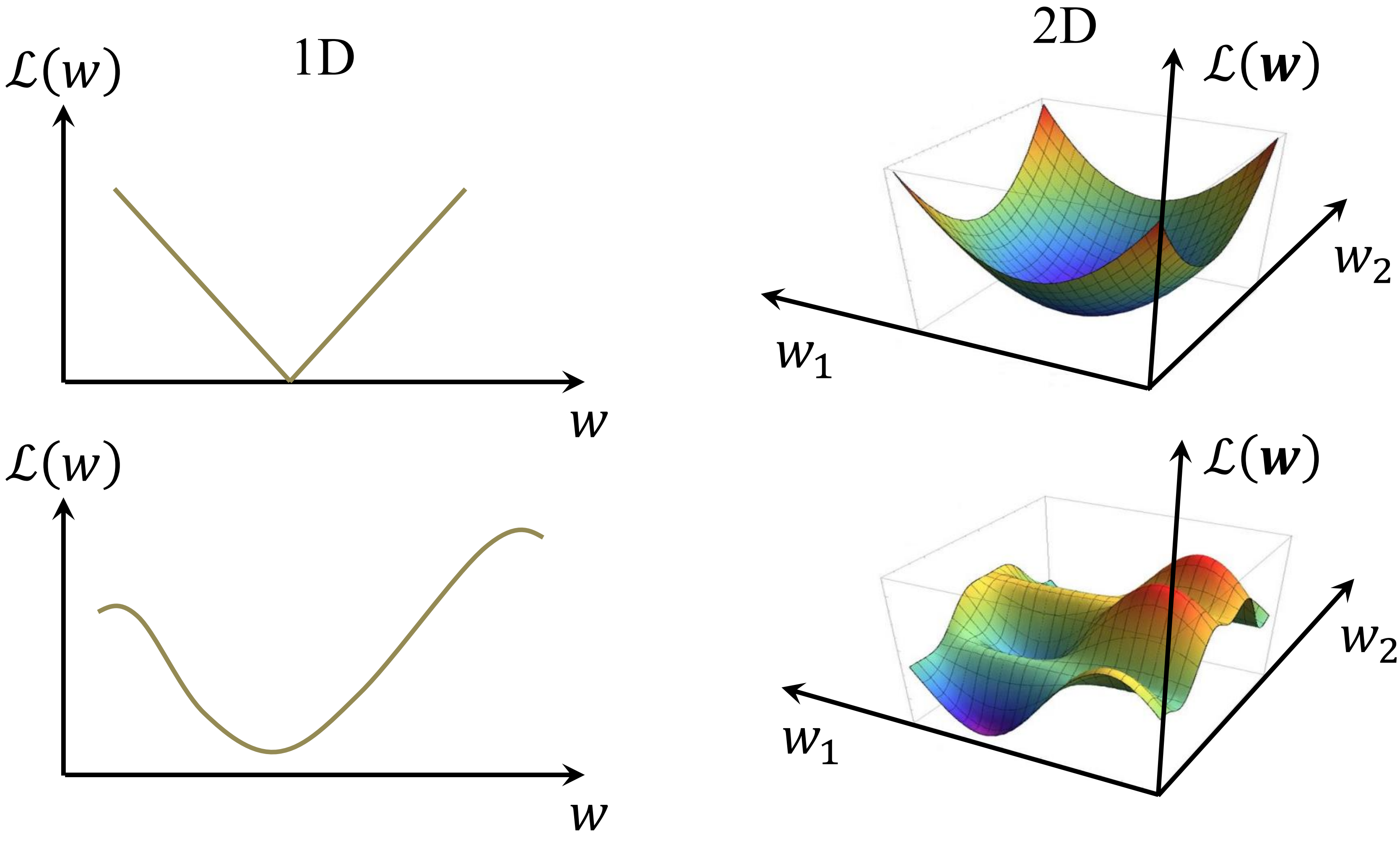


Ground Truth — Prediction

x	o^*	o				
		$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$
1	4	2	3	4	5	6
2	7	3	5	7	9	11
3	10	4	7	10	13	16
4	13	5	9	13	17	21
5	16	6	11	16	21	26

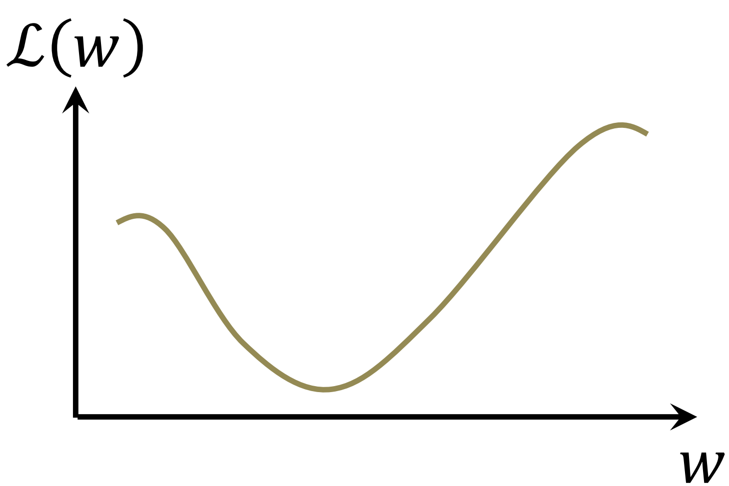
x	o^*	$Error$				
		$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$
1	4	2	0.5	0	0.5	2
2	7	8	2	0	2	8
3	10	18	4.5	0	4.5	18
4	13	32	8	0	8	32
5	16	50	12.5	0	12.5	50
MSE		110	27.5	0	27.5	110

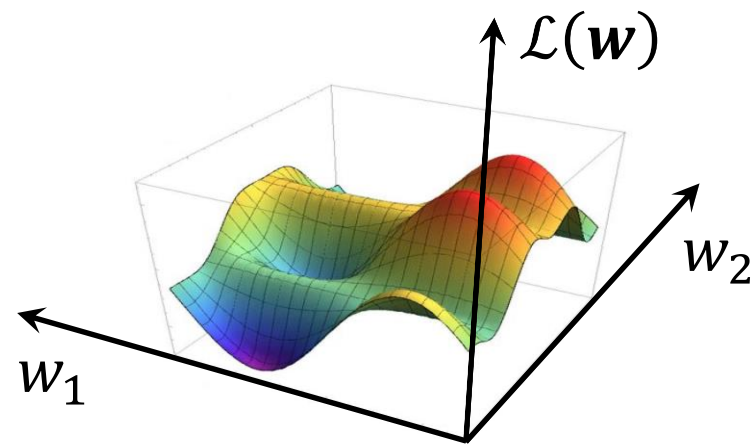
Error/Loss Landscape



Learning Neural Network

- How can we find the best values of the parameters $\boldsymbol{w}$?
  - Exhaustive search
  - Random search
  - …

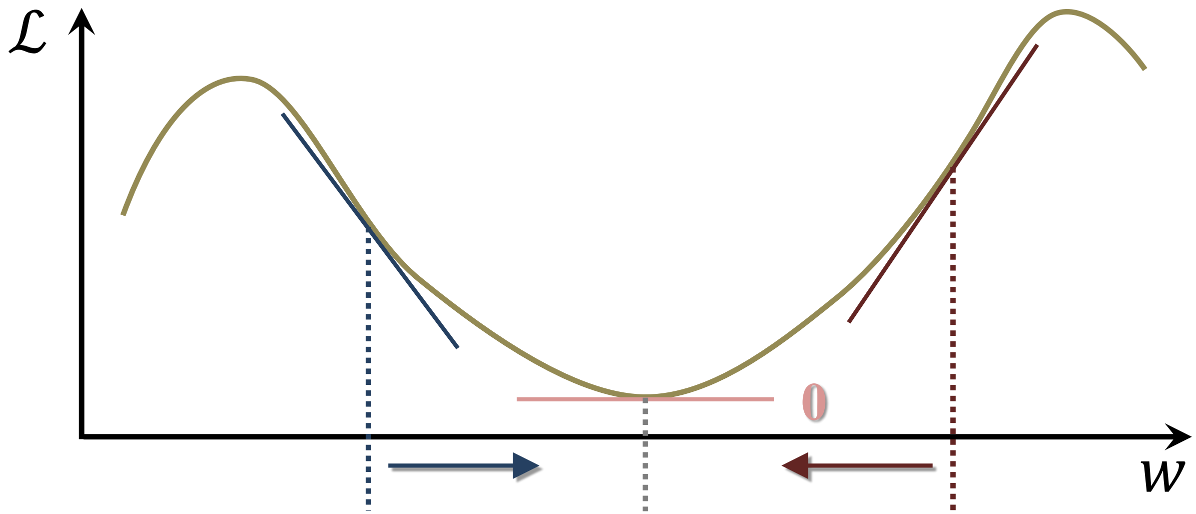




Error/Loss Function & Derivatives

- How to decrease the error function $\mathcal{L}$?
  - If $\mathcal{L}'(w) > 0$: Move to the left
  - If $\mathcal{L}'(w) < 0$: Move to the right

$$w^{new} = w^{old} - \frac{\partial \mathcal{L}}{\partial w}$$



$\frac{\partial \mathcal{L}}{\partial w} < 0$ $\frac{\partial \mathcal{L}}{\partial w} = 0$ $\frac{\partial \mathcal{L}}{\partial w} > 0$

Gradient Descent

$$\mathcal{L}(\boldsymbol{w}) = \mathcal{L}(w_1, w_2, \dots, w_d)$$

- To minimize $\mathcal{L}(\boldsymbol{w})$,
  take and use partial derivatives of $\mathcal{L}(\boldsymbol{w})$

- Gradient $\nabla\mathcal{L}(\boldsymbol{w})$ points in direction
  of steepest *increase* of $\mathcal{L}(\boldsymbol{w})$

- $-\nabla\mathcal{L}(\boldsymbol{w})$ points in direction
  of steepest *decrease* of $\mathcal{L}(\boldsymbol{w})$

Gradient

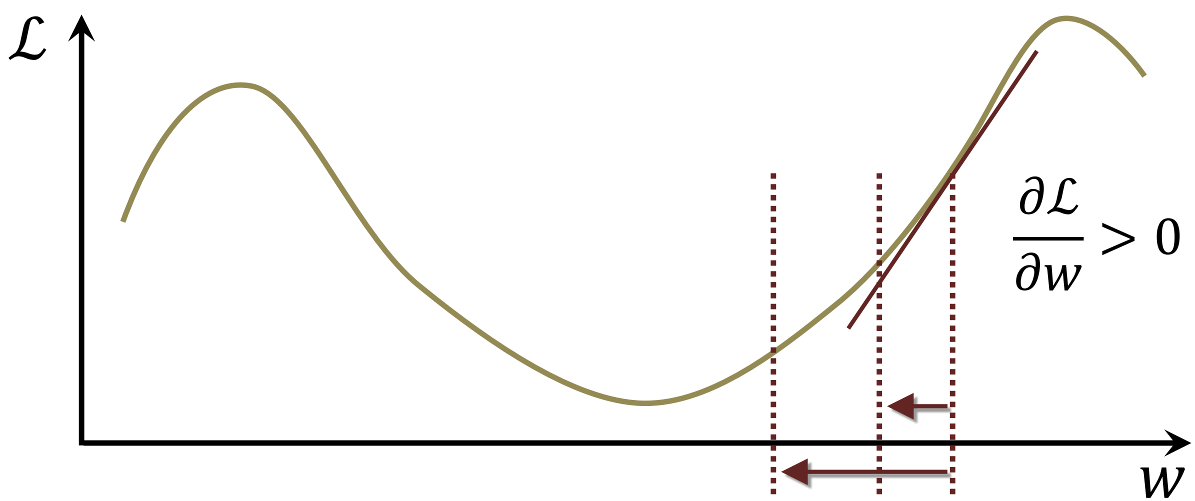
$$\nabla\mathcal{L} = \begin{bmatrix} \frac{\partial\mathcal{L}}{\partial w_1} \\ \frac{\partial\mathcal{L}}{\partial w_2} \\ \vdots \\ \frac{\partial\mathcal{L}}{\partial w_d} \end{bmatrix}$$

Gradient Descent: Update Rule

- Gradient descent update rule:

$$\boldsymbol{w}^{new} = \boldsymbol{w}^{old} - \eta \nabla \mathcal{L}(\boldsymbol{w}) \qquad w_i^{new} = w_i^{old} - \eta \frac{\partial \mathcal{L}}{\partial w_i}$$

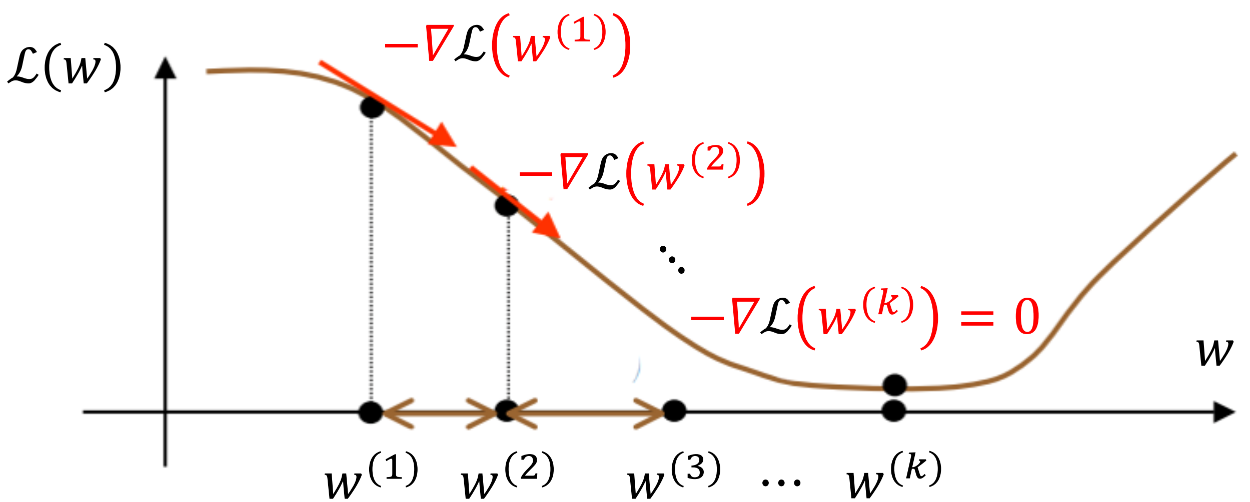
  - η (learning rate): a hyperparameter that determines the step size in adjusting the weights



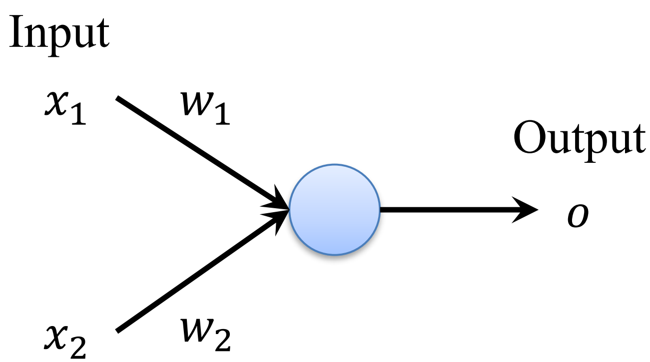
*hyperparameter : Parameter to control the ML learning process

Gradient Descent: Algorithm

- Initialization: $\boldsymbol{w}^{(0)}$ with some initial guess, $k = 0$
- While $|\nabla\mathcal{L}(\boldsymbol{w})| > \varepsilon$
  1. Choose a learning rate: $\eta^{(k)}$
  2. Update weights: $\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta^{(k)}\nabla\mathcal{L}(\boldsymbol{w})$
  3. $k = k + 1$

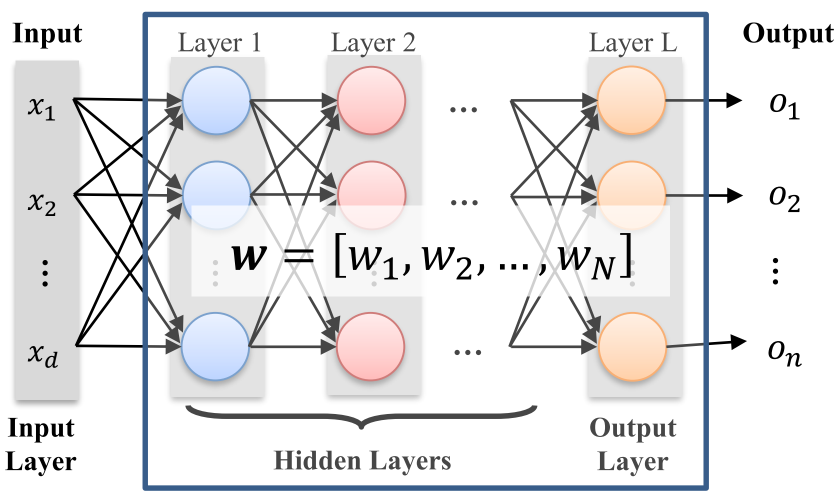


Gradient Descent: Update



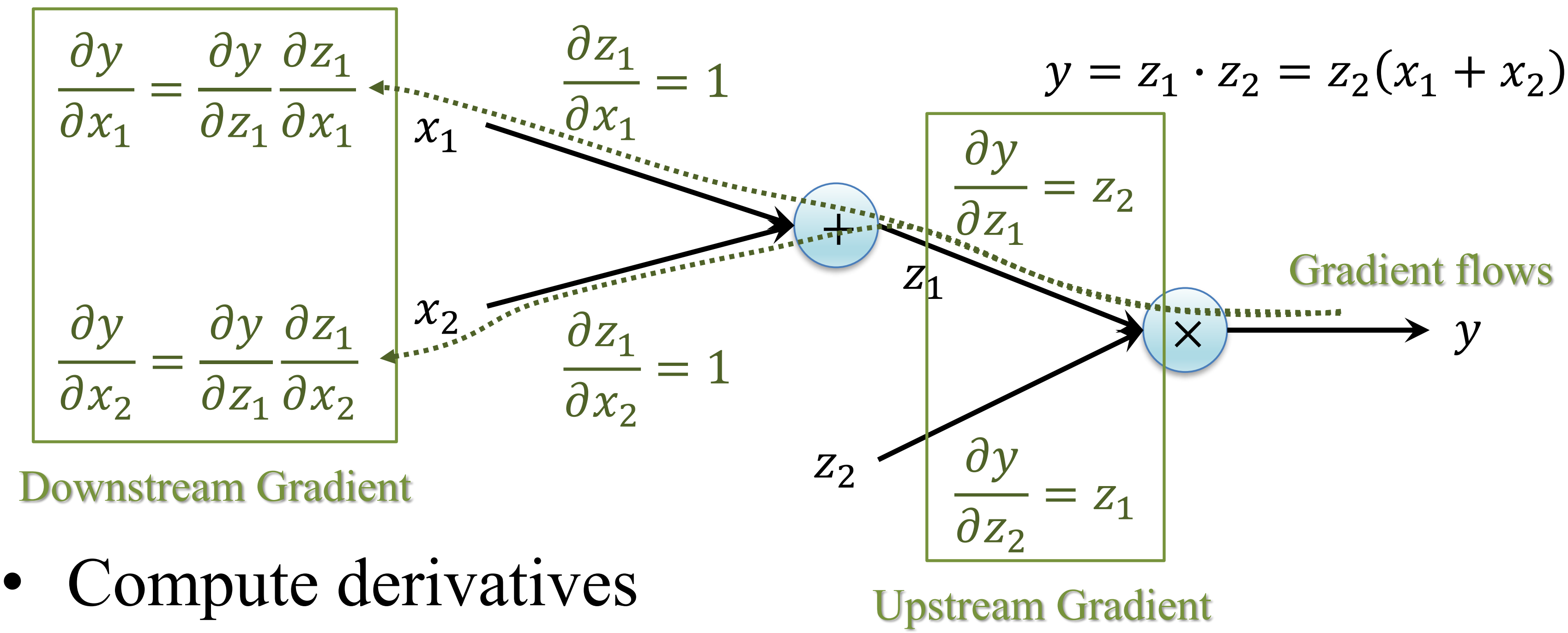
$$o = f(x) = \sum_{i=1}^{2} w_i \cdot x_i$$

$$\frac{\partial o}{\partial w_i} = \frac{\partial}{\partial w_i} f(x) = x_i$$



What about the weights in the hidden layers?

Computational Graph

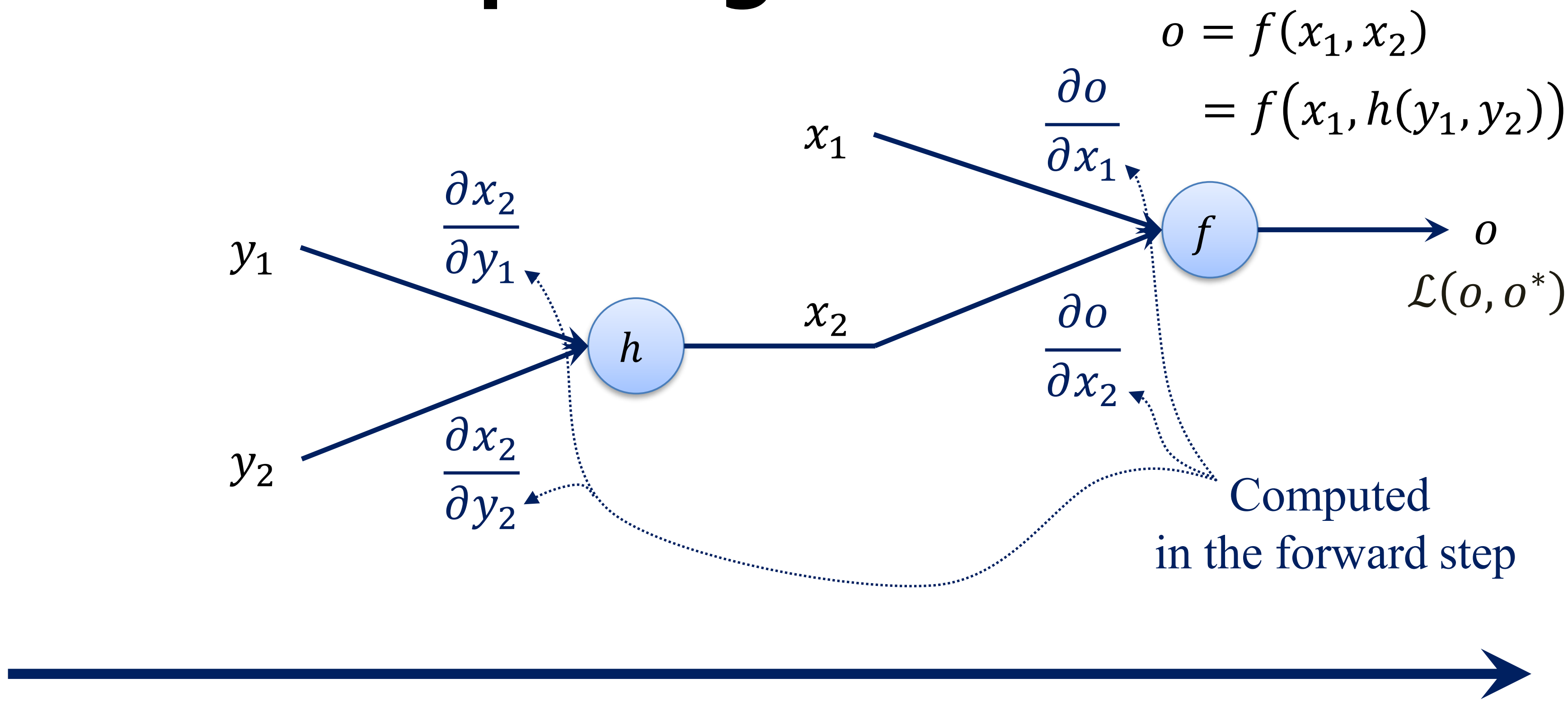


- Compute derivatives

$$\frac{\partial y}{\partial z_1} = z_2 \qquad \frac{\partial y}{\partial x_1} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial x_1} = z_2 \cdot 1$$

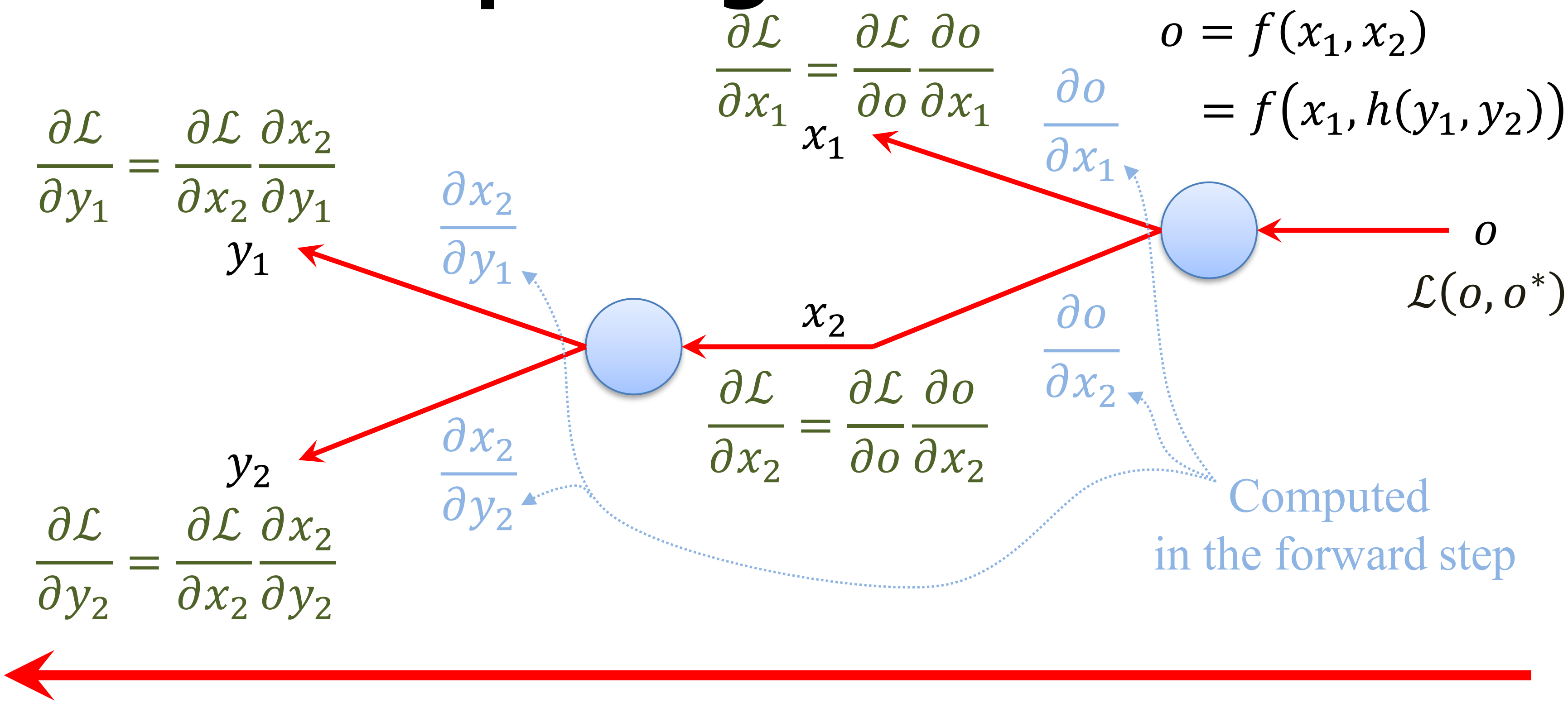
$$\frac{\partial y}{\partial z_2} = z_1 \qquad \frac{\partial y}{\partial x_2} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial x_2} = z_2 \cdot 1$$

Backpropagation: Computing Gradients



- Forward step: Produce the output and compute the loss
  Compute and save local gradients

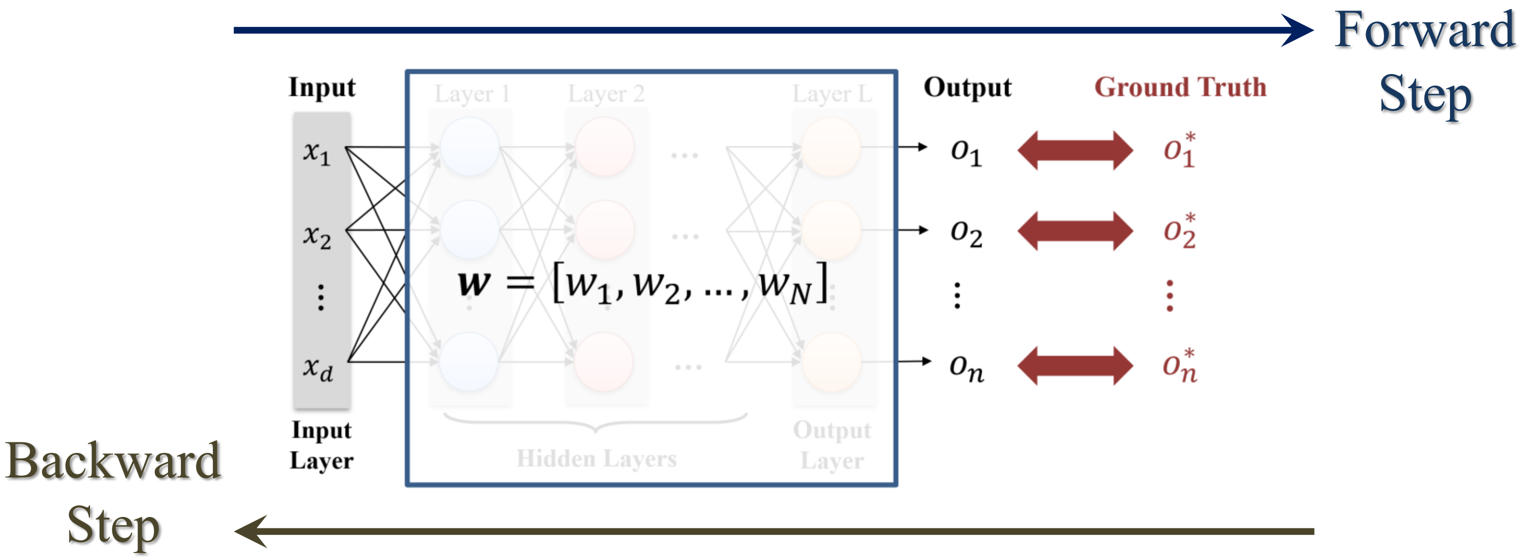
Backpropagation: Computing Gradients



- Backward step: Gradients from the last layer flows backward through the network, i.e., complete the calculation of the gradients

Training Procedure

1. Compute the actual output (prediction): $\boldsymbol{o}$
2. Compare to the desired output (ground truth): $\boldsymbol{o}^*$
3. Compute the error/loss: $\mathcal{L}(\boldsymbol{o}, \boldsymbol{o}^*; \boldsymbol{w})$
4. Determine the effect of each weight on the error/loss: $\nabla\mathcal{L}$
5. Adjust the weights $\boldsymbol{w}$ using gradient descent update rule



Network Training: Multi-Layer

$$\mathcal{L} = \frac{1}{2}\sum_k (o_k - o_k^*)^2$$

$$\frac{\partial \mathcal{L}}{\partial w_{kj}} = \frac{\partial \mathcal{L}}{\partial o_k}\frac{\partial o_k}{\partial z_k}\frac{\partial z_k}{\partial w_{kj}} = \delta_k^z h_j$$

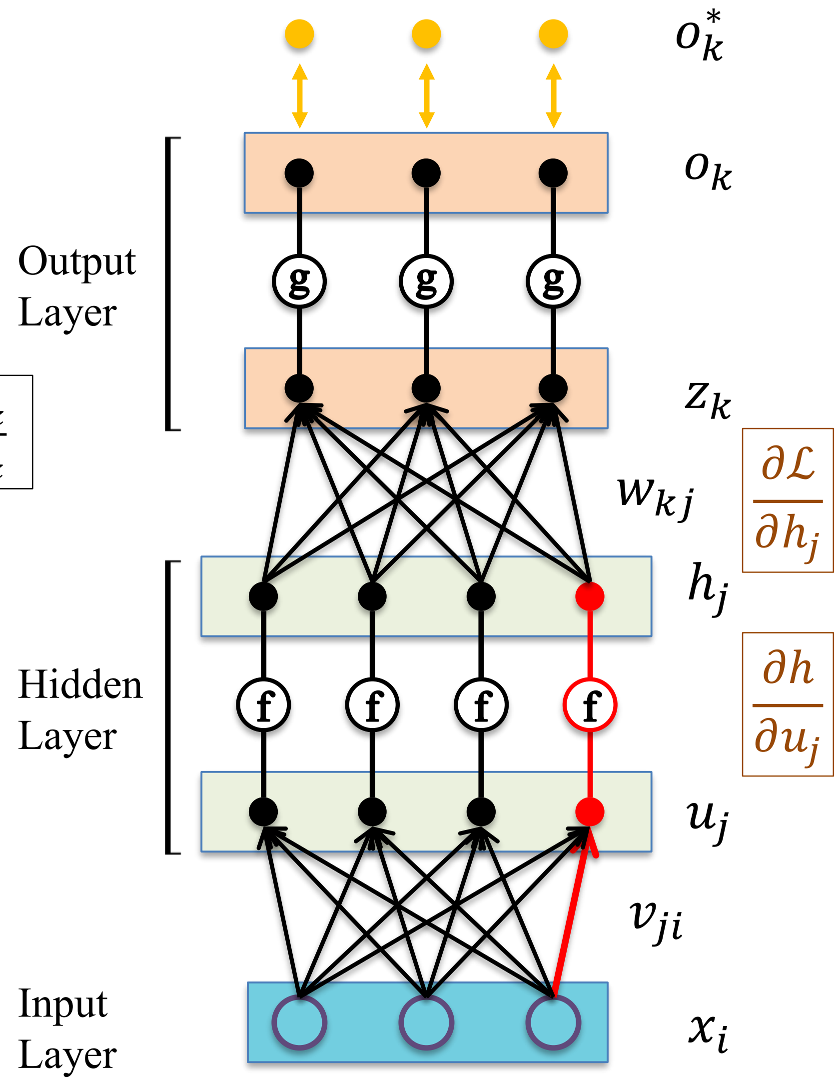
$$\boxed{h_j = \frac{\partial z_k}{\partial w_{kj}}}$$

$$\boxed{\delta_k^z = \frac{\partial \mathcal{L}}{\partial o_k}\frac{\partial o_k}{\partial z_k}}$$

Gradients on **hidden weights**:

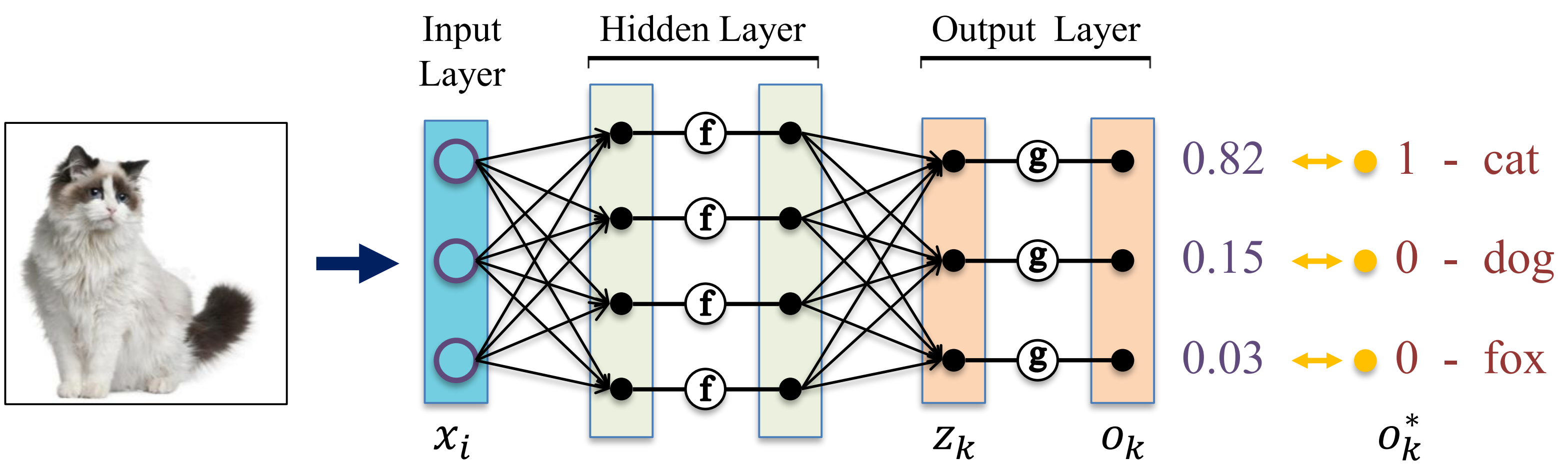
$$\frac{\partial \mathcal{L}}{\partial v_{ji}} = \frac{\partial \mathcal{L}}{\partial h_j}\frac{\partial h_j}{\partial v_{ji}} = \frac{\partial \mathcal{L}}{\partial h_j}\frac{\partial h_j}{\partial u_j}\frac{\partial u_j}{\partial v_{ji}} = \frac{\partial \mathcal{L}}{\partial h_j}\frac{\partial h_j}{\partial u_j} x_i$$

$$\boxed{x_i = \frac{\partial u_j}{\partial v_{ji}}}$$



Softmax Activation

- Represent the outputs of the network as probabilities that the input belongs to each class



$$o_k = \frac{e^{z_k}}{\sum_i e^{z_i}} = p(y = k \mid \boldsymbol{x})$$

$$\boldsymbol{z} = [z_1 \quad z_2 \quad z_3] = [2.8 \quad 1.1 \quad -0.5]$$

$$e^{\boldsymbol{z}} = [e^{z_1} \quad e^{z_2} \quad e^{z_3}] = [e^{2.8} \quad e^{1.1} \quad e^{-0.5}]$$

$$\boldsymbol{o} = [o_1 \quad o_2 \quad o_3] = [0.82 \quad 0.15 \quad 0.03]$$

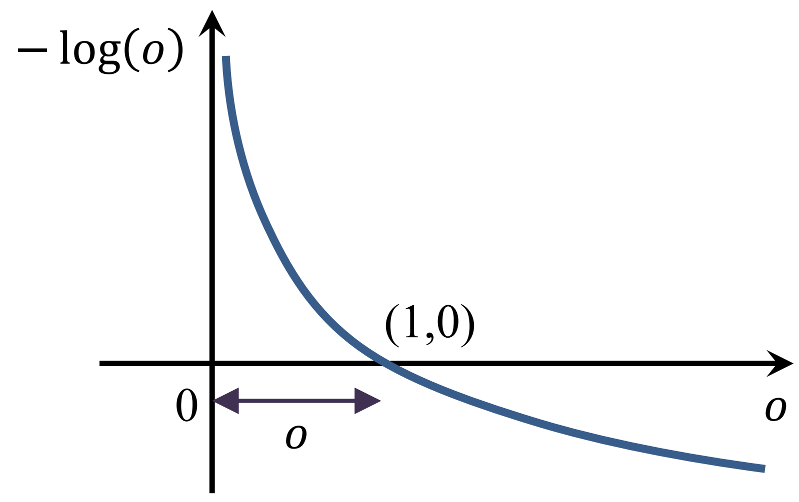
Cross-Entropy Loss

- Cross-entropy loss is one of the most popular loss functions for classification problems
  - Binary classification: $o^* = 1 \; or \; 0$

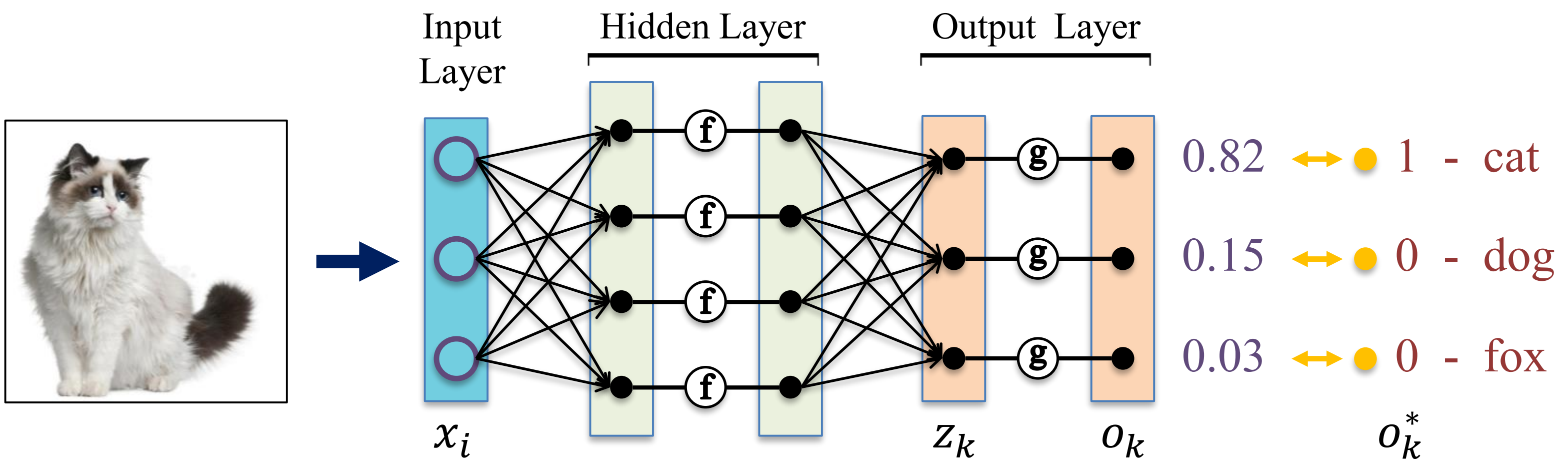
$$\mathcal{L} = -\sum_{n=1}[o^{n*} \log o^n + (1 - o^{n*}) \log(1 - o^n)]$$

  - Multi-class classification

$$\mathcal{L} = -\sum_{n}\sum_{k} o_k^{n*} \log o_k^n$$



Cross-Entropy Loss: Example



$$\mathcal{L} = -\sum_n \sum_k o_k^{n*} \log o_k^n$$

$$\boldsymbol{z} = [z_1 \quad z_2 \quad z_3] = [2.8 \quad 1.1 \quad -0.5]$$

$$\boldsymbol{o} = [o_1 \quad o_2 \quad o_3] = [0.82 \quad 0.15 \quad 0.03]$$

$$\mathcal{L} = -(1 \cdot \log 0.82 + 0 \cdot \log 0.15 + 0 \cdot \log 0.03) = 0.1985$$

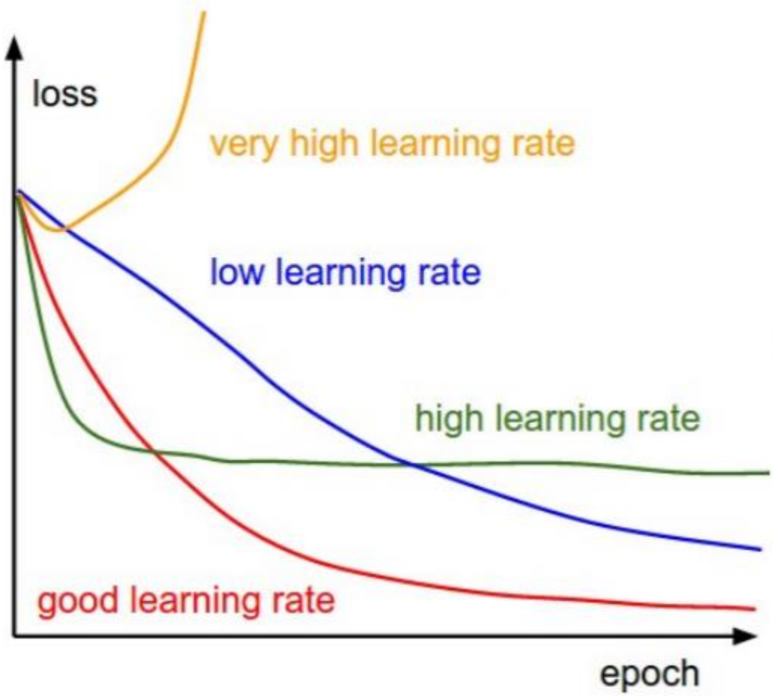
$$\boldsymbol{z} = [z_1 \quad z_2 \quad z_3] = [-0.5 \quad 1.1 \quad 2.8]$$

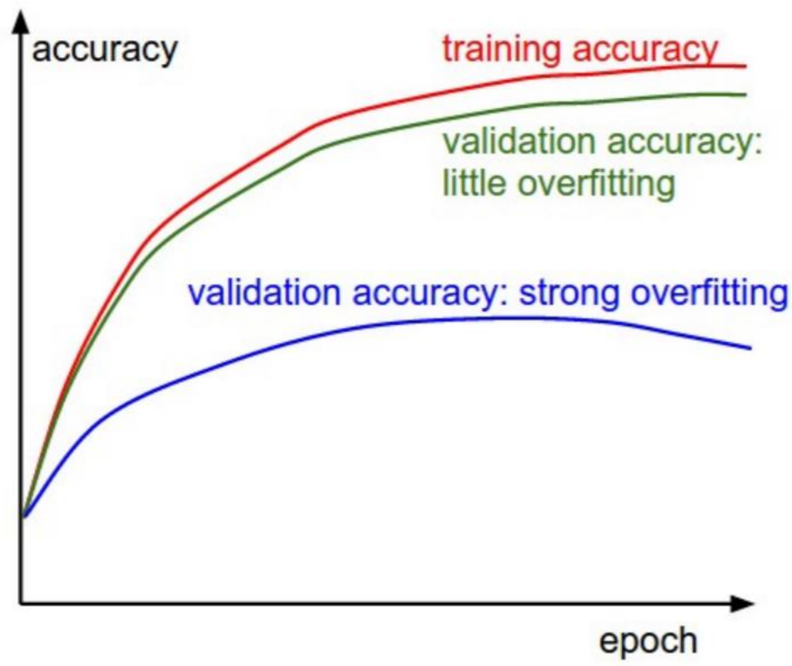
$$\boldsymbol{o} = [o_1 \quad o_2 \quad o_3] = [0.03 \quad 0.15 \quad 0.82]$$

$$\mathcal{L} = -(1 \cdot \log 0.03 + 0 \cdot \log 0.15 + 0 \cdot \log 0.82) = 3.5066$$

Optimization Algorithm

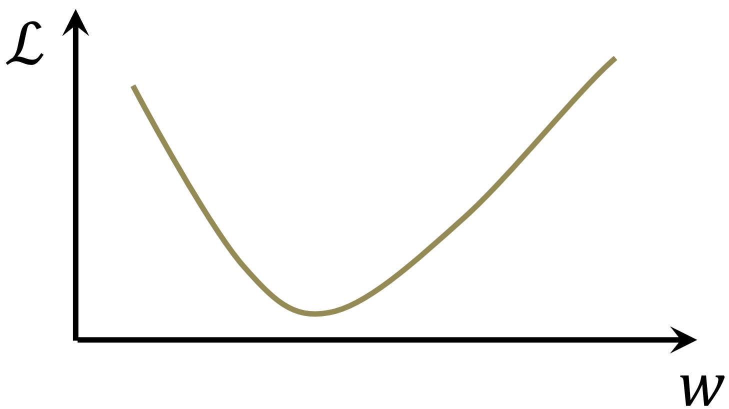
Training Procedure: Monitor Error (Loss)

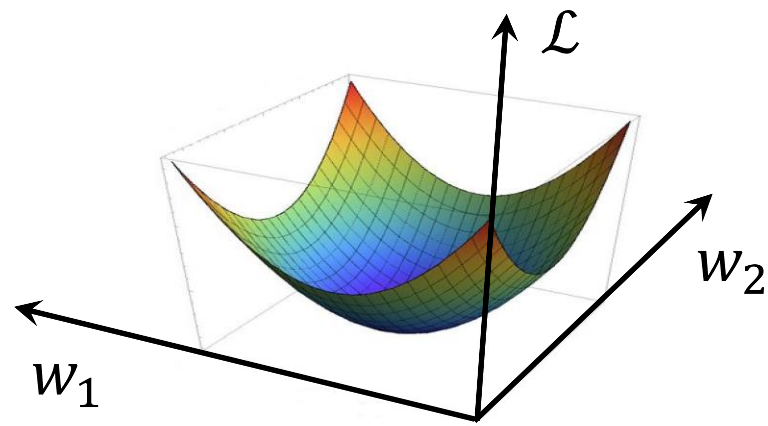




Error/Loss Landscape

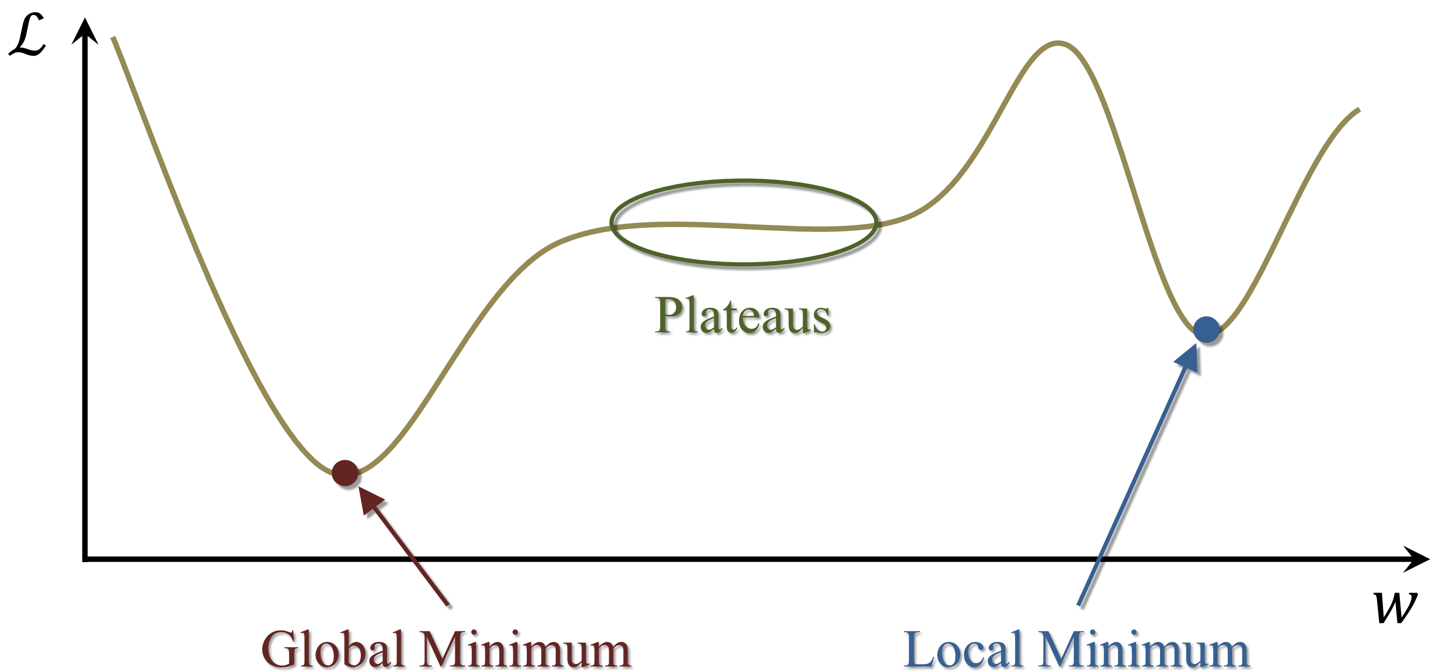
- Nice & smooth error/loss functions





- Are the actual error/loss functions are usually nice?

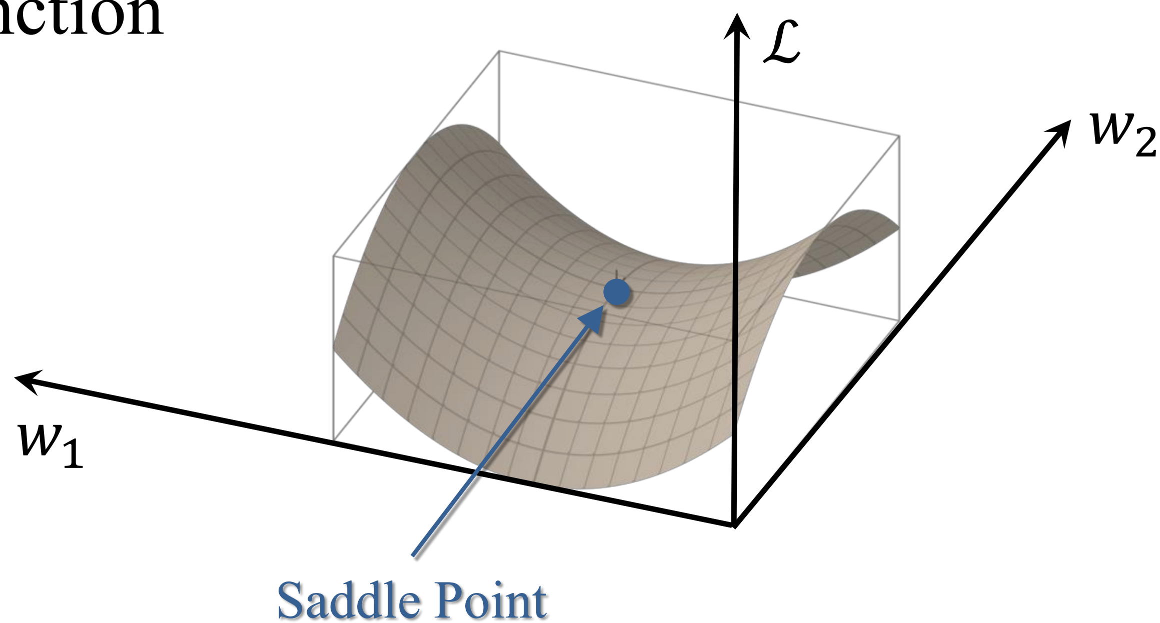
Error/Loss Landscape



- At plateau, training can be too slow
- In deep learning (lots of parameters), local minimum tends to be not much worse than the global minimum

Error/Loss Landscape

- Saddle Point
  - A point where the slopes (derivates) in orthogonal directions are all zero, but not a local extreme of a function



Gradient Descent: Large Dataset

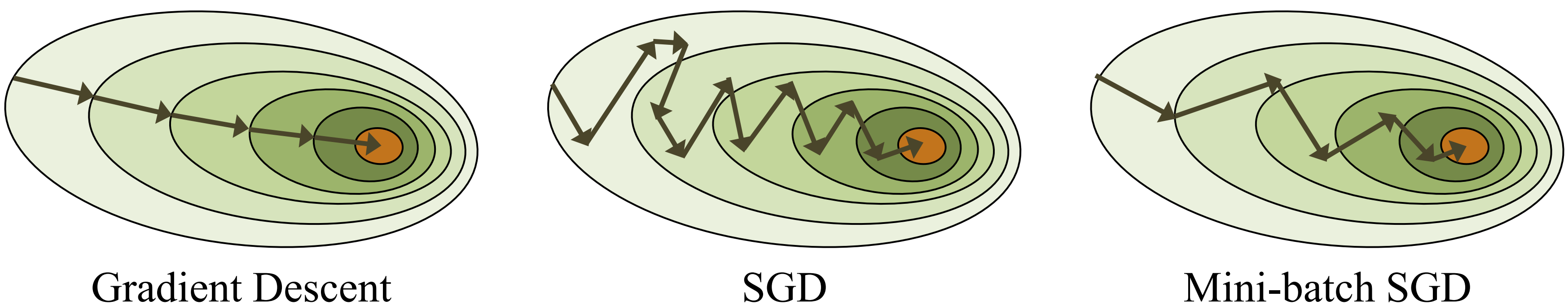
- Gradient descent
  - Use the entire training data to compute the error and update parameters

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}^i\left(x^i, y^i, \boldsymbol{w}\right)$$

  - When N is large, the full sum become too expensive and slow!

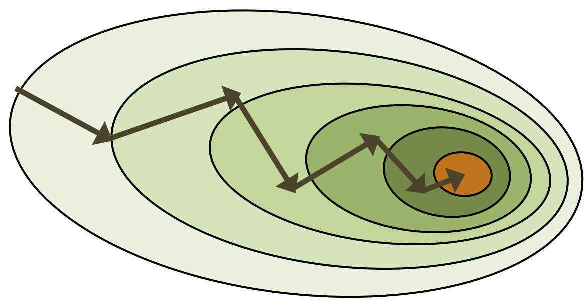
Stochastic Gradient Descent

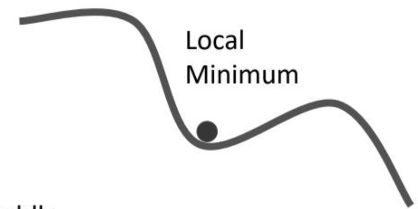
- Stochastic gradient descent (SGD)
  - Select one (or a subset) of the training data at *random* to compute the error and update parameters
  - Mini-batch SGD: Select and use a subset of the training data

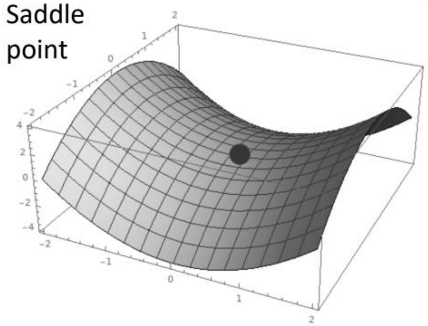


Gradient Descent (SGD): Issues

- Gradient Descent (SGD) “almost surely” converges to a global/local minimum



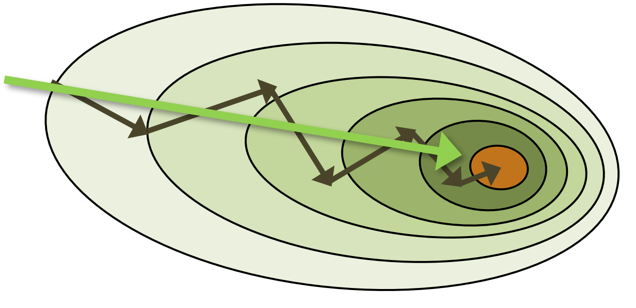




- However,
  - It uses a gradient, estimated on a simple or small batch
  - The estimated gradient may be noisy and oscillate substantially
  - It can be too slow at plateaus and get stuck at saddle points

Gradient Descent with Momentum

- Continue move in the general direction as the previous iterations
  - Maintain a running average of all gradients until the current step



Gradient Descent with Momentum

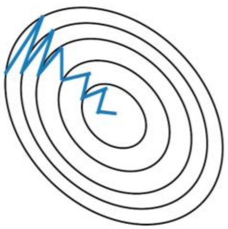
- Gradient descent:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta^{(k)} \nabla \mathcal{L}(\boldsymbol{w})$$

- Gradient descent with momentum:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \nabla \boldsymbol{w}^{(k+1)}$$

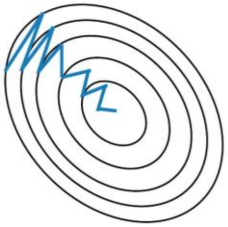
$$\nabla \boldsymbol{w}^{(k+1)} = \beta \nabla \boldsymbol{w}^{(k)} + \eta^{k} \nabla \mathcal{L}(\boldsymbol{w})$$



– $\nabla \boldsymbol{w}^{(k+1)}$ is called momentum

Gradient Descent with Momentum

- Gradient descent with momentum:



$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \nabla\boldsymbol{w}^{(k+1)}$$

$$\nabla\boldsymbol{w}^{(k+1)} = \beta\nabla\boldsymbol{w}^{(k)} + \eta^{k}\nabla\mathcal{L}(\boldsymbol{w})$$

  - $\nabla\boldsymbol{w}^{(k+1)}$ is called momentum
    - Accumulate the gradients from the past
    - Similar to the momentum of a ball rolling down the hill
  - β is a momentum coefficient, typically set to 0.9
  - Update the weights in the direction of the weighted average of the past gradients

Adaptive Optimizers

- Gradient descent algorithms and learning rate schedulers thus far, by and large, rely on manual selection of the associated hyperparameters

- Adaptive gradient descent algorithms or adaptive learning rate methods seek to adaptively adjust learning rates (or gradients) during training

Root Mean Square Propagation (RMSprop)

- Root Mean Square Propagation (RMSprop) maintains a running average of the mean squared gradients and
  rescales the learning rates using an inverse of the root mean squared gradients

Root Mean Square Propagation (RMSprop)

- Update the weights

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \widehat{\boldsymbol{V}}^{(k+1)}$$

- Scale the weighted values

$$\widehat{\boldsymbol{V}}^{(k+1)} = \frac{\eta^{(k)}}{\sqrt{\boldsymbol{V}^{(k+1)}} + \epsilon} \nabla\mathcal{L}(\boldsymbol{w})$$

- Compute a weighted average of the past squared gradients

$$\boldsymbol{V}^{(k+1)} = \alpha\boldsymbol{V}^{(k)} + (1-\alpha)\big(\nabla\mathcal{L}(\boldsymbol{w})\big)^2$$

Gradient descent with momentum:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \nabla\boldsymbol{w}^{(k+1)}$$

$$\nabla\boldsymbol{w}^{(k+1)} = \beta\nabla\boldsymbol{w}^{(k)} + \eta^{k}\nabla\mathcal{L}(\boldsymbol{w})$$

Default $\alpha = 0.9$

Root Mean Square Propagation (RMSprop)

- RMSprop:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \widehat{\boldsymbol{V}}^{(k+1)}$$

$$\widehat{\boldsymbol{V}}^{(k+1)} = \frac{\eta^{(k)}}{\sqrt{\boldsymbol{V}^{(k+1)}} + \epsilon} \nabla \mathcal{L}(\boldsymbol{w})$$

Adjust only the learning rate

$$\boldsymbol{V}^{(k+1)} = \alpha \boldsymbol{V}^{(k)} + (1 - \alpha)\left(\nabla \mathcal{L}(\boldsymbol{w})\right)^2$$

- Scale the learning rate by the inverse of the root mean squared gradients
  - Scale down the learning rates with large mean squared gradients (prevent the gradients from being too large)
  - Scale up the learning rates with small mean squared gradients (prevent the gradients from being too small)

Momentum & RMSprop

- Gradient descent with momentum smooths the gradient

Gradient descent with momentum:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \nabla\boldsymbol{w}^{(k+1)}$$

$$\nabla\boldsymbol{w}^{(k+1)} = \beta\nabla\boldsymbol{w}^{(k)} + \eta^{k}\nabla\mathcal{L}(\boldsymbol{w})$$

- RMSprop adjust the learning rate

RMSprop:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \widehat{\boldsymbol{V}}^{(k+1)}$$

$$\widehat{\boldsymbol{V}}^{(k+1)} = \frac{\eta^{(k)}}{\sqrt{\boldsymbol{V}^{(k+1)}} + \epsilon}\nabla\mathcal{L}(\boldsymbol{w})$$

$$\boldsymbol{V}^{(k+1)} = \alpha\boldsymbol{V}^{(k)} + (1-\alpha)\big(\nabla\mathcal{L}(\boldsymbol{w})\big)^{2}$$

- Can we do both?

Adaptive Moment Estimation (Adam)

- Adaptive Moment Estimation (Adam) maintains a running average of the mean gradients ,
  maintains a running average of the mean squared gradients and
  rescales the learning rates using an inverse of the root mean squared gradients

Adaptive Moment Estimation (Adam)

- Update the weights

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^{k} - \frac{\eta^{k}}{\sqrt{\widehat{\boldsymbol{V}}^{k+1}} + \epsilon}\widehat{\boldsymbol{U}}^{k+1}$$

- Scale the weighted values

$$\widehat{\boldsymbol{U}}^{k+1} = \frac{\boldsymbol{U}^{k+1}}{1-\beta_1}, \qquad \widehat{\boldsymbol{V}}^{k+1} = \frac{\boldsymbol{V}^{k+1}}{1-\beta_2}$$

- Compute a weighted average of the past gradients and past squared gradients

$$\boldsymbol{U}^{k+1} = \beta_1 \nabla \boldsymbol{U}^{k} + (1-\beta_1)\nabla\mathcal{L}(\boldsymbol{w})$$

$$\boldsymbol{V}^{k+1} = \beta_2 \boldsymbol{V}^{k} + (1-\beta_2)\left(\nabla\mathcal{L}(\boldsymbol{w})\right)^2$$

Adaptive Moment Estimation (Adam)

- Adam:

$$\boldsymbol{w}^{k+1} = \boldsymbol{w}^{k} - \frac{\eta^{k}}{\sqrt{\widehat{\boldsymbol{V}}^{k+1}} + \epsilon}\widehat{\boldsymbol{U}}^{k+1}$$

$\beta_1 = 0.9$
$\beta_2 = 0.999$
$\epsilon = 10^{-8}$

$$\widehat{\boldsymbol{U}}^{k+1} = \frac{\boldsymbol{U}^{k+1}}{1-\beta_1}, \qquad \widehat{\boldsymbol{V}}^{k+1} = \frac{\boldsymbol{V}^{k+1}}{1-\beta_2}$$

$$\boldsymbol{U}^{k+1} = \beta_1 \boldsymbol{U}^{k} + (1-\beta_1)\nabla\mathcal{L}(\boldsymbol{w})$$

$$\boldsymbol{V}^{k+1} = \beta_2 \boldsymbol{V}^{k} + (1-\beta_2)\big(\nabla\mathcal{L}(\boldsymbol{w})\big)^2$$

RMSprop:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \widehat{\boldsymbol{V}}^{(k+1)}$$

$$\widehat{\boldsymbol{V}}^{(k+1)} = \frac{\eta^{(k)}}{\sqrt{\boldsymbol{V}^{(k+1)}} + \epsilon}\nabla\mathcal{L}(\boldsymbol{w})$$

$$\boldsymbol{V}^{(k+1)} = \alpha\boldsymbol{V}^{(k)} + (1-\alpha)\big(\nabla\mathcal{L}(\boldsymbol{w})\big)^2$$

Generalization & Regularization

Regularization: Weight Decay

$$\mathcal{L}_{reg}(\boldsymbol{w}) = \mathcal{L}(\boldsymbol{w}) + \frac{\lambda}{2}\sum_i w_i^2$$

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta^{(k)}\nabla\mathcal{L}(\boldsymbol{w}) - \eta^{(k)}\lambda\boldsymbol{w}^{(k)}$$

- Regularization term penalizes large weights
- During gradient descent update, weights are decayed linearly toward zero
  - λ determines how dominant the regularization is