

Spring 2024

# CMPE 258-01

# Deep Learning

*Dr. Kaikai Liu, Ph.D. Associate Professor*

*Department of Computer Engineering*

*San Jose State University*

*Email: [kaikai.liu@sjsu.edu](mailto:kaikai.liu@sjsu.edu)*

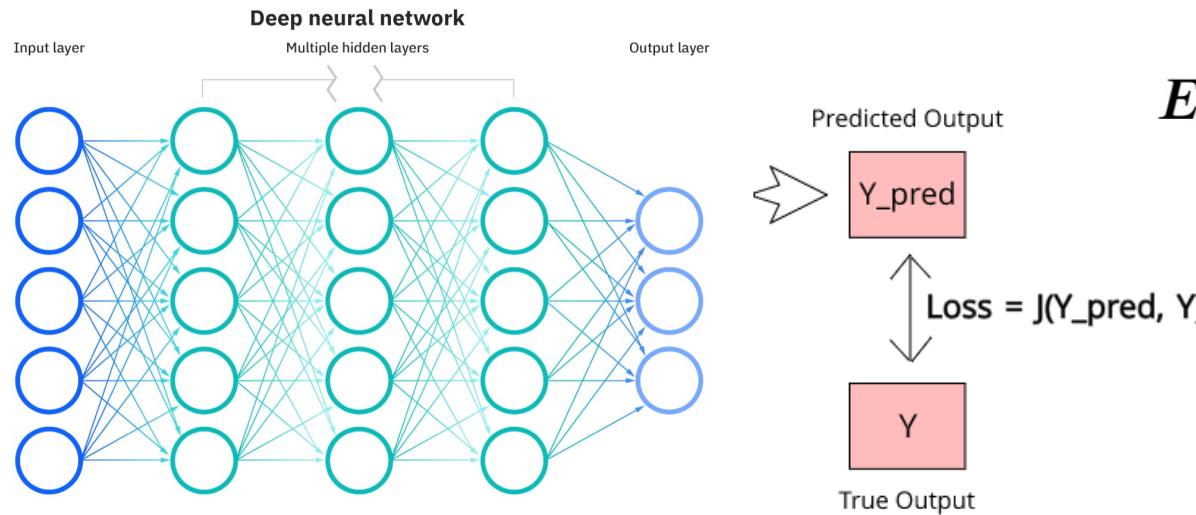
*Website: <https://www.sjsu.edu/cmpe/faculty/tenure-line/kaikai-liu.php>*



# Neural Network Regression

- Neural Network for Regression:

- Similar to Linear Regression that it also uses a set of inputs and weights to produce an output.
- Regression Loss Functions: Mean Squared Error
- Backpropagation allows us to calculate and attribute the error associated with each neuron, allowing us to adjust and fit the parameters of the model(s) appropriately.



$$E(y_{\text{output}}, y_{\text{target}}) = \frac{1}{2}(y_{\text{output}} - y_{\text{target}})^2$$

$$w_{ij} = w_{ij} - \alpha \frac{dE}{dw_{ij}}$$

# Gradient descent

- Gradient descent is an optimization algorithm for minimizing the loss of a predictive model with regard to a training dataset.
  - Which step in this algorithm is most time consuming?
  - Typically the loss function is really the **average loss** over a **large dataset**.
  - **Loading and computing** on all the data is **expensive**

Gradient Descent Algorithm

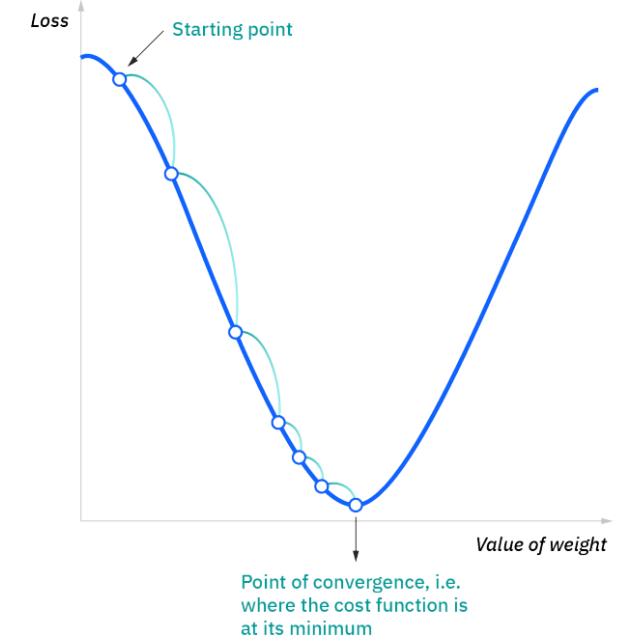
$$\theta^{(0)} \leftarrow \text{initial vector (random, zeros ...)}$$

For  $\tau$  from 0 to convergence:

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \rho(\tau) \left( \nabla_{\theta} \mathbf{L}(\theta) \Big|_{\theta=\theta^{(\tau)}}^{\text{Evaluated at}} \right)$$

$$\nabla_{\theta} \mathbf{L}(\theta) \Big|_{\theta=\theta^{(\tau)}} = \frac{1}{n} \sum_{i=1}^n \nabla_{\theta} \text{loss}(y, f_{\theta}(x)) \Big|_{\theta=\theta^{(\tau)}}$$

This update is simultaneously done for all the weights.



# Stochastic gradient descent

- Batch gradient descent computes the gradient using the whole dataset.
  - This is great for convex, or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution, either local or global.
- Stochastic gradient descent (SGD) computes the gradient using random sample.
  - Most applications of SGD actually use a minibatch of several samples.
  - SGD works well for error manifolds that have lots of local maxima/minima. In this case, the somewhat noisier gradient calculated using the reduced number of samples tends to jerk the model out of local minima into a region that hopefully is more optimal.
  - Single samples are really noisy, while **minibatches** tend to average a little of the noise out. Thus, the amount of jerk is reduced when using minibatches.

# Stochastic Gradient Descent

- Draw a simple random sample of data indices
  - Often called a **batch** or **mini-batch**
  - Choice of **batch size** trade-off **gradient quality** and **speed**

- Compute **gradient estimate** and uses as **gradient**

$\theta^{(0)} \leftarrow$  initial vector (random, zeros ...)

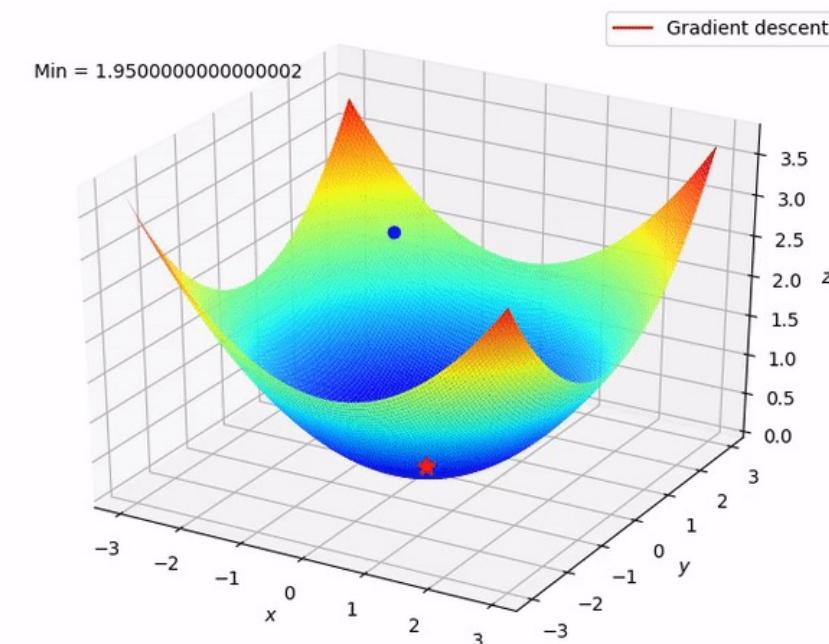
For  $\tau$  from 0 to convergence:

$\mathcal{B} \sim$  Random subset of indices

$$\theta^{(\tau+1)} \leftarrow \theta^{(\tau)} - \rho(\tau) \left( \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_{\theta} \mathbf{L}_i(\theta) \Big|_{\theta=\theta^{(\tau)}} \right)$$

- Loss can be written as a sum of the loss on each record.

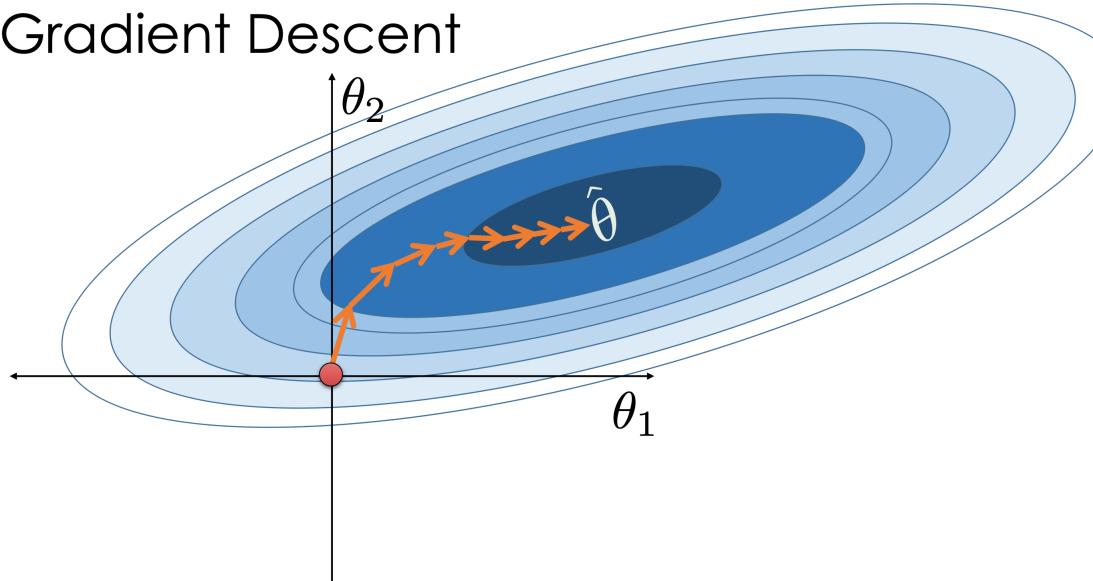
Decomposable Loss  $\mathbf{L}(\theta) = \sum_{i=1}^n \mathbf{L}_i(\theta) = \sum_{i=1}^n \mathbf{L}(\theta, x_i, y_i)$



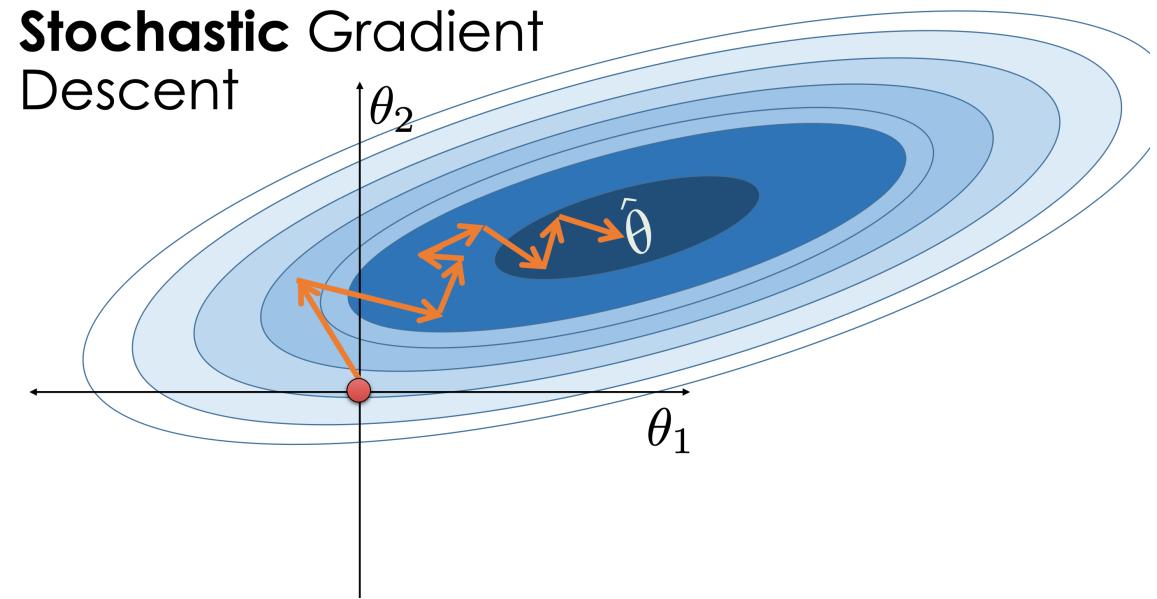
# Stochastic Gradient Descent

- Dominant algorithm in modern machine learning and AI
  - Some important variations
    - Momentum terms (SGD with Momentum)
    - Automatic learning rates (ADAM)
  - Core algorithm driving progress in deep learning
    - TensorFlow and Pytorch designed around SGD

Gradient Descent



Stochastic Gradient  
Descent



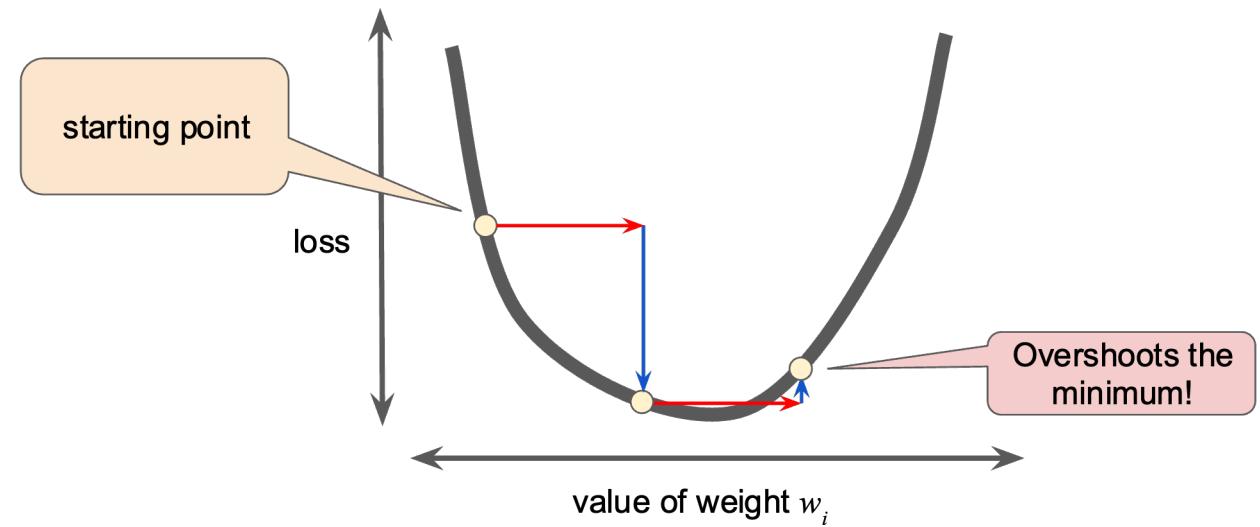
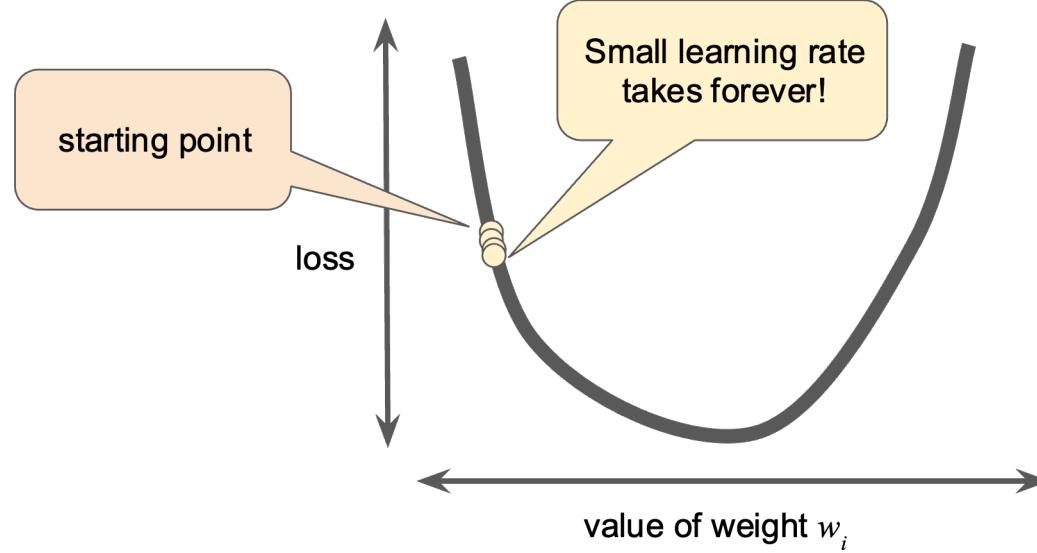
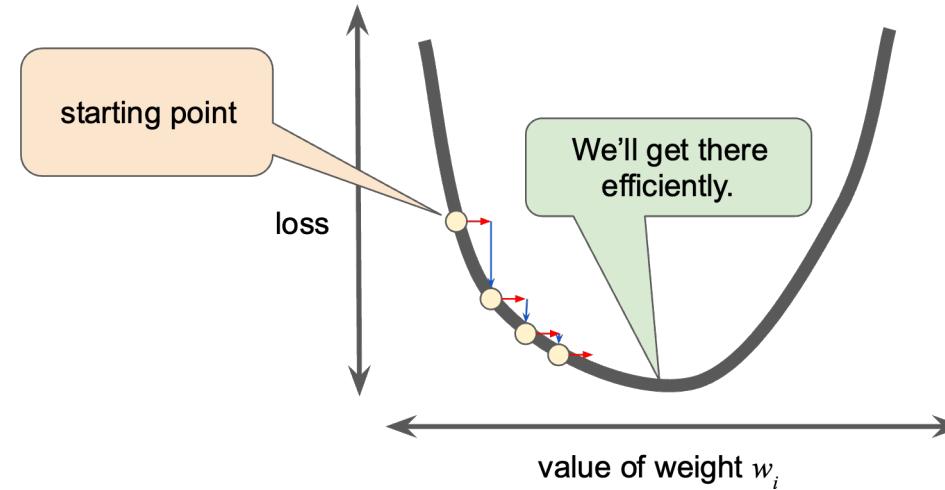
# Learning Rate

- Learning rate

Repeat until convergence {

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

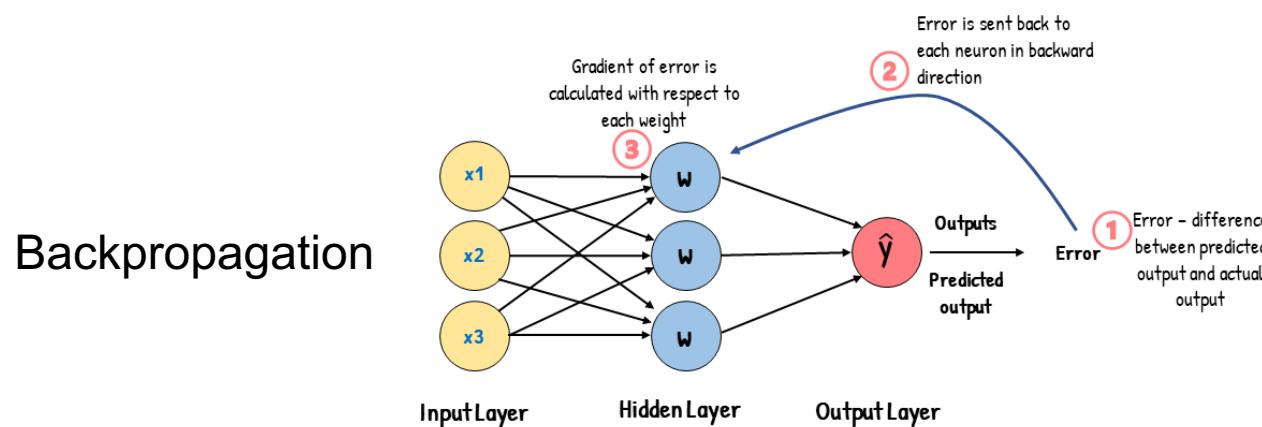
}



# Neural Network Training

- **Stochastic Gradient Descent With Back-propagation**

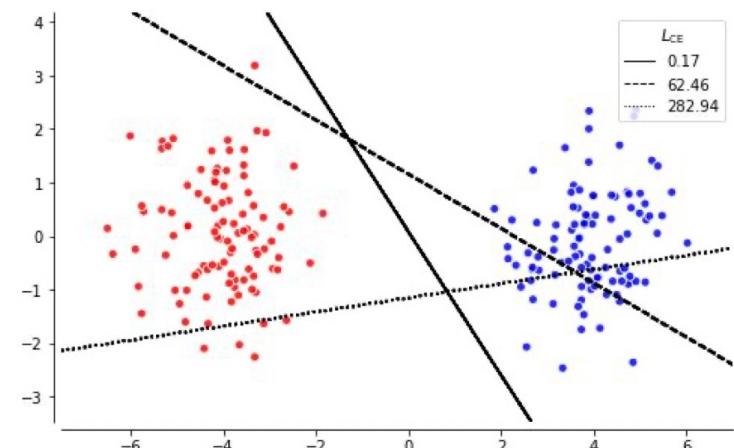
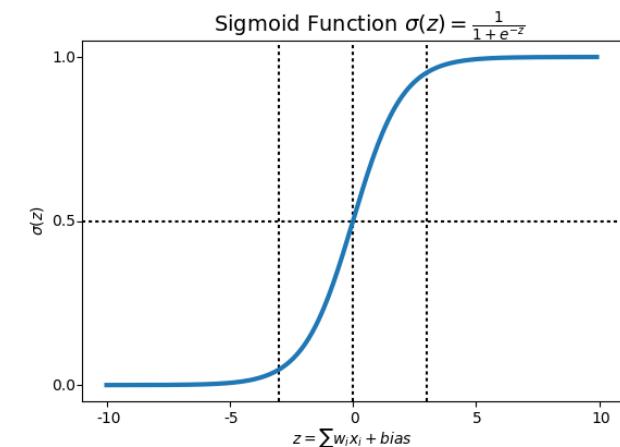
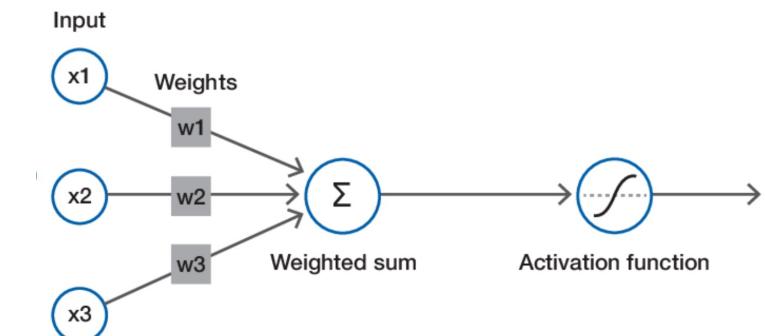
- Stochastic gradient descent is an optimization algorithm for minimizing the loss of a predictive model with regard to a training dataset.
- Back-propagation is an automatic differentiation algorithm for calculating gradients for the weights in a neural network graph structure.
  - It makes gradient descent feasible for multi-layer neural networks.
- Stochastic gradient descent and the back-propagation of error algorithms together are used to train neural network models.



# Neural Classifier

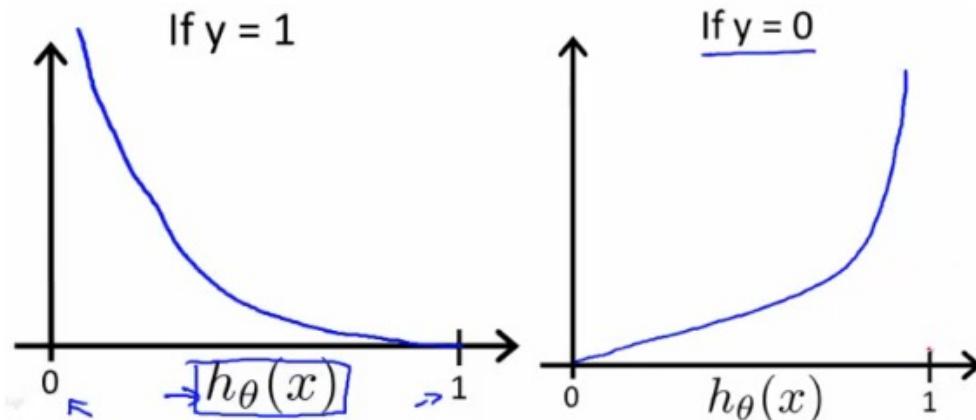
- The simplest “neural” classifier
  - Equivalent to logistic regression: **sigmoid function** output (0~1)
  - **Cross entropy loss** is also called negative log likelihood or logistic loss.
    - two separate cost functions: one for  $y=1$  and one for  $y=0$ .
  - Being additive over samples allows for efficient learning.
  - Encodes negation of logarithm of probability of entirely correct classification

$$L_{CE}(\mathbf{p}, \mathbf{t}) = - \sum_{i=1}^n [\mathbf{t}^{(i)} \log \mathbf{p}^{(i)} + (1 - \mathbf{t}^{(i)}) \log(1 - \mathbf{p}^{(i)})]$$



# Cost Function

- The benefits of taking the logarithm reveal themselves when you look at the cost function graphs for  $y=1$  and  $y=0$ . These smooth monotonic functions (always increasing or always decreasing) make it easy to calculate the gradient and minimize cost.
- Image from Andrew Ng's slides on logistic regression



$\text{Cost}(h_\theta(x), y) = 0$  if  $h_\theta(x) = y$   
 $\text{Cost}(h_\theta(x), y) \rightarrow \infty$  if  $y = 0$  and  $h_\theta(x) \rightarrow 1$   
 $\text{Cost}(h_\theta(x), y) \rightarrow \infty$  if  $y = 1$  and  $h_\theta(x) \rightarrow 0$

- The key thing to note is the cost function penalizes confident and wrong predictions more than it rewards confident and right predictions

# Cost Function

- We can compress our cost function's two conditional cases into one case (when  $y$  is equal to 1, then the second term will be zero and will not affect the result. If  $y$  is equal to 0, then the first term will be zero and will not affect the result):

$$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$

- We can fully write out our entire cost function as follows:

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

- Gradient Descent

Repeat {

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

}

Repeat {

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

}

# Softmax

- Softmax function is also a type of sigmoid function but is handy when we are trying to handle classification problems.
  - The sigmoid function was able to handle just two classes.
  - Softmax is the most commonly used final activation in classification (produces probability estimate) and is the Multi-dimensional generalisation of sigmoid
  - Has simple derivatives
  - The softmax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs. This essentially gives the probability of the input being in a particular class.

$$p_i = \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}$$

Softmax function takes an N-dimensional vector of real numbers and transforms it into a vector of real number in range (0,1) which add upto 1.

# Softmax

- The property of softmax function that it outputs a probability distribution makes it suitable for probabilistic interpretation in classification tasks.
  - The numerical range of floating point numbers in numpy is limited. For exponential, its not difficult to overshoot that limit, in which case python returns nan
  - To make our softmax function numerically stable, we simply normalize the values in the vector, by multiplying the numerator and denominator with a constant
  - We can choose an arbitrary value for  $\log(C)$  term, but generally  $\log(C)=-\max(a)$  is chosen, as it shifts all of elements in the vector to negative to zero

$$\begin{aligned} p_i &= \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}} \\ &= \frac{Ce^{a_i}}{C \sum_{k=1}^N e^{a_k}} \\ &= \frac{e^{a_i + \log(C)}}{\sum_{k=1}^N e^{a_k + \log(C)}} \end{aligned}$$

```
def stable_softmax(X):
    exps = np.exp(X - np.max(X))
    return exps / np.sum(exps)
```

# Derivative of Softmax

- Due to the desirable property of softmax function outputting a probability distribution, we use it as the final layer in neural networks.
- For this we need to calculate the derivative or gradient and pass it back to the previous layer during backpropagation.

$$\frac{\partial p_i}{\partial a_j} = \frac{\partial \frac{e^{a_i}}{\sum_{k=1}^N e^{a_k}}}{\partial a_j} \quad \frac{\partial p_i}{\partial a_j} = \begin{cases} p_i(1 - p_j) & \text{if } i = j \\ -p_j \cdot p_i & \text{if } i \neq j \end{cases}$$

- Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is.
  - It is used when node activations can be understood as representing the probability that each hypothesis might be true, i.e. when the output is a probability distribution. Thus it is used as a loss function in neural networks which have softmax activations in the output layer.

$$H(y, p) = - \sum_i y_i \log(p_i)$$

# Softmax + Cross entropy

- Softmax + Cross entropy

- Cross Entropy Loss with Softmax function are used as the output layer extensively.
- Encodes negation of logarithm of probability of entirely correct classification
- Equivalent to multinomial logistic regression model
- Numerically stable combination

$$L = - \sum_i y_i \log(p_i)$$

$$\begin{aligned}\frac{\partial L}{\partial o_i} &= - \sum_k y_k \frac{\partial \log(p_k)}{\partial o_i} \\ &= - \sum_k y_k \frac{\partial \log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i} \\ &= - \sum_k y_k \frac{1}{p_k} \times \frac{\partial p_k}{\partial o_i}\end{aligned}$$

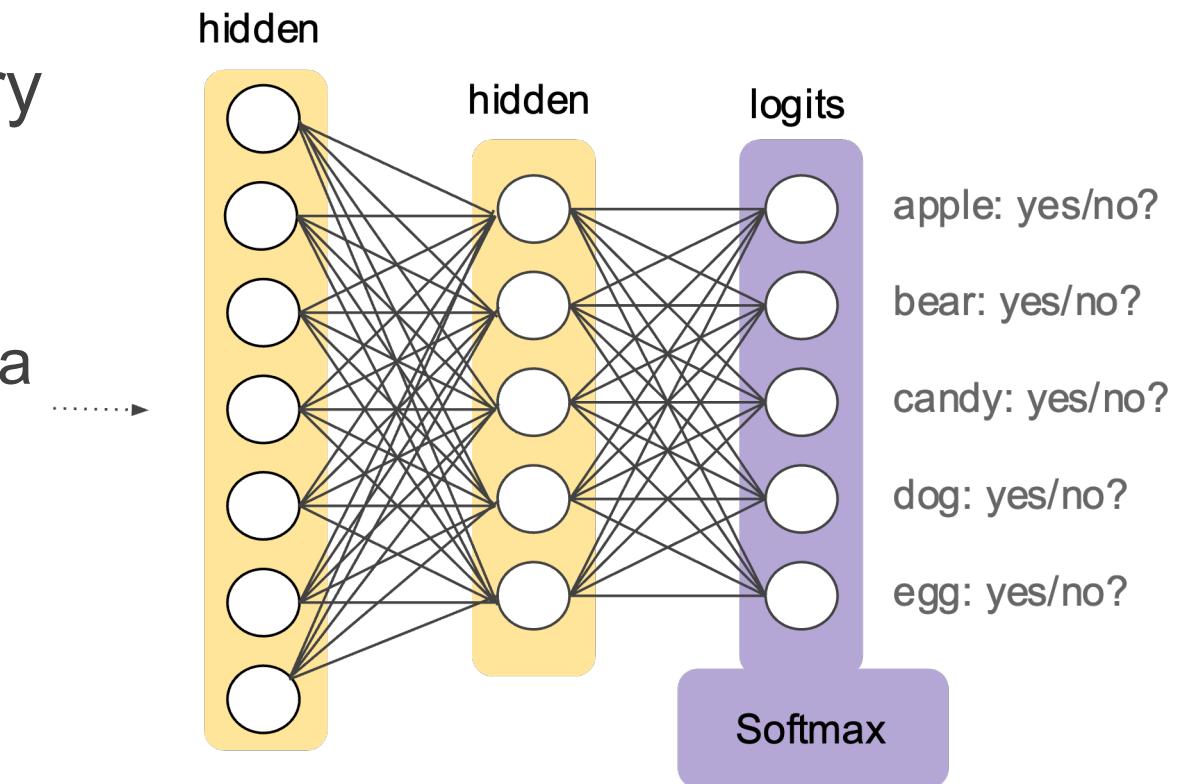


$$\frac{\partial L}{\partial o_i} = p_i - y_i$$

which is a very simple and elegant expression

# Multi-Class Neural Networks

- Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate binary classifiers—one binary classifier for each possible outcome.
  - During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification question.

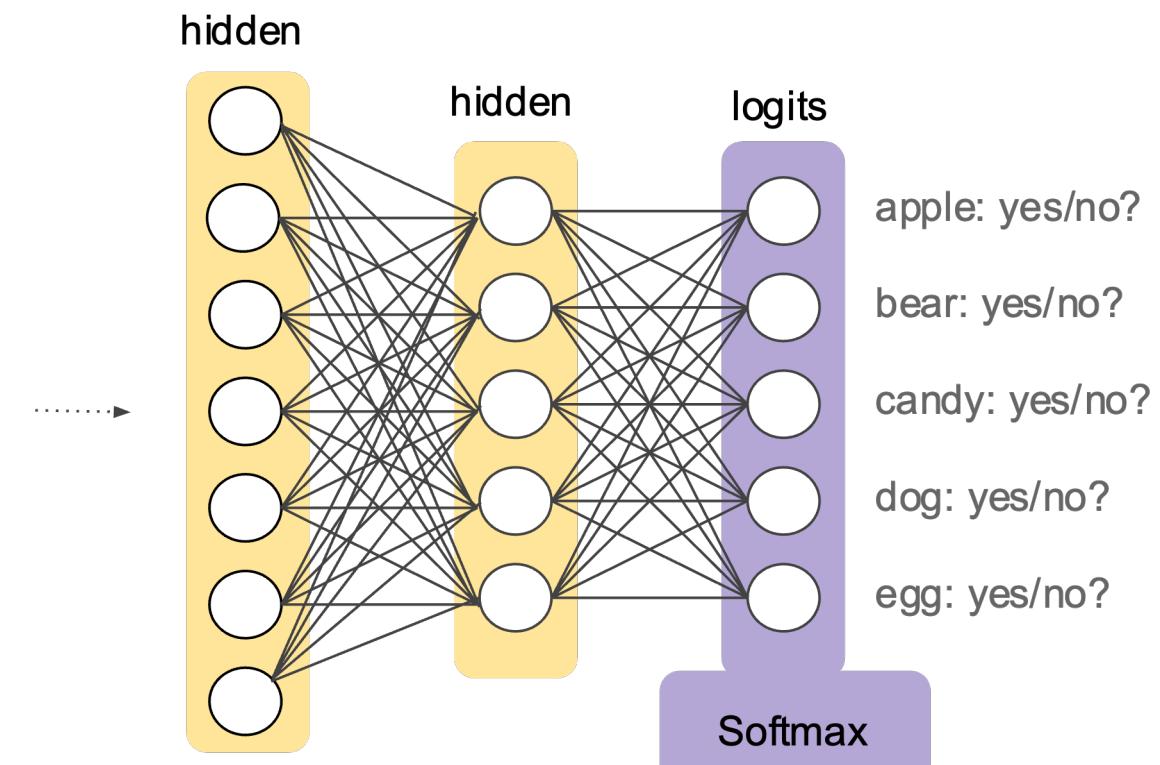


# Multi-Class Neural Networks

- **Softmax** extends the probability idea into a multi-class world.

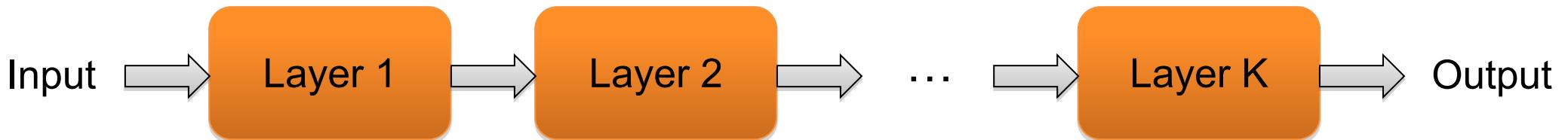
- Softmax assigns decimal probabilities to each class in a multi-class problem.
- Those decimal probabilities must add up to 1.0.
- This additional constraint helps training converge more quickly than it otherwise would.

$$p(y = j|\mathbf{x}) = \frac{e^{(\mathbf{w}_j^T \mathbf{x} + b_j)}}{\sum_{k \in K} e^{(\mathbf{w}_k^T \mathbf{x} + b_k)}}$$

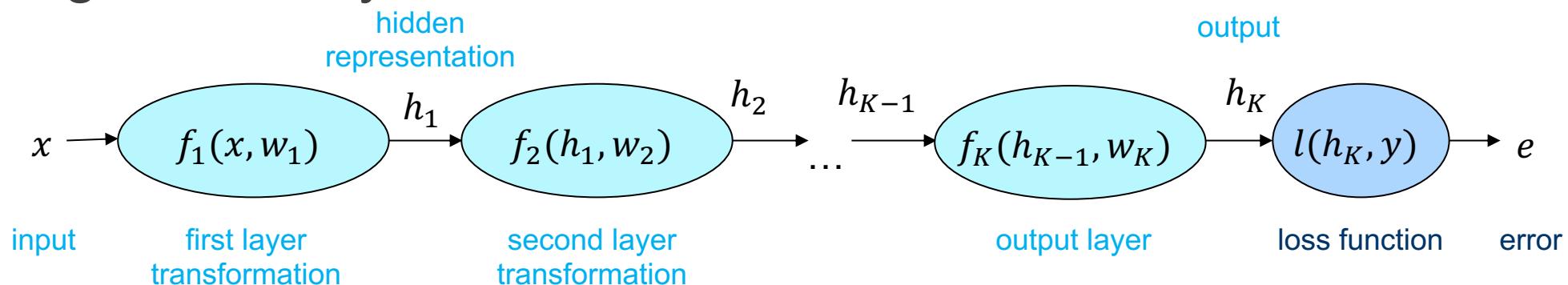


# Neural Network Training

- The function computed by the network is a composition of the functions computed by individual layers (e.g., linear layers and nonlinearities):



- Training a multi-layer network

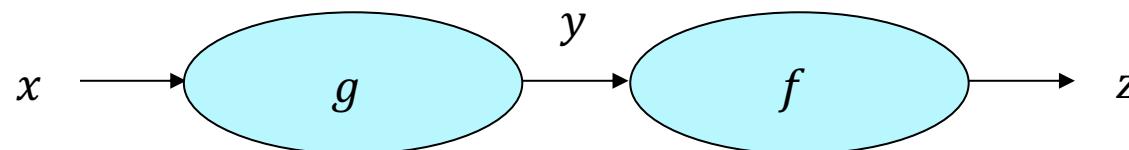


- To train the network, we need to find the gradient of the error w.r.t. the parameters of each layer

$$\text{kth layer: } w_k \leftarrow w_k - \eta \frac{\partial e}{\partial w_k}$$

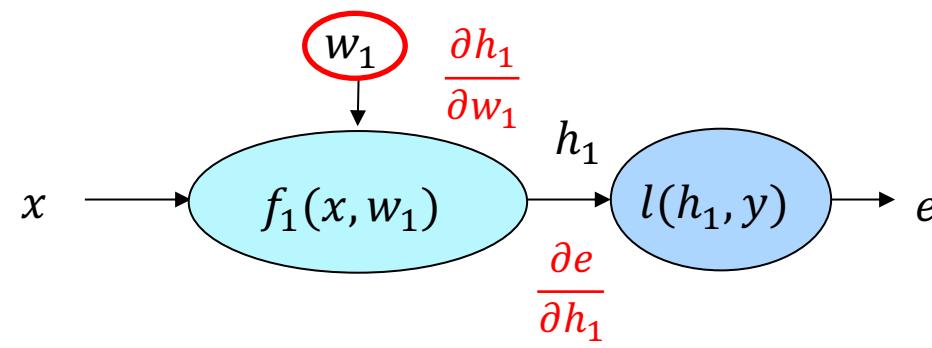
# Neural Network Training

- Chain rule



- Applying the chain rule

- Start with k=1



$$e = l(f_1(x, w_1), y)$$

$$e = (y - w_1^T x)^2$$

$$h_1 = f_1(x, w_1) = w_1^T x$$

$$e = l(h_1, y) = (y - h_1)^2$$

$$\frac{\partial h_1}{\partial w_1} = x$$

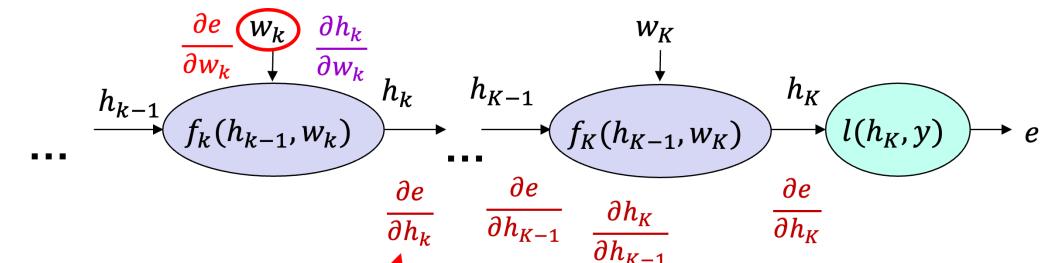
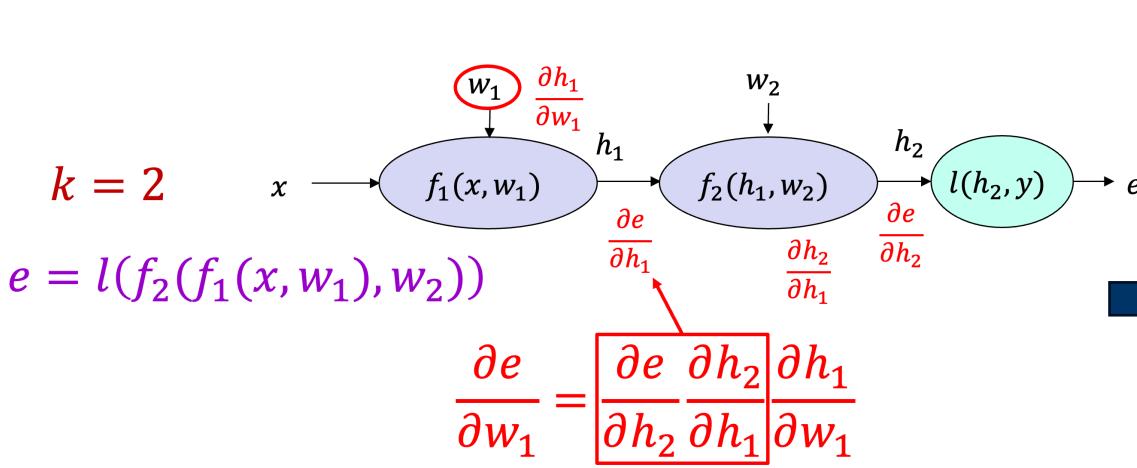
$$\frac{\partial e}{\partial h_1} = -2(y - h_1) = -2(y - w_1^T x)$$

$$\frac{\partial e}{\partial w_1} = \frac{\partial e}{\partial h_1} \frac{\partial h_1}{\partial w_1} = -2x(y - w_1^T x)$$

# Neural Network Training

- Applying the chain rule

- $k=2$



$$\frac{\partial e}{\partial w_k} = \boxed{\frac{\partial e}{\partial h_K} \frac{\partial h_K}{\partial h_{K-1}} \dots \frac{\partial h_{k+1}}{\partial h_k}} \frac{\partial h_k}{\partial w_k}$$

Upstream gradient

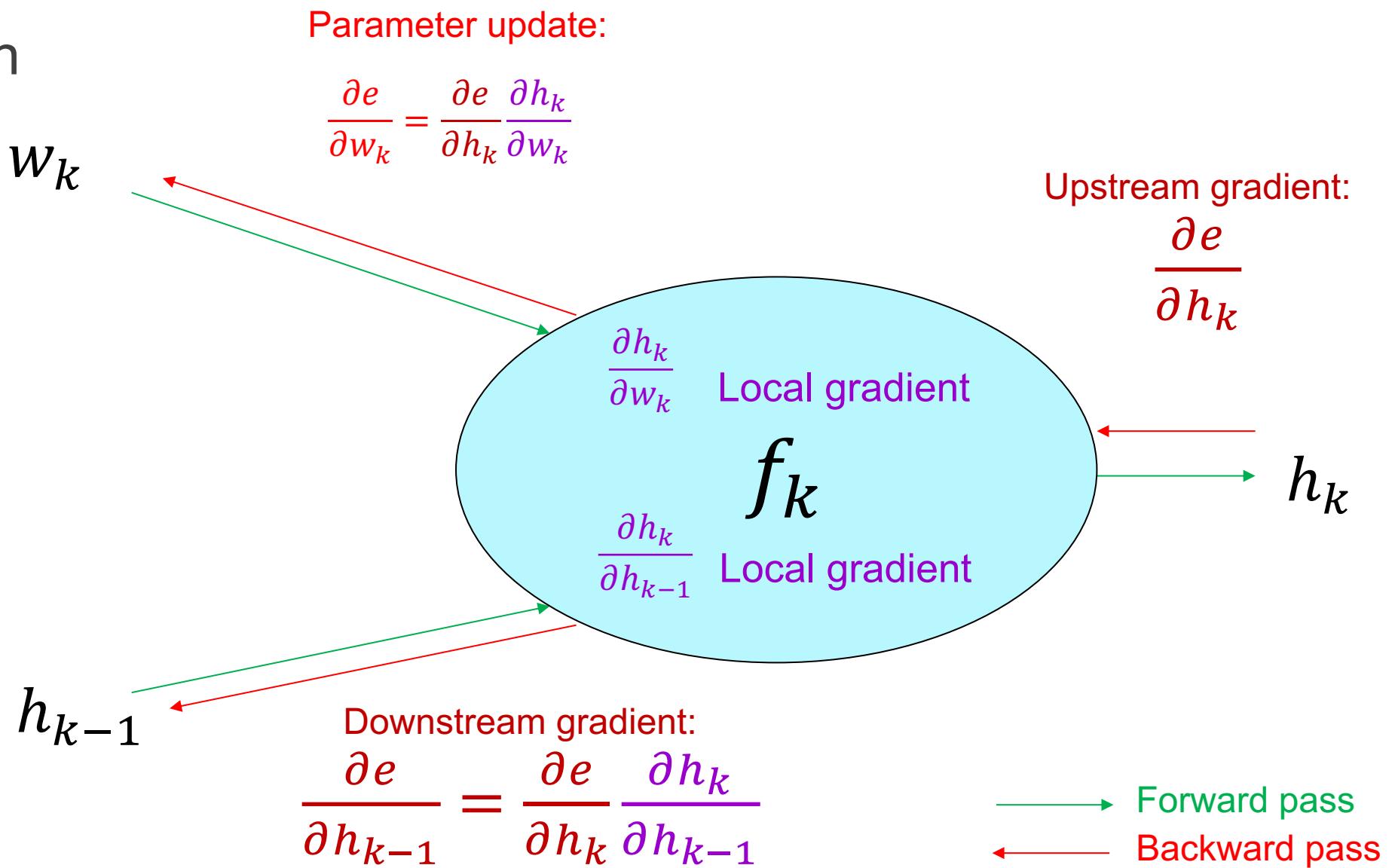
$$\frac{\partial e}{\partial h_k}$$

Local gradient

$$\frac{\partial h_k}{\partial w_k}$$

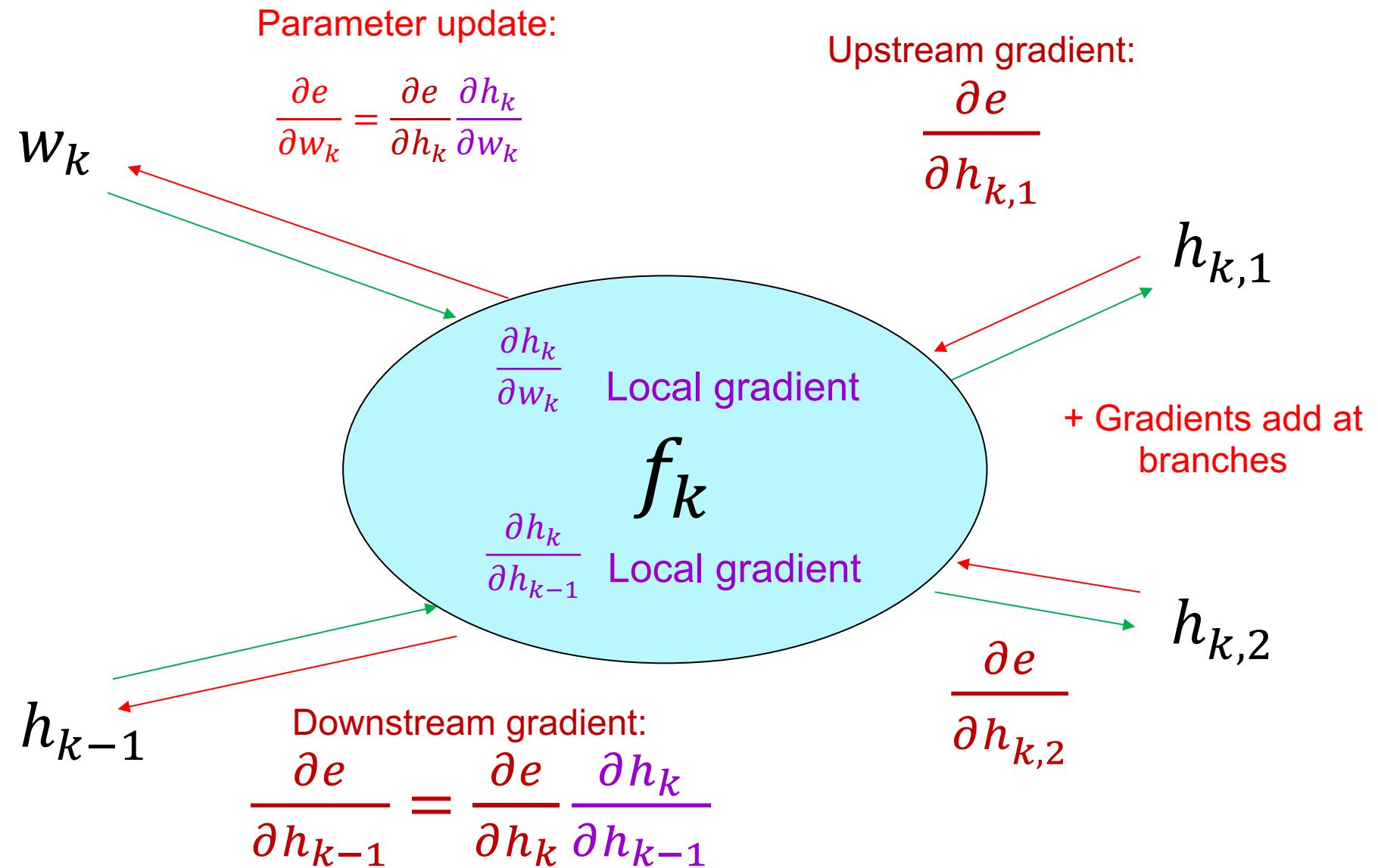
# Neural Network Training

- Backpropagation



# Neural Network Training

- With branches
  - E.g., Resnet



# Neural Network Training

- In vector calculus, the Jacobian matrix of a vector-valued function of several variables is the matrix of all its first-order partial derivatives
- The Jacobian matrix collects all first-order partial derivatives of a multivariate function that can be used for backpropagation.

Suppose  $\mathbf{f} : \mathbf{R}^n \rightarrow \mathbf{R}^m$  is a function such that each of its first-order partial derivatives exist on  $\mathbf{R}^n$ . This function takes a point  $\mathbf{x} \in \mathbf{R}^n$  as input and produces the vector  $\mathbf{f}(\mathbf{x}) \in \mathbf{R}^m$  as output. Then the Jacobian matrix of  $\mathbf{f}$  is defined to be an  $m \times n$  matrix, denoted by  $\mathbf{J}$ , whose  $(i,j)$ th entry is  $\mathbf{J}_{ij} = \frac{\partial f_i}{\partial x_j}$ , or explicitly

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \dots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \nabla^T f_1 \\ \vdots \\ \nabla^T f_m \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

where  $\nabla^T f_i$  is the transpose (row vector) of the [gradient](#) of the  $i$ -th component.

# Neural Network Training

- Gradient

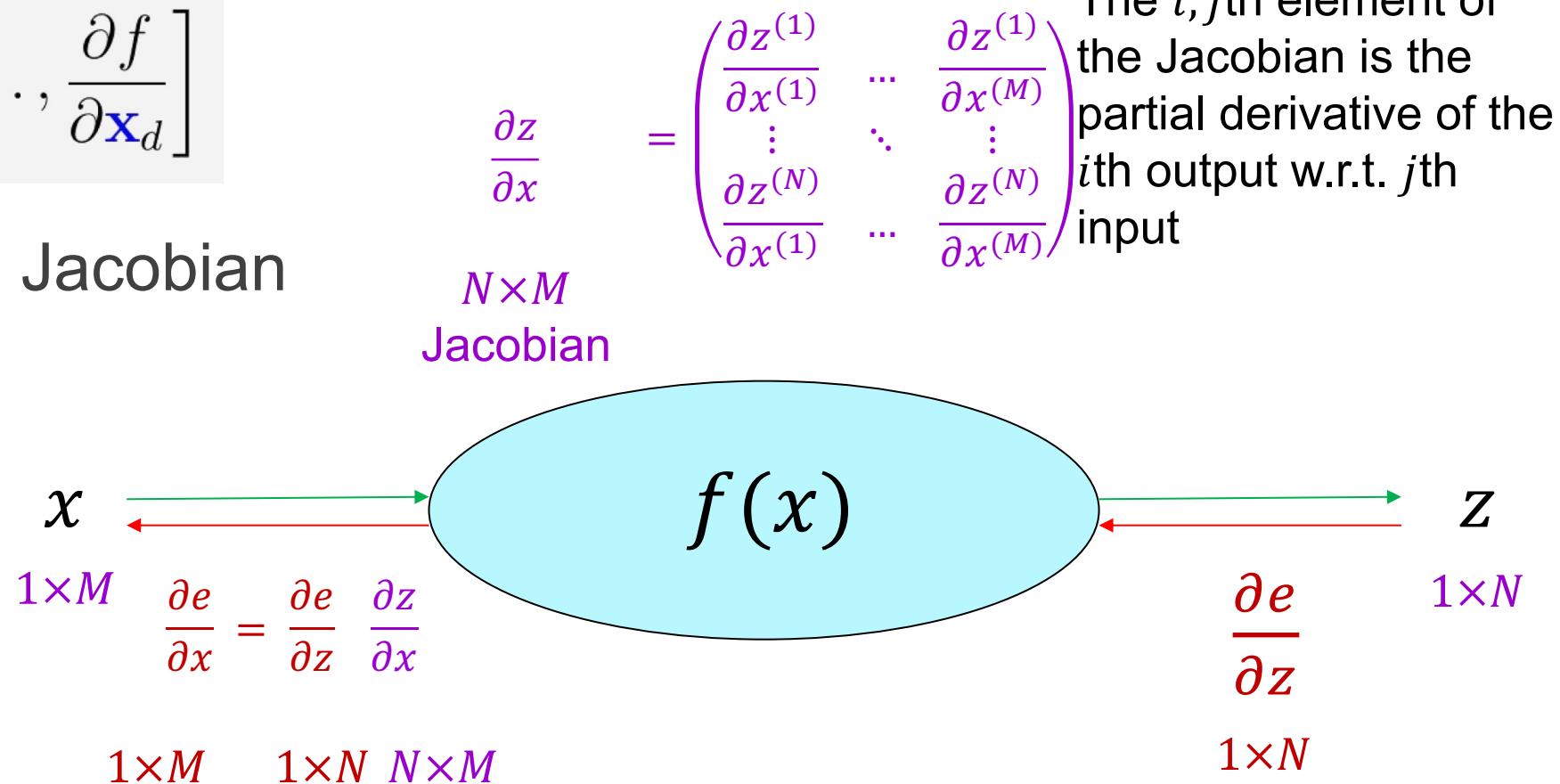
$$y = f(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$$

$$\frac{\partial y}{\partial \mathbf{x}} = \nabla_{\mathbf{x}} f(\mathbf{x}) = \left[ \frac{\partial f}{\partial \mathbf{x}_1}, \dots, \frac{\partial f}{\partial \mathbf{x}_d} \right]$$

- Dealing with vectors: Jacobian

$$\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}^k$$

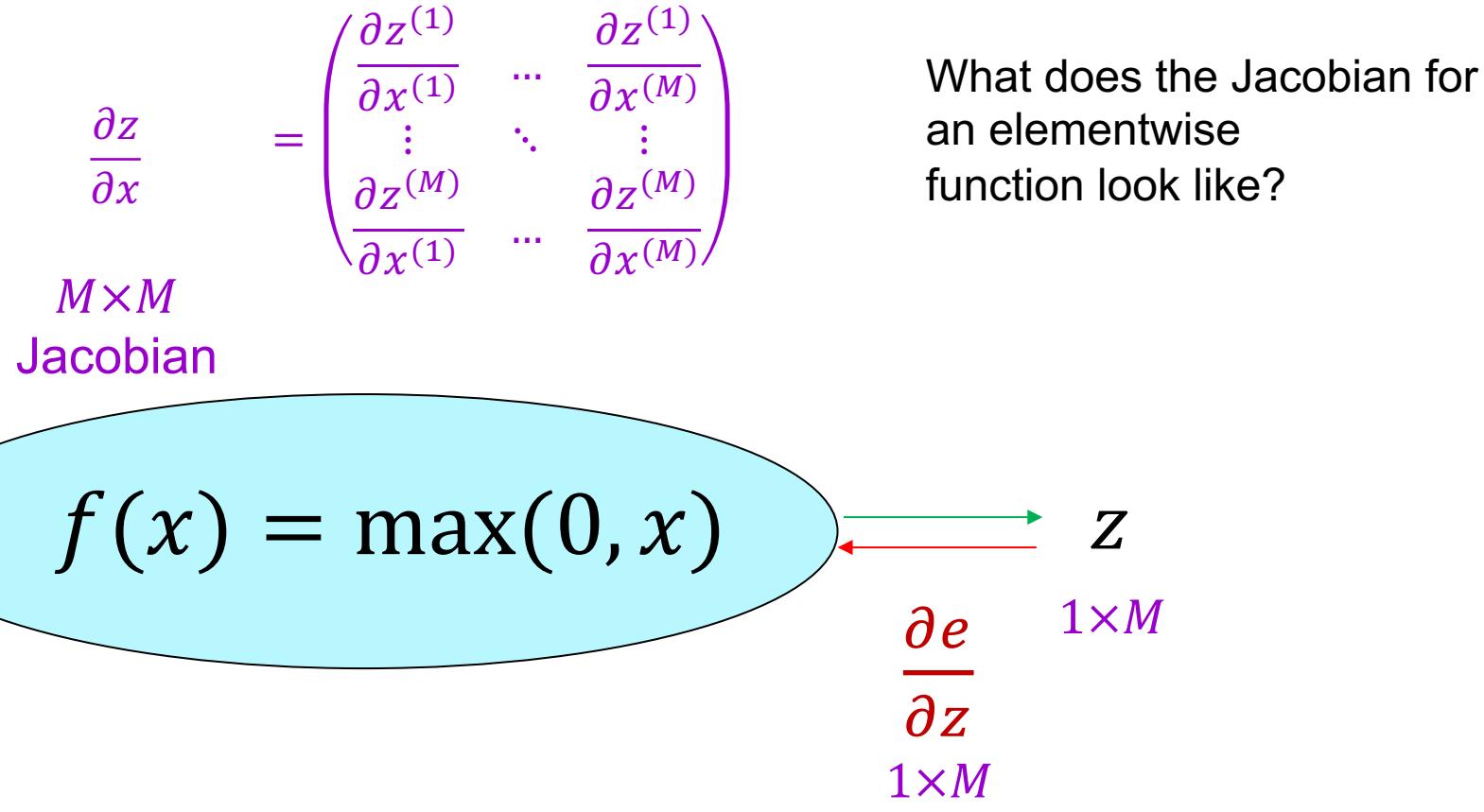
$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{J}_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial \mathbf{x}_1} & \dots & \frac{\partial f_1}{\partial \mathbf{x}_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial \mathbf{x}_1} & \dots & \frac{\partial f_k}{\partial \mathbf{x}_d} \end{bmatrix}$$



Jacobian: row indices correspond to outputs, column indices correspond to inputs. The  $i, j$ th element of the Jacobian is the partial derivative of the  $i$ th output w.r.t.  $j$ th input

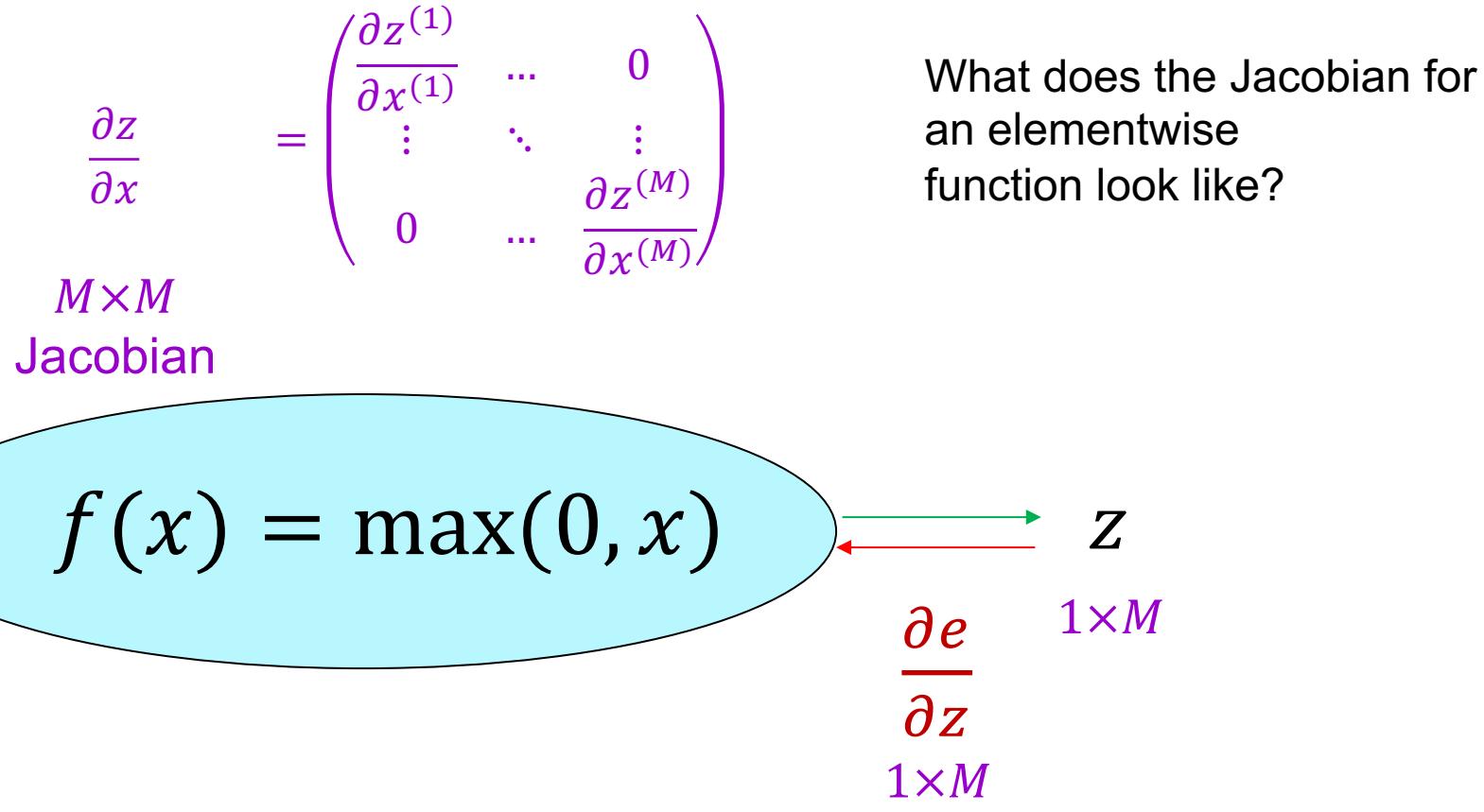
# Neural Network Training

- Simple case: Elementwise operation (ReLU layer)



# Neural Network Training

- Simple case: Elementwise operation (ReLU layer)



# Backpropagation

- Simple example

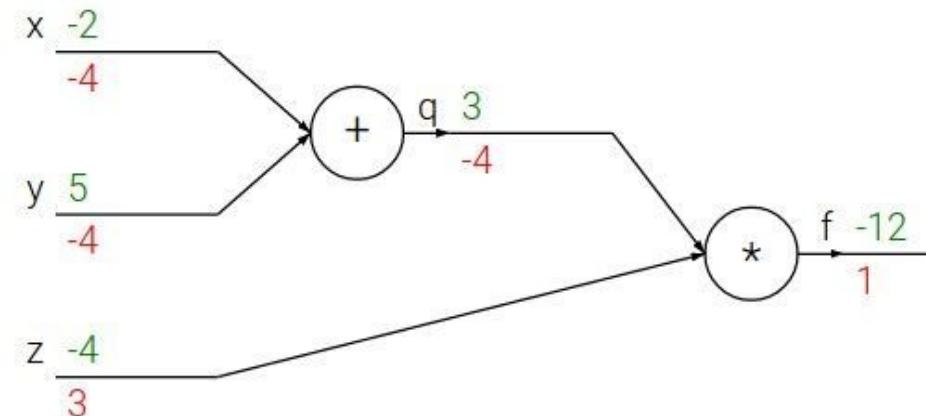
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2$ ,  $y = 5$ ,  $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



# Backpropagation: a simple example

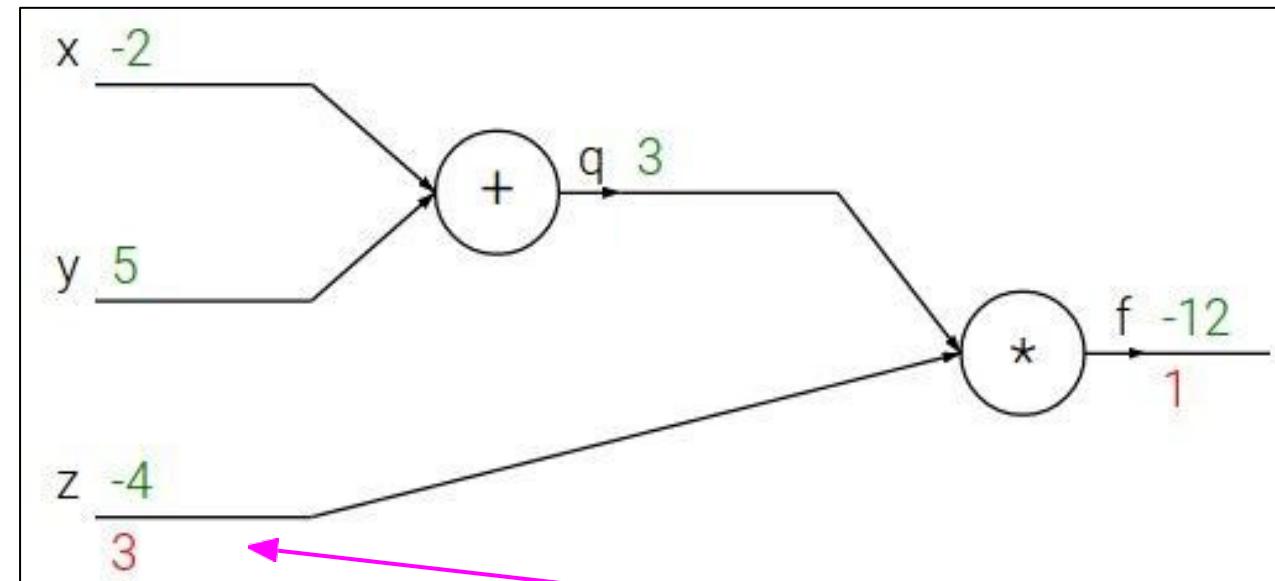
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2, y = 5, z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial z}$$

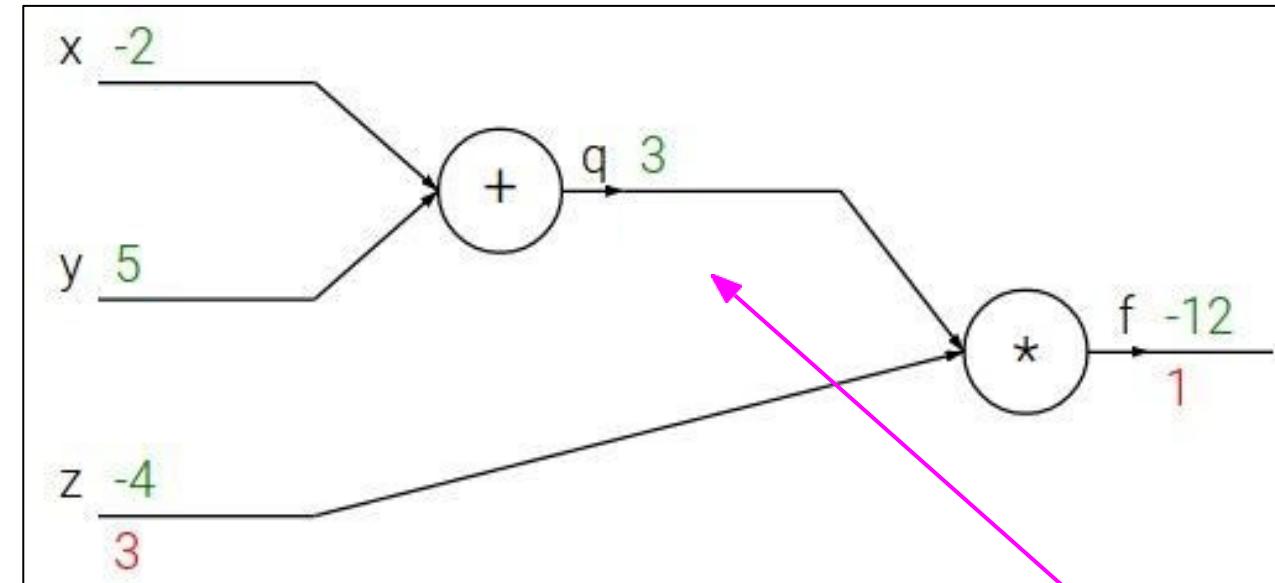
# Backpropagation: a simple example

$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2$ ,  $y = 5$ ,  $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

$$\frac{\partial f}{\partial q}$$

# Backpropagation: a simple example

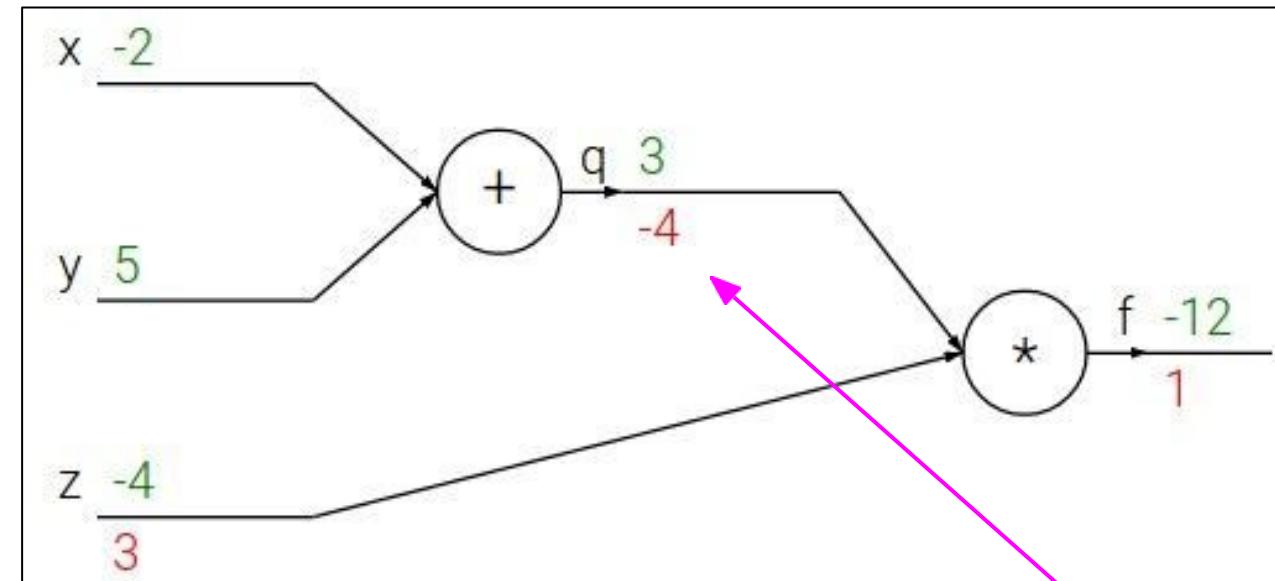
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2$ ,  $y = 5$ ,  $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



$$\frac{\partial f}{\partial q}$$

# Backpropagation: a simple example

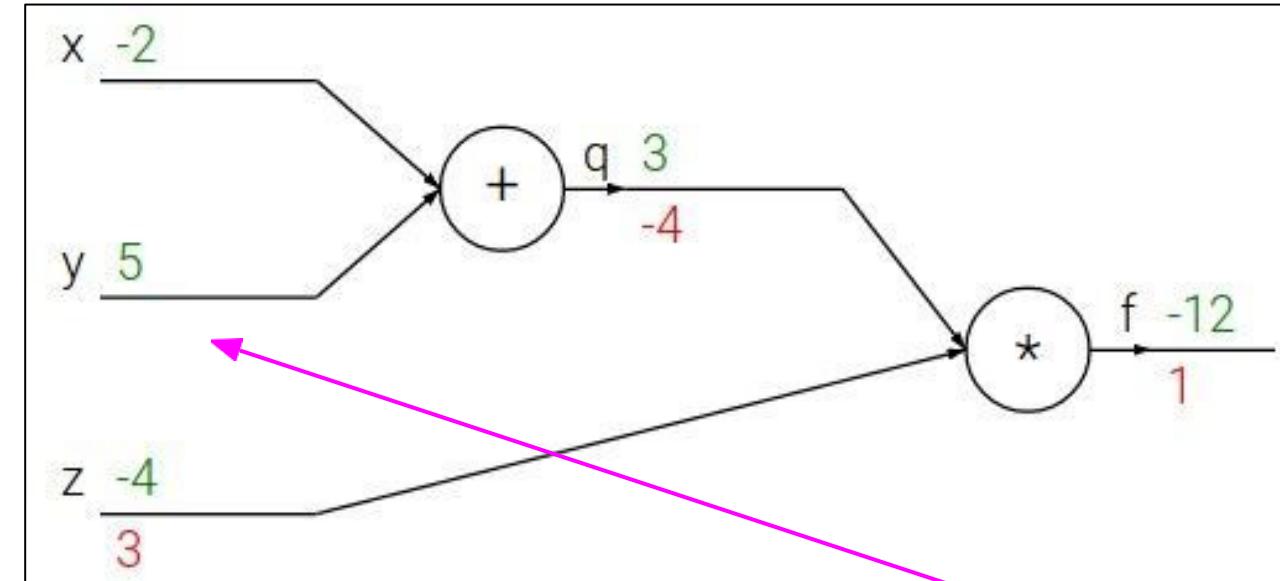
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2$ ,  $y = 5$ ,  $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}$$

Upstream gradient      Local gradient

$$\frac{\partial f}{\partial y}$$

# Backpropagation: a simple example

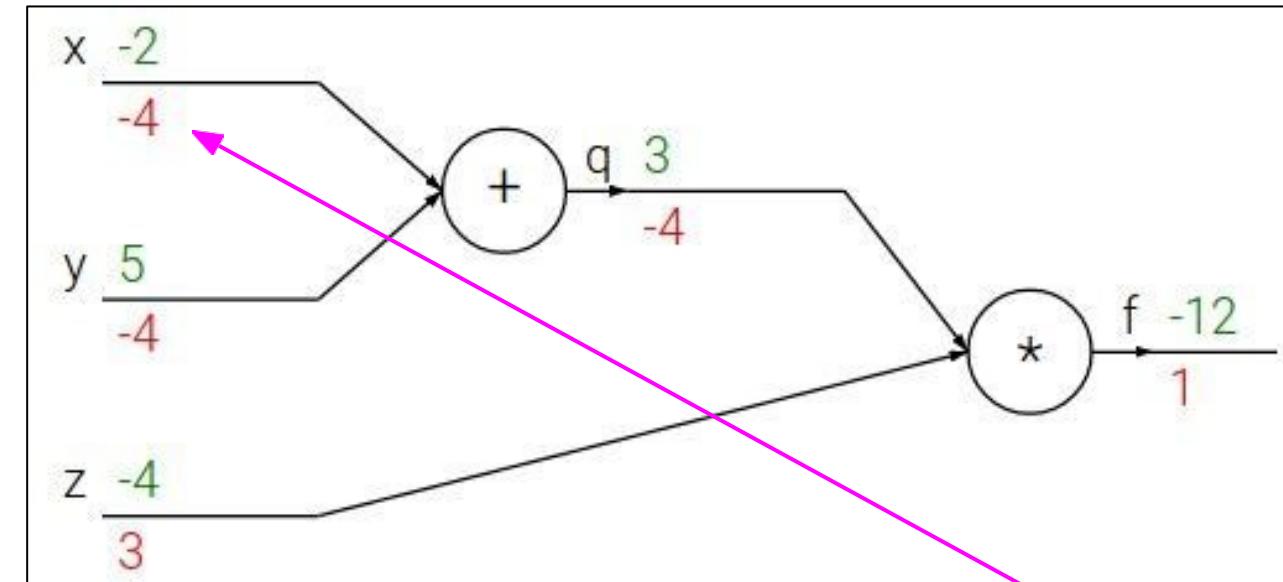
$$f(x, y, z) = (x + y)z$$

e.g.  $x = -2$ ,  $y = 5$ ,  $z = -4$

$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want:  $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

Upstream  
gradient

Local  
gradient

$$\frac{\partial f}{\partial x}$$

# Backpropagation

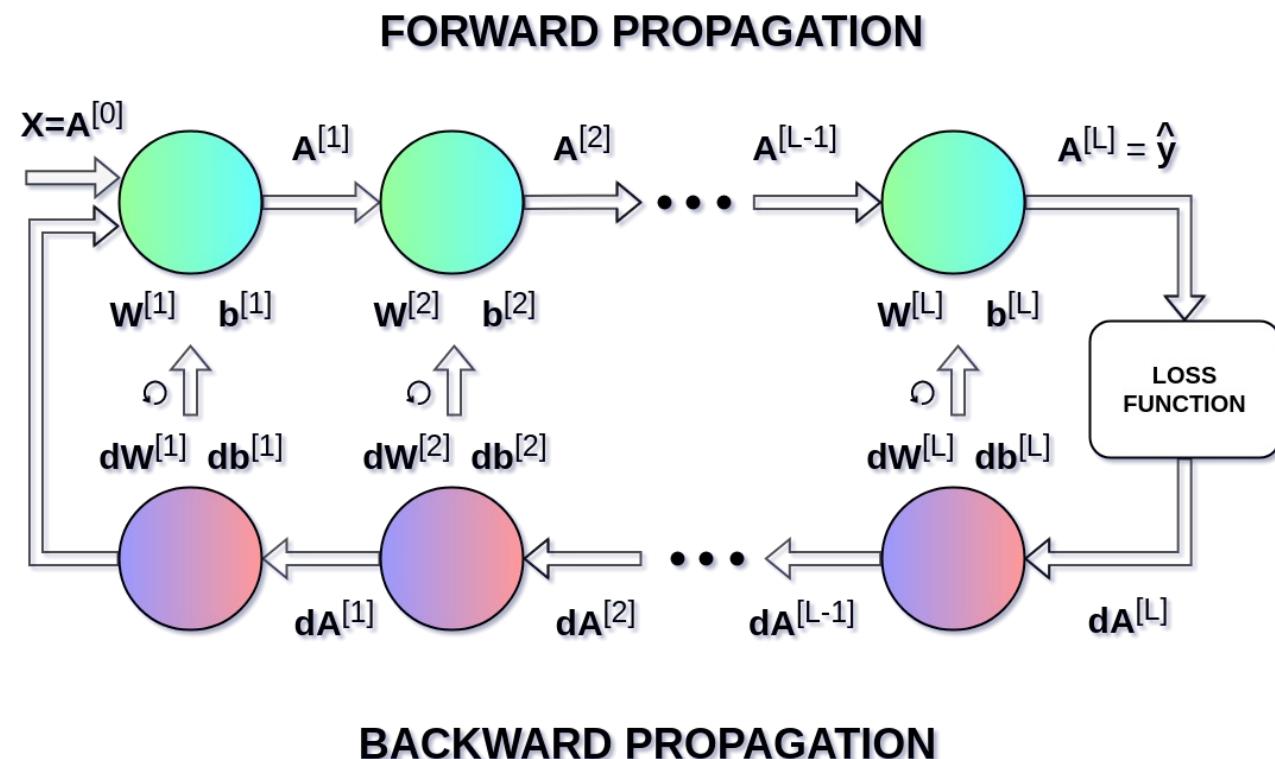
- Backpropagation is an algorithm that allows us to calculate a very complicated gradient, like the one we need.

- The parameters of the neural network are adjusted according to the following formulae.

$$\mathbf{w}^{[l]} = \mathbf{w}^{[l]} - \alpha \mathbf{dW}^{[l]}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \mathbf{db}^{[l]}$$

- $\alpha$  represents learning rate - a hyperparameter which allows you to control the value of performed adjustment.



# Optimizer algorithms

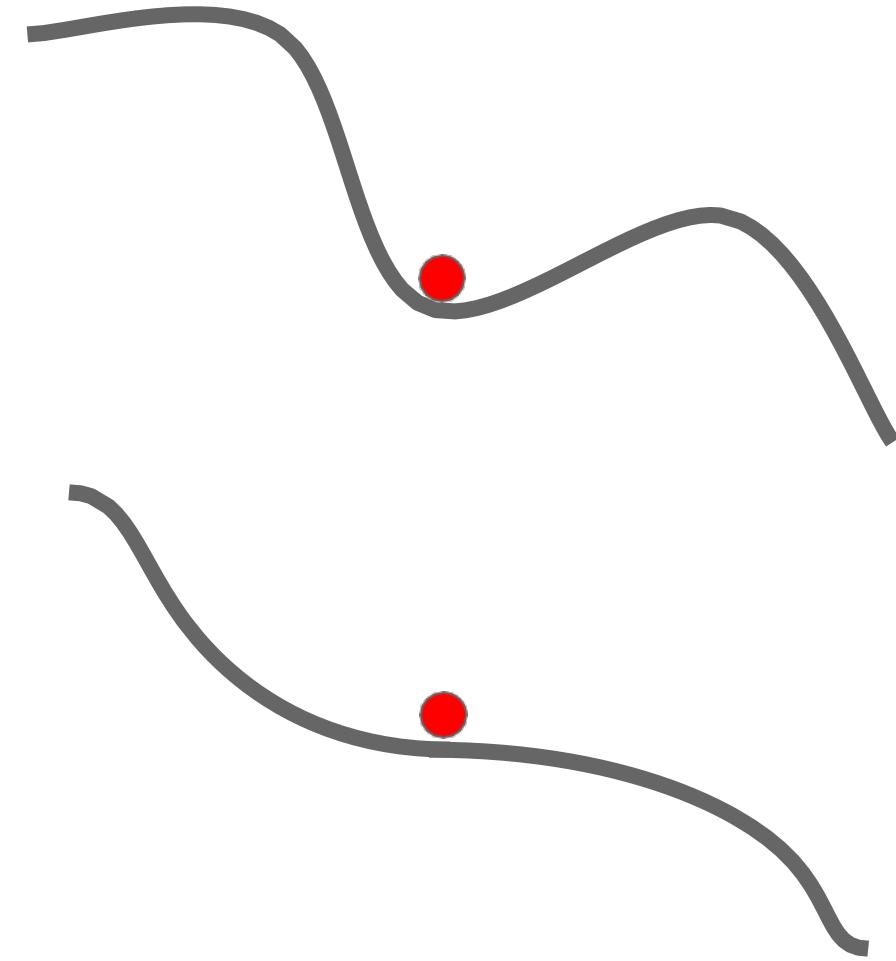
- Optimizer algorithms are optimization method that helps improve a deep learning model's performance.
- While training the deep learning optimizers model, modify each epoch's weights and minimize the loss function.
  - An **optimizer** is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates.
  - Thus, it helps in reducing the overall loss and improving accuracy.
- **Gradient Descent Deep Learning Optimizer**
- **Stochastic Gradient Descent Deep Learning Optimizer**
  - Gradient descent works best for most purposes. However, it has some downsides too. It is expensive to calculate the gradients if the size of the data is huge. Gradient descent works well for convex functions, but it doesn't know how far to travel along the gradient for nonconvex functions.

# Optimization: Problems with SGD

- What if the loss function has a **local minima** or **saddle point**?
- Saddle points much more common in high dimension
- Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^N \nabla_W L_i(x_i, y_i, W)$$



# SGD + Momentum Optimizer

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:  
    dx = compute_gradient(x)  
    x -= learning_rate * dx
```

## SGD+Momentum

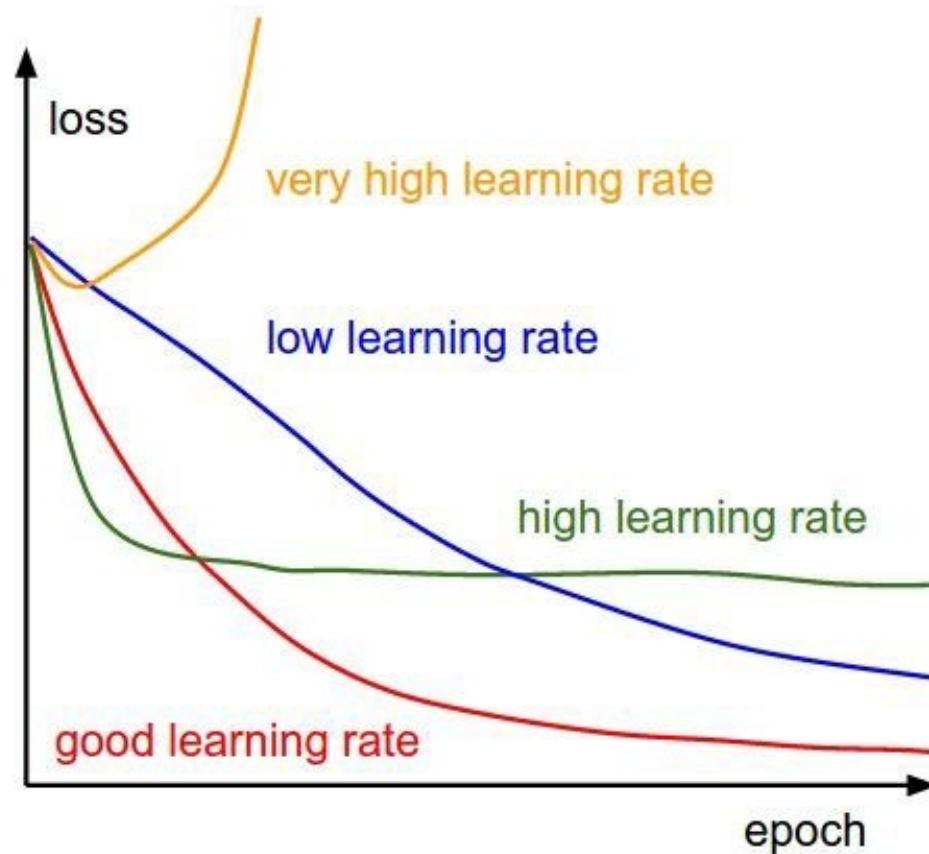
$$\begin{aligned} v_{t+1} &= \rho v_t + \nabla f(x_t) \\ x_{t+1} &= x_t - \alpha v_{t+1} \end{aligned}$$

```
vx = 0  
while True:  
    dx = compute_gradient(x)  
    vx = rho * vx + dx  
    x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

# Learning Rate

- Learning rate as a hyperparameter
  - Q: Which one of these learning rates is best to use?
  - A: All of them! Start with large learning rate and decay over time



# Other Optimizers

- Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer
  - Uses different learning rates for each iteration
  - It performs smaller updates for parameters associated with frequently occurring features, and larger updates for parameters associated with infrequently occurring features.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} g_{t,i}$$

## RMSProp

- Root Mean Square Propagation (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight
- RMSProp keep moving average of the squared gradients for each weight. And then we divide the gradient by square root the mean square.

$$E[g^2](t) = \beta E[g^2](t-1) + (1-\beta) \left( \frac{\partial c}{\partial w} \right)^2$$

$$w_{ij}(t) = w_{ij}(t-1) - \frac{\eta}{\sqrt{E[g^2]}} \frac{\partial c}{\partial w_{ij}}$$

# Other Optimizers

- Adam is an adaptive learning rate optimization algorithm that utilises both momentum and scaling, combining the benefits of RMSProp and SGD w/th Momentum.

- Adam paper: <https://arxiv.org/pdf/1412.6980.pdf>
- The algorithm updates exponential moving averages of the gradient ( $m_t$ ) and the squared gradient ( $v_t$ ) where the hyper-parameters  $\beta_1, \beta_2 \in [0, 1]$  control the exponential decay rates of these moving averages.

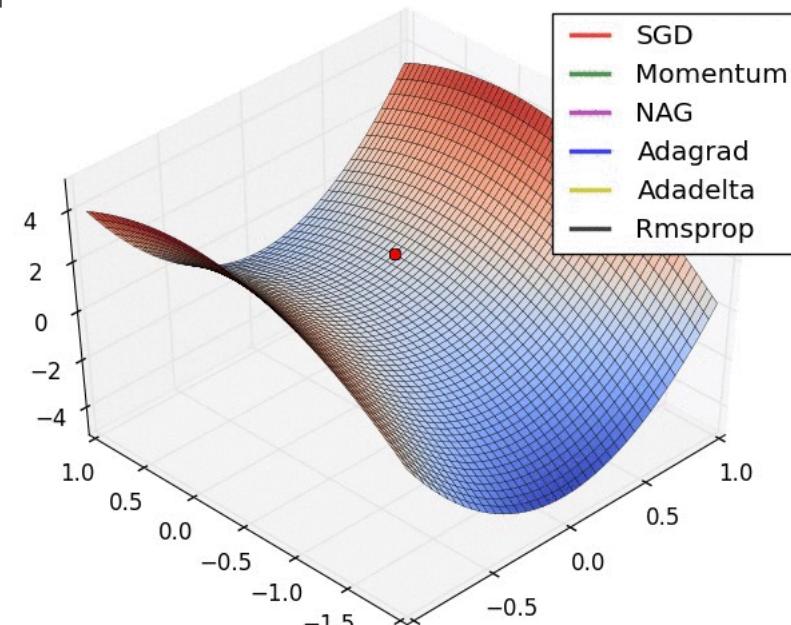
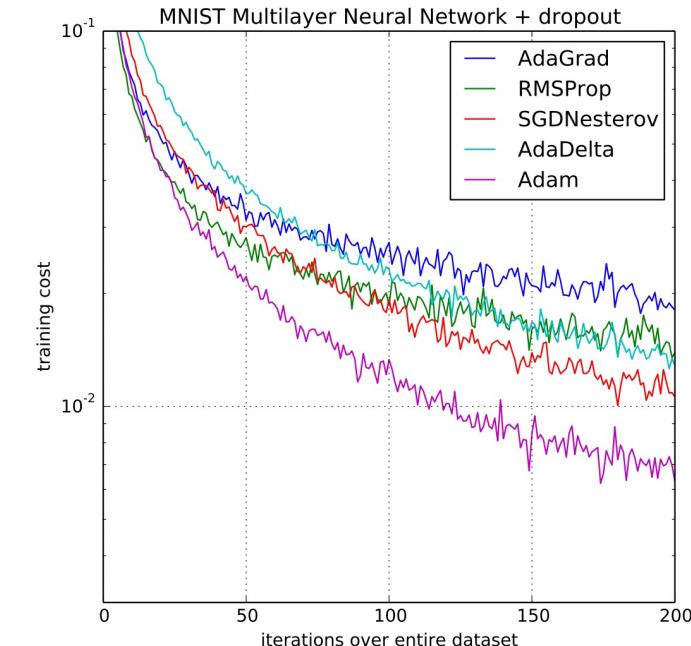
$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$



# Torch Optim

- torch.optim is a package implementing various optimization algorithms.

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
optimizer = optim.Adam(model.parameters(), lr=0.05)
```

- Optimizers also support specifying per-parameter options.

- contain a params key, containing a list of parameters belonging to it.

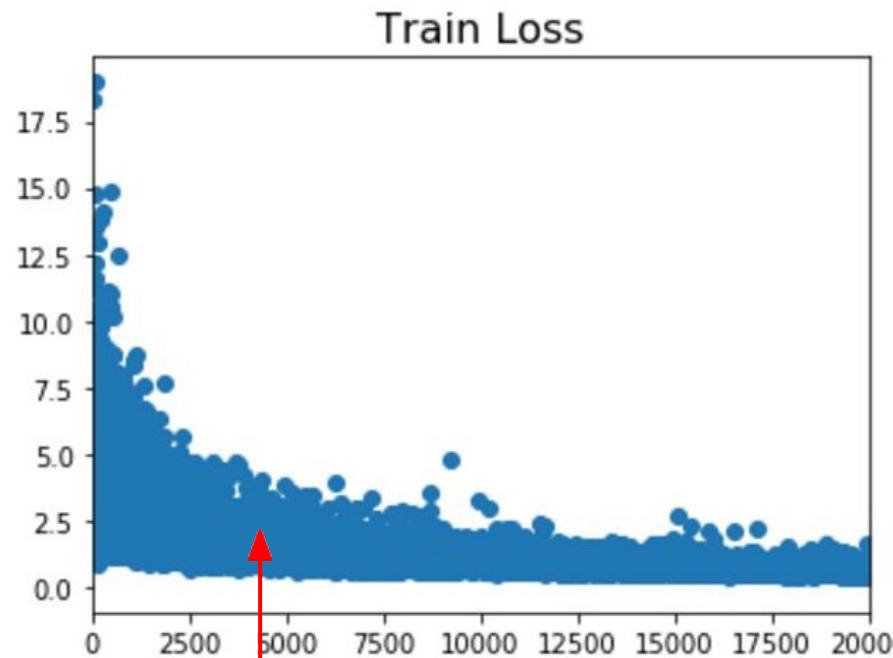
```
optim.SGD([
    {'params': model.base.parameters()},
    {'params': model.classifier.parameters(), 'lr': 1e-3}
], lr=1e-2, momentum=0.9)
```

This means that model.base's parameters will use the default learning rate of 1e-2, model.classifier's parameters will use a learning rate of 1e-3, and a momentum of 0.9 will be used for all parameters.

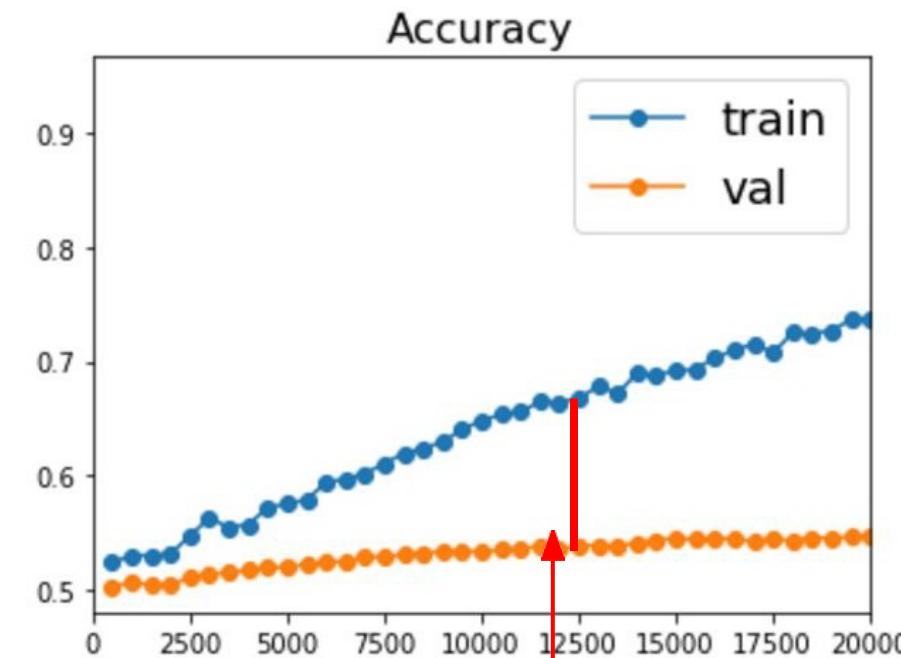
- Taking an optimization step

```
for input, target in dataset:
    optimizer.zero_grad()
    output = model(input)
    loss = loss_fn(output, target)
    loss.backward()
    optimizer.step()
```

# Beyond Training Error



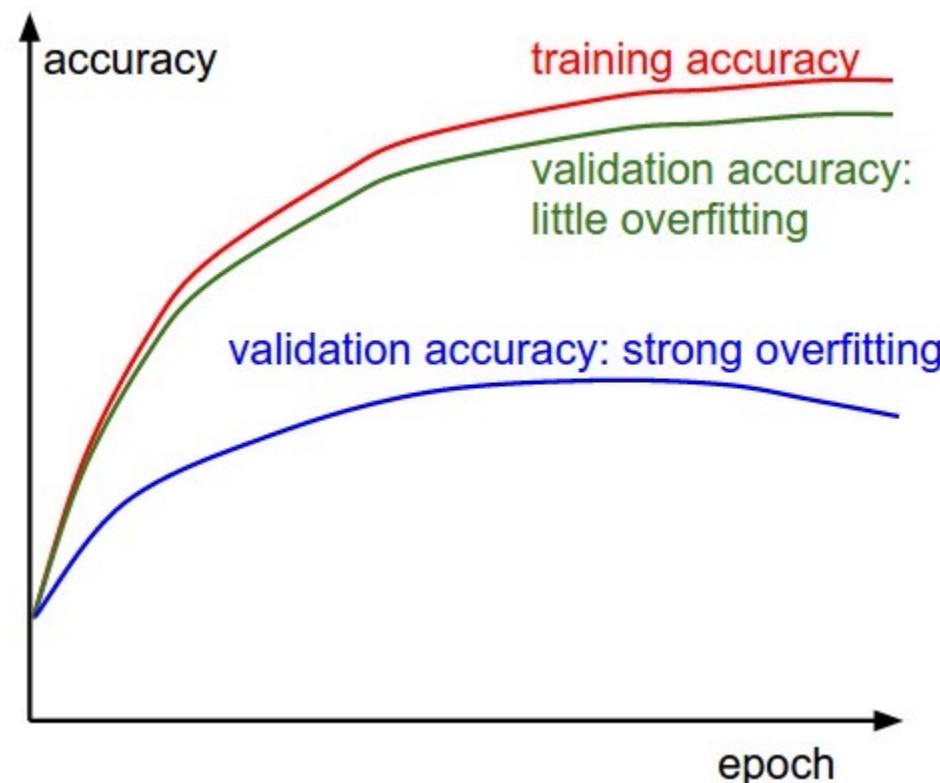
Better optimization algorithms  
help reduce training loss



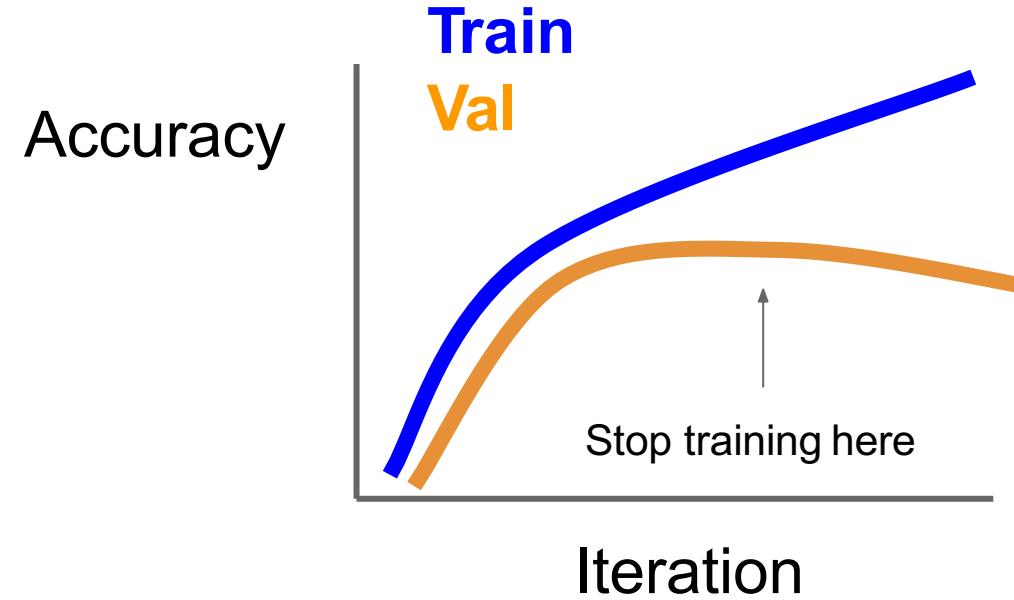
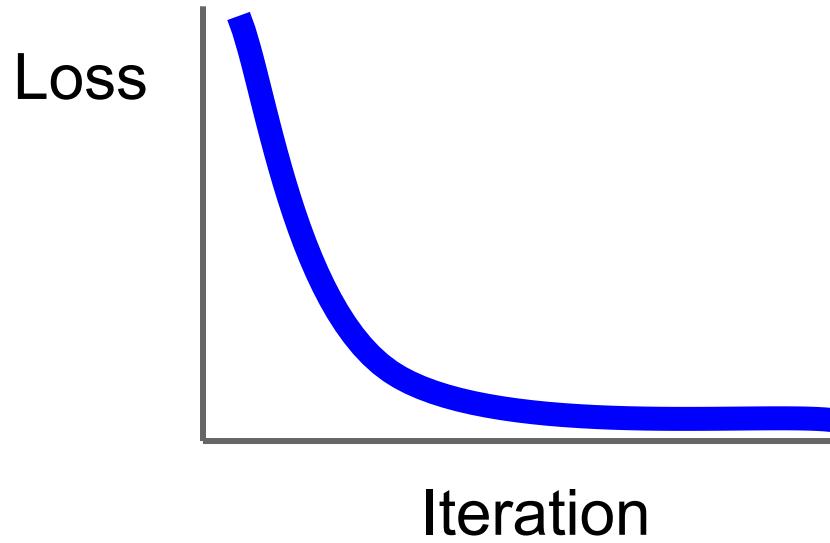
But we really care about error on  
new data - how to reduce the gap?

# Overfitting

- The gap between the training and validation accuracy indicates the amount of overfitting.

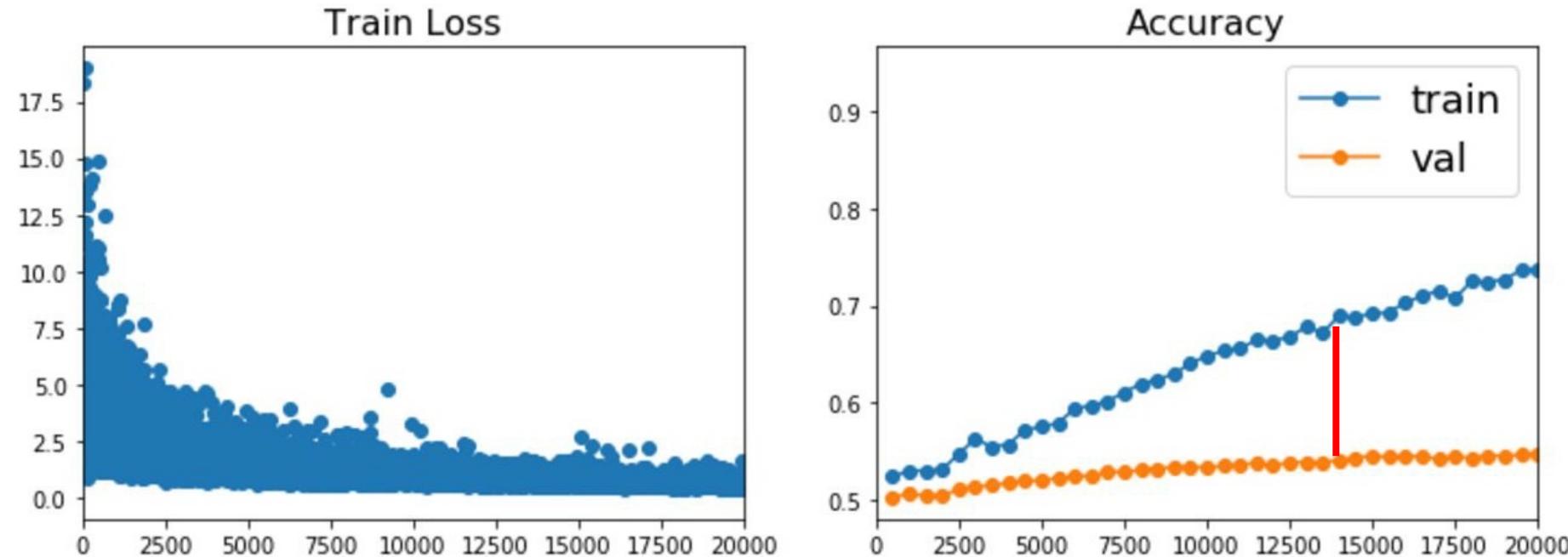


# Early Stopping: Always do this



Stop training the model when accuracy on the validation set decreases  
Or train for a long time, but always keep track of the model snapshot  
that worked best on val

# How to improve single-model performance?



Regularization

# Regularization: reduce overfit

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use: **L2 regularization**  
**L1 regularization**  
Elastic net (L1 + L2)

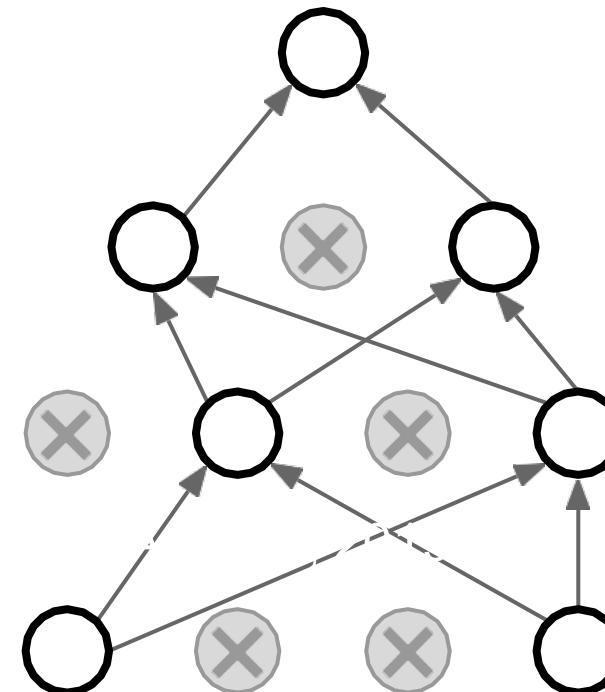
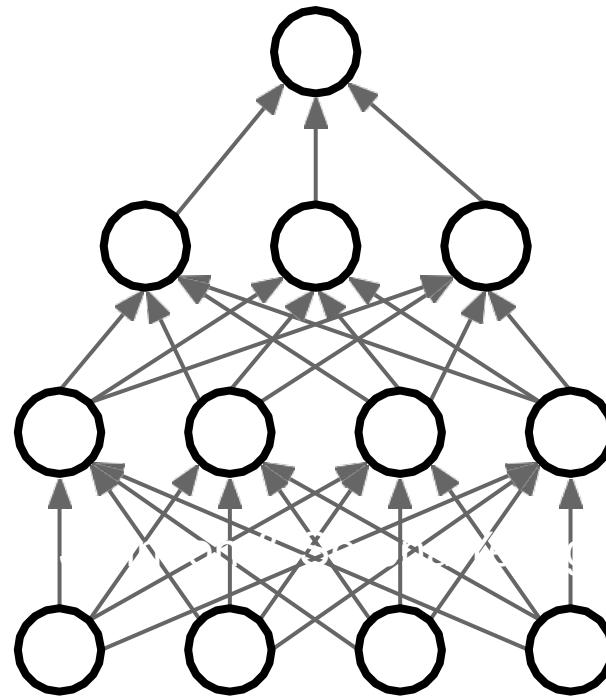
$$R(W) = \sum_k \sum_l W_{k,l}^2 \quad (\text{Weight decay})$$

$$R(W) = \sum_k \sum_l |W_{k,l}|$$

$$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$$

# Regularization: Dropout

In each forward pass, randomly set some neurons to zero. Probability of dropping is a hyperparameter; 0.5 is common.

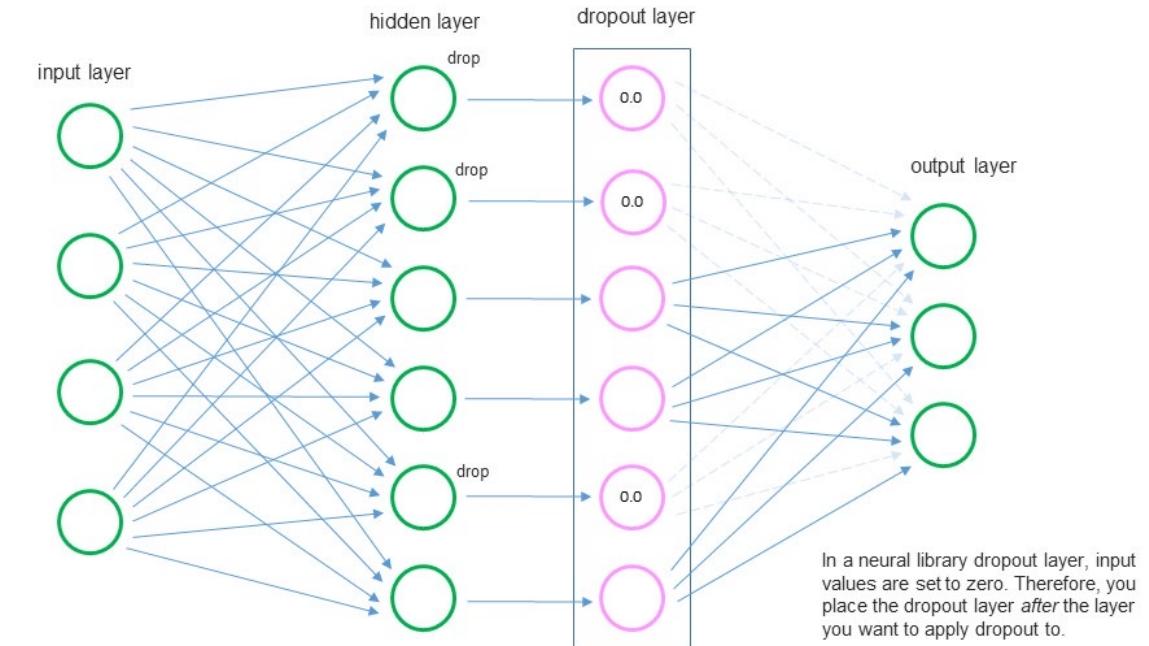


# Dropout

## • Dropout

- Dropout is a solution proposed to this problem by Nitish Srivastava, Geoffrey Hinton and few other students at the University of Toronto in 2012. Hinton is now an employee at Google, leading to the giant picking up the patent for the technology.

• <https://patents.google.com/patent/WO2014105866A1/en>



OPINIONS

## Google Activates ‘Dropout’ Patent, Neural Network Engineers To Be On Alert?

ANIRUDH VK - 3 WEEKS AGO

# Sample Code

- Repository
  - <https://github.com/lkk688/DeepDataMiningLearning>
- Tensorflow introduction
  - <https://colab.research.google.com/drive/188d4pSon4mSAzhGG54zXjWctTOo7Ds53?usp=sharing>
- Pytorch introduction
  - <https://colab.research.google.com/drive/1KZKXqa8FkaJpruUI1XzE7vjb4pHfMoS?usp=sharing>

# Thank You



**Address:**  
ENG257, SJSU



**Email Address:**  
[Kaikai.liu@sjsu.edu](mailto:Kaikai.liu@sjsu.edu)