Spring 2024

CMPE 258-01 Deep Learning

Dr. Kaikai Liu, Ph.D. Associate Professor

Department of Computer Engineering

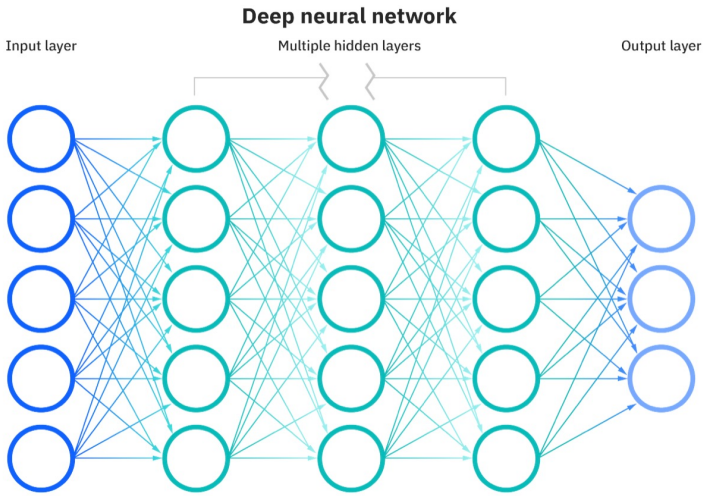
San Jose State University

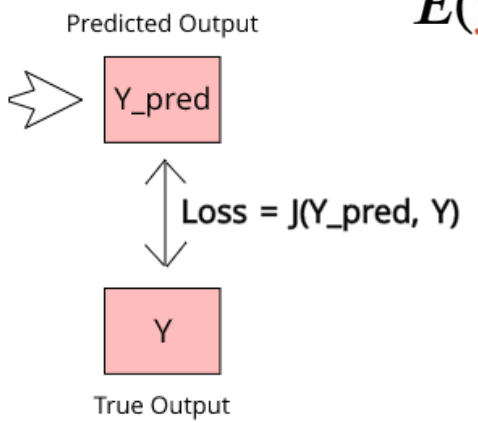
Email: kaikai.liu@sjsu.edu

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.





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

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

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

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

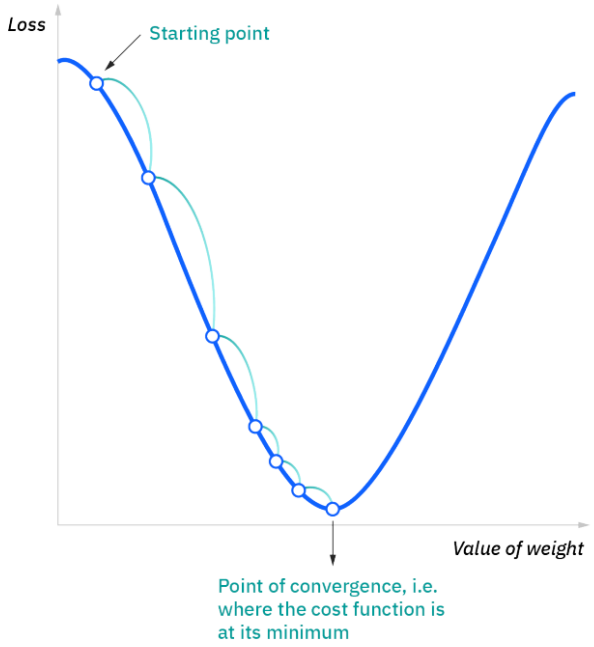
For τ from 0 to convergence:

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

(Evaluated at $\theta=\theta^{(\tau)}$)

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

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

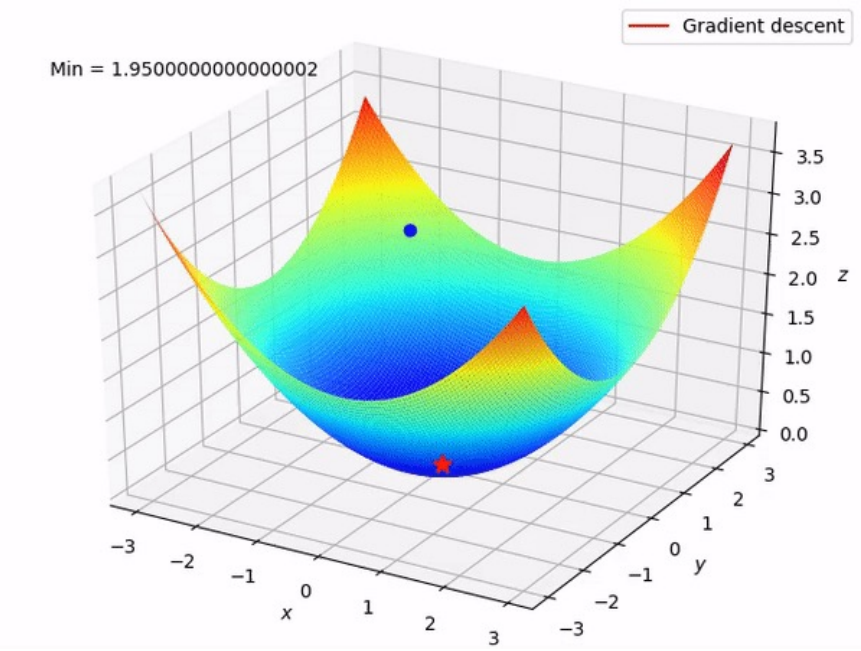
- Draw a simple random sample of data indices
  - Often called a **batch** or **mini-batch**
  - Choice of **batch size** trade-off **gradient quality** and **speed**
- Compute **gradient estimate** and uses as **gradient**

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

For τ from 0 to convergence:

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

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



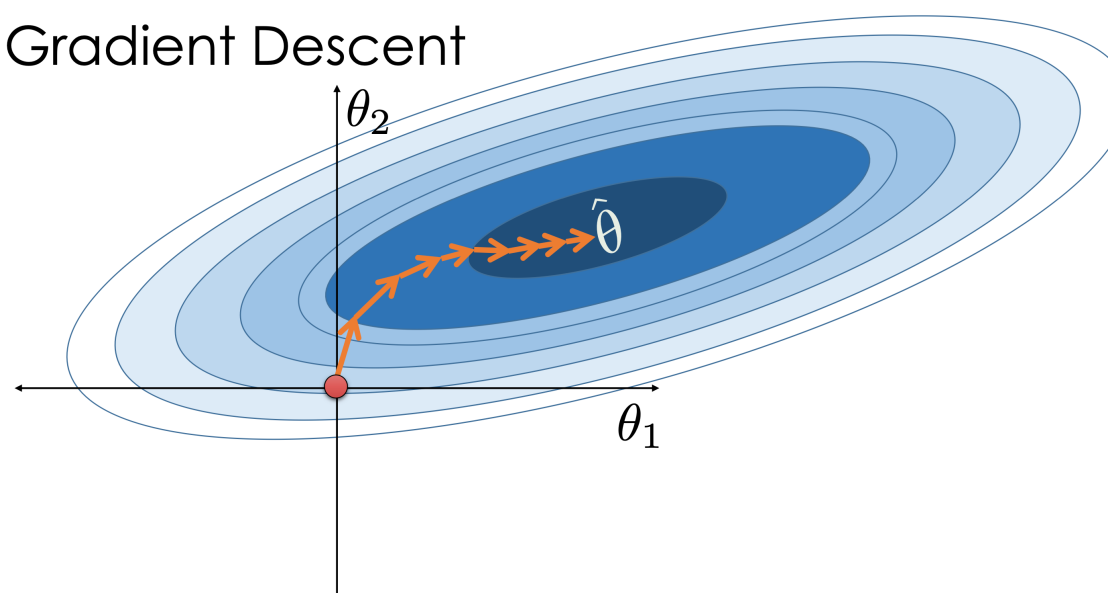
- Loss can be written as a sum of the loss on each record.

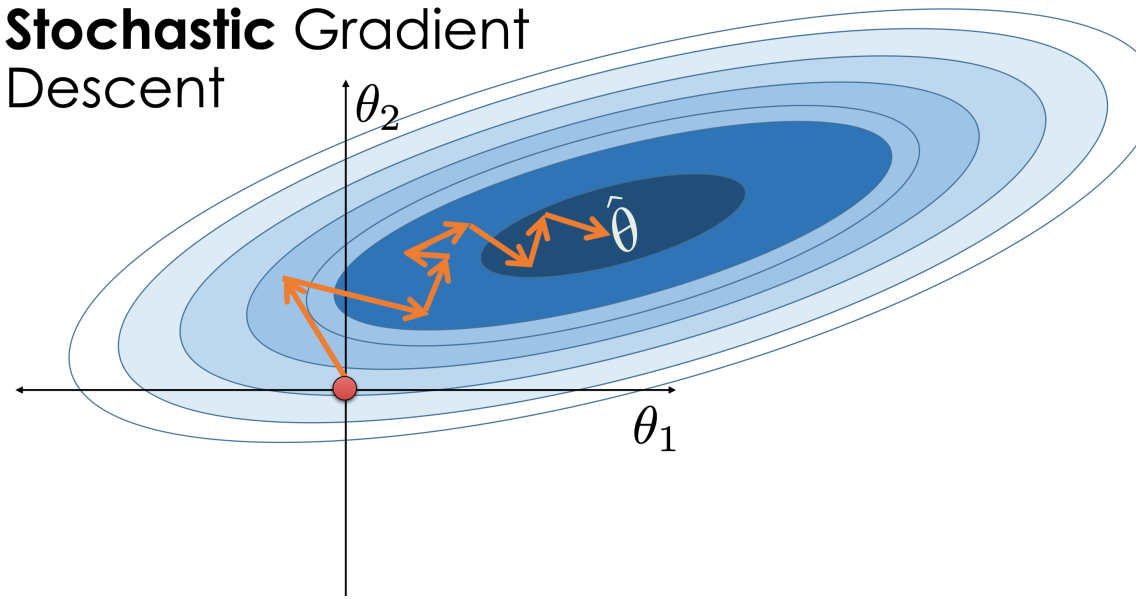
Decomposable Loss

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

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





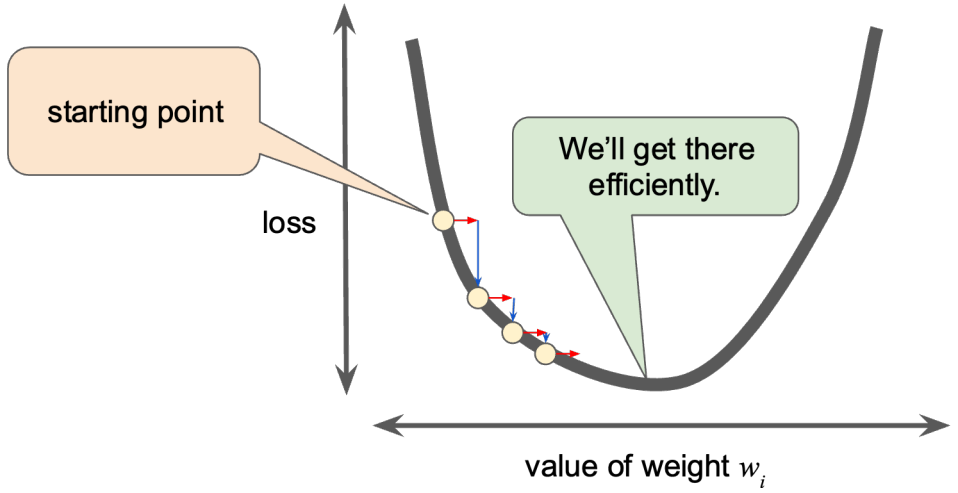
Learning Rate

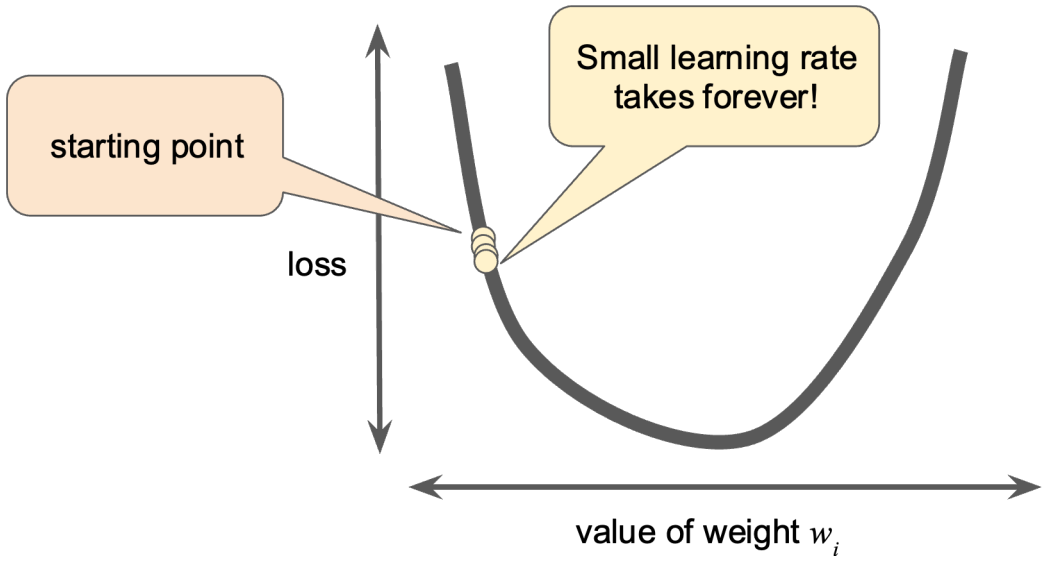
- Learning rate

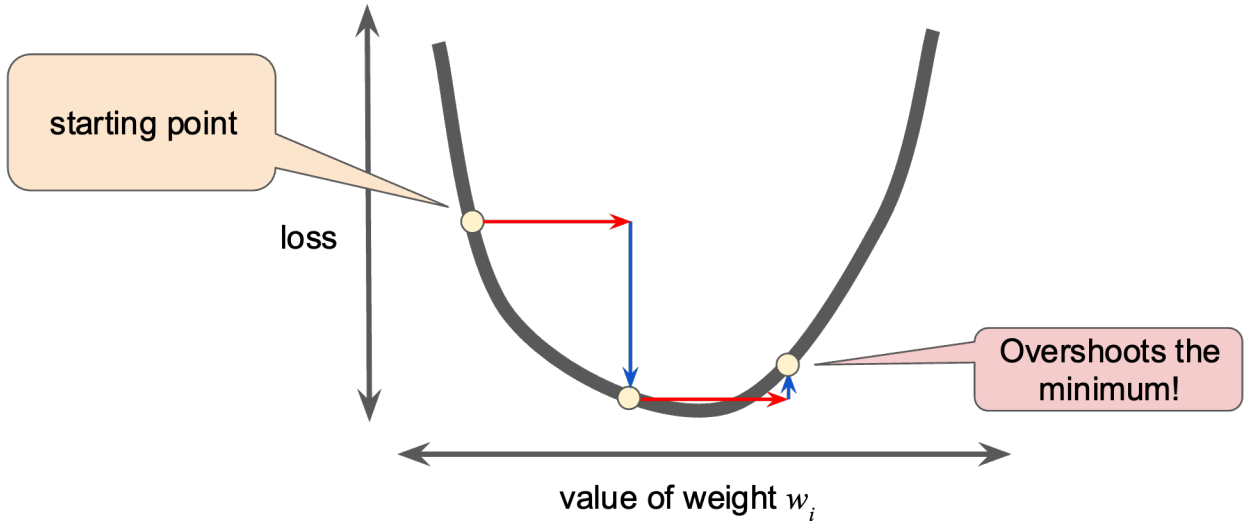
Repeat until convergence {

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

}



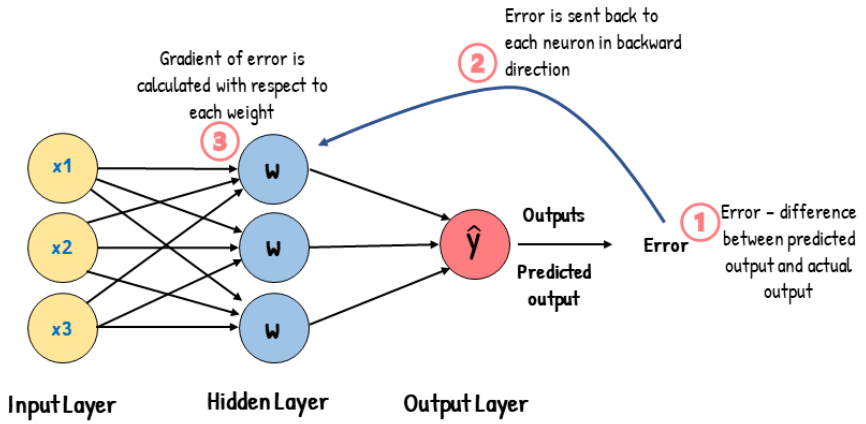




Neural Network Training

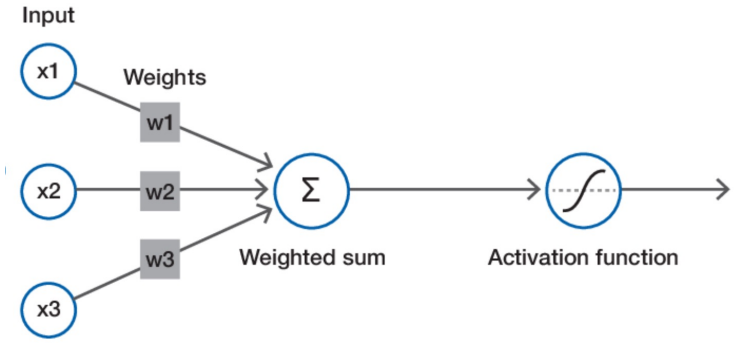
- Stochastic Gradient Descent With Back-propagation
  - Stochastic gradient descent is an optimization algorithm for minimizing the loss of a predictive model with regard to a training dataset.
  - Back-propagation is an automatic differentiation algorithm for calculating gradients for the weights in 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.

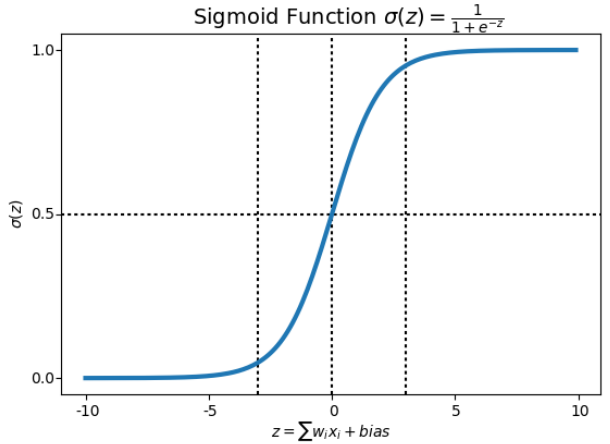
Backpropagation



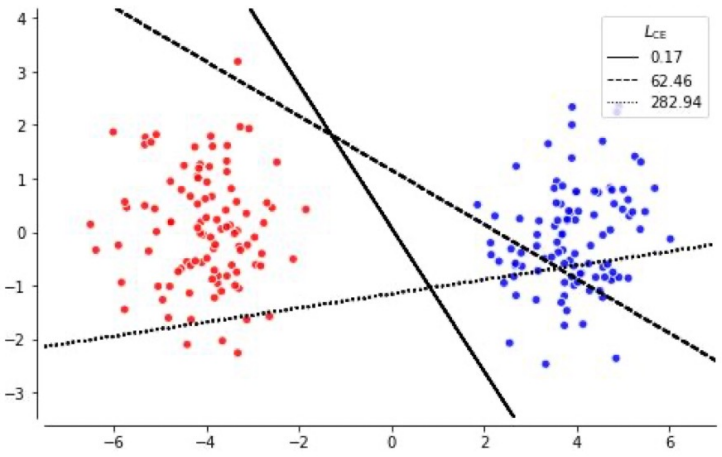
Neural Classifier

- 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



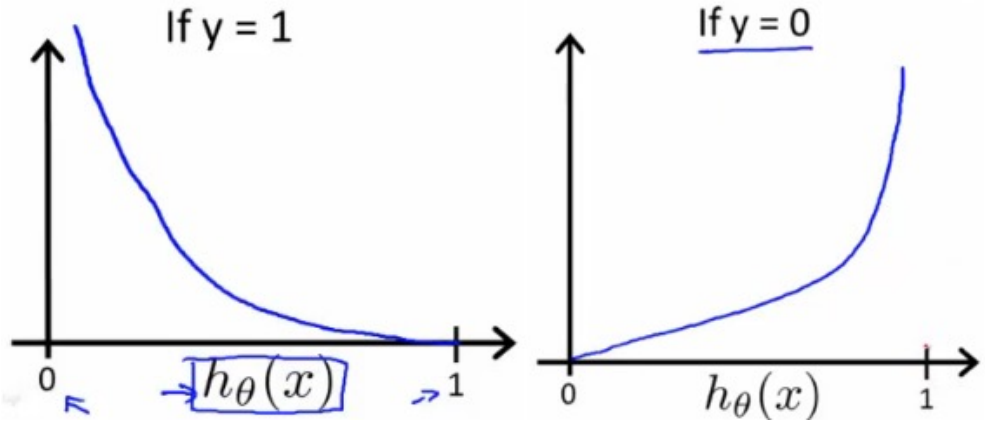


$$L_{\mathrm{CE}}(\mathbf{p}, \mathbf{t}) = -\sum_{i=1}^{n} [\mathbf{t}^{(i)} \log \mathbf{p}^{(i)} + (1 - \mathbf{t}^{(i)}) \log(1 - \mathbf{p}^{(i)})]$$



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



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

  - 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

$$\begin{aligned} &Repeat \ \{ \\ &\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \\ &\} \end{aligned} \quad \Rightarrow \quad \begin{aligned} &Repeat \ \{ \\ &\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \\ &\} \end{aligned}$$

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.

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

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.

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

- 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

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

$$\frac{\partial L}{\partial o_i} = -\sum_k y_k \frac{\partial log(p_k)}{\partial o_i}$$

$$= -\sum_k y_k \frac{\partial log(p_k)}{\partial p_k} \times \frac{\partial p_k}{\partial o_i}$$

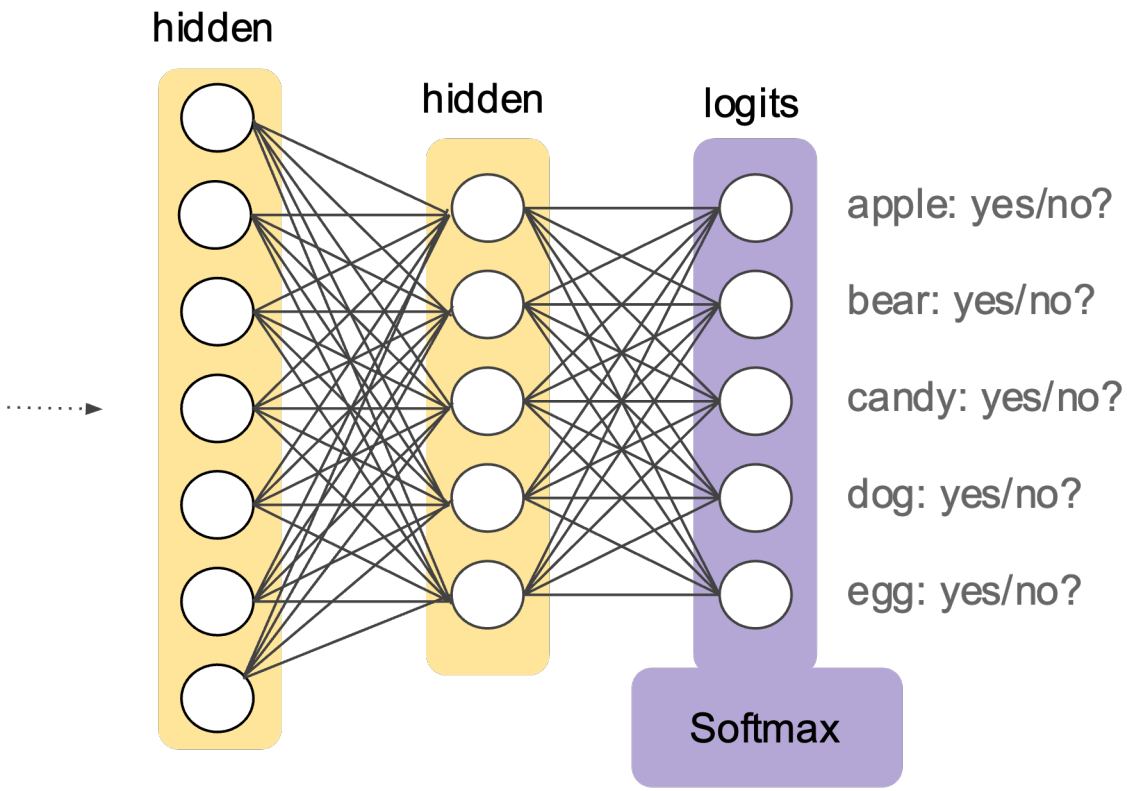
$$= -\sum y_k \frac{1}{p_k} \times \frac{\partial p_k}{\partial o_i}$$

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

which is a very simple and elegant expression

Multi-Class Neural Networks

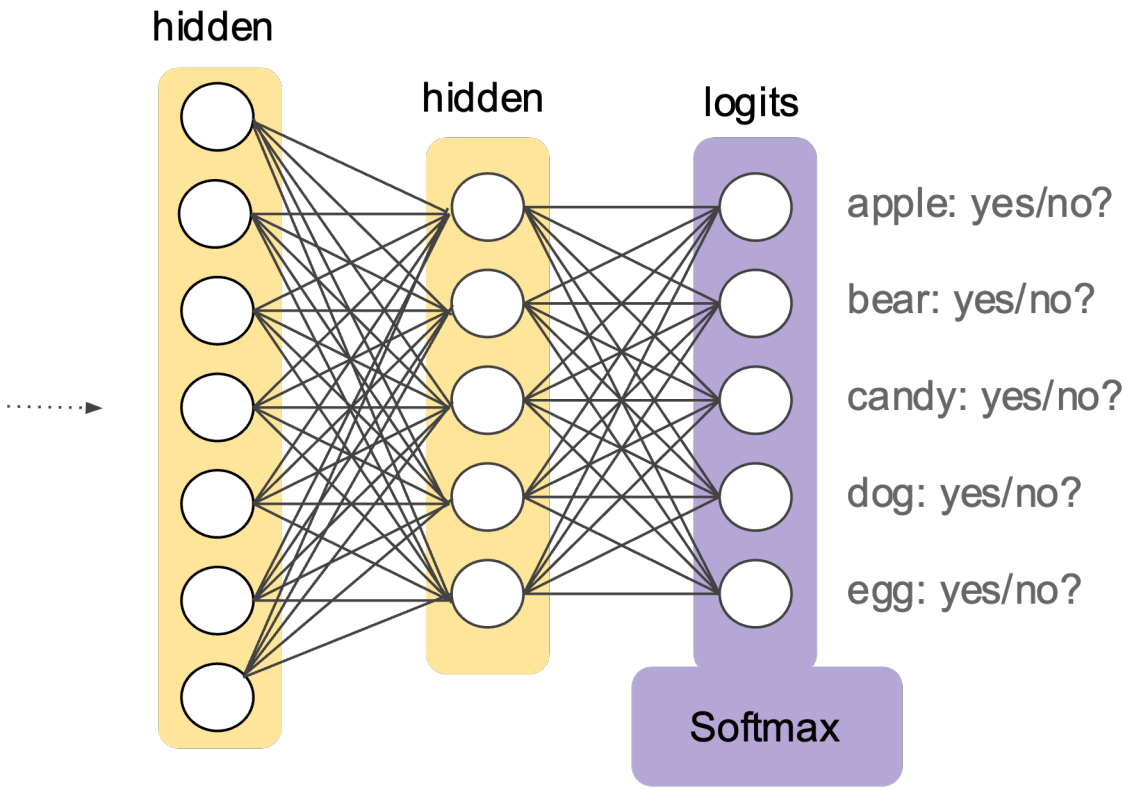
- Given a classification problem with N possible solutions, a one-vs.-all solution consists of N separate binary classifiers—one binary classifier for each possible outcome.
  - During training, the model runs through a sequence of binary classifiers, training each to answer a separate classification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



Multi-Class Neural Networks

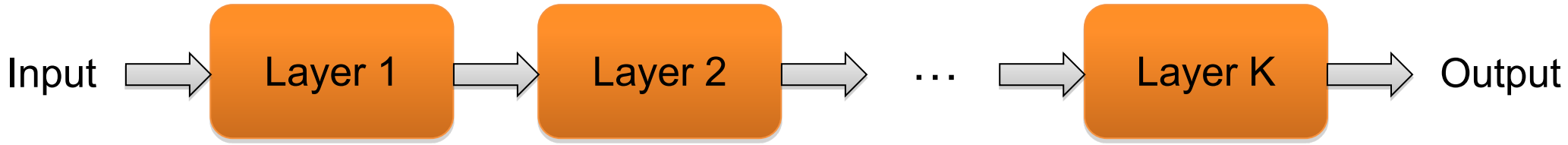
- **Softmax** extends the probability idea into a multi-class world.
  - Softmax assigns decimal probabilities to each class in a multi-class problem.
  - Those decimal probabilities must add up to 1.0.
  - This additional constraint helps training converge more quickly than it otherwise would.

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

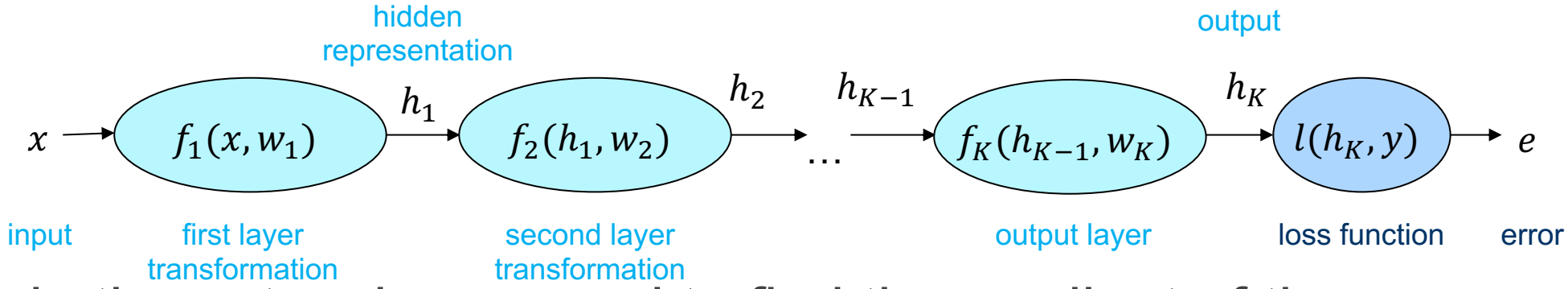


Neural Network Training

- 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

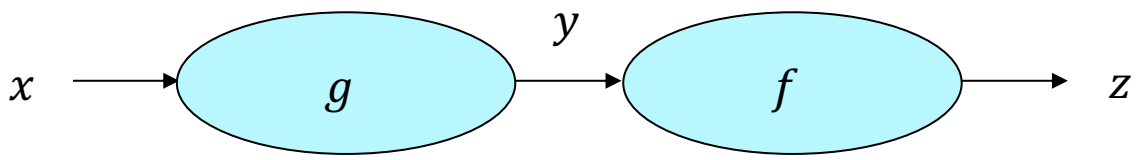


- To train the network, we need to find the gradient of the error w.r.t. the parameters of each layer

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

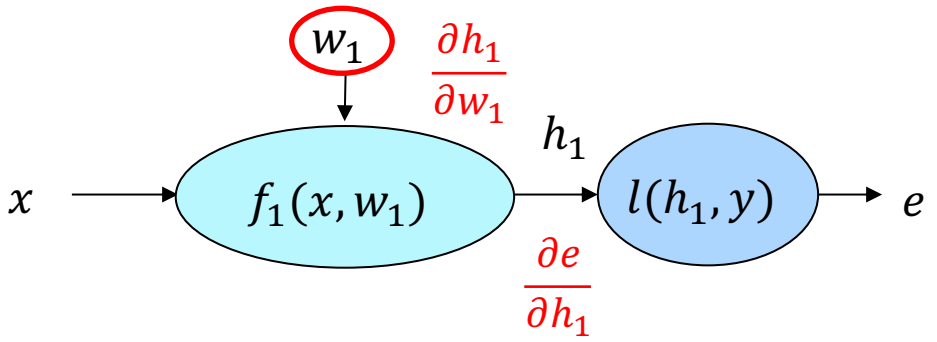
Neural Network Training

- Chain rule



$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Applying the chain rule
  - Start with k=1



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

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

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

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

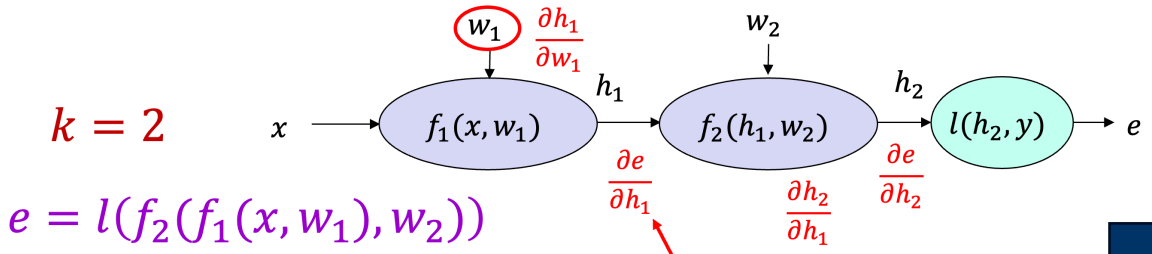
$$\frac{\partial h_1}{\partial w_1} = x$$

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

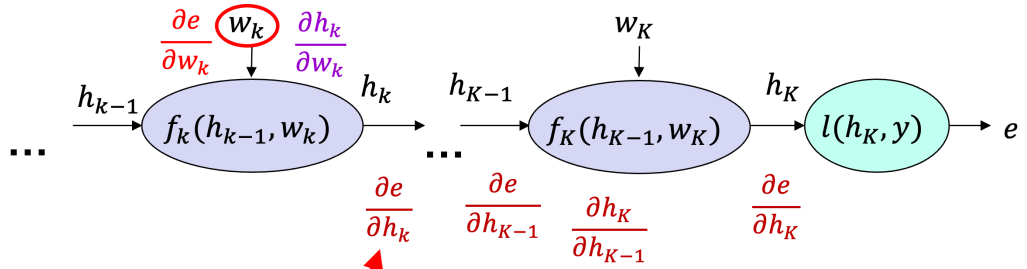
$$\frac{\partial e}{\partial w_1} = \frac{\partial e}{\partial h_1}\frac{\partial h_1}{\partial w_1} = -2x(y - w_1^T x)$$

Neural Network Training

- Applying the chain rule
  - k=2



$$\frac{\partial e}{\partial w_1} = \boxed{\frac{\partial e}{\partial h_2}\frac{\partial h_2}{\partial h_1}}\frac{\partial h_1}{\partial w_1}$$



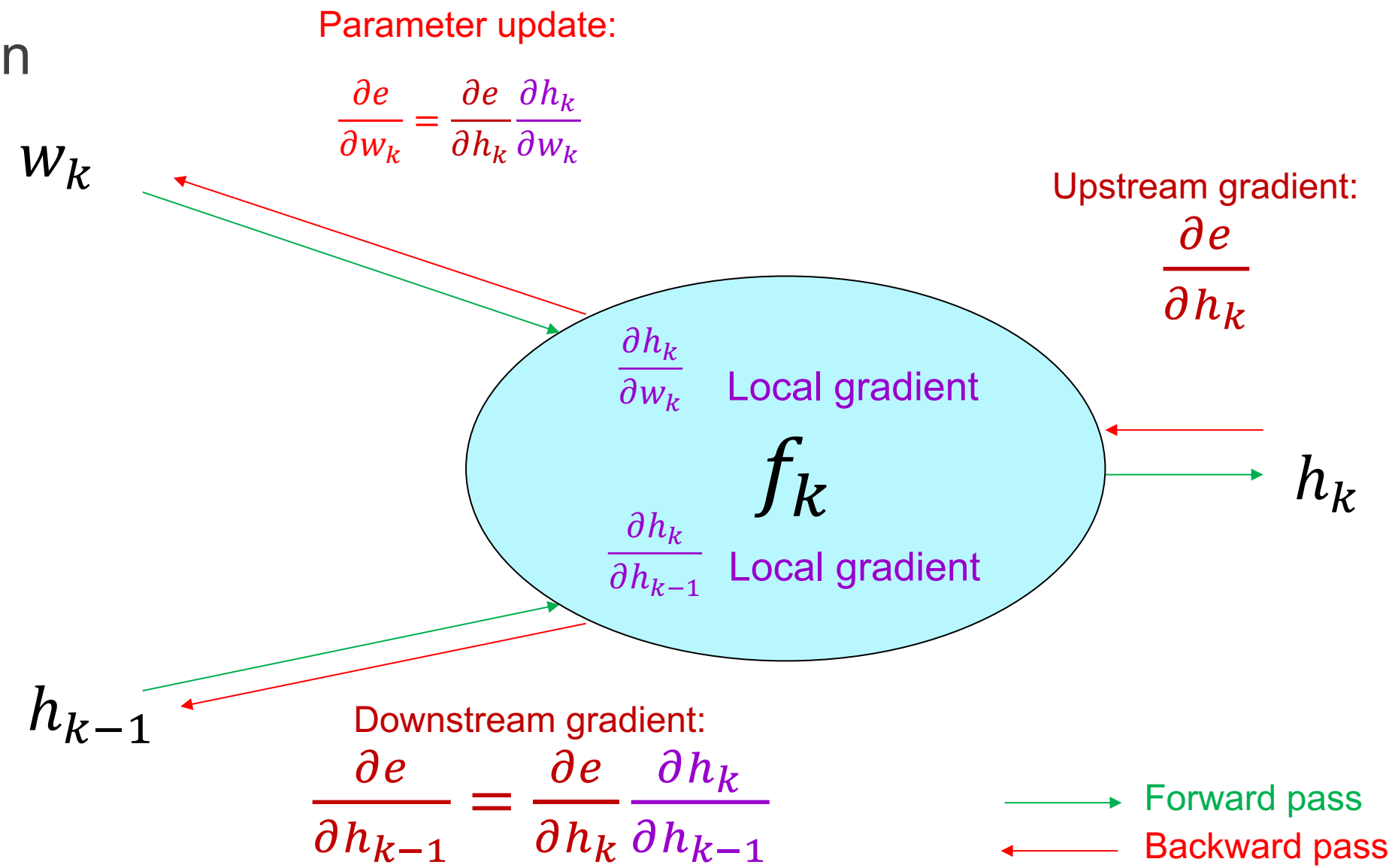
$$\frac{\partial e}{\partial w_k} = \boxed{\frac{\partial e}{\partial h_K}\frac{\partial h_K}{\partial h_{K-1}}\cdots\frac{\partial h_{k+1}}{\partial h_k}}\frac{\partial h_k}{\partial w_k}$$

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

Local gradient

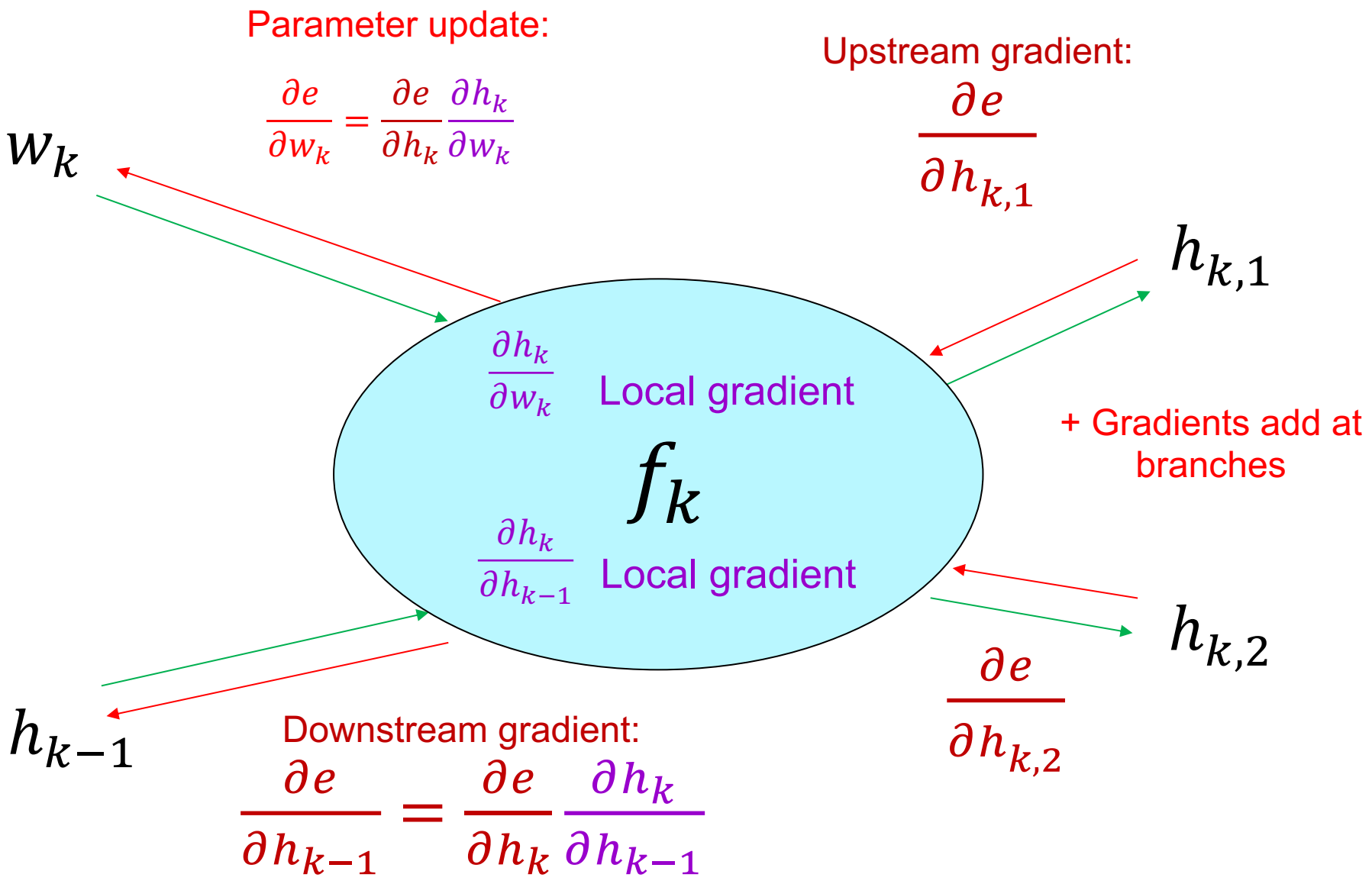
Neural Network Training

- Backpropagation



Neural Network Training

- With branches
  - E.g., Resnet



Thank You

Address:
ENG257, SJSU

Email Address:
Kaikai.liu@sjsu.edu