CS 615 – Deep Learning

Objective Functions & Gradient Rules

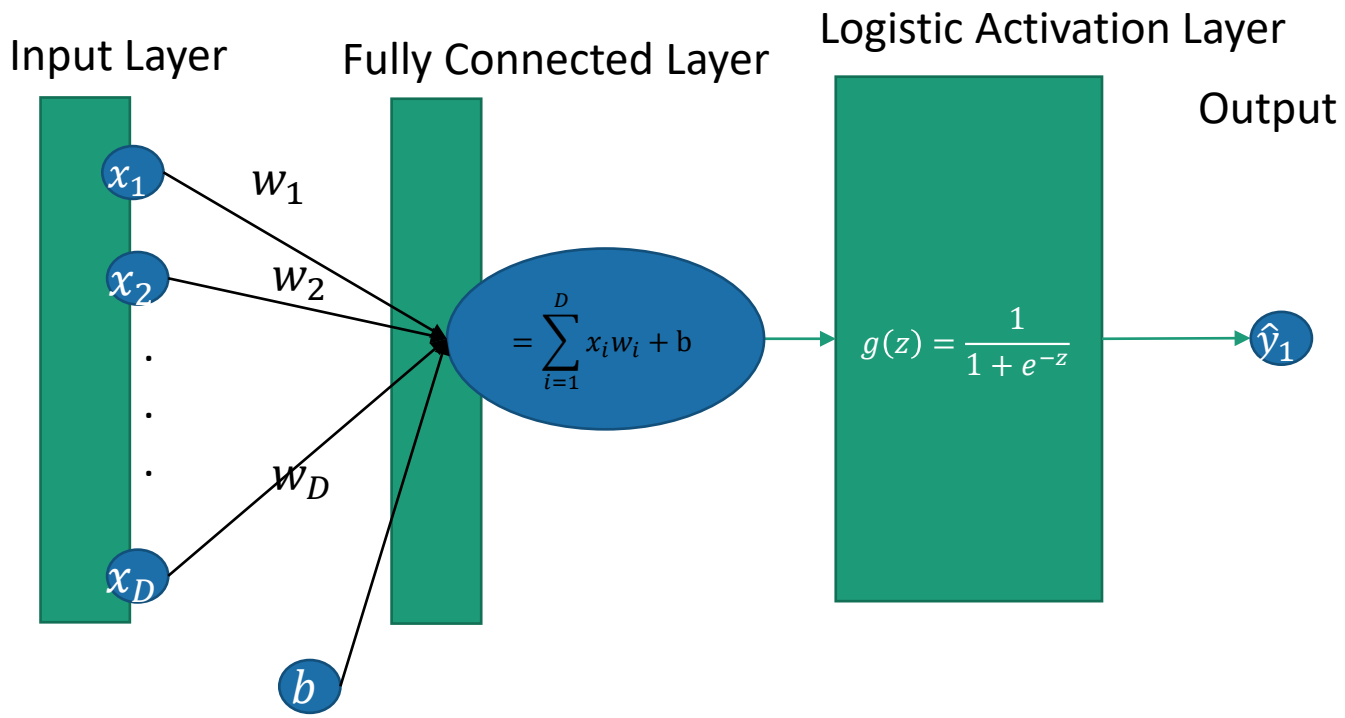
Slides adapted from material created by E. Alpaydin
Prof. Mordohai, Prof. Greenstadt, Pattern Classification (2nd Ed.),
Pattern Recognition and Machine Learning

Objectives

- Objective Functions
- Gradient Rules

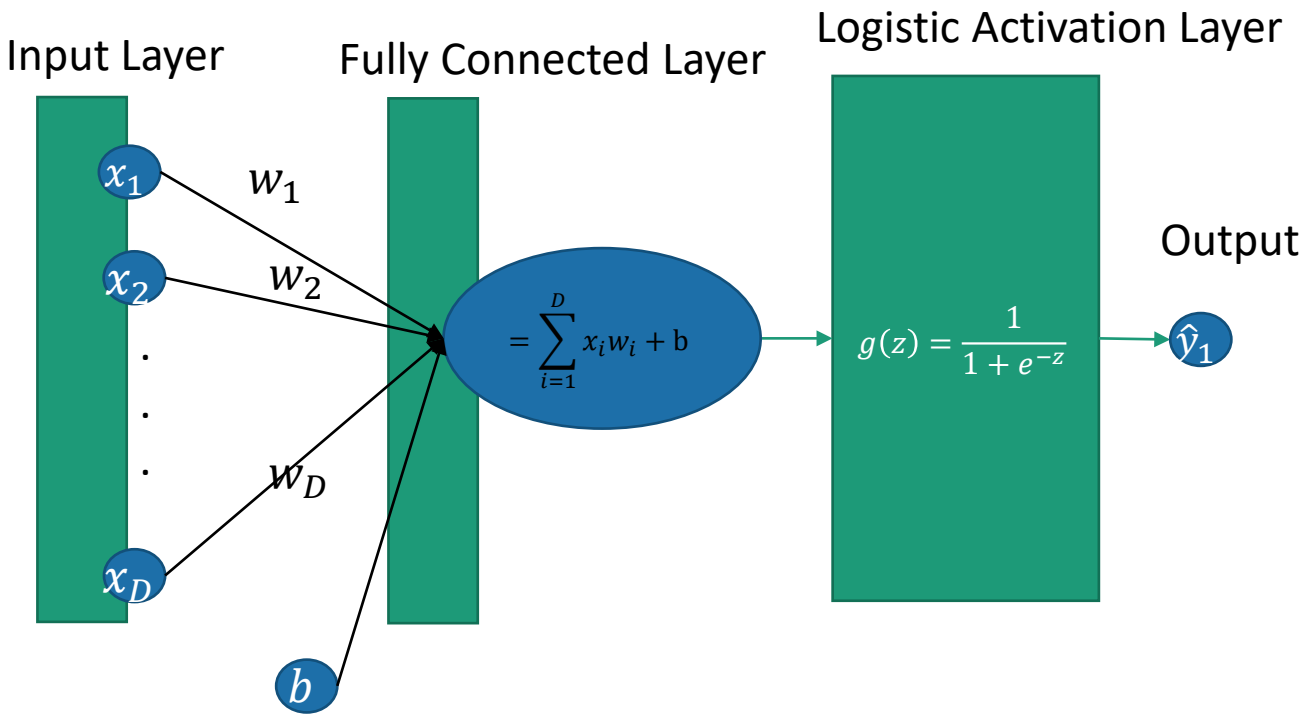
Learning

- Ok, so now we have this idea of deep learning architecture including several potential types of layers
  - Input
  - Fully Connected
  - Activation
- Where does the *learning* come in?



Learning

- To learn we must have:
  - A goal
  - Data
- And what we learn is the value of the weights (and biases) in the layers, as needed to best reach our goal.
- In this example the weights to be learned are all in the fully connected layer as $w = [w_1, w_2, \dots, w_D]^T, b$
- So, what are common "goals"?



Objective Functions

- To learn we must be targeting maximizing or minimizing some value.
- We call the function that we're attempting to minimize or maximize the *objective function.*
- If our objective function is something we're attempting to minimize, this is often referred to as our *loss* function.
- There are several commonly used objective functions.
- Let's look at a few.

Squared Error Objective Function

- A commonly used objective function when our target values are continuous and unbounded, is the *squared error.*
- If our target value is y and our estimated value is $\hat{y}$ then the squared error is defined as:

$$J = (y - \hat{y})^2$$

- The process of finding the weights that minimize a squared error objective function is called *least squares estimate (LSE)*

Likelihood Objective Function

- If our target value is a probability, then perhaps we want to maximize the probability of the correct class.
- Let y be the correct binary class $y \in (0,1)$ and $\hat{y}$ is the probability of our observation coming from class 1.
- Then
  - If $y = 1$ we want $\hat{y}$ to be as large as possible (ideally 1).
  - If $y = 0$ then we want $\hat{y}$ to be as small as possible (ideally 0)
    - Or $(1 - \hat{y})$ to be as large as possible.

Likelihood Objective Function

- We could then have our objective function be case-based:

$$J = \begin{cases} \hat{y}, & y = 1 \\ 1 - \hat{y}, & y = 0 \end{cases}$$

- Or we can conveniently write this as a single expression:

$$J = \hat{y}^{y}(1 - \hat{y})^{1-y}$$

- This expression is called the *likelihood*

Log Likelihood/Loss Objective Function

$$\boxed{J = \hat{y}^{y}(1-\hat{y})^{1-y}}$$

- To avoid exponents and products in this expression, its common to take the log of it:

$$J = \ln(\hat{y}^{y}(1-\hat{y})^{1-y}) = yln(\hat{y}) + (1-y)\ln(1-\hat{y})$$

- This objective function is called the *log likelihood* and the process of finding the weights to *maximize* it is called the *log likelihood estimate (LLE).*
- It is also common to negate this expression in order to have a cost function:

$$J = -(yln(\hat{y}) + (1-y)\ln(1-\hat{y}))$$

- Now we all this objective function the *log loss.*

Log Loss Objective Function

$$J = -(yln(\hat{y}) + (1 - y)\ln(1 - \hat{y}))$$

- Since the log loss function involves logs, and $\log(0) = -\infty$, it is typically to add in a small *numeric stability* constant within the logs, say $\epsilon = 10^{-7}$

$$J = -(yln(\hat{y} + \boldsymbol{\epsilon}) + (1 - y)\ln(1 - \hat{y} + \boldsymbol{\epsilon}))$$

Cross Entropy Objective Function

- And finally, if our target output is a distribution, it is common to use the cross-entropy objective function.
- From basic statistics, entropy of a distribution y is computed as:

$$J = -\sum_{k=1}^{K} y_k \ln(y_k)$$

  - Note: For the entropy to be $[0,1]$ the base of the log should be k, but since minimizing one base minimizes the other, we'll just stick with the natural log.
- With cross entropy we are computing the entropy between two distributions, y and $\hat{y}$

$$J = -\sum_{k=1}^{K} y_k \ln(\hat{y}_k)$$

Cross Entropy Objective Function

$$J = -\sum_{k=1}^{K} y_k \ln(\hat{y}_k)$$

- It is worth noting that now both the target and the estimation are *vectors* (probability distributions).
- Thus, we can write this as:

$$J = -y \ln\left(\hat{y}^T + \epsilon\right)$$

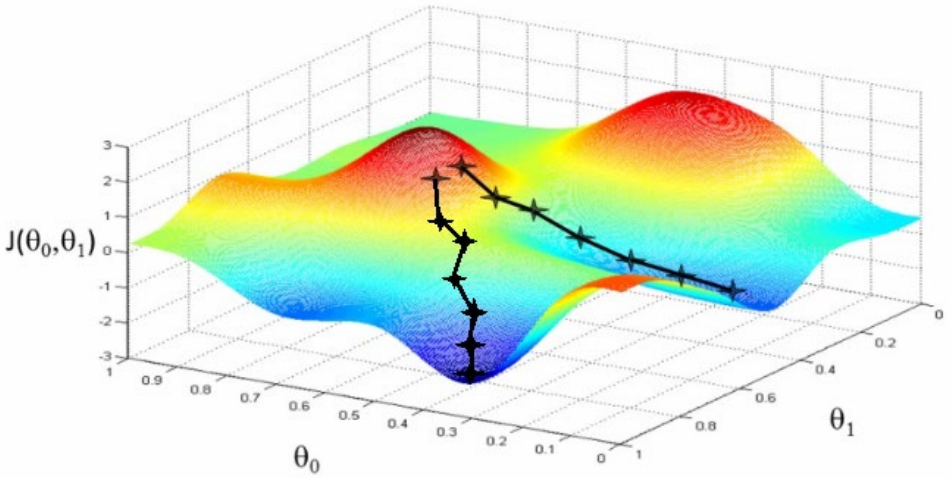
Numeric instability warning!!!!

Gradient-Based Learning

- Ok, so given an architecture and a chosen objective function, how do we find the weights to minimize it or maximize it?
- In this course we'll going to use a technique called *gradient ascent/descent*
- The *gradient* of a function is its slope with respect to a variable.
- Recall that we can find the slope of a function by taking its derivative.
- Therefore, to find the gradient of an objective function with regards to one of our unknowns (weights), we need to determine/compute *partial gradients.*

Gradient Ascent/Descent

- The general idea is, until convergence
  - Computer the partial gradients with respect to each unknown.
  - "Move" the value of the unknowns by some amount of their gradients.



The graphic depicts gradient assent with two different initial values of (θ_0, θ_1) and updating each parameter simultaneously

Gradient Learning

- So, for each set of weight, w, we need to compute the gradient of our objective function with regards to that weight:

$$\frac{\partial J}{\partial w}$$

- We could write this function for each set of weights and take the derivative of it.
- For instance, imagine we have an input layer, connected to a fully connected layer (with weights and bias w, b), followed by a logistic activation function.
- Our output is then:

$$\hat{y} = \frac{1}{1 + e^{-(xw+b)}}$$

Gradient Learning

$$\hat{y} = \frac{1}{1 + e^{-(xw+b)}}$$

- If our activation function is the log loss, we then have:

$$J = -\left(yln\left(\frac{1}{1 + e^{-(xw+b)}} + \boldsymbol{\epsilon}\right) + (1 - y)\ln\left(1 - \frac{1}{1 + e^{-(xw+b)}} + \boldsymbol{\epsilon}\right)\right)$$

- Now we can compute $\frac{\partial J}{\partial w}$
- Hopefully you can imagine how ugly this gets as our architectures get deeper and more complicated….

Chain Rule

- Fortunately, we can leverage the *chain rule* from calculus to allow us to compute *per-module* gradients and chain them together to form our final gradient rules!
- Recall, the chain rule:

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

- Let's see how this applies to our problem….

Chain Rule

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

- Imagine we need $\frac{\partial J}{\partial w}$ and that
  - Our output, $\hat{y}$, was created using input data in
  - And input data in was created using the weights w
- Using the chain rule, we can write $\frac{\partial J}{\partial w}$ as:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial in} \cdot \frac{\partial in}{\partial w}$$

Backpropagation

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial in} \cdot \frac{\partial in}{\partial w}$$

- In fact, we can start at the output $\hat{y}$ and *accumulate* the gradient as we go *back* from the output, to our weight(s).
- Let δ be our current gradient.
- Initial $\delta = \frac{\partial J}{\partial \hat{y}}$
- Then $\delta = \delta \cdot \frac{\partial \hat{y}}{\partial in}$
- And finally, $\frac{\partial J}{\partial w} = \delta \cdot \frac{\partial in}{\partial w}$
- This idea of keeping track of the accumulated gradient as we move *back* from the output is aptly called ***backpropagation***.

Backpropagation

- So, to use backpropagation with the chain rule, we're going to need partial gradients for our modules.
- In particular we'll need:
  - The gradient of our objective functions with regards to output $\hat{y}$
  - The output of each activation layer's output, with respect to its input.
  - The output of each fully connected layer's output, with respect to its input.

Gradients

- Most resources use various simplifications of the equations (without showing the steps, and often making assumptions) that lead to computational efficiency.
- Here we'll show the entire process, and mention areas where there can be simplifications.

SE Gradient

- Let's start with the gradients of our objective functions with regards to their inputs!
- Recall that the squared error is:

$$J = (y - \hat{y})^2$$

- What is $\frac{\partial J}{\partial \hat{y}}$ for this?

$$\frac{\partial J}{\partial \hat{y}} = -2(y - \hat{y})$$

Log Loss Gradient

- Recall that the log loss for a single observation is:

$$J = -\left((yln(\hat{y}) + (1-y)\ln(1-\hat{y}))\right)$$

- What is $\frac{\partial J}{\partial \hat{y}}$ for this?

$$\frac{\partial J}{\partial \hat{y}} = -\frac{y-\hat{y}}{\hat{y}(1-\hat{y})+\epsilon}$$

Numeric instability warning!!!!

Cross Entropy Gradient

- Cross entropy is a bit different in it involves multiple outputs (**and therefore will have a gradient that is a vector**).
  - This idea hold true for multi-output squared error or log loss objective functions as well.
- Recall the objective function:

$$J = -\sum_{k=1}^{K} y_k \ln(\hat{y}_k + \epsilon) = -yln\left(\hat{y}^T + \epsilon\right)$$

- The gradient is then:

$$\frac{\partial J}{\partial \hat{y}} = -\frac{y}{\hat{y} + \epsilon} \in \mathbb{R}^{1\times K}$$

Numeric instability warning!!!!

Cross Entropy Gradient

- It's worth noting that the Softmax objective function is typically used to generate the output when using a Cross Entropy objective function.
- Due to one-hot-encoding, most the target outputs are zero, allowing for the combination of the gradient from cross entropy and softmax to be greatly simplified (and more efficient).
- However, for the sake of generalizability, we will not be combining their gradients, but will process them independently.

Gradient: Activation Functions

- Next let's establish the gradients of the output of our activation functions with regards to their inputs.
- Here we're taking the derivative of a vector function.
  - K inputs and K outputs.
- The result is a *Jacobian matrix.*
  - Which, in this case, will be a $K \times K$ matrix.
  - Section 5.3 in the Mathematics for Machine Learning book is a good reference.

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

Gradient: Linear Activation

- Let's start with the linear activation layer:

$$g(z) = z$$

- What is $\frac{\partial g(z)}{\partial z}$?
- We need to compute $\frac{\partial g_j(z)}{\partial z_i} \forall i, j$ to create a *matrix* of gradients.
- Since $g_j(z)$ is only dependent on z_j, the gradient matrix will be all zeros except where $i = j$.
- And what will $\frac{\partial g_j(z)}{\partial z_j}$ be?
  - One!
- So, our matrix is just an identity matrix!

Gradient: ReLu Activation

- ReLu activation layer?

$$g_i(z) = \begin{cases} 0, & z_i < 0 \\ z_i, & z_i \geq 0 \end{cases}$$

- Likewise, the output $g_j(z)$ from a ReLu activation function only depends on z_j, so it too will be zeros everywhere except on the diagonal.

- And in this case, what are the values on the diagonal, $\frac{\partial g_j(z)}{\partial z_j}$?

$$\frac{\partial g_j(z)}{\partial z_j} = \begin{cases} 0, & z_j < 0 \\ 1, & z_j \geq 0 \end{cases}$$

Gradient: Logistic Activation

- Logistic Activation Layer?

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

- Likewise, the output $g_j(z)$ from a Logistic activation function only depends on z_j, so it too will be zeros everywhere except on the diagonal.
- On the diagonals we'll have (we'll work through this in class):

$$\frac{\partial g_j(z)}{\partial z_j} = g(z_j)\left(1 - g(z_j)\right) + \epsilon$$

Gradient: Tanh Activation

- Hyperbolic Tangent Activation Layer?

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

- The output $g_j(z)$ from a tanh activation function also only depends on z_j, so it too will be zeros everywhere except on the diagonal.
- On the diagonals we'll have:

$$\frac{\partial g_j(z)}{\partial z_j} = \left(1 - g_j^2(z)\right) + \epsilon$$

Gradient: Softmax Activation

- And finally, the softmax activation layer:

$$g(z) = \frac{e^z}{\sum_i e^{z_i}}$$

- This one is a bit trickier since any given output, $g_j(z)$, depends on all the inputs (due to the denominator).

Gradient: Softmax Activation

$$g(z) = \frac{e^z}{\sum_i e^{z_i}}$$

- So, what is $\frac{\partial g_j(z)}{\partial z_i}$?
- On the diagonals where $i = j$ we get:

$$\frac{\partial g_j(z)}{\partial z_j} = g_j(z)\left(1 - g_j(z)\right)$$

- How about off-diagonal (when $i \neq j$)?

$$\frac{\partial g_j(z)}{\partial z_i} = -g_i(z)g_j(z)$$

Gradient: Softmax Activation

$$g(z) = \frac{e^z}{\sum_i e^{z_i}}$$

- All together

$$\frac{\partial g_j(z)}{\partial z_i} = \begin{cases} g_j(z)\left(1 - g_j(z)\right), & i = j \\ -g_i(z)g_j(z), & i \neq j \end{cases}$$

- Or

$$\frac{\partial g_j(z)}{\partial z_i} = g_i(z)\left((i == j) - g_j(z)\right)$$

Chaining Check

- To verify the chain rule, we'll assume that there's an activation function that provides our final output $\hat{y}$
- If the input to that activation function is the vector h, we then can compute:

$$\frac{\partial J}{\partial h} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h}$$

- Verifying our dimensions….

$$\frac{\partial J}{\partial h} \in \left(\mathbb{R}^{1\times K}\right) \cdot \left(\mathbb{R}^{K\times K}\right) = \mathbb{R}^{1\times K}$$

Speed-Up

- It's worth nothing that for all the activation functions mentioned, other than softmax, are diagonal matrices.
- Therefore, we can just have a *vector* of the elements of the diagonal as our gradient for those, and apply the *element-wise (Hadamard)* product to the backpropagating gradient.

Gradient: Fully Connected Layer

- Our last layer is the fully-connected layer.
- Let our FC layer be a function $g(z): \mathbb{R}^{1\times D} \rightarrow \mathbb{R}^{1\times K}$
- What is the size of $\frac{\partial g(z)}{\partial z}$?
- A Jacobian matrix of size $\mathbb{R}^{K\times D}$

Gradient: Fully Connected Layer

- What are the elements of $\frac{\partial g(z)}{\partial z}$?
- Recall:

$$g(z) = zW + b$$

- What is $\frac{\partial g_j(z))}{\partial z_i}$?

$$\frac{\partial g_j(z)}{\partial z_i} = W_{ji}$$

- So, the entire Jacobian matrix $\frac{\partial g(z)}{\partial z}$ is just W^T!

Chaining Check

- To verify the chain rule, let's continue with our previous architecture, but now add on a FC layer.
- Let $h \in \mathbb{R}^{1\times K}$ be the output of the FC layer and $x \in \mathbb{R}^{1\times D}$ be the input to it.
- We then want

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h} \cdot \frac{\partial h}{\partial x}$$

- Verifying our dimensions….

$$\frac{\partial J}{\partial x} \in \left(\mathbb{R}^{1\times K}\right) \cdot \left(\mathbb{R}^{K\times K}\right) \cdot \left(\mathbb{R}^{K\times D}\right) = \mathbb{R}^{1\times D}$$

Backpropagation

- Now we have all the parts we'll need to do backprop!
- We start off our gradient at the output:

$$\frac{\partial J}{\partial \hat{y}}$$

- And then we keep updating it as we go backwards, towards the input layer.

Backpropagation

- For instance, the gradient of the objective function with regards to the input to the layer that computed $\hat{y}$, which I'll call $h^{(M)}$, is:

$$\frac{\partial J}{\partial h^{(M)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h^{(M)}}$$

- To compute the partial of the objective function with regards to input to the layer that created $h^{(M)}$ (which I'll call $h^{(M-1)}$) we have:

$$\frac{\partial J}{\partial h^{(M-1)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h^{(M)}} \cdot \frac{\partial h^{(M)}}{\partial h^{(M-1)}}$$

- Etc…

Note: If a layer's gradient is a vector (instead of the Jacobian matrix), multiplication is element-wise.

Backpropagation

$$\frac{\partial J}{\partial h^{(M-1)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h^{(M)}} \cdot \frac{\partial h^{(M)}}{\partial h^{(M-1)}}$$

- If we let δ *accumulate* the gradient as we back propagate, then this can be written as:
  1. $\delta = \frac{\partial J}{\partial \hat{y}}$
  2. $\delta \rightarrow \delta \cdot \frac{\partial \hat{y}}{\partial h^{(M)}}$
  3. $\delta \rightarrow \delta \cdot \frac{\partial h^{(M)}}{\partial h^{(M-1)}}$
- Which can lead us to some pseudo-code…

Let $\delta = \frac{\partial J}{\partial \hat{y}}$ //start gradient off as partial of objective function with regards to the output
Let $m = M$ //m is our current layer out of M total layers
while $m > 1$ //backprop until we hit the input layer
 //update the gradient using the partial of the output of layer m, $h^{(m)}$, with regards to its input $h^{(m-1)}$
 $\delta = \delta \cdot \frac{\partial h^{(m)}}{\partial h^{(m-1)}}$

Modules

- Now we want to implement our objective functions and add `gradient` methods to our other layers.
- The objective function classes should have:
  - A method that returns the current objective function's value, give a target value, and an output (estimated) value.
  - A method that returns the gradient of the objective function with regards to its input (the estimated values).

Modules

- Here's an example, in Python, for a `LeastSquares` objective function.

```
class LeastSquares():
  def eval(self,y,yhat):
    return (y - yhat)*(y - yhat)

  def gradient(self,y,yhat):
    return -2*(y-yhat)
```