DNN Introduction 2

Instructor: Xuechen Zhang

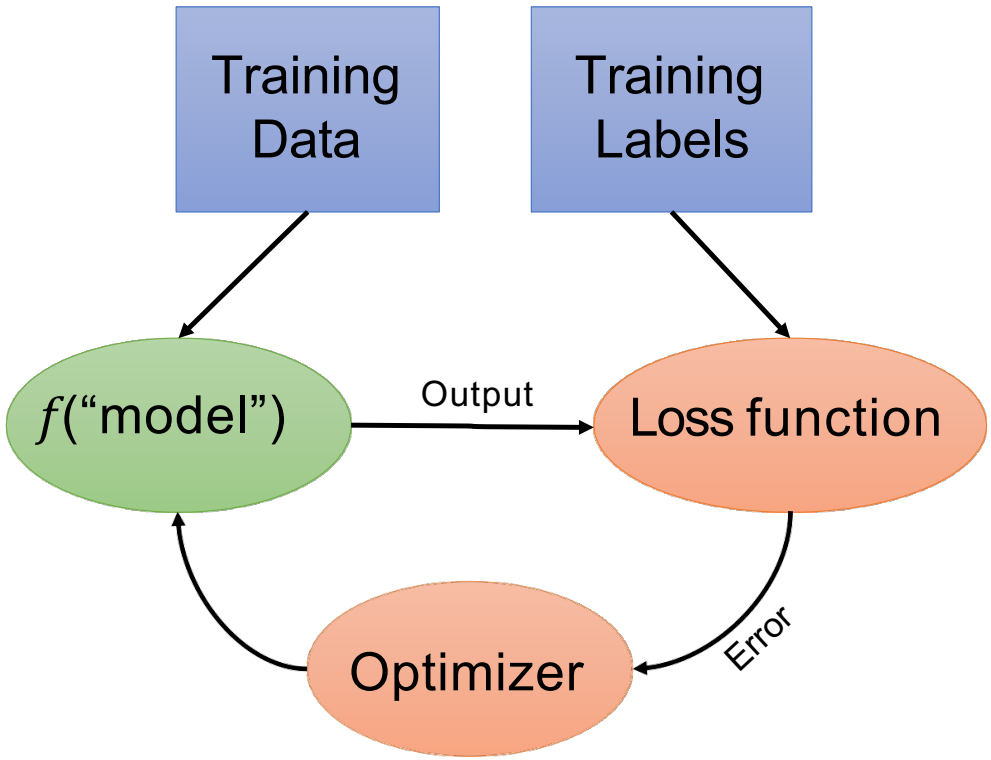
Washington State University Vancouver

Slides adapted from R. Singh@Brown and K. Wong@UTK

Outline

- **Model optimizer**
- Backpropagation and autodiff
- Multi-layer NNs and activation

Next critical ingredient for our new approach: Optimizer



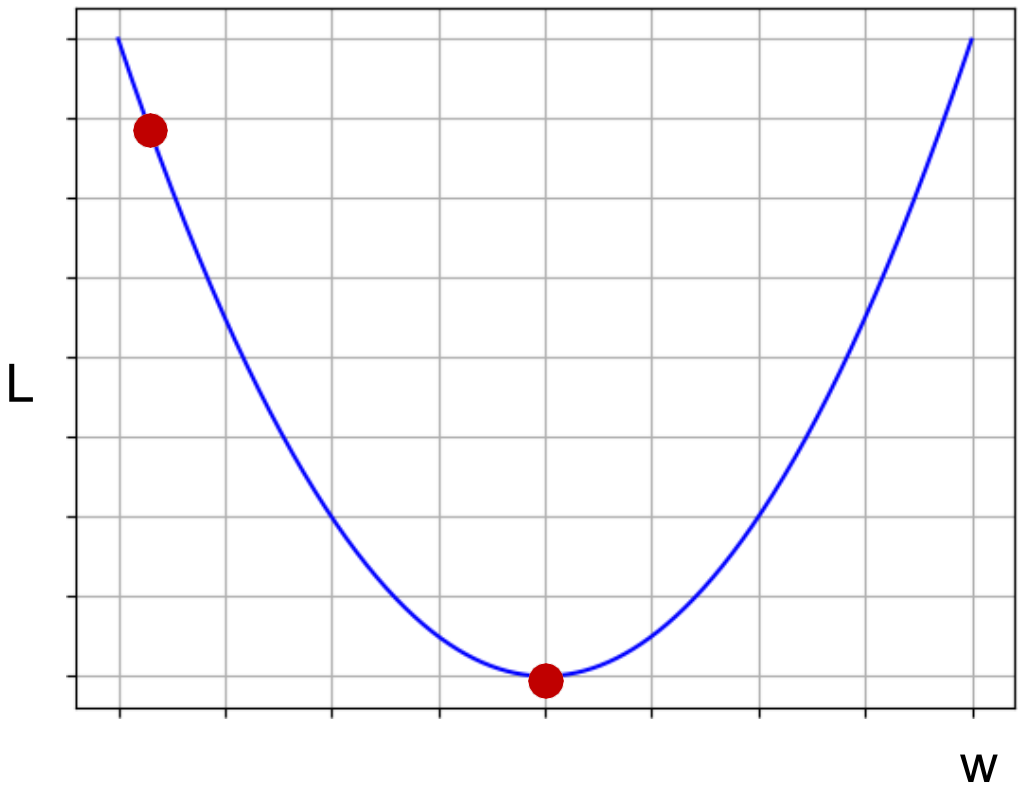
Today's goal – learn about the optimizer

- What does it mean to optimize?
- Gradient descent for linear regression
- Start building a neural network
- Calculating gradients for composite functions (Chain rule)

What does it mean to optimize?

1. Calculate the parameter update values

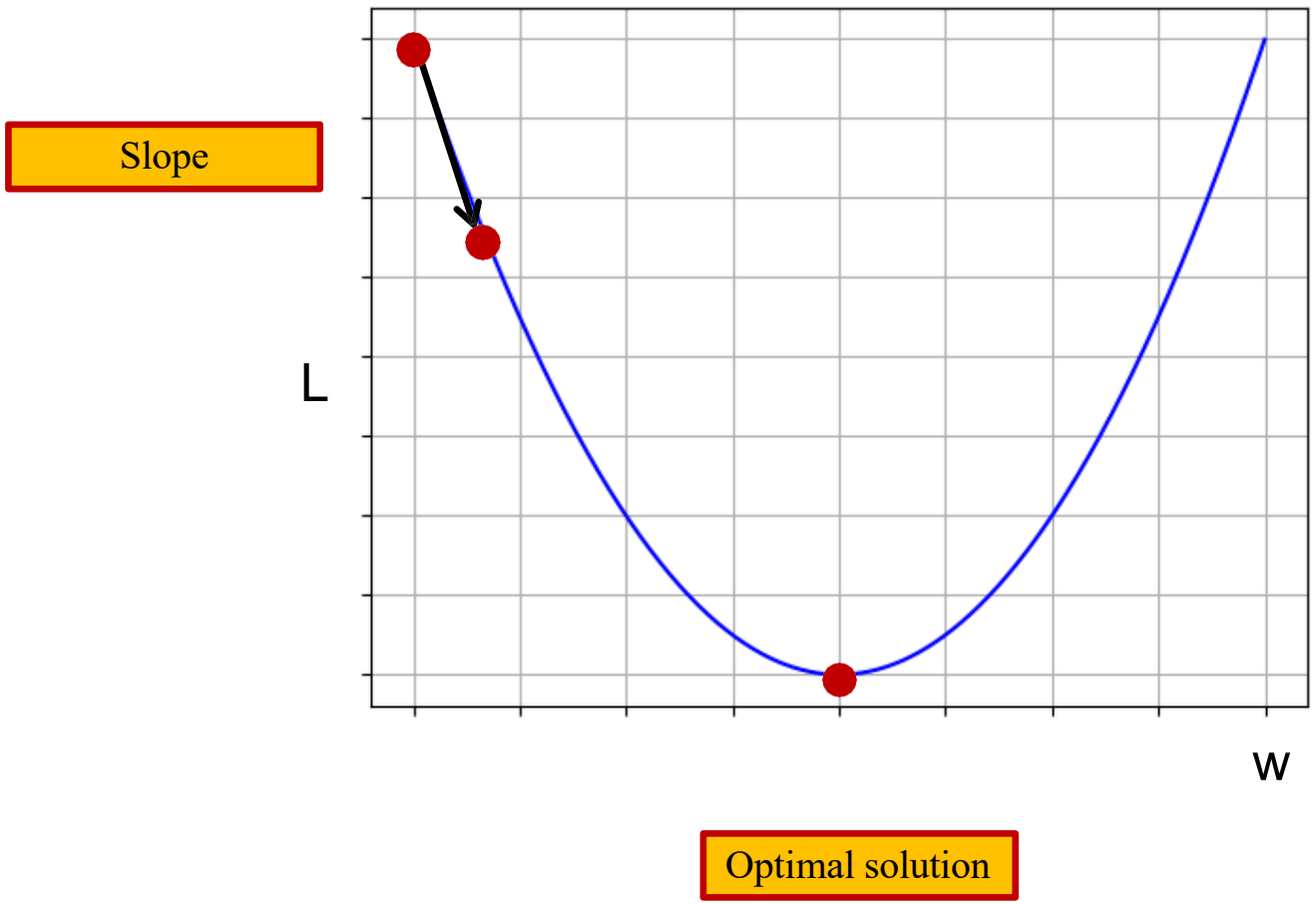
2. Update the parameters



Optimal solution

Gradient (measuring the change)

- Calculating partial derivative of the Loss with respect to the weights/parameters



Courtesy: https://statisticsbyjim.com/regression/mean-squared-error-mse/

Vector Calculus Recap

- Partial derivative: the derivative of a multivariable function with respect to one of its variables
- Example: f(x,w,b) = wx+b
- The partial derivative of f with respect to w is $\frac{\partial f}{\partial w}$
- How to compute? – treat all other variables as constants and differentiate
- $\frac{\partial f}{\partial w} = \frac{\partial}{\partial w}(wx + b) = \frac{\partial}{\partial w}(wx) + \frac{\partial}{\partial w}(b) = x + 0 = x$

Gradient

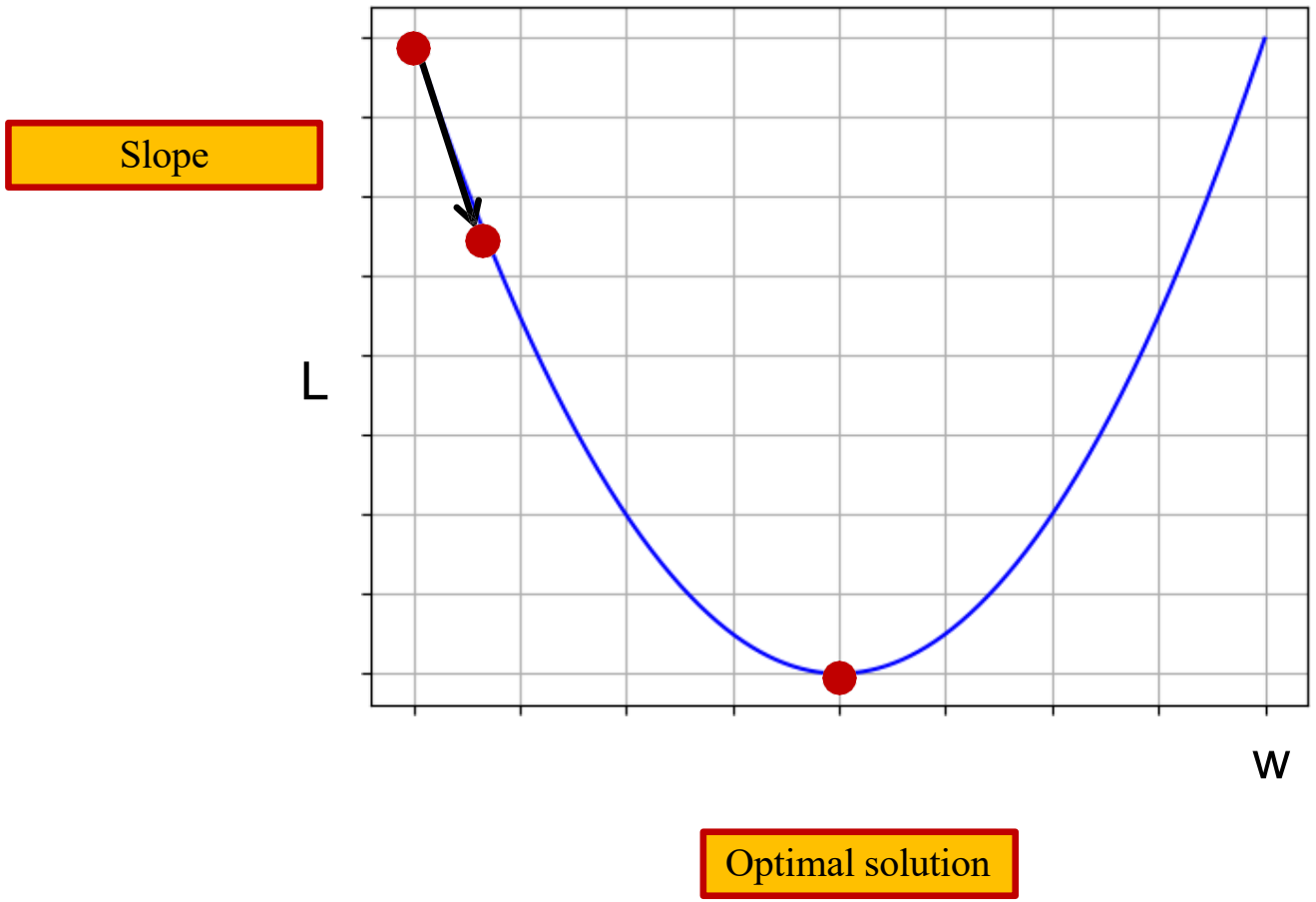
- A gradient simply measures the change in all weights with regard to the change in error.
- You can also think of a gradient as the slope of a function.
- The higher the gradient, the steeper the slope and the faster a model can learn.
- If the slope is zero, the model stops learning.

Gradient descent

- Think of gradient descent as hiking down to the bottom of a valley
- $w_{i+1} = w_i - \alpha \nabla f(w)$
- w_{i+1} is the next position; w_i is the current position.
- α is the learning rate.
- $\nabla f(w)$ is the direction of the descent.

Loss gradient descent in deep learning

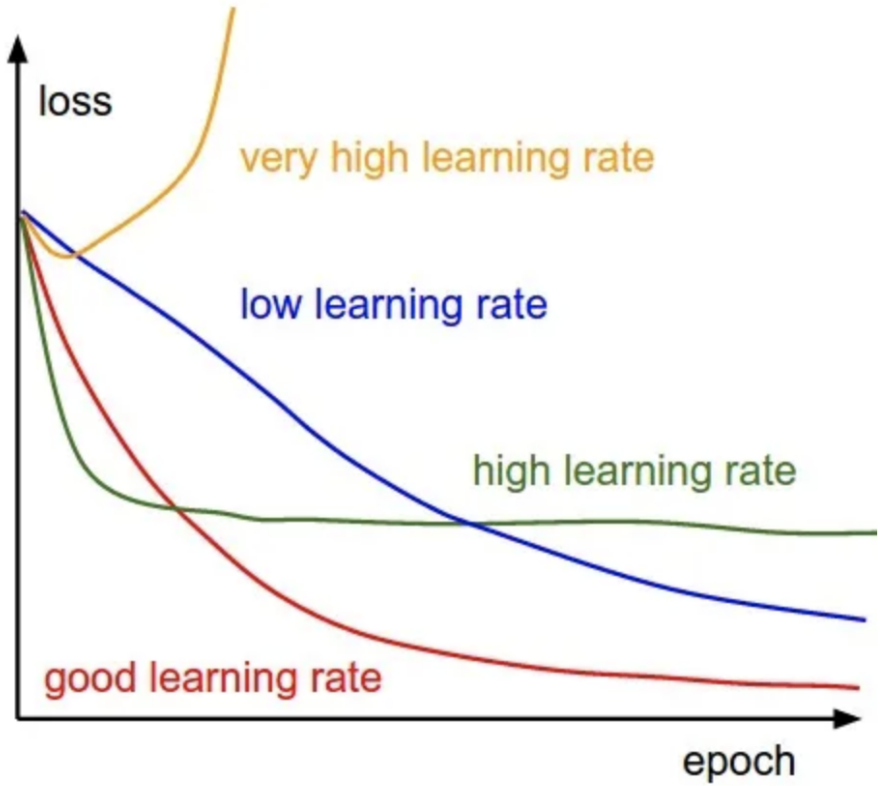
- $\triangle w = -\alpha * \frac{\partial L}{\partial w}$



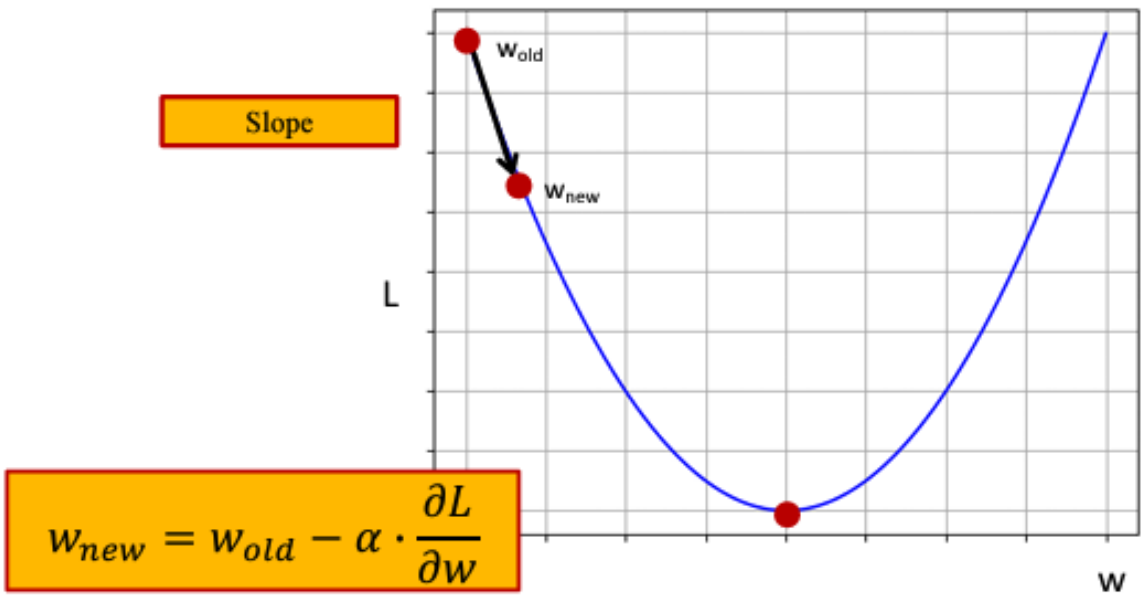
Courtesy: https://statisticsbyjim.com/regression/mean-squared-error-mse/

Impact of learning rate

- Determine how fast or slow we will move towards the optimal weights
- Too small: slow convergence
- Too big: overshoot



Gradient descent (updating parameters)



Gradient descent (updating parameters)

Average squared residual (residual: difference between predicted and true value)

Decreasing the MSE = the model has less error = data points fall closer to the regression line

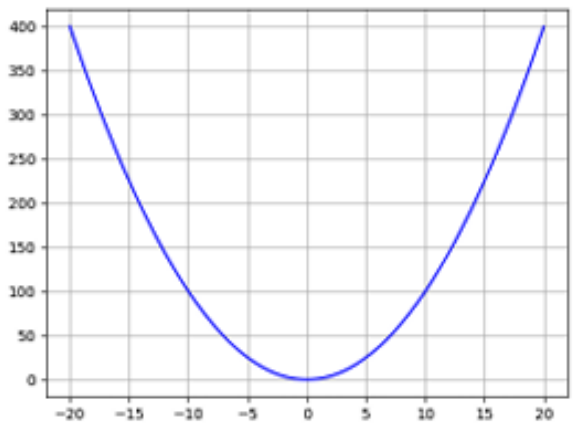
$$MSE = \frac{\sum_{k=1}^{n}(y^k - \hat{y}^k)^2}{n}$$

y^k : *true output value*

$\hat{y}^k$: *predicted output value*

n: *number of samples*

What could be the purpose of squaring the distance?



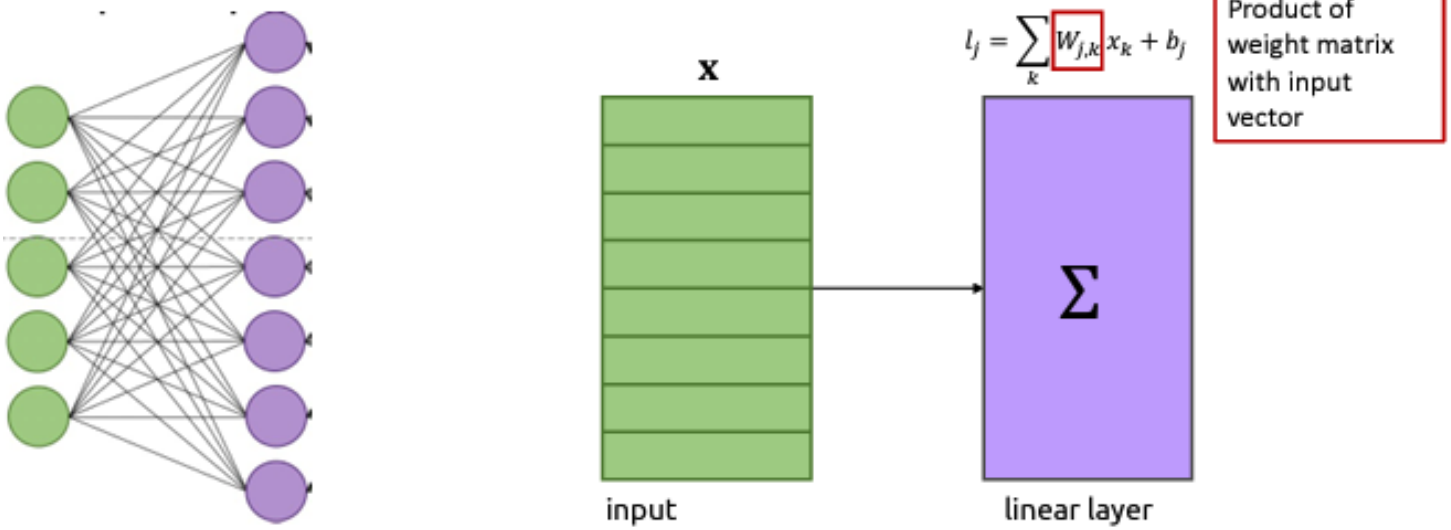
Gradient descent of MSE (1 sample)

$$
\begin{aligned}
L &= (y - \hat{y})^2 \\
&= (y - f(x))^2 \\
&= y^2 + f(x)^2 - 2yf(x) \\
&= y^2 + (wx + b)^2 - 2y(wx + b) \\
&= y^2 + w^2x^2 + b^2 + 2wxb - 2ywx - 2yb
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial w} &= 2wx^2 + 2xb - 2yx \\
\frac{\partial L}{\partial w} &= 2x(wx + b - y)
\end{aligned}
$$

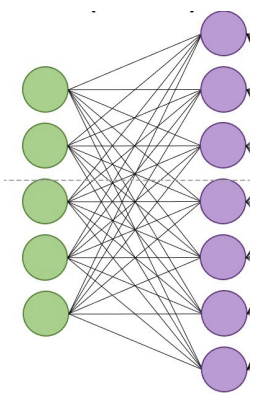
Let’s start building our neural network model

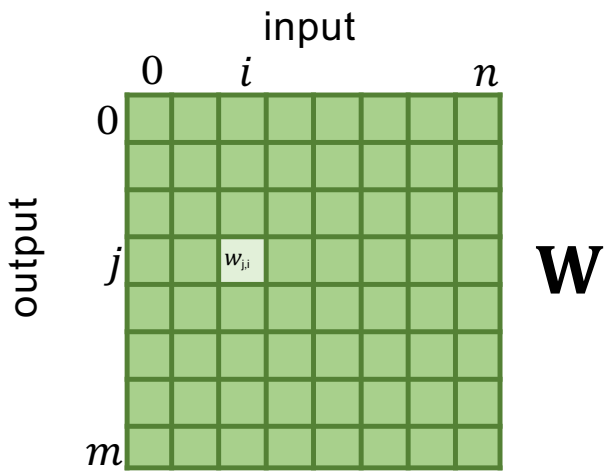
- This is a simplified view of our model with an input and a linear layer



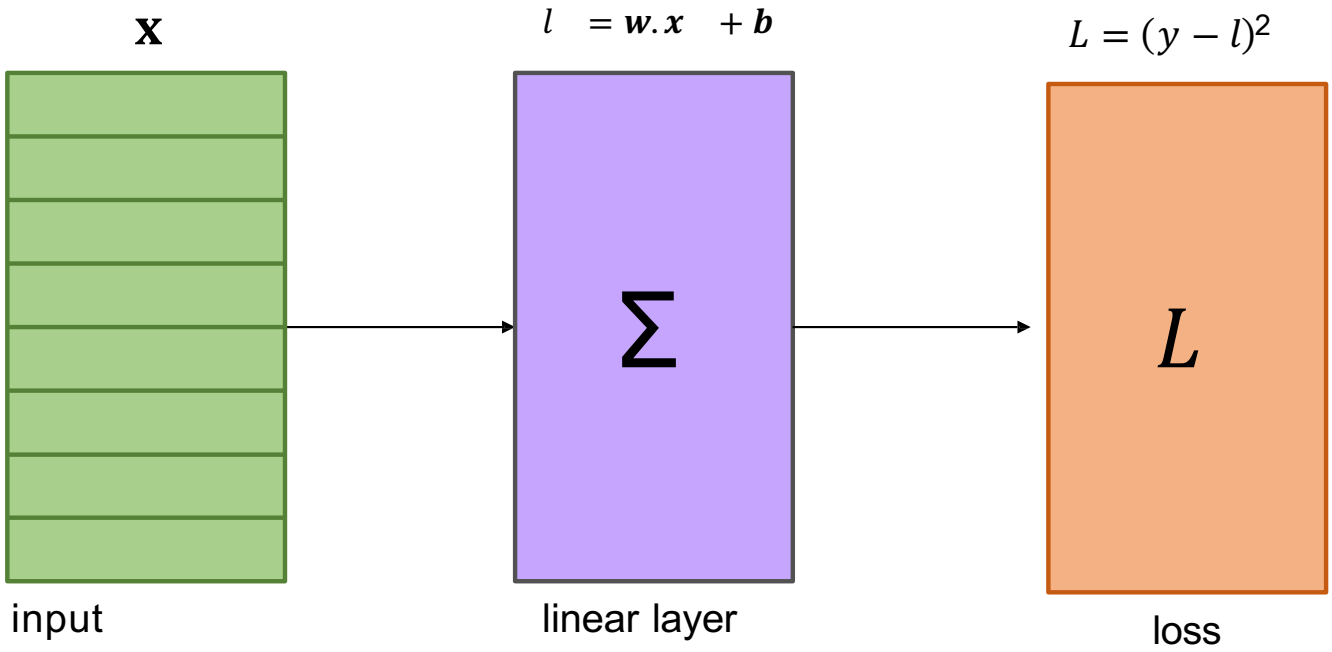
Our Weight Matrix

- We have an input vector of size n and an output vector of size m, so our weights matrix W is of dimensionality $m * n$.
- $w_{j,i}$ is the jth row and the ith column of our matrix, or the weight multiplied by the ith index of the input which is used to create the jth in the output.

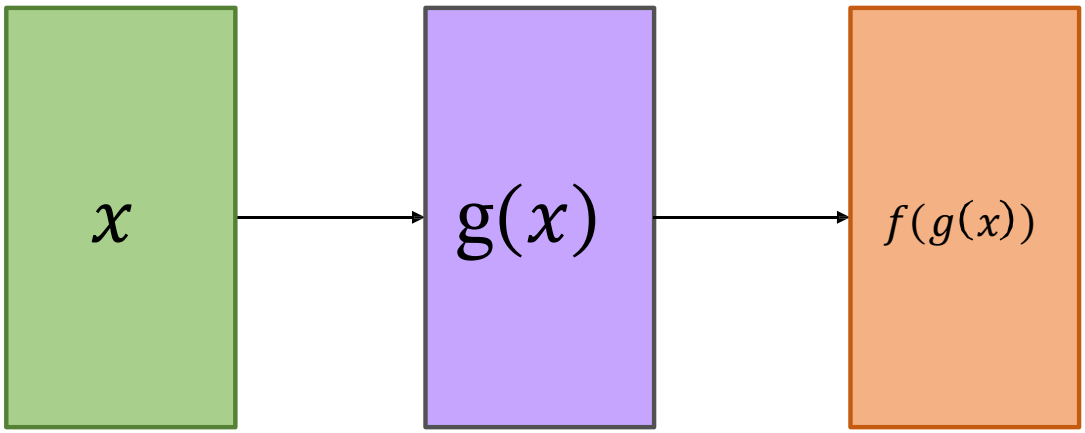




Adding MSE Loss to Our Network



Looking at composite function!



Using gradient descent to update parameters

- Recall the parameter update for Gradient Descent: $\triangle w = -\alpha * \frac{\partial L}{\partial w}$
- L is a composition of a series of functions (linear layers, loss layer, maybe more...)
- How do we compute the derivative of a composition of functions?

Chain rule

If f and g are both differentiable and $F(x)$ is the composite function defined by $F(x) = f(g(x))$ then F is differentiable and F' is given by the product

$$F'(x) = f'(g(x))\, g'(x)$$

Differentiate outer function

Differentiate inner function

Applying chain rule [Example]

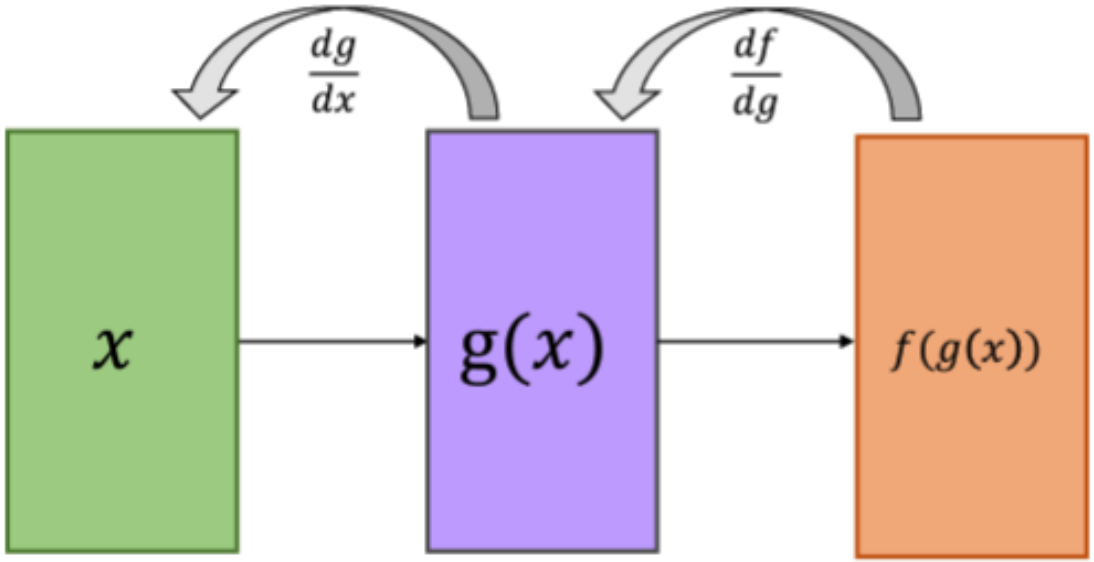
- f(x)=x^2; g(x) = $2x^2 + 1$; F(x) = f(g(x))
- F(x) = $(2x^2 + 1)^2$
- What is $F'(x)$?

Applying chain rule [Example]

- f(x)=x^2; g(x) = $2x^2 + 1$; F(x) = f(g(x))
- F(x) = $(2x^2 + 1)^2$
- What is $F'(x)$?
- $F'(x) = 16x^3 + 8x$

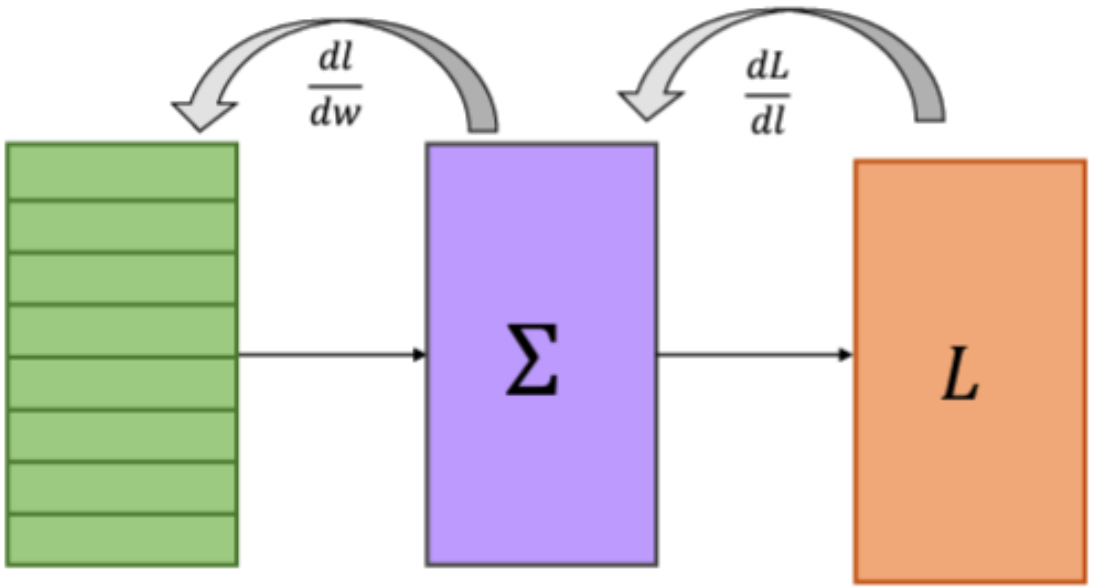
The Chain Rule (for Differentiation)

- Each layer computes the gradients with respect to its variables and passes the result backwards
- Backpropagation (or backward pass)
- Given arbitrary function: $f(g(x)) \rightarrow \frac{df}{dx} = \frac{df}{dg} * \frac{dg}{dx}$



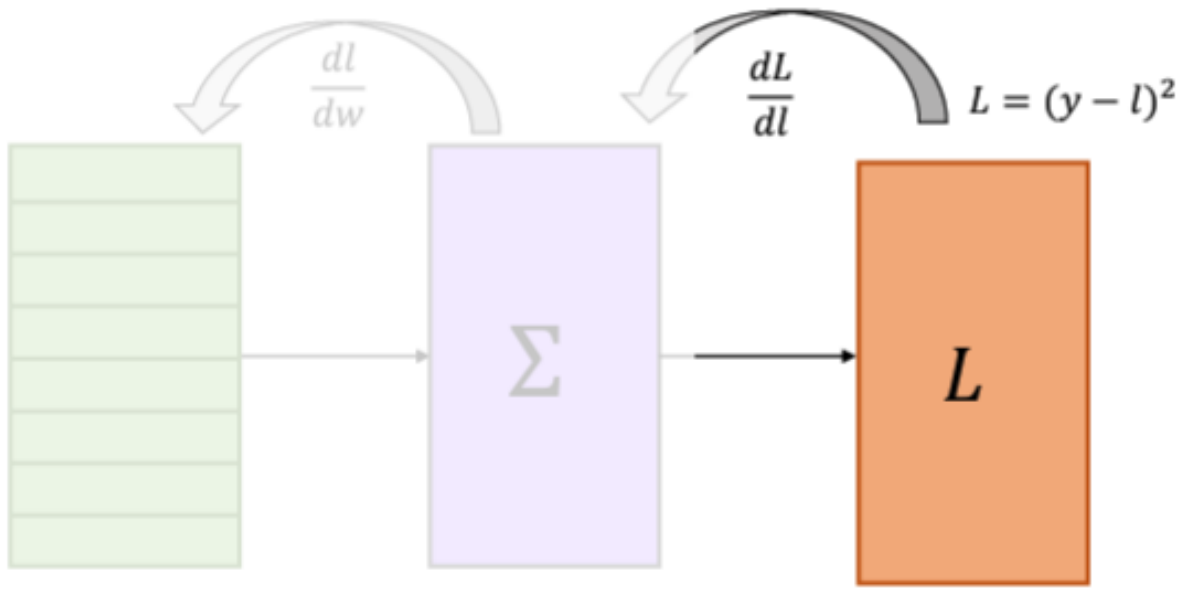
The Chain Rule in our Network

- Here's our function: $L(l(w)) \rightarrow \frac{dL}{dw} = \frac{dL}{dl} * \frac{dl}{dw}$



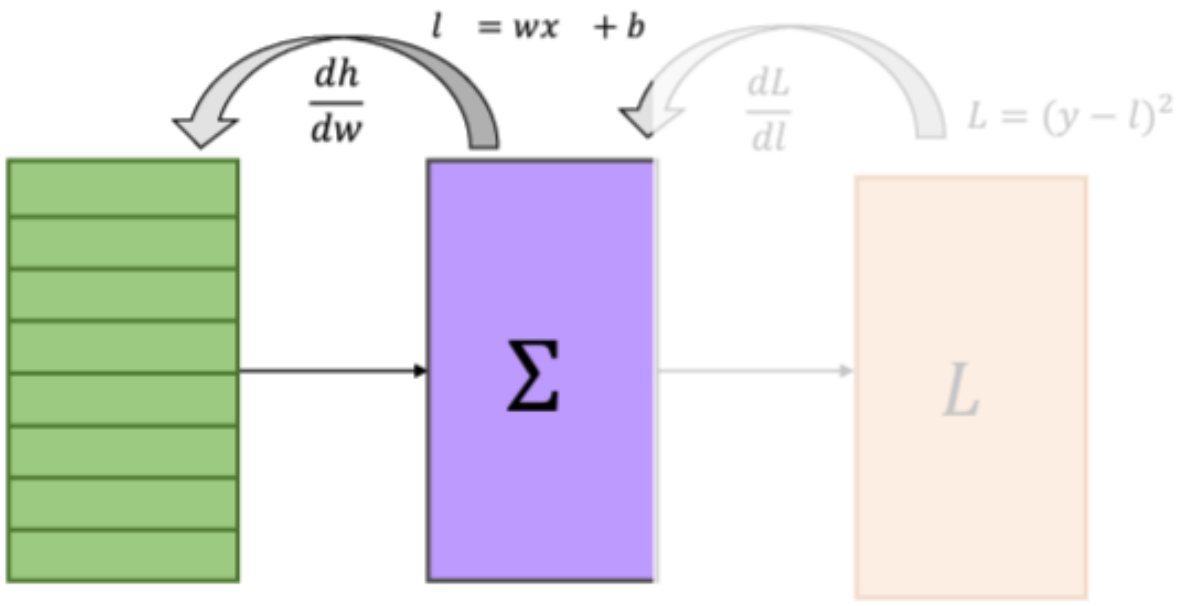
Derivative of loss layer

- $\frac{dL}{dl} = \frac{d(y-l)^2}{dl}$



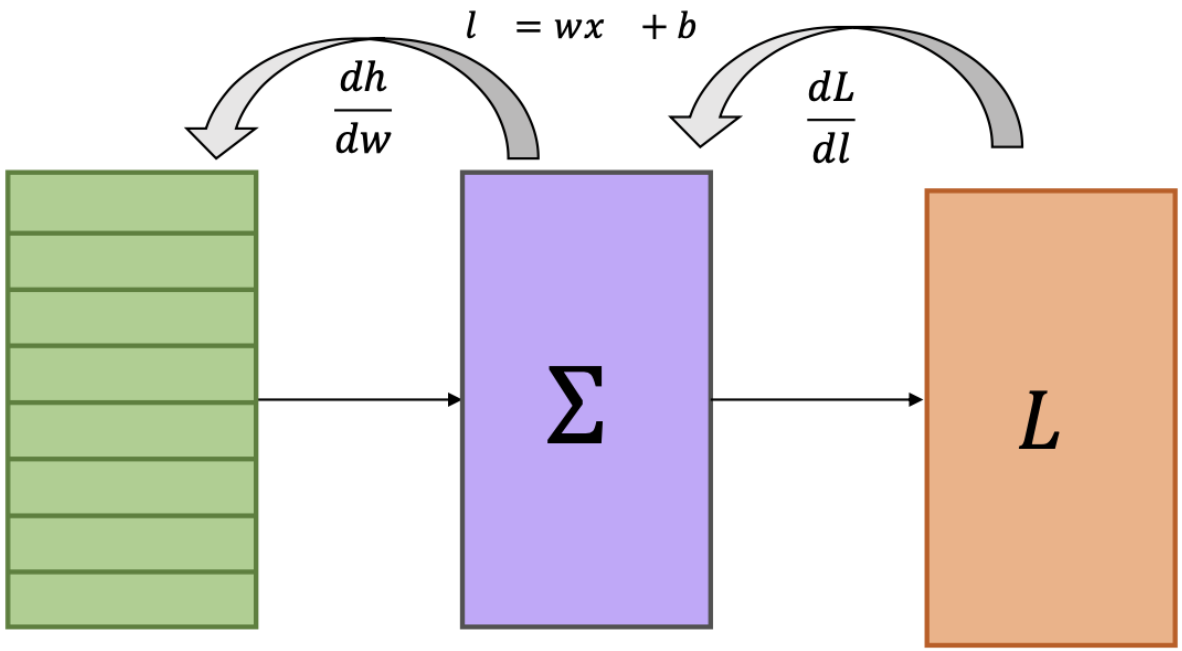
Derivative of linear layer

- $\frac{dl}{dw} = \frac{d(wx+b)}{dw}$



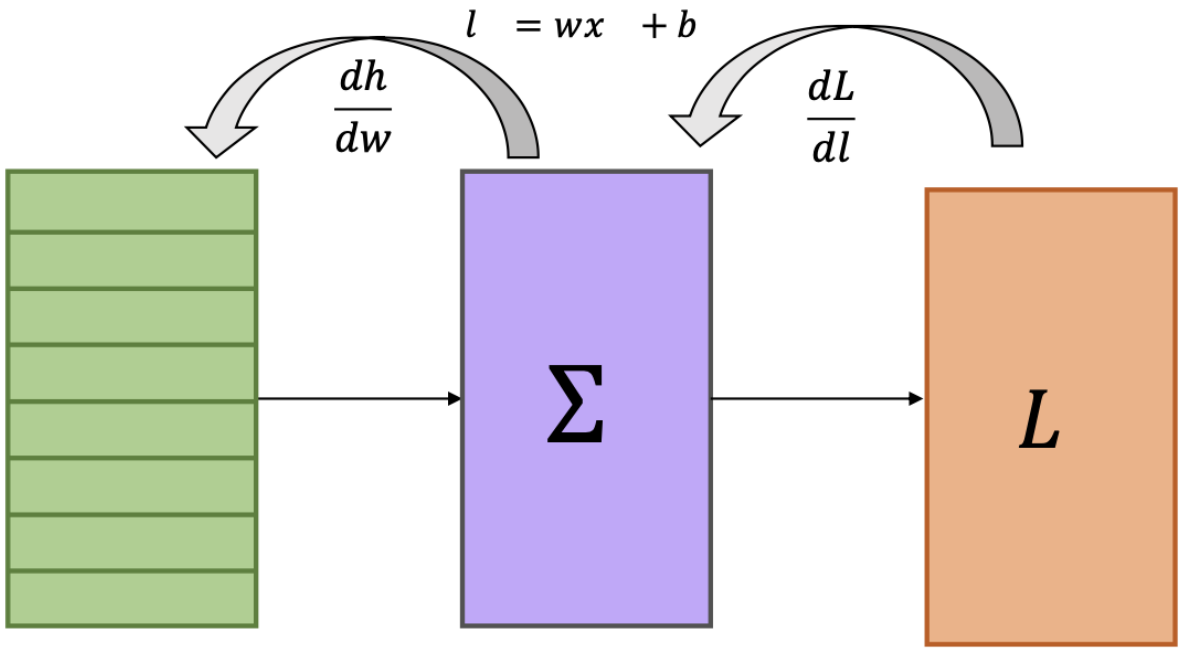
Putting it all together

- $\frac{dl}{dw} = \frac{dL}{dl} * \frac{dl}{dw} =$



Putting it all together

- $\frac{dl}{dw} = \frac{dL}{dl} * \frac{dl}{dw} = -2(y - l) * x = -2x(y - wx - b) = 2x(wx + b - y)$



Recall: Gradient Descent of MSE (1 sample)

- $\triangle w = -\alpha \frac{dl}{dw}$

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

$$= (y - f(x))^2$$

$$= y^2 + f(x)^2 - 2yf(x)$$

$$= y^2 + (wx + b)^2 - 2y(wx + b)$$

$$= y^2 + (wx + b)^2 - 2y(wx + b)$$

$$= y^2 + (wx + b)^2 - 2y(wx + b)$$

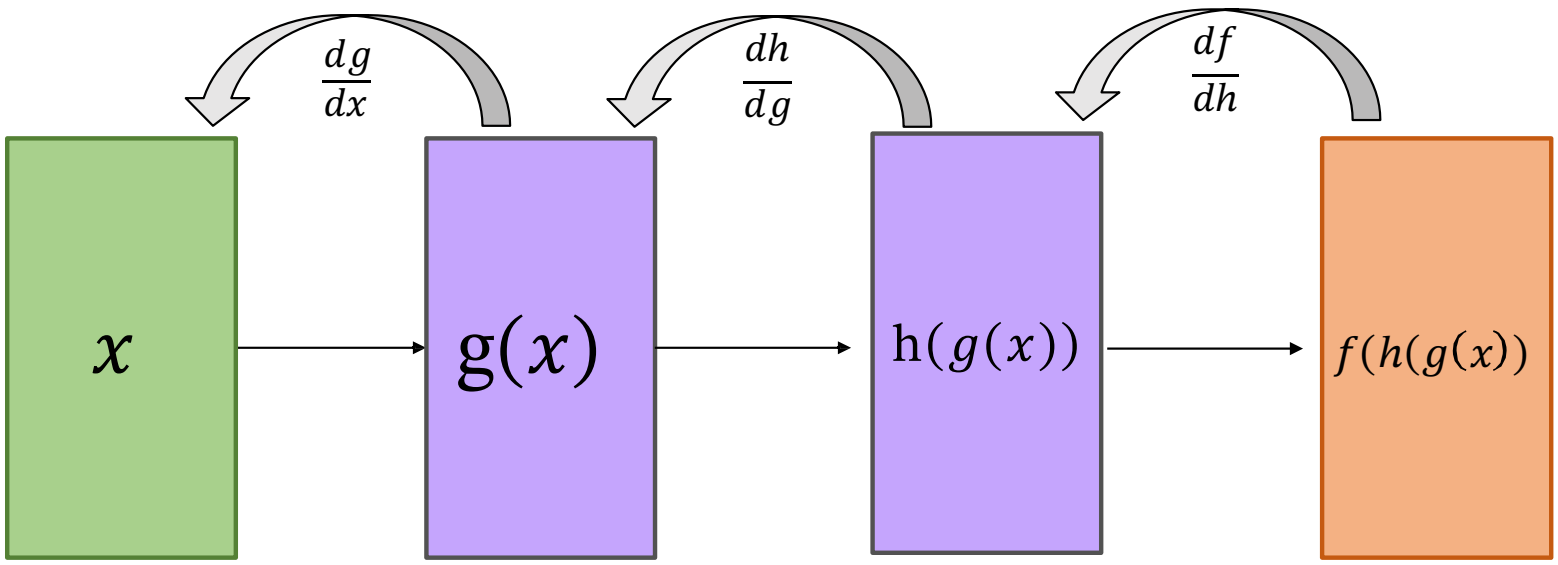
$$= y^2 + w^2x^2 + b^2 + 2wxb - 2ywx - 2yb$$

$$\frac{\partial L}{\partial w} = 2wx^2 + 2xb - 2yx$$

$$\frac{\partial L}{\partial w} = 2x(wx + b - y)$$

Adding more layers

- Can we add any function?
- $f(h(g(x))) \rightarrow \frac{df}{dx} = \frac{df}{dh} * \frac{dh}{dg} * \frac{dg}{dx}$



Few more important points: Backpropagation

- The process of calculating gradients of functions via chain rule in a neural network
- Is a part of and NOT the whole learning algorithm
- Can be calculated with respect to any variable of choice
- For learning in neural networks we calculate gradients with respect to the weights

Outline

- Model optimizer
- **Backpropagation and autodiff**
- Multi-layer NNs and activation

Today's goal – continue learning about backpropagation

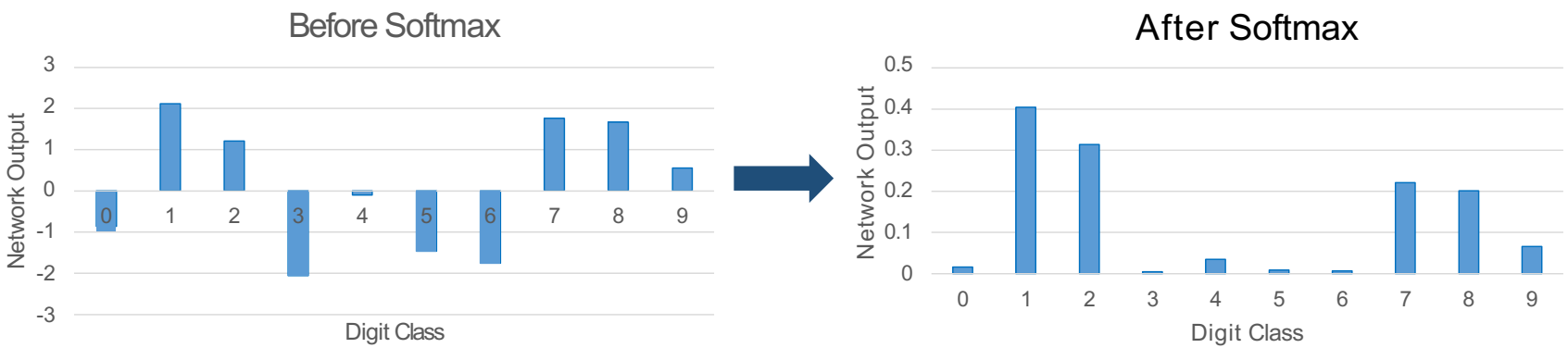
- Building a simple neural network for multi-class classification
- Backpropagation of our network (via Chain Rule)
- Computation graph for neural networks

Our new probability layer

- What does a probablity distribution, p looks like?
  - For any digit j: $p_j \in [0, 1]$
  - $\sum_k p_k = 1$
- Currently, our outputs l do not satisfy these properties
  - For any digit j: $l_j \in R$
  - $\sum_k l_k = R$
- How to make our network output satisfy these properties?

The Softmax Function

- The formula: $p_j = \frac{e^{l_j}}{\sum_k e^{l_k}}$
- Using exponents e^{l_j} means every number is positive
- Dividing by $\sum_k e^{l_k}$ means every p_j is between 0 and 1, and that $\sum_k P_k = 1$.



We call these numbers ***logits*** (l notation)

Recap: Cross Entropy Loss (for Multi-class classification)

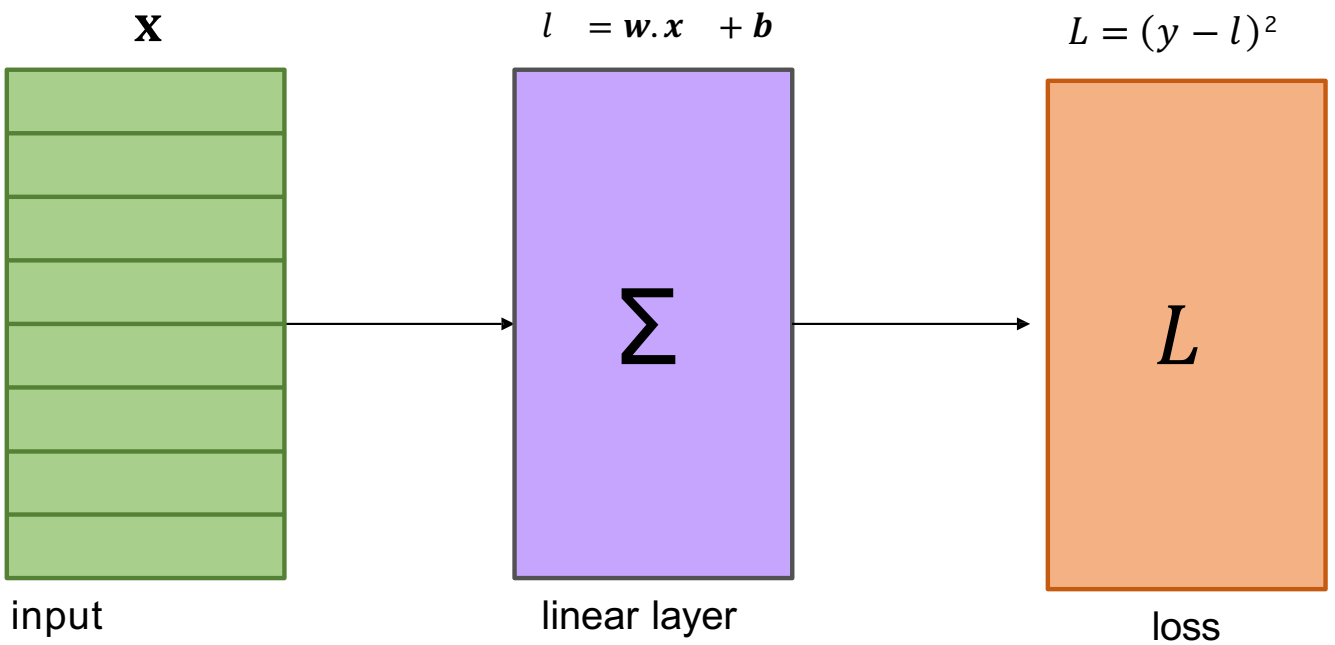
- Loss = $-\sum_{j=1}^{m} y_i log(p_j)$

p	Classes (m)	y
0.3	"0"	0
0.2	"1"	0
0.5	"2"	1

We can get these probabilities by using a Softmax function

We want model to assign high probability to the true class and low to others

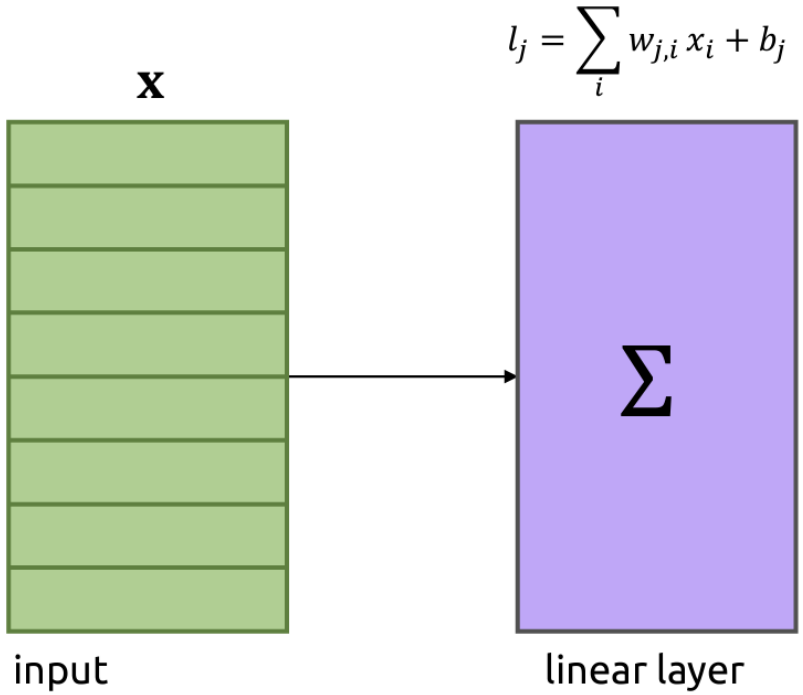
What changes do we make for this task?



- Remove the MSE loss layer
- Increase the number of outputs to m classes
- Add a softmax layer
- Add a cross entropy loss layer

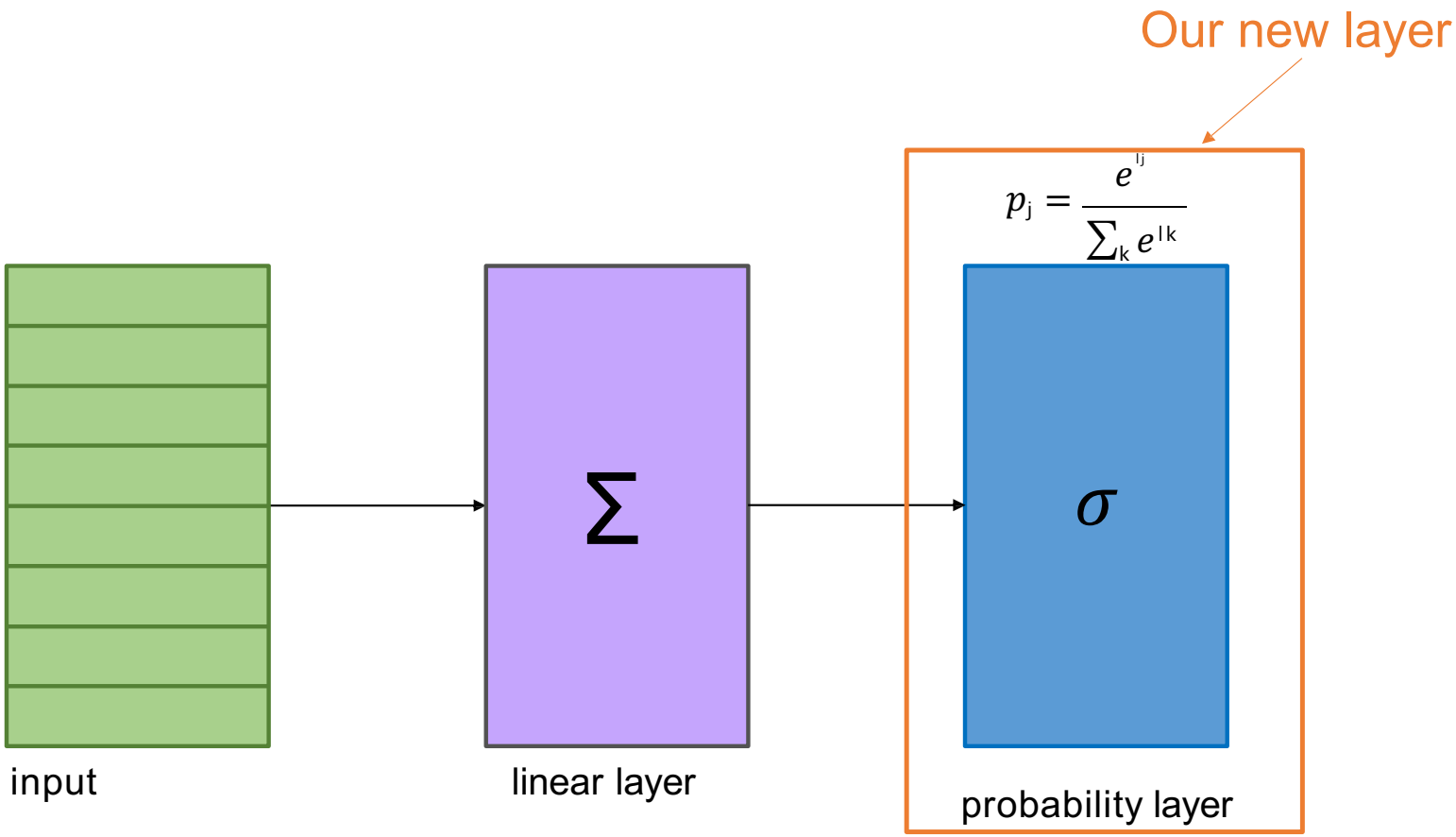
Our model before

- This is a simplified view of our model with an input and a linear layer

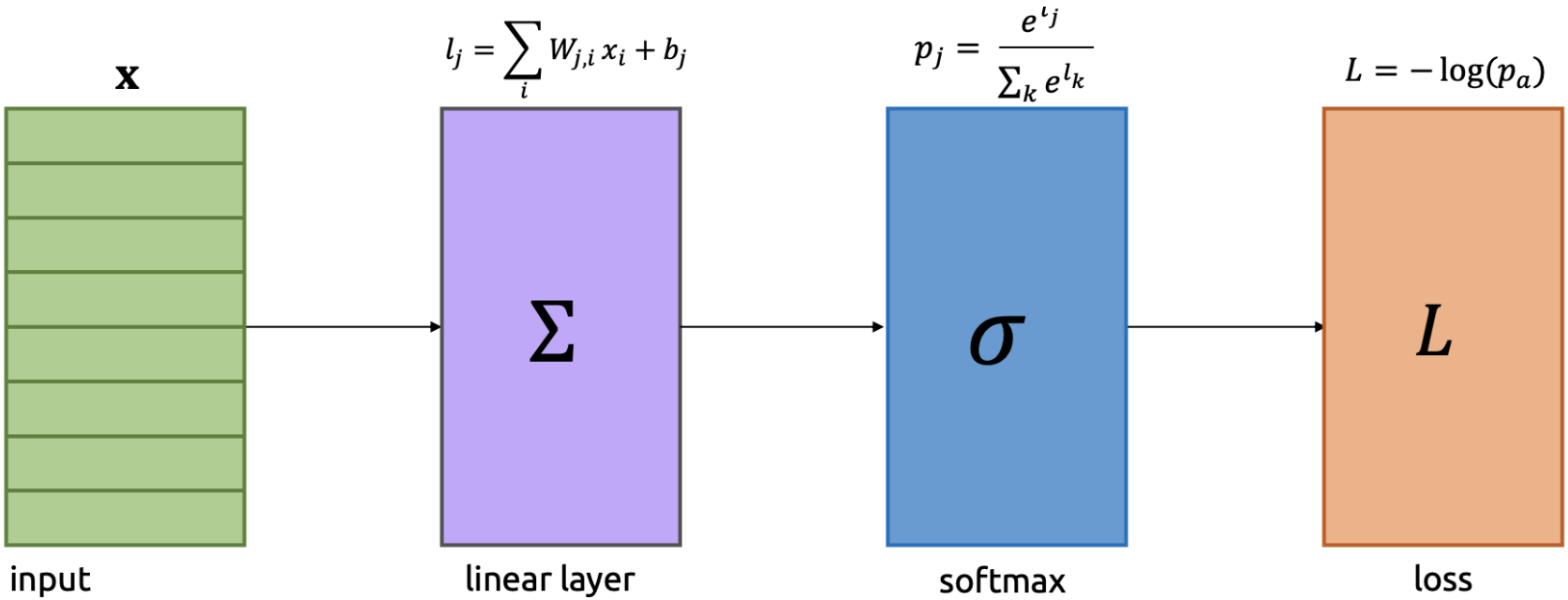


Our model after

- This is our model with the new probabability layer

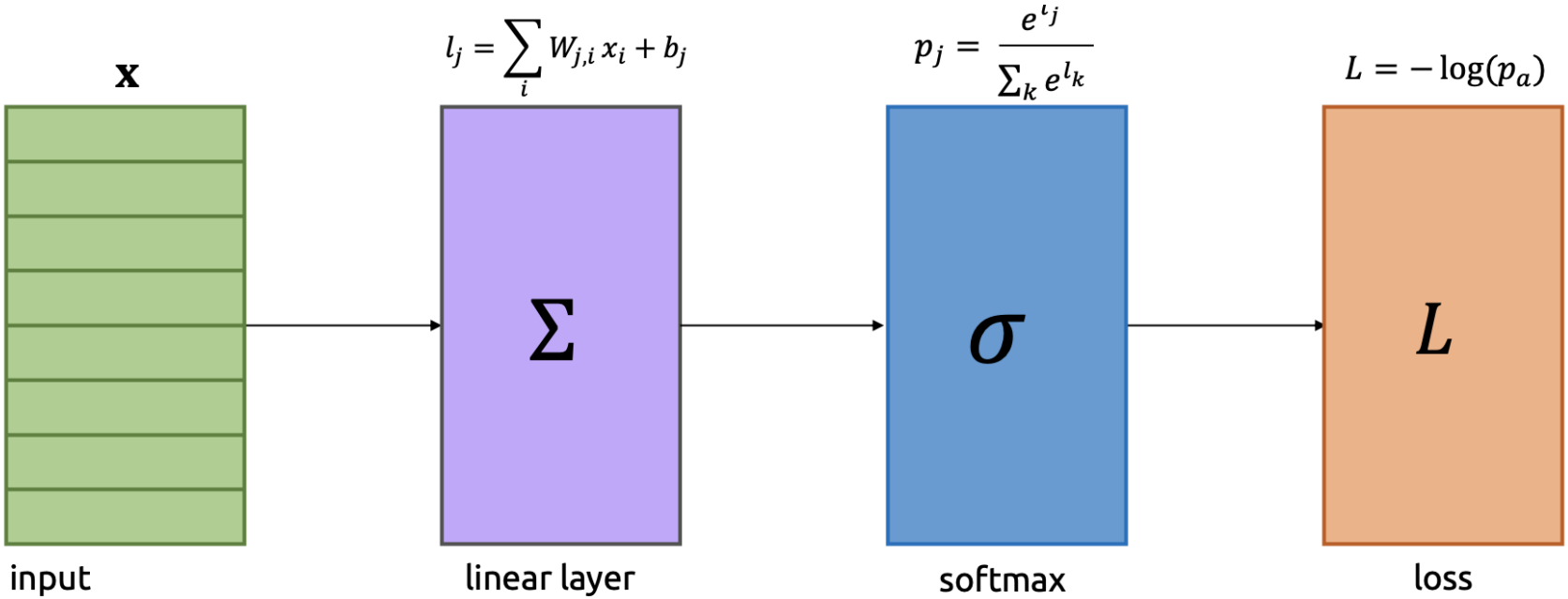


Adding Cross Entropy Loss to Our Network



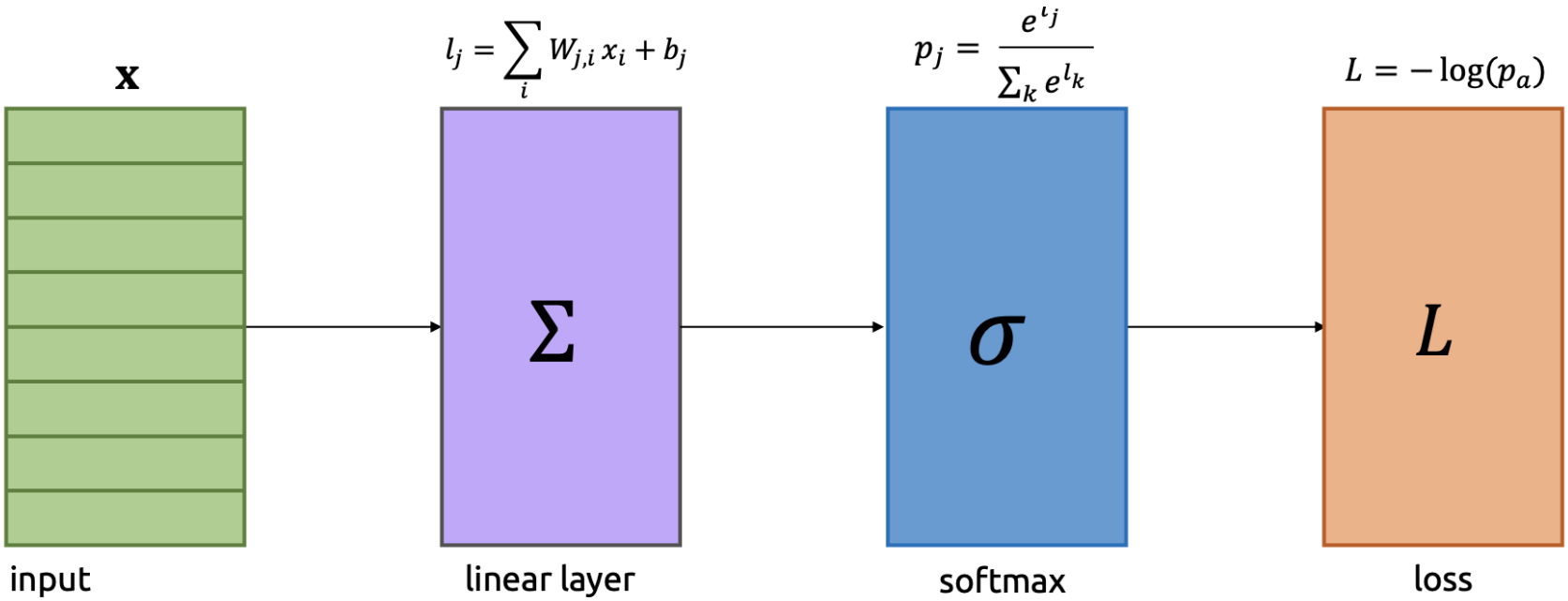
What is the Chain Rule in Our Network?

- Here is our function: L(p(l(w))) $\rightarrow$



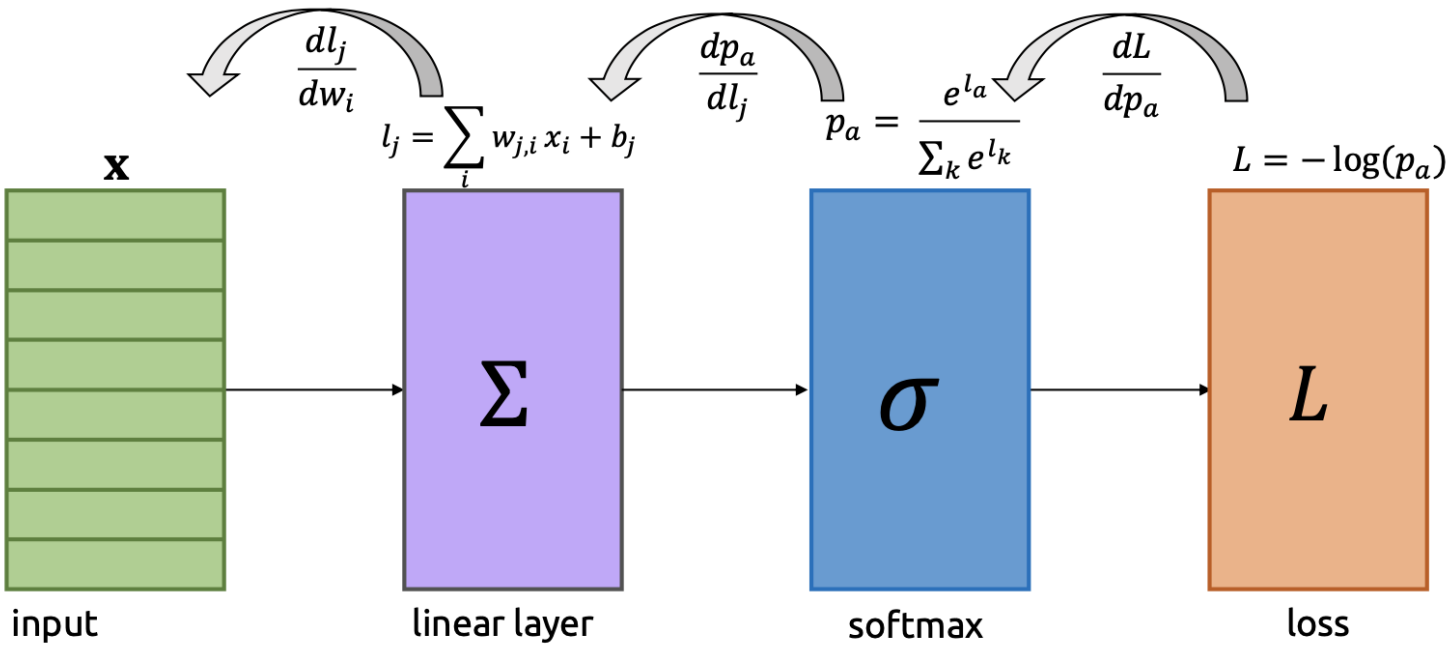
What is the Chain Rule in Our Network?

- Here is our function: $L(p(l(w))) \rightarrow \frac{dL}{dw} = \frac{dL}{dp} * \frac{dp}{dl} * \frac{dl}{dw}$



Chain Rule Put Together

$$\Delta w_{j,i} = -\alpha \frac{\partial L}{\partial w_{j,i}} = -\alpha \cdot \frac{\partial L}{\partial p_a} \cdot \frac{\partial p_a}{\partial l_j} \cdot \frac{\partial l_j}{\partial w_{j,i}} = -\alpha \cdot \left(\frac{-1}{p_a}\right) \cdot \left(p_a(y_j - p_j)\right) \cdot (x_i) = -\alpha \cdot (p_j - y_j) \cdot x_i$$

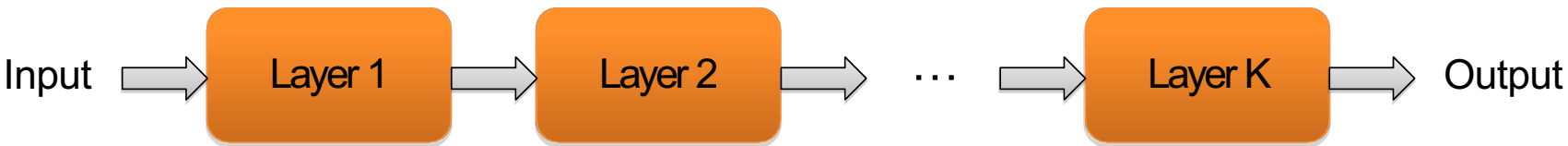


Chain Rule Put Together

- $\triangle w_{j,i} = -\alpha * (p_j - y_j) * x_i$
  - α: learning rate
  - p_j: probability in class j.
  - x_i: the input
  - y_j: the true label of class j.
- We use this to descend along the gradient toward the minimum loss value
- We used chain rule to propagate backwards through the entire network while doing the derivative - backpropagation

Backpropagation for Deeper Networks

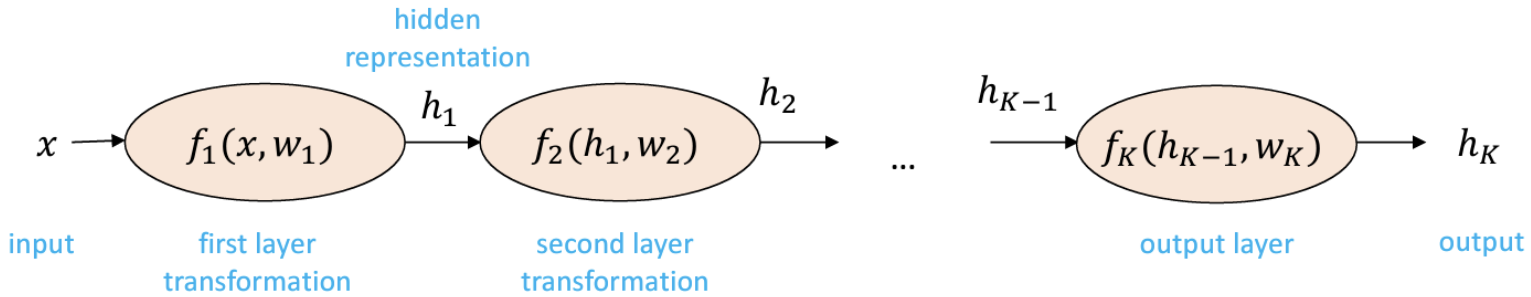
- The function computed by the network is a composition of the functions computed by individual layers (e.g., linear layers and nonlinearities)



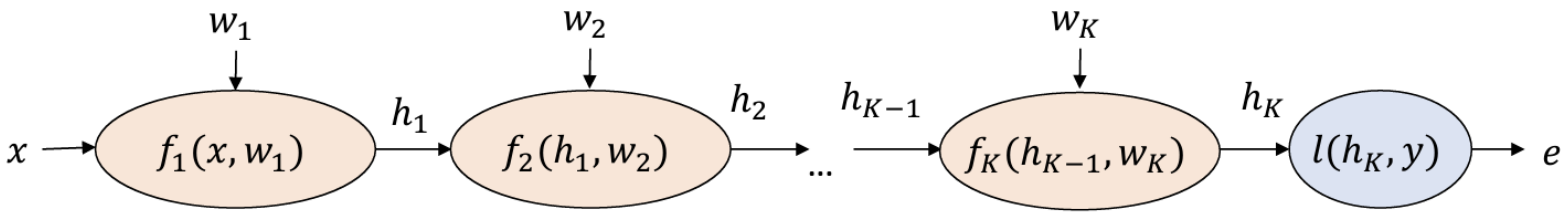
Computation Graph

- A directed acyclic graph (DAG) that is used to specify mathematical computations:
  - Each edge represents a data dependency (i.e. feed a variable as input to the function)
  - Each node represents a function, or a variable (scalars, vectors, matrices, tensors)
- Recall that neural networks are compositions of functions
- A computation graph can be used to specify a general neural network

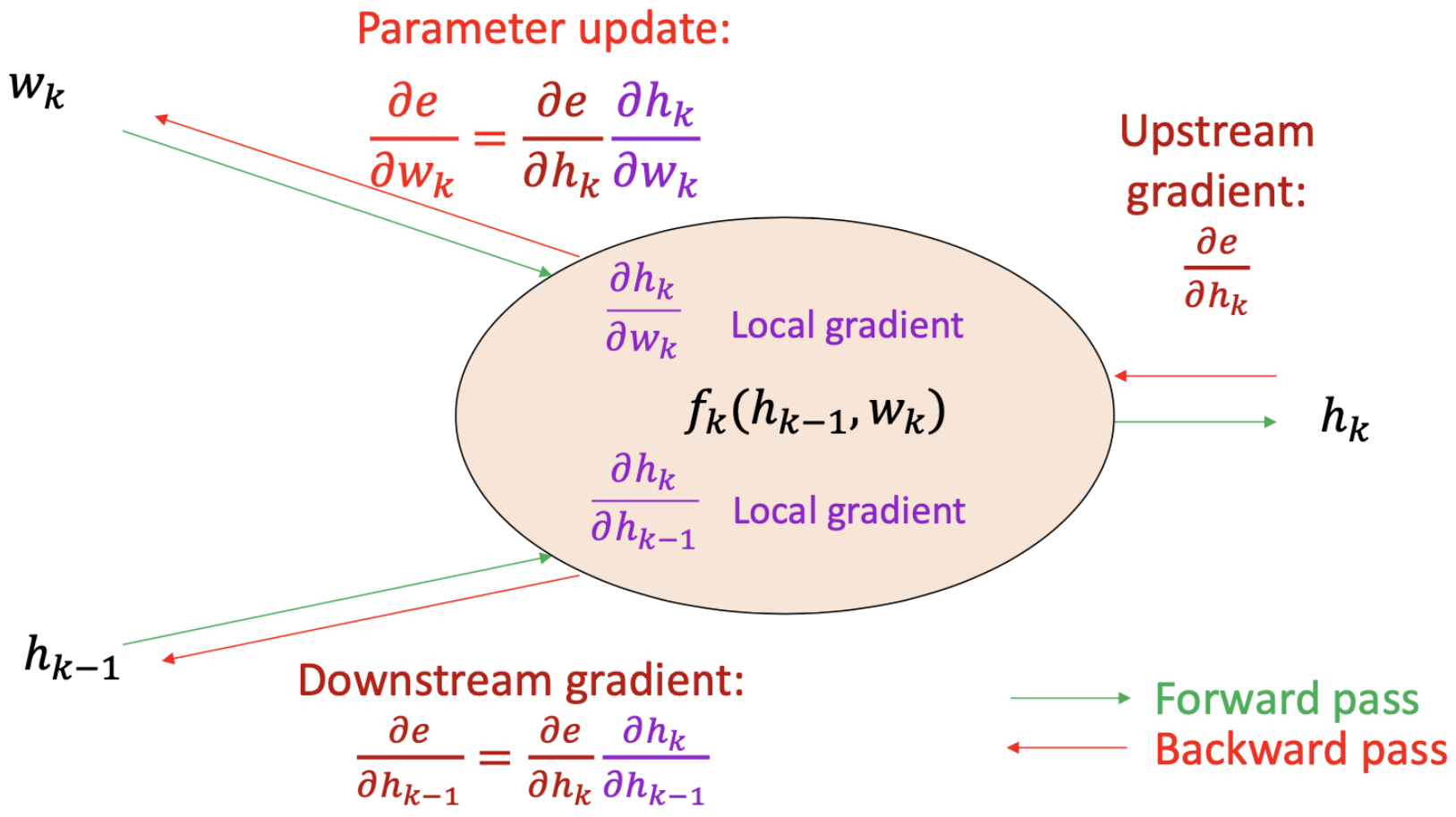
Computation Graph



Example Computation Graph for a Neural Network with a Loss Layer:

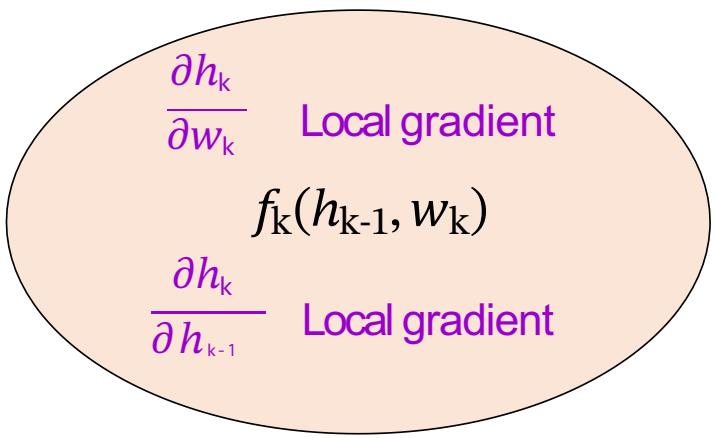


Backpropagation: Summary



Backpropagation: Layer Abstraction

- Layer is an abstraction of a function (linear layer, softmax layer, ReLU layer)
- Forward pass: Just need to implement the function itself $f_k(h_{k-1}, w_k)$
- Backward pass requires two functions to compute: $\frac{\partial h_k}{\partial h_{k-1}}$ and also $\frac{\partial h_k}{\partial w_k}$ if the function has parameters.



Recap

- Softmax function
- Building a simple model with new layers
- Chain rule for multi-class classification
- Computation graph

Today's goal – learn about deep learning frameworks

- Gradient Descent pseudocode
- Stochastic Gradient Descent (SGD)
- Automatic differentiation

Putting Everything Together: Gradient Descent

```
# delta_W is 2-D matrix of 0's in the shape of W
for each input and corresponding answer a:
    probabilities = run_network(input)
    for j in range(len(probabilities)):
        y_j = 1 if j == a else 0
        for i in range(len(input):
            delta_W[j][i] += alpha * (y_j – probabilities[j]) * input[i]
W += delta_W
```

Forward pass

Backward pass:

Compute $\frac{\partial L}{\partial W_{i,j}}$ for every $W_{i,j}$

Over the entire dataset

Putting Everything Together: Gradient Descent

```
# delta_W is 2-D matrix of 0's in the shape of W
for each input and corresponding answer a:
    probabilities = run_network(input)
    for j in range(len(probabilities)):
        y_j = 1 if j == a else 0
        for i in range(len(input):
            delta_W[j][i] += alpha * (y_j – probabilities[j]) * input[i]
W += delta_W
```

Forward pass

Backward pass:

Compute $\frac{\partial L}{\partial W_{i,j}}$ for every $W_{i,j}$

Over the entire dataset

Gradient descent update

Gradient Descent: Limitation?

```
# delta_W is 2-D matrix of 0's in the shape of W
for each input and corresponding answer a:
  probabilities = run_network(input)
  for j in range(len(probabilities)):
    y_j = 1 if j == a else 0
    for i in range(len(input):
      delta_W[j][i] += alpha * (y_j – probabilities[j]) * input[i]
W += delta_W
```

We iterate over the ***entire*** dataset...

...to update the weights only ***once***

Stochastic Gradient Descent (SGD)

- Alternative is to train on batches: small subsets of the training data
- Why stochastic: Each batch is randomly sampled from the full training data
- We update the parameters after each batch

Stochastic Gradient Descent: Pseudocode

```
for each batch:
  # delta_W is 2-D matrix of 0's in the shape of W
  for each input and corresponding answer a in batch:
    probabilities = run_network(input)
    for j in range(len(probabilities)):
      y_j = 1 if j == a else 0
      for i in range(len(input)):
        delta_W[j][i] += alpha * (y_j – probabilities[j]) * input[i]
  W += delta_W
```

Stochastic Gradient Descent: Pseudocode

```
for each batch:
  # delta_W is 2-D matrix of 0's in the shape of W
  for each input and corresponding answer a  in batch:
    probabilities = run_network(input)
    for j in range(len(probabilities)):
      y_j = 1 if j == a else 0
      for i in range(len(input)):
        delta_W[j][i] += alpha * (y_j – probabilities[j]) * input[i]
  W += delta_W
```

Now we update weights ***after every batch***

13

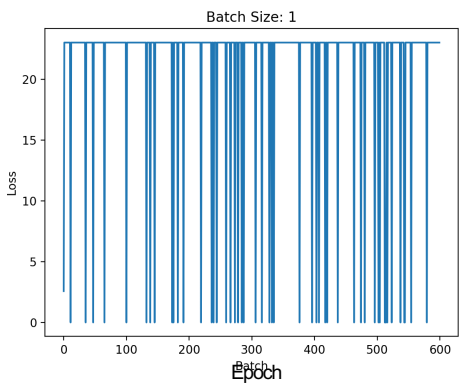
Stochastic Gradient Descent (SGD)

- Train on batches: small subsets of the training data
- We update the parameters after each batch
- This makes the training process stochastic or non-deterministic:
  - batches are a random subsample of the data
  - do not provide the gradient that the entire dataset as a whole would provide at once
- Formally: the gradient of a randomly-sampled batch is an unbiased estimator of the gradient over the whole dataset
  - “Unbiased”: expected value == the true gradient, but may have large variance (i.e. the gradient may ‘jitter around’ a lot)

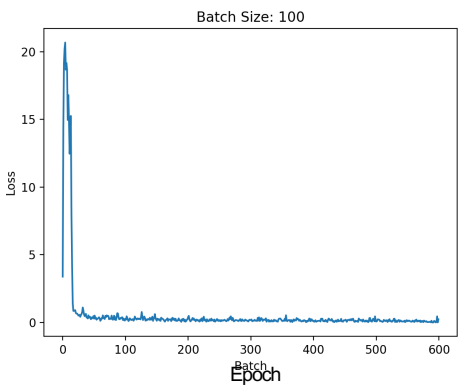
What size should the batch be?

- Rule of thumb nowadays: Pick the largest batch size you can fit on your GPU!

Small batch size:
Fast, jittery updates



Large batch size:
Slower, stable updates

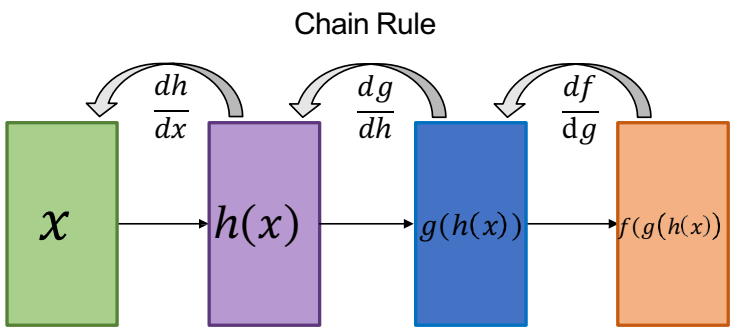


Generalizing Backpropagation

- What if we want to add another layer to our model?
- Calculating derivatives by hand again is a lot of work
- Can the computers do this for us?

Automatic differentiation

- Computer-based Derivatives
- Use the chain rule at runtime



Automatic differentiation

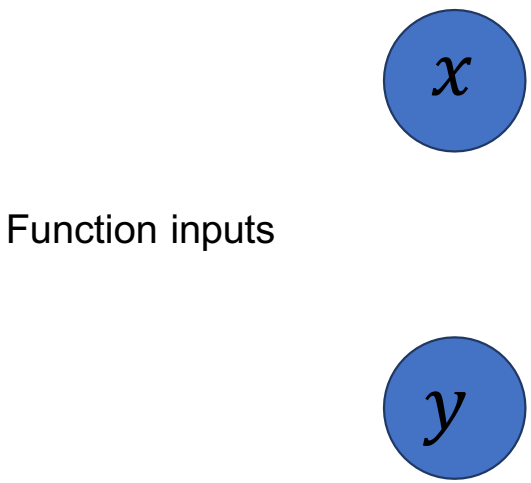
- Use the chain rule at runtime
- Gives exact results
- Handles dynamics (loops, etc.)
- Easier to implement
- Can't simplify expressions using identity, e.g., $sin^2 x + cos^2 x = 1$.

Two Main "Flavors" of Autodiff

- Forward Mode Autodiff
  - Compute derivatives alongside the program as it is running
- Reverse Mode Autodiff
  - Run the program, then compute derivatives (in reverse order)

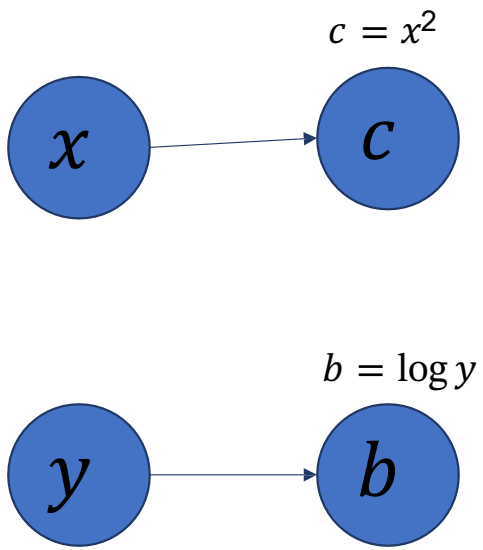
Forward Mode Autodiff

- Given $f(x, y) = x^2 + logy$



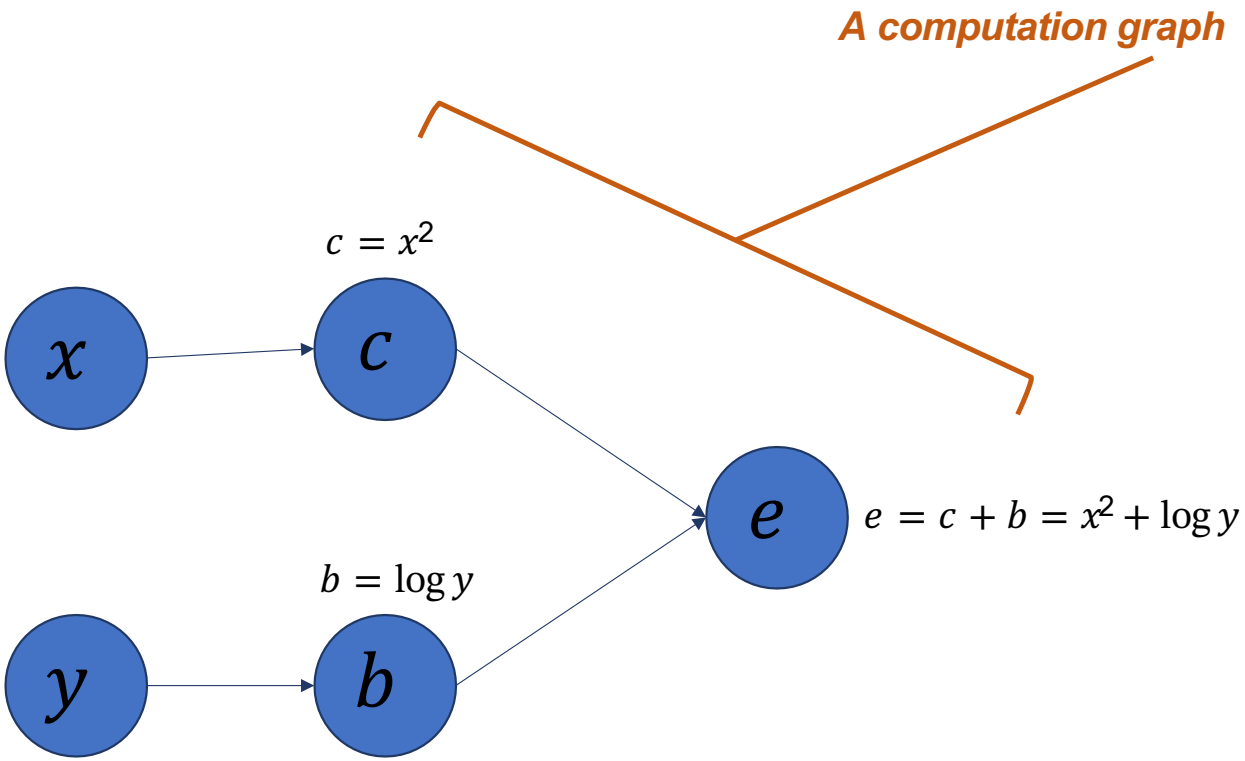
Forward Mode Autodiff

- Given $f(x, y) = x^2 + \log y$

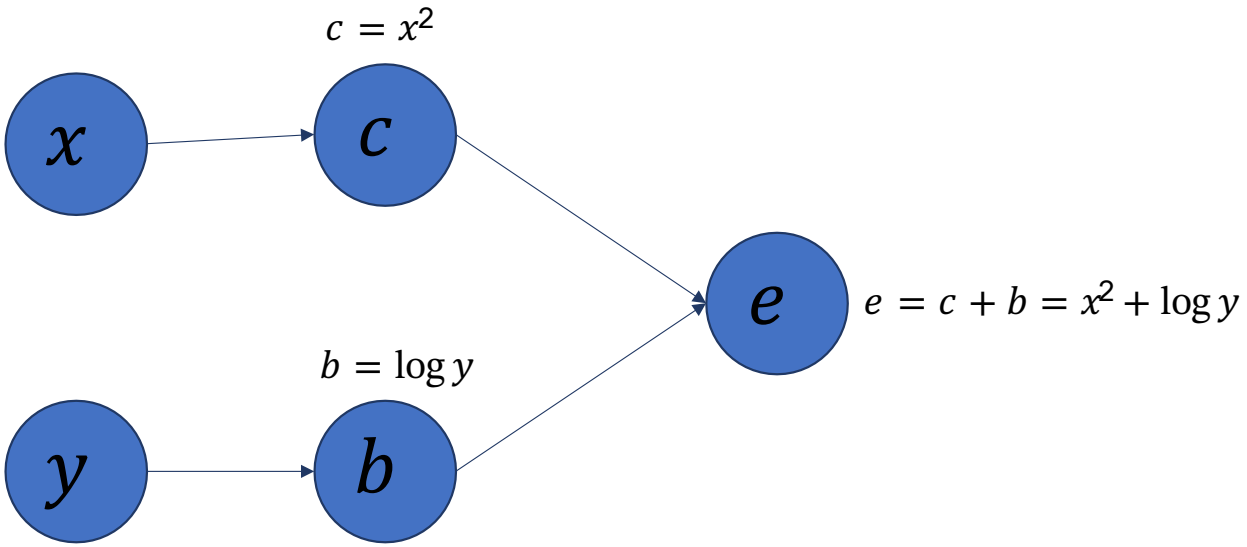


Forward Mode Autodiff

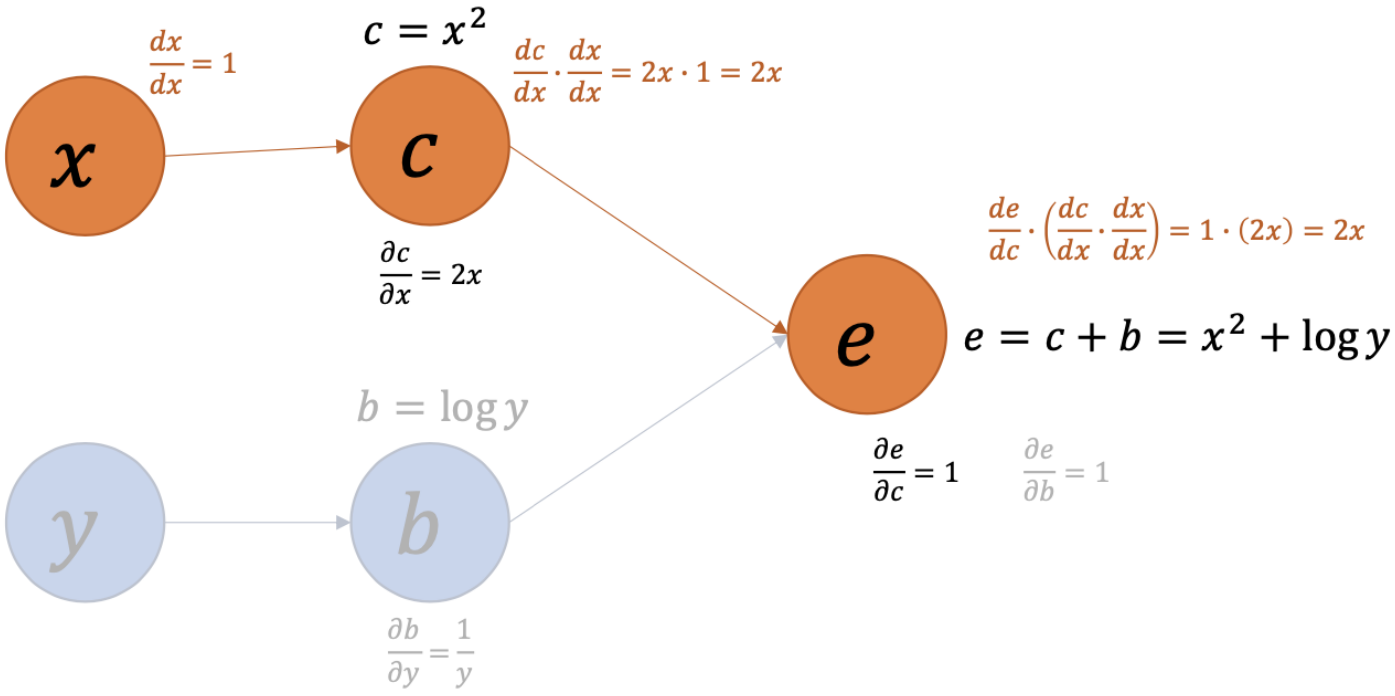
- Given $f(x, y) = x^2 + \log y$



What is the chain rule for de/dx and de/dy?

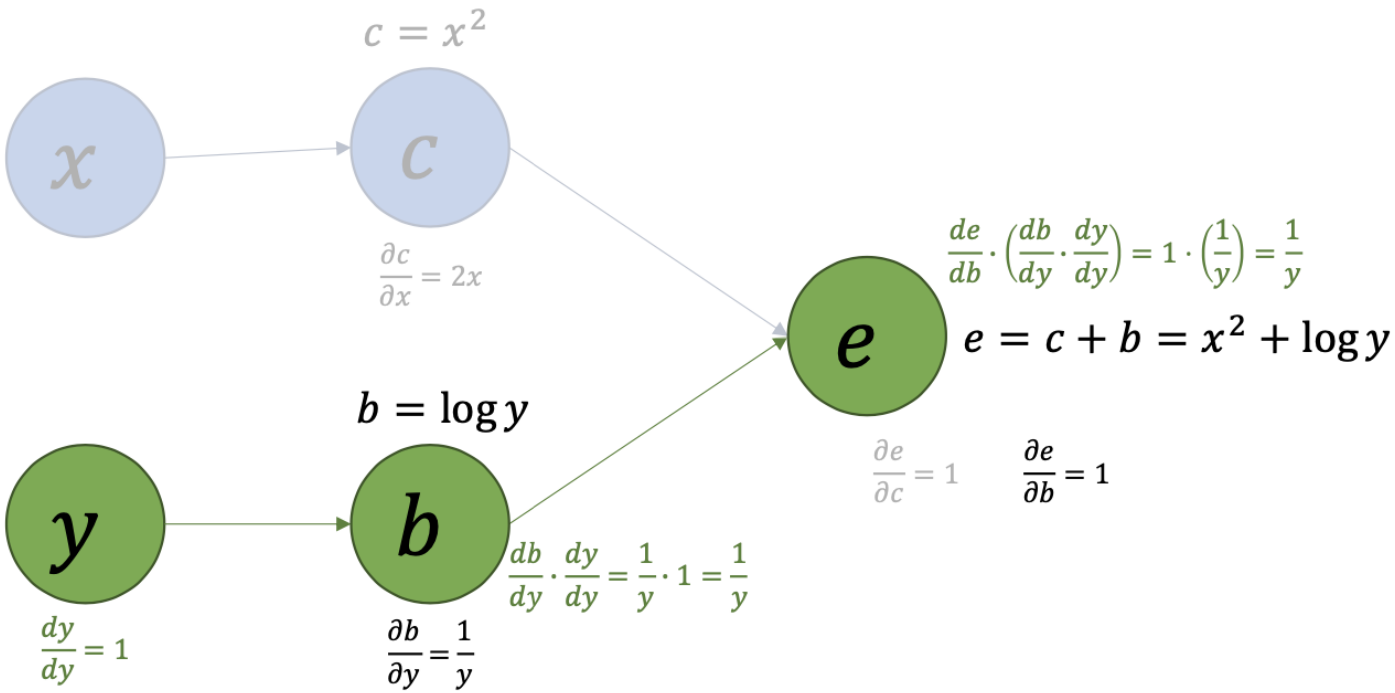


Forward Mode Autodiff



Forward Mode Autodiff

- Can do the same thing starting from the second input:



Time and Memory Complexity

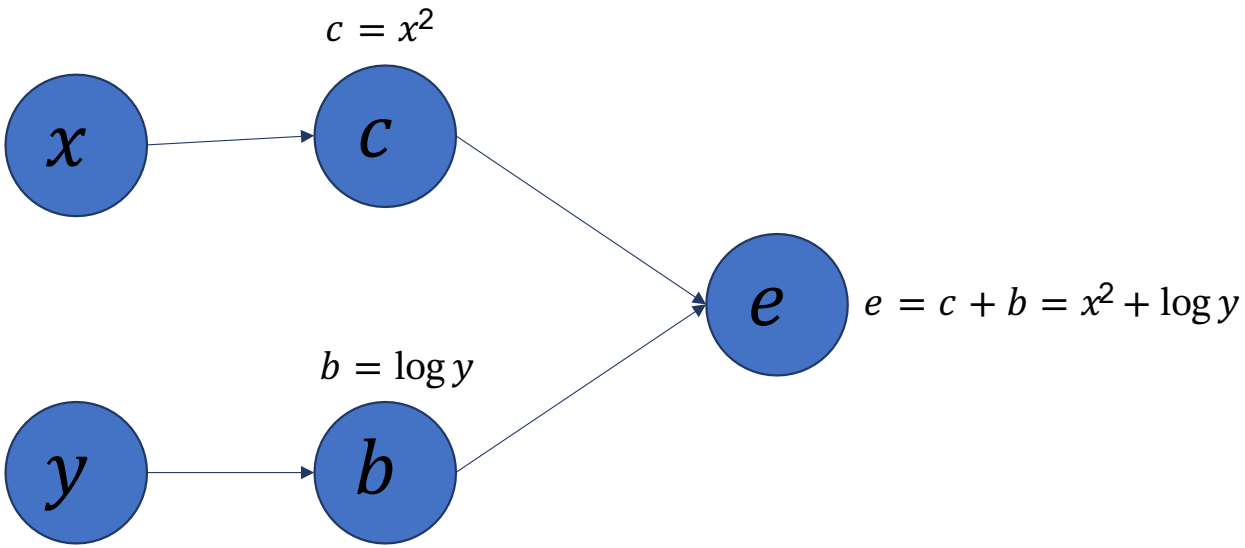
- N: Number of input features to the network
- K: Number of nodes in the graph
- Time: O(N*K)
- Memory: O(1)

Two Main "Flavors" of Autodiff

- Forward Mode Autodiff
  - Compute derivatives alongside the program as it is running
- **Reverse Mode Autodiff**
  - Run the program, then compute derivatives (in reverse order)

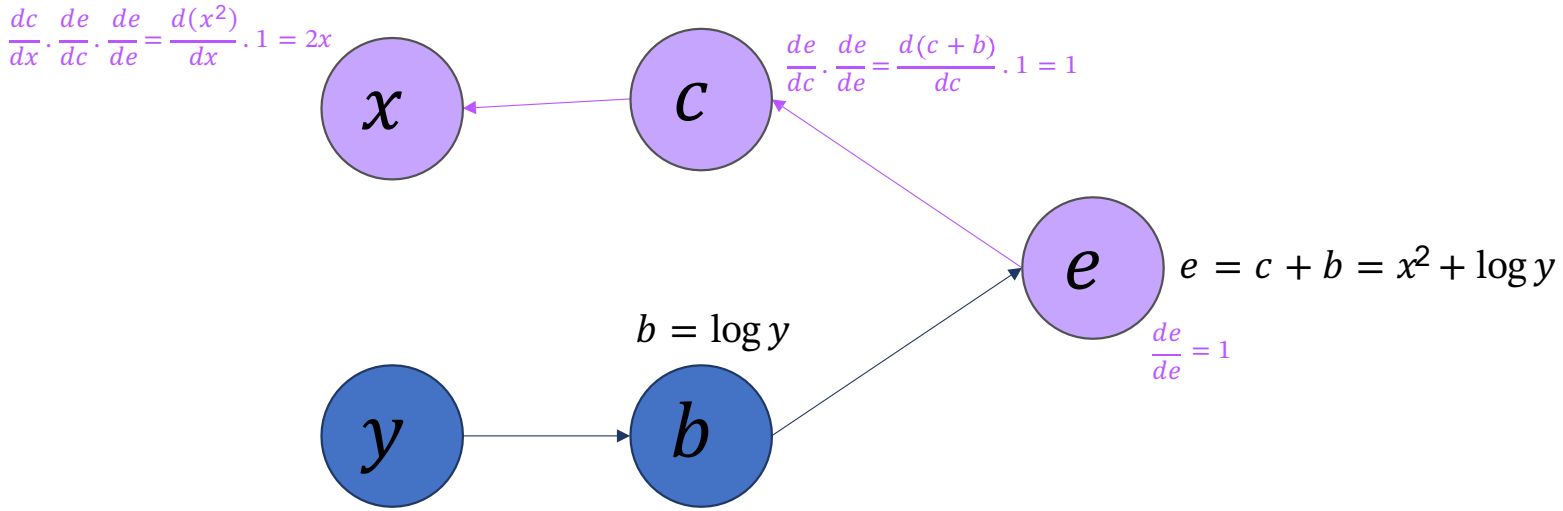
Reverse Mode Autodiff

- Idea: first, run the function toward to the produce the graph
- $f(x, y) = x^2 + logy$



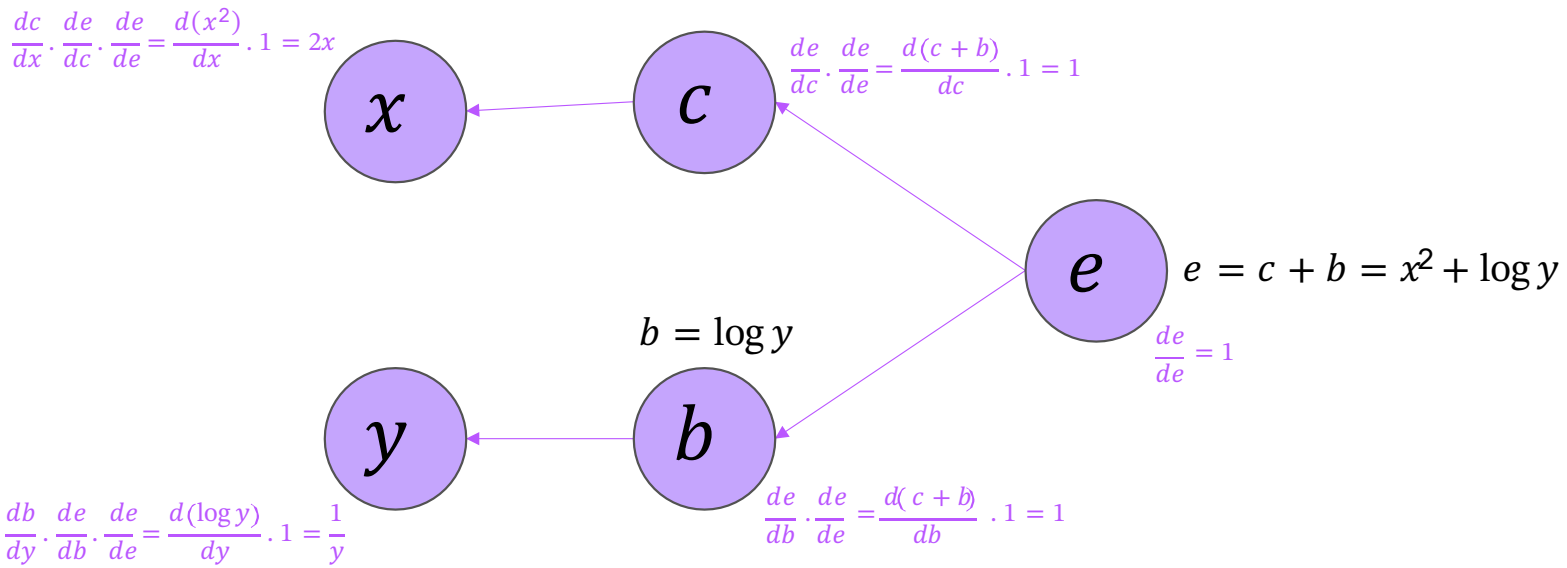
Reverse Mode Autodiff

- Then, compute derivatives backward from the final node toward the inputs



Reverse Mode Autodiff

- Then, compute derivatives backward from the final node toward the inputs



Time and Memory Complexity

- N: Number of input features to the network
- K: Number of nodes in the graph
- Time: O(K)
- Memory: O(K)

Reverse Mode Autodiff is Time Efficient

- Forward mode: $O(N * K)$ time, $O(1)$ memory
- Reverse mode: $O(K)$ time, $O(K)$ memory
- The memory cost comes from having to keep the entire graph from the forward pass in order to then differentiate backwards

Outline

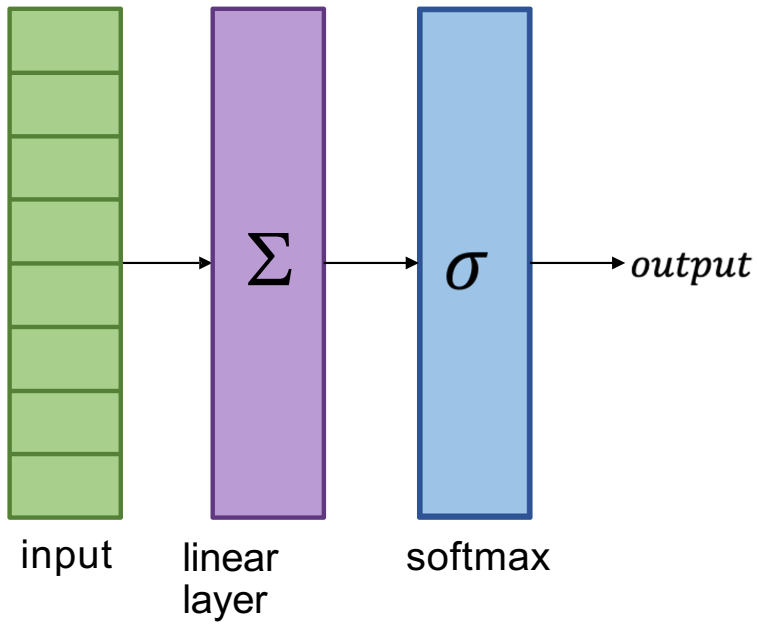
- Model optimizer
- Backpropagation and autodiff
- **Multi-layer NNs and activation**

Today's goal – learn to build multi-layer neural networks

- Adding more layers to the network
- Introducing non-linearity (Activation functions)
- Multi-layer neural network with non-linearity

Single Layer Fully Connected Feed Forward Neural Network

- This network can achieve 90% accuracy on the MNIST test set.



Single Layer Fully Connected Feed Forward Neural Network

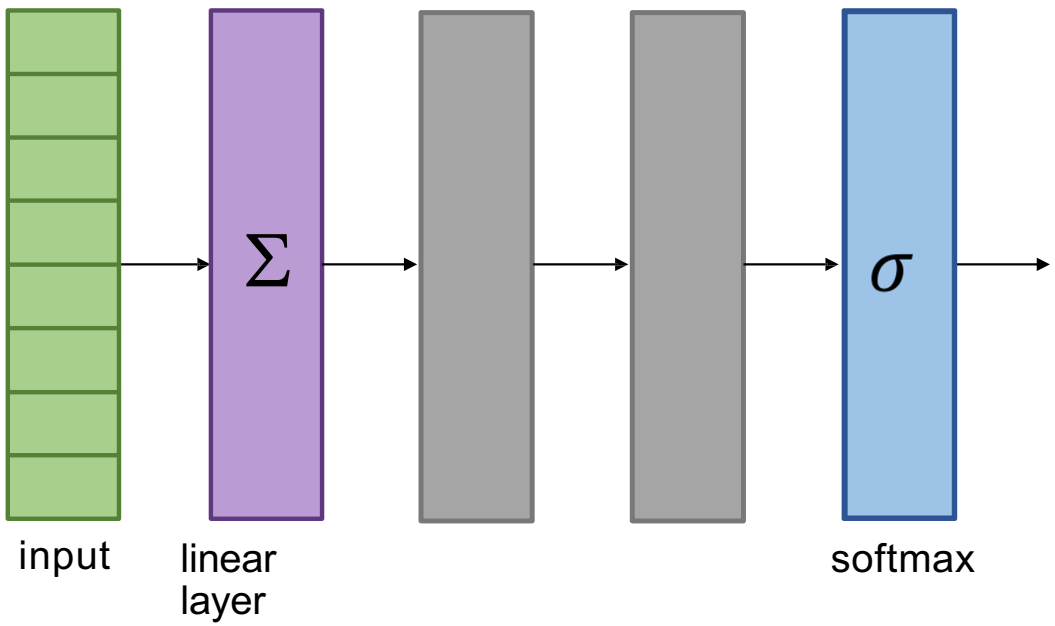
- This network can achieve 90% accuracy on the MNIST test set.
- How can we do better?

Single Layer Fully Connected Feed Forward Neural Network

- This network can achieve 90% accuracy on the MNIST test set.
- How can we do better?
- Go deeper!

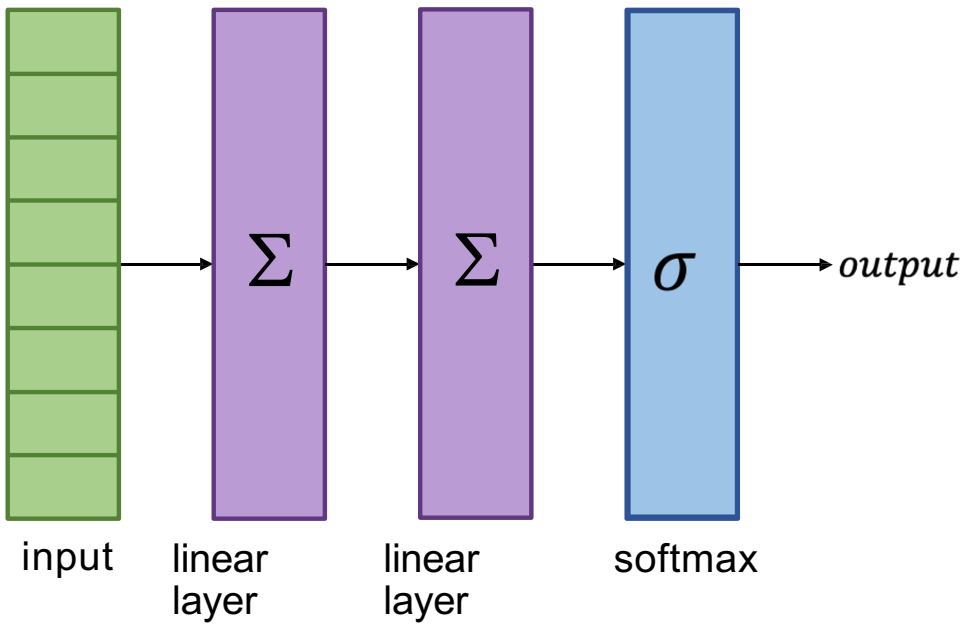
Multi-layer Neural Networks

- Each new layer adds another function to the network
  - f(g(h(... z(x)...)))
  - More composed function -> can represent more complex computations
- Each new layers has its own tunable parameters.
  - More parameters to tune -> can capture more complex patterns in the data.

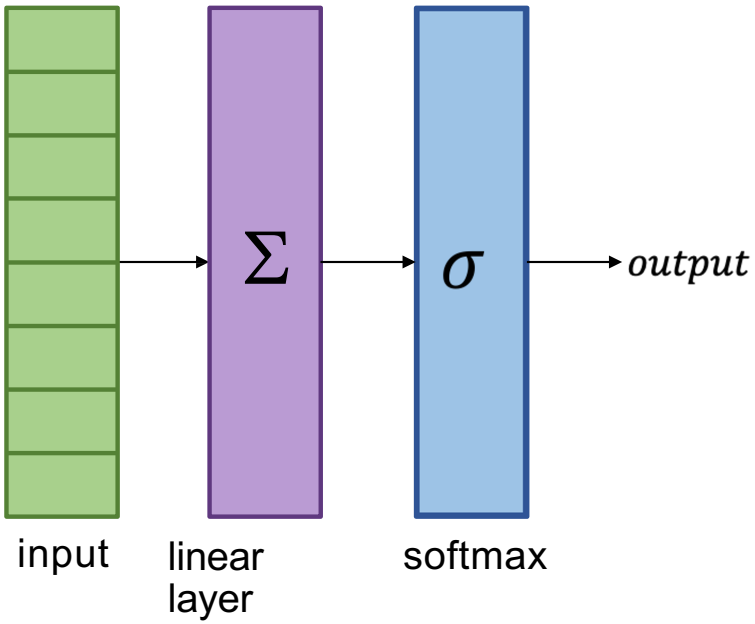


One Way to Make a Multi-layer Neural Network

- Obvious idea: just stack more linear layers
- Let's examine the consequences of this design decision...

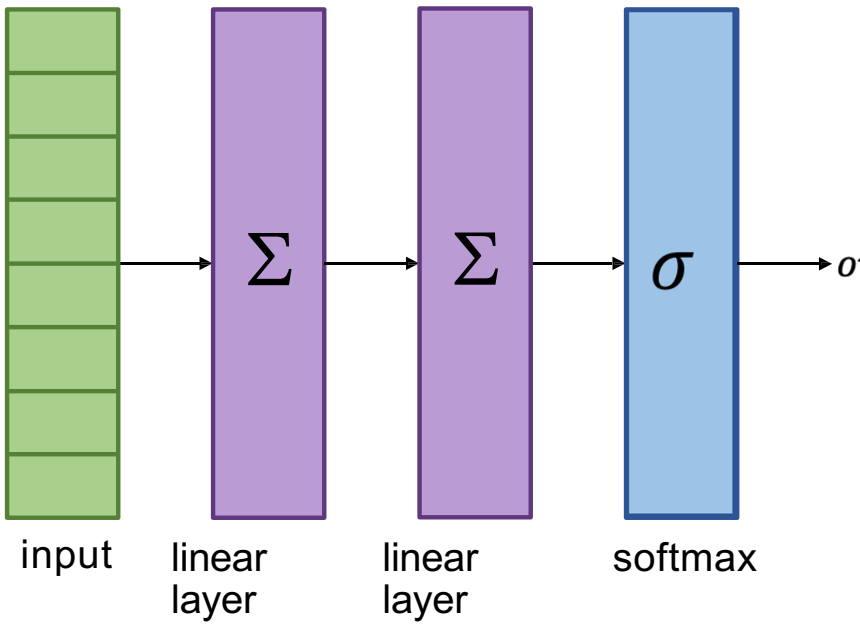


Single-Layer Network (in math)



$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Multi-Layer Network (in math)



$$\sigma\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right)$$

Let's simplify this a bit...

Simplifying multi-layer math ...

$$\sigma\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\left(\begin{bmatrix} w_1 & b_1 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)\right)$$

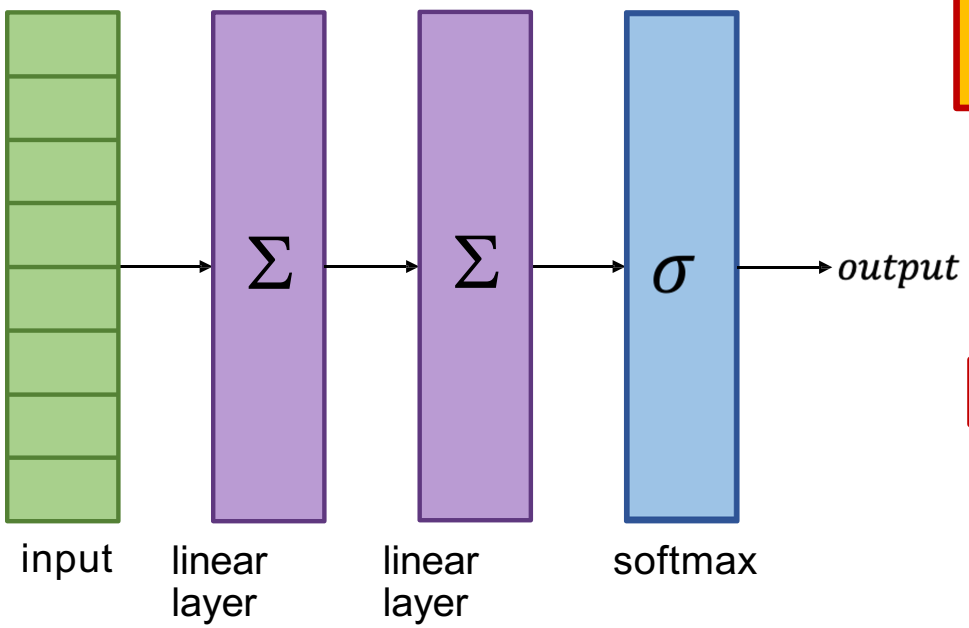
Apply associativity…

$$\sigma\left(\left(\begin{bmatrix} w_2 & b_2 \end{bmatrix}\begin{bmatrix} w_1 & b_1 \end{bmatrix}\right)\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$$

Multiply the matrices...

Same as a one-layer network → $\sigma\left(\begin{bmatrix} w_{12} & b_{12} \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}\right)$

Takeaway: Stacking Linear Layers Isn't Enough

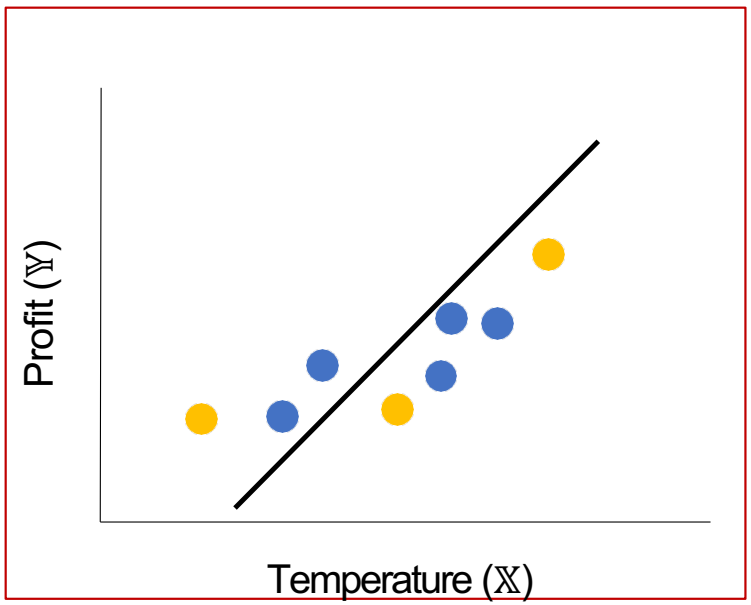


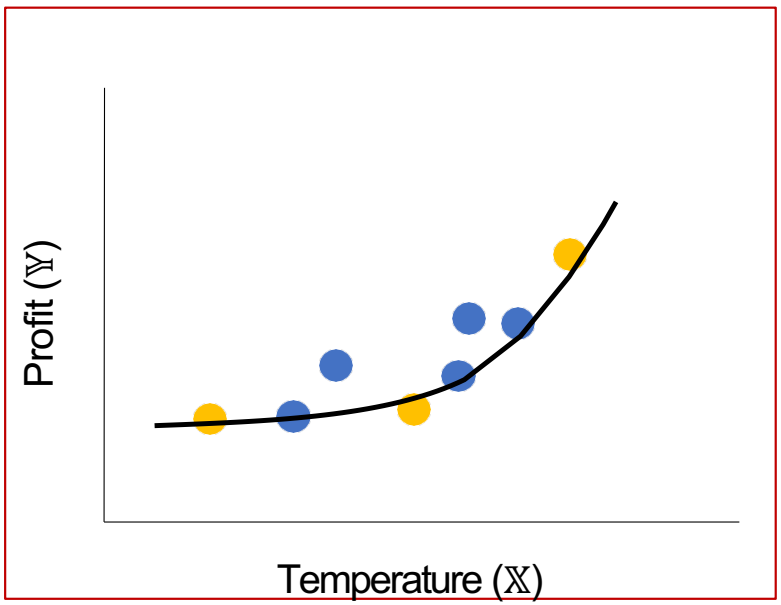
Combination of linear functions is another linear function.

Why is this a problem?

Linear functions may not be sufficient

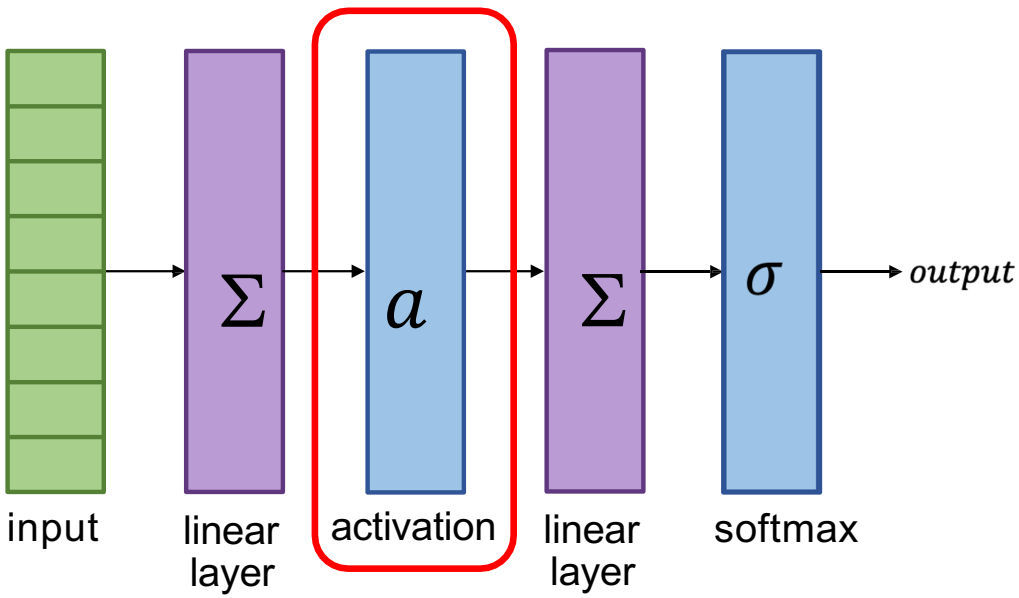
- Root cause of our problem: a composition of linear functions is still linear





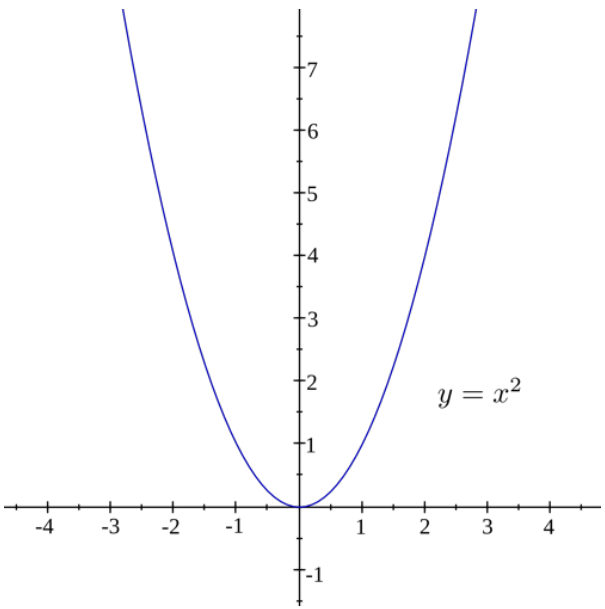
Incorporate non-linearity - Activation Functions

- Root cause of our problem: a composition of linear functions is still linear
- Need some kind of nonlinear function between each linear layer.
- Called an activation function
  - Origin of the name: a neuron “activates” if it gets enough electrochemical input

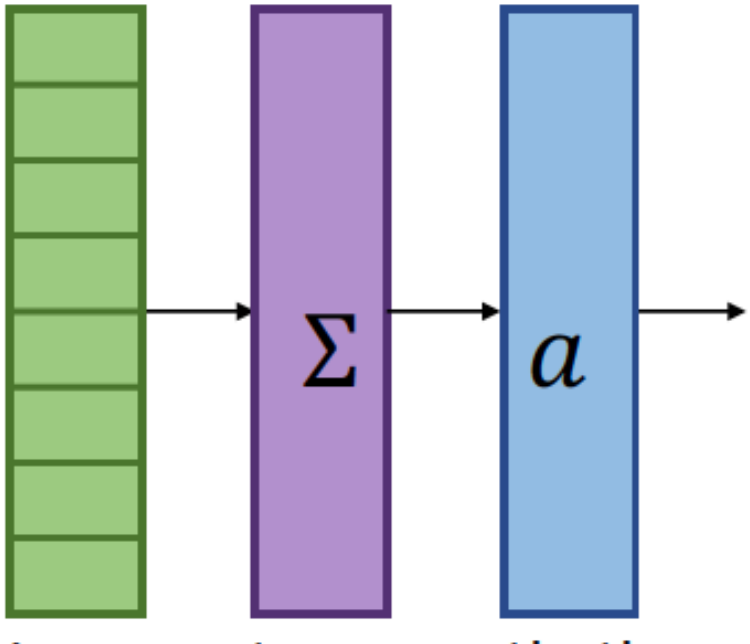


What is a good activation function?

- How about $a(x) = x^2$?
  - Linear -> Quadratic
  - Let's examine the consequences of this design decision
  - In particular, let's look at what happens to the gradient



The gradient of $a(x) = x^2$



$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial a_j}{\partial l_j} \cdot \frac{\partial l_j}{\partial w_{j,i}}$$

$$\frac{\partial a_j}{\partial w_{j,i}} = \frac{\partial (l_j)^2}{\partial l_j} \cdot x_i$$

$$\frac{\partial a_j}{\partial w_{j,i}} = 2l \cdot x_i$$

We have a problem

- New gradient is, in general, larger in magnitude.
- With more layers, gradient gets bigger and bigger...
- Known as the **Exploding Gradient Problem**
- Thus, $a(x) = x^2$ is not a good function.

Previous Gradient

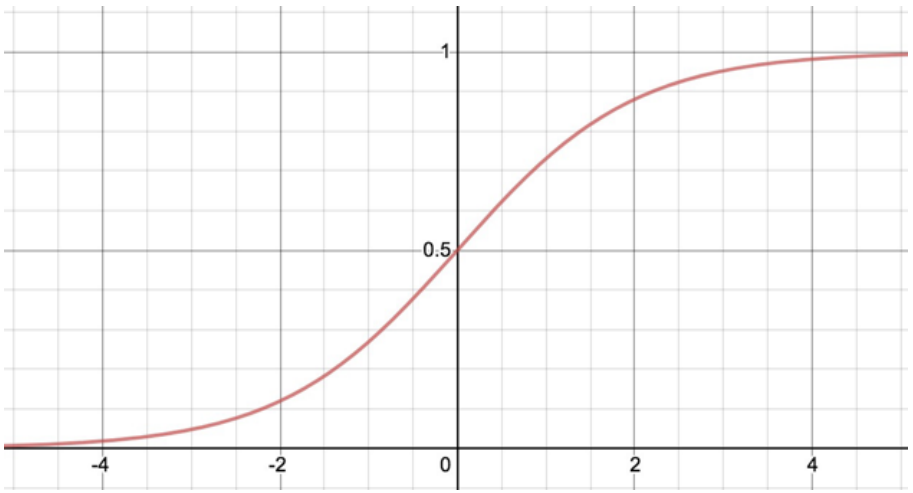
$$\frac{\partial l_j}{\partial w_{j,i}} = x_i$$

New Gradient

$$\frac{\partial a_j}{\partial w_{j,i}} = 2l \cdot x_i$$

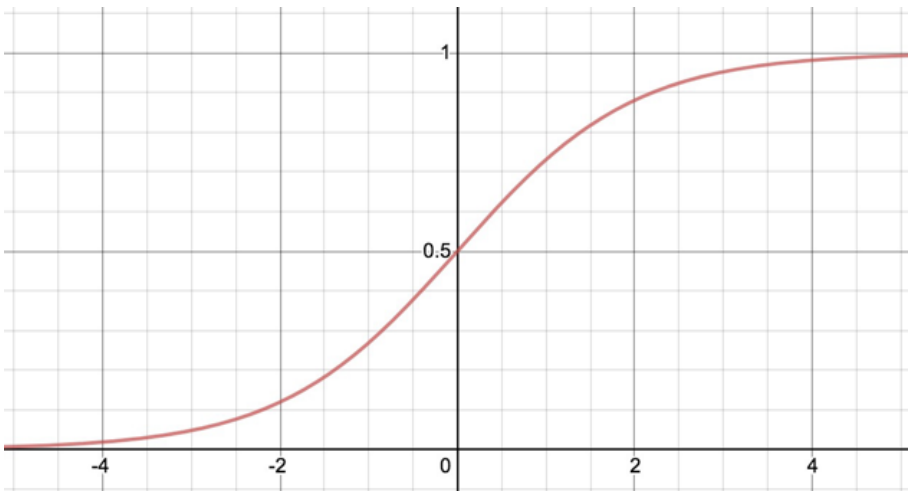
The Sigmoid Activation Function

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
- Historically very popular activation function
- Takes real value and squashes it to range between 0 and 1



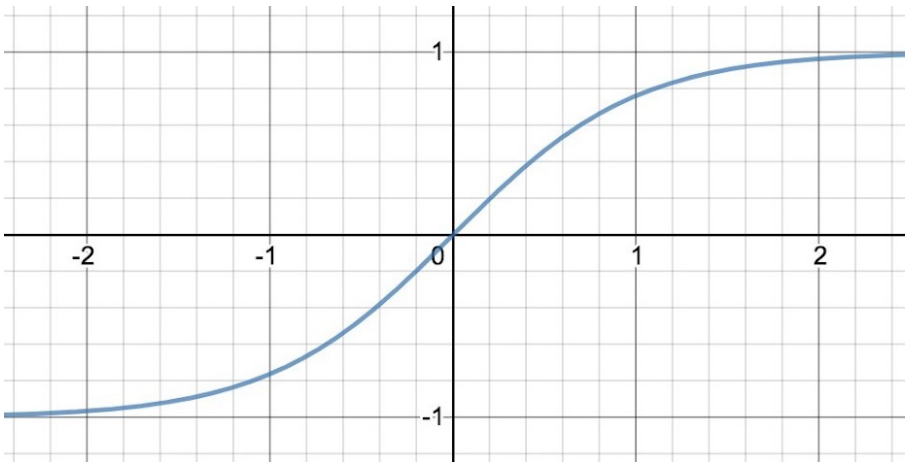
The Sigmoid Activation Function

- $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
- Large negative numbers become 0 and large positive numbers become 1
- Bounded: guarantees gradient cannot grow without bound!



Another Activation Function: Tanh

- $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Output range: [-1,1]
- Somewhat desirable property of keeping the signal that passes through the network “centered” around zero.
- Vanishing gradient problem: we have zero derivative when the function approaches the boundary and the network stops learning.

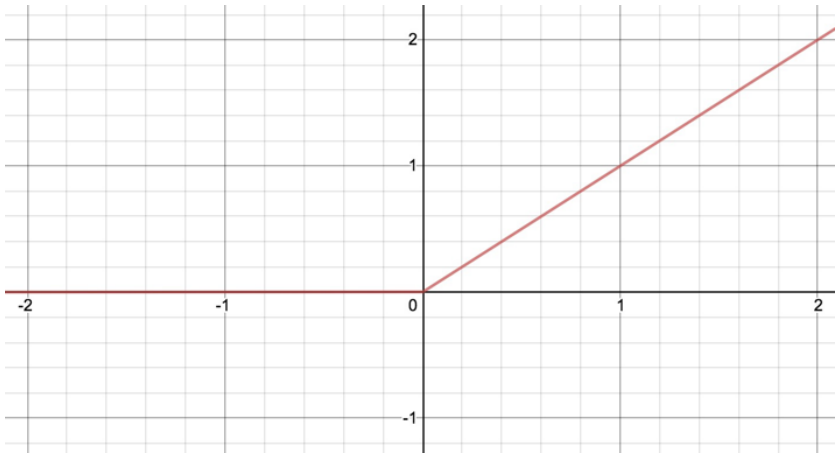


Consequences of Vanishing Gradients

- Problem is exacerbated by stacking multiple layers (gradients shrink more the deeper you go)
- Led to the belief that in practice, neural nets could only ever be a few layers deep...

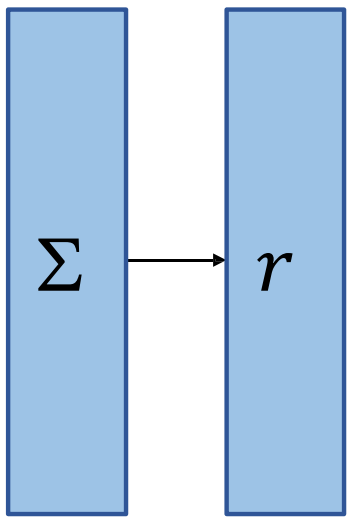
The Rectifier Function

- ReLU(x) = max(0,x)



More commonly known as ReLU

- Rectified Linear Unit
- Technically: Linear layer followed by the rectifier function
- But in most contexts, you will see the rectifier function called "ReLU"



Rectified Linear Unit

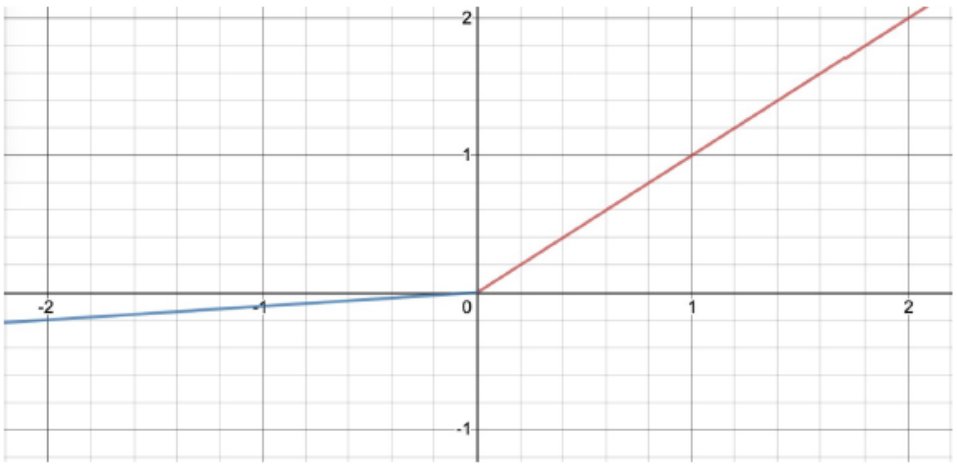
Advantages of ReLU

- Does not suffer from vanishing or exploding gradients!
- Super computationally efficient (avoids the exp calls in sigmoid/tanh)
- Most popular, de-facto ‘standard’ activation function

Leaky ReLU

- $LeakyReLU(x) = max(0, x) + \alpha * min(0, x)$
- We give a tiny positive slop for negative inputs
- Some activation “leaks” through the barrier

$$f(x) = \begin{cases} x, & x > 0 \\ ax, & else \end{cases}$$



Other Activation Functions

Softplus

CLASS `torch.nn.Softplus(beta=1, threshold=20)` [SOURCE]

Applies the element-wise function:

$$\text{Softplus}(x) = \frac{1}{\beta} * \log(1 + \exp(\beta * x))$$

SoftPlus is a smooth approximation to the ReLU function and can be used to constrain the output of a machine to always be positive.

For numerical stability the implementation reverts to the linear function for inputs above a certain value.

Hardshrink

CLASS `torch.nn.Hardshrink(lambd=0.5)` [SOURCE]

Applies the hard shrinkage function element-wise:

$$\text{HardShrink}(x) = \begin{cases} x, & \text{if } x > \lambda \\ x, & \text{if } x < -\lambda \\ 0, & \text{otherwise} \end{cases}$$

LogSigmoid

CLASS `torch.nn.LogSigmoid` [SOURCE]

Applies the element-wise function:

$$\text{LogSigmoid}(x) = \log\left(\frac{1}{1 + \exp(-x)}\right)$$

CELU

CLASS `torch.nn.CELU(alpha=1.0, inplace=False)` [SOURCE]

Applies the element-wise function:

$$\text{CELU}(x) = \max(0, x) + \min(0, \alpha * (\exp(x/\alpha) - 1))$$

More details can be found in the paper Continuously Differentiable Exponential Linear Units .

Parameters

- **alpha** – the α value for the CELU formulation. Default: 1.0
- **inplace** – can optionally do the operation in-place. Default: `False`

Shape:

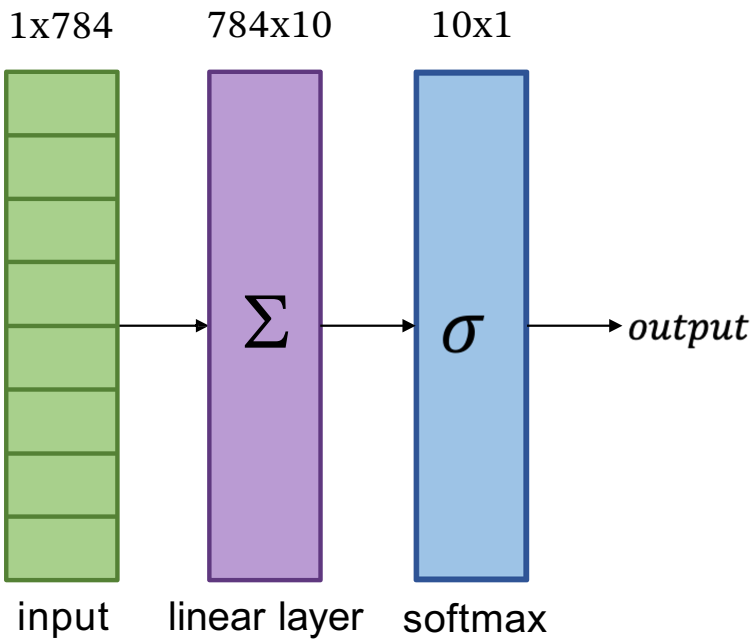
- Input: $(N, *)$ where * means, any number of additional dimensions
- Output: $(N, *)$, same shape as the input

Reasons to use other activation functions

- Bounding network outputs to a particular range
  - Tanh: [-1, 1]
  - Sigmoid: [0, 1]
  - Softplus: $[0, \infty]$
- Example: Predicting a person's age from other biological features
  - Age is a strictly positive quantity
  - We can help our network learn by restricting it to output only positive numbers
  - Use a Softplus activation on the output

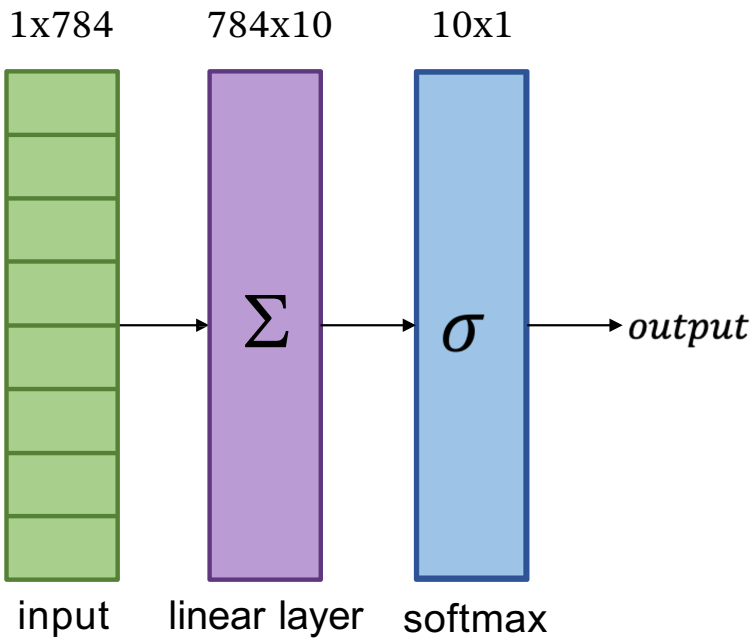
Building a multi-layer network

- Previously:

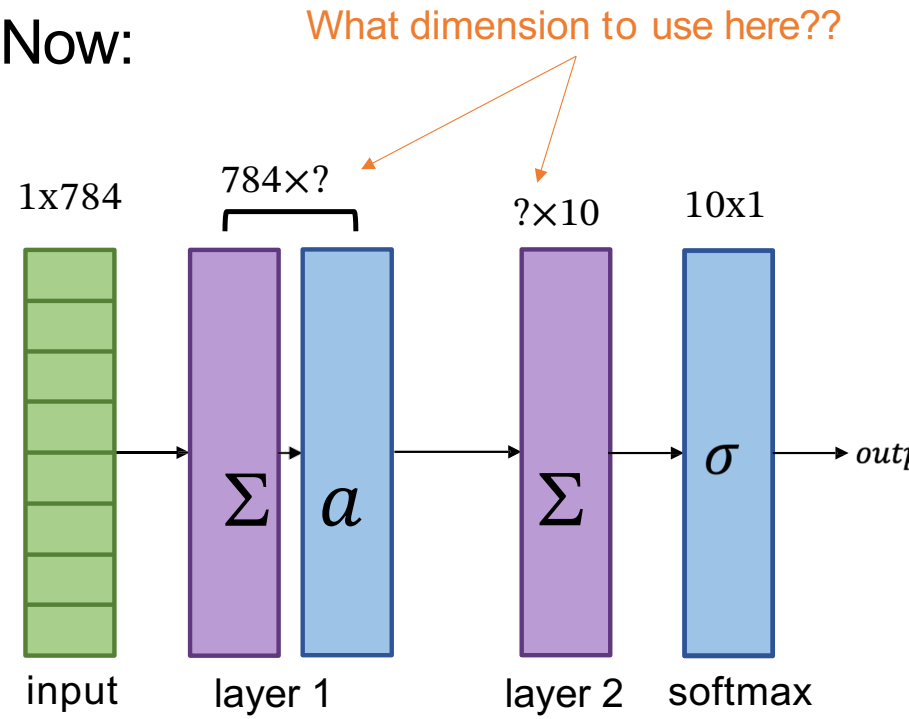


Consequences of adding activation layers

- Previously:

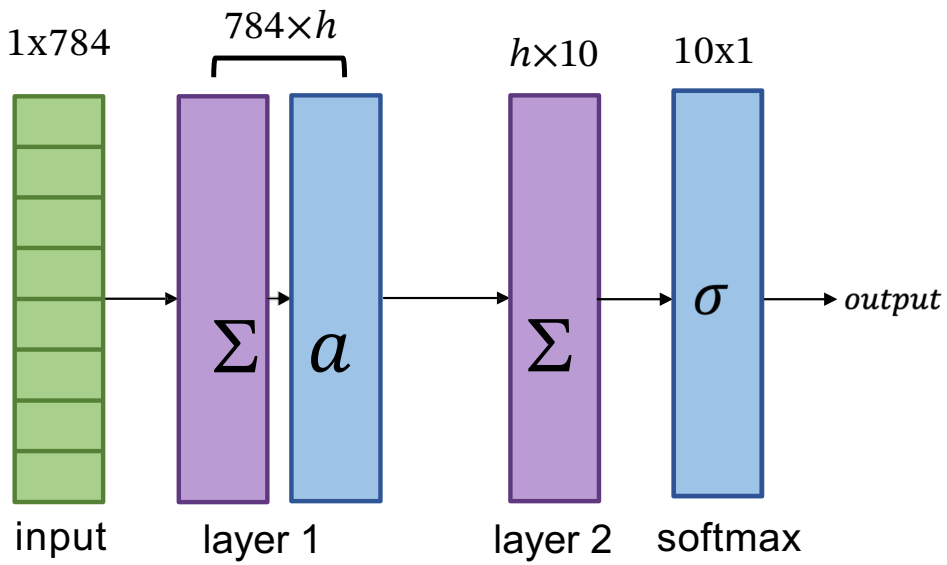


- Now:



“Hidden Layers”

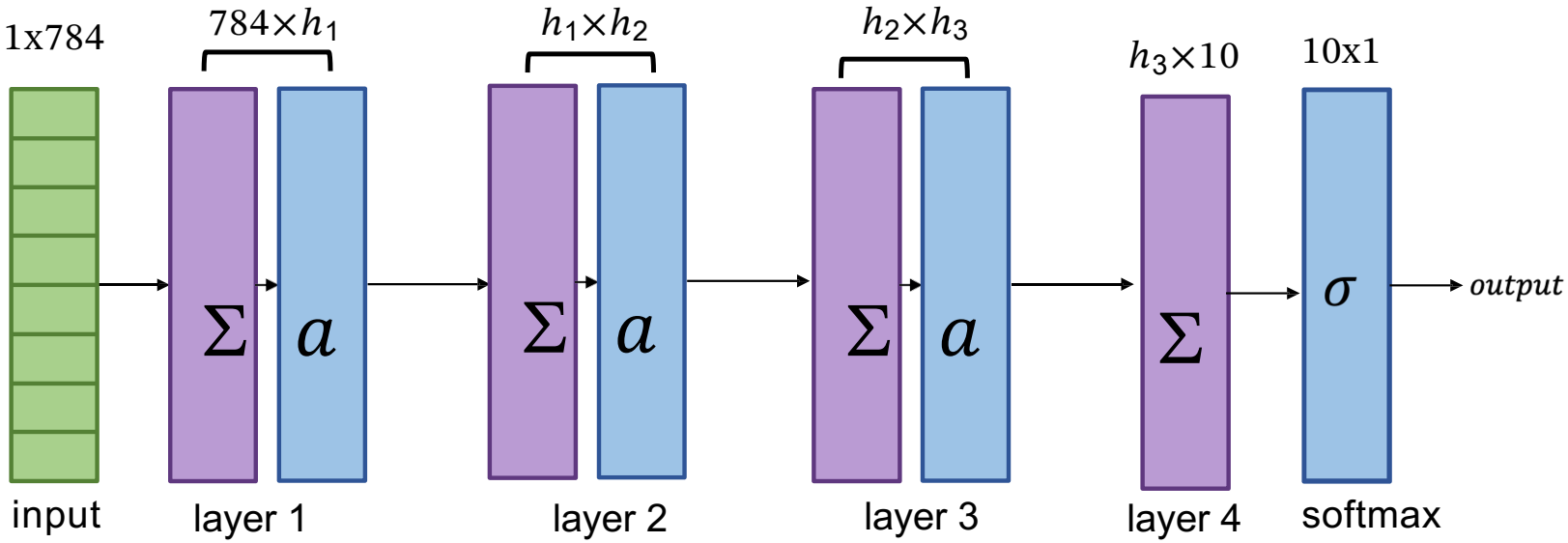
- The output of a function that doesn’t feed into the output layer (like softmax) is called a **hidden layer**.
- Have to set the size *h* of these hidden layers
- More linear units -> more hidden layer sizes



Hyperparameters

- Hidden layer sizes are a **hyperparameter** — configuration external to model, value usually set before training begins
  - Number of epochs, batch size, etc.
  - Contrast this with a learnable parameter, we keep talking about
- Rule of thumb
  - Start out making hidden layers the same size as the input
  - Then, tweak it to see the effect
- There are more principled (and time-consuming) ways to set them
  - Grid search, random search, Bayesian optimization...
  - See here for an overview and more references

What a multi-layer neural network could look like?



What functions can a one-hidden-layer neural net learn?

- Remarkably, a one-hidden-layer network can actually represent any function (under the following assumptions):
  - Function is continuous
  - We are modeling the function over a closed, bounded subset of R^n
  - Activation function is sigmoidal (i.e. bounded and monotonic)