

Backpropagation

Tripp Deep Learning F23

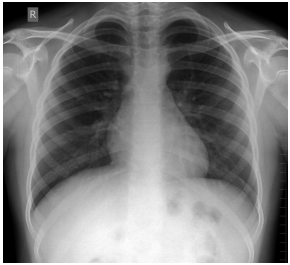


TODAY'S GOAL

By the end of the class, you should understand the motivation for backpropagation and alternatives to it, be able to explain how it works, and be able to use it in your code.

Summary

1. The gradient can be estimated using parameter perturbations, but this is inefficient
2. The gradient can be calculated using the chain rule, but this can be inefficient
3. Backpropagation is an efficient way to use the chain rule in a neural network
4. Backpropagation requires storage of activations throughout the network
5. Backpropagation is done automatically by deep learning software
6. PyTorch creates computational graphs dynamically



Labelled data
Gold-standard examples

Viral pneumonia: 0
Bacterial pneumonia: 0
Not pneumonia: 1

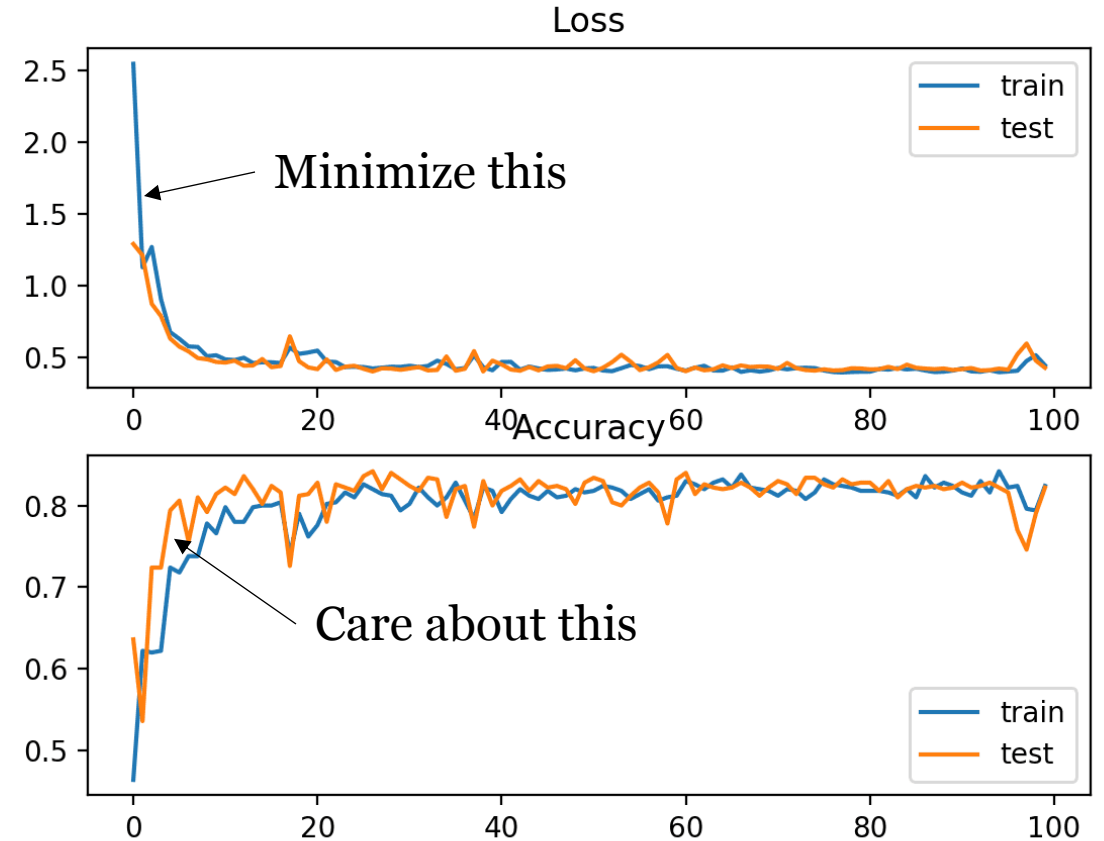
How to change network parameters to reduce loss

Untrained Deep Network
(Capable of a wide variety of functions
given suitable parameter changes.)

↓
Loss
↓ ↑
.2
.65
0.15

Minimizing loss on training data is a proxy

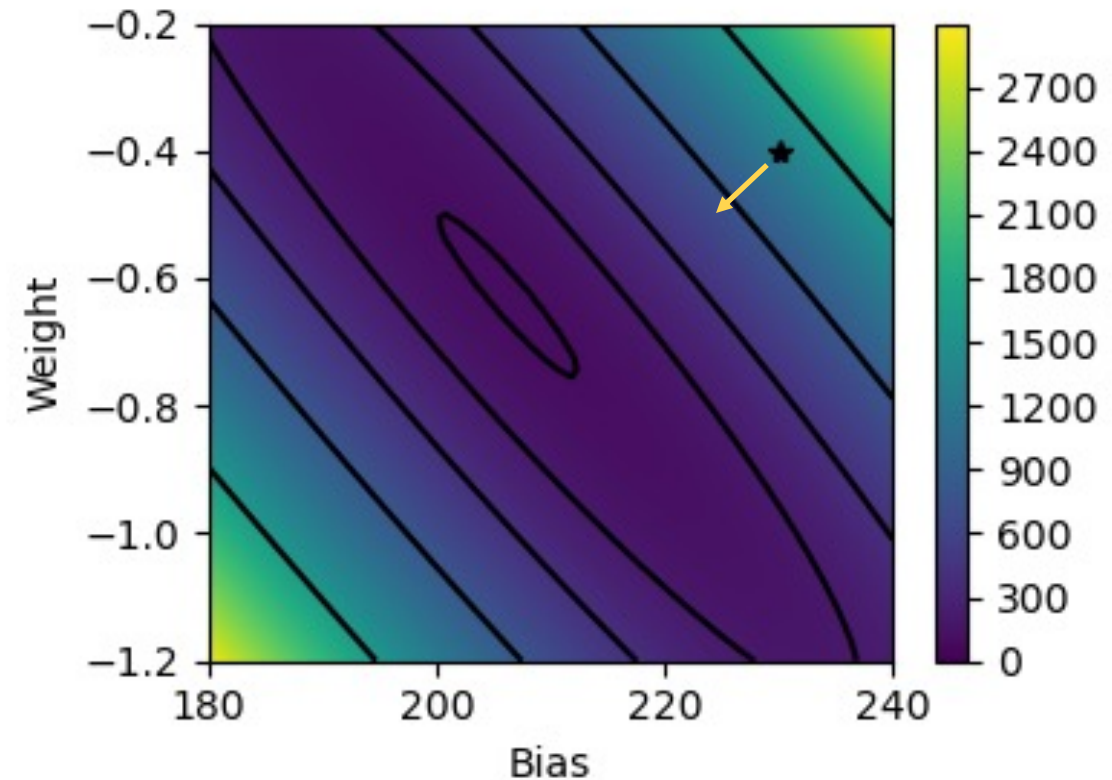
- What we want is to good *performance* on *held-out* data
- What we can do is optimize *loss* on *training* data
 - Predicting labelled data is useless, but supervised learning needs labels
 - Performance metrics we care about (e.g., accuracy) may not be differentiable, but it's only practical to optimize differentiable functions
- This works if loss correlates with performance and the network generalizes



<https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/>

Gradient descent

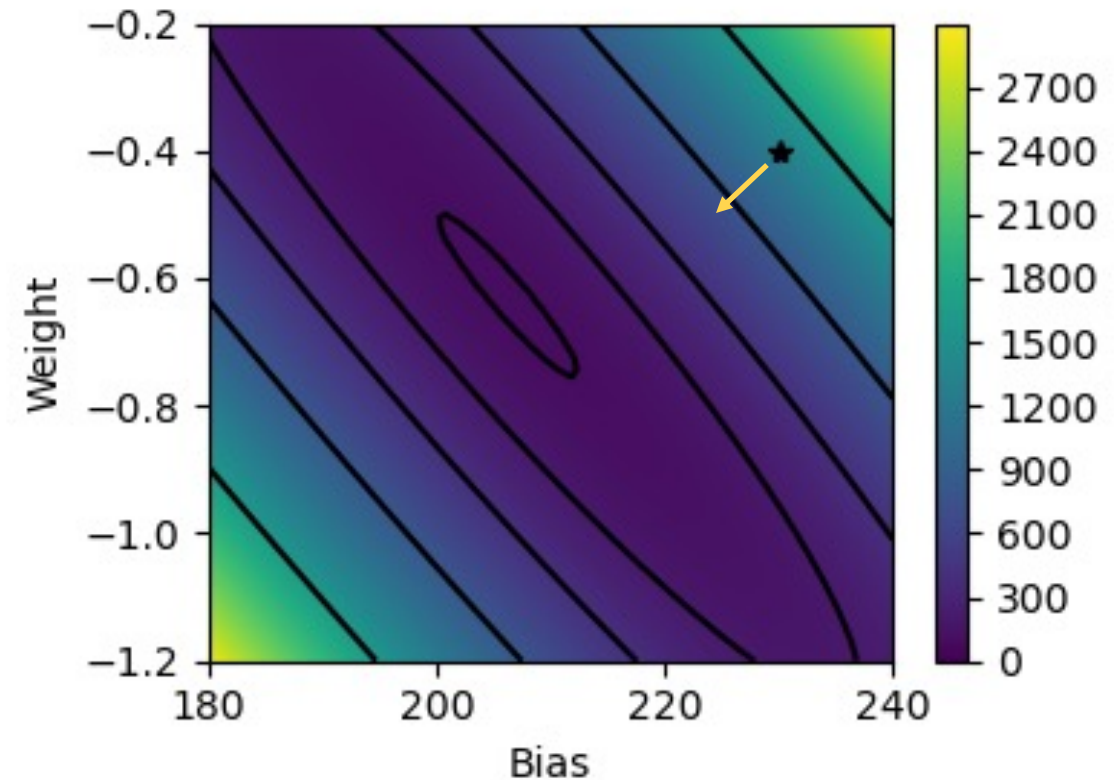
- The gradient of a scalar function of multiple parameters is a vector field that indicates the direction and rate of fastest increase of the function at each point, θ , in the parameter space
- We descend the gradient of the loss, i.e., we move the parameters in the opposite direction
- This usually reduces the loss, although we can't move too far because the gradient is different at each point



Gradient descent

The gradient is also the list of partial derivatives at a given point,

$$\nabla L(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial L}{\partial \theta_1} \\ \vdots \\ \frac{\partial L}{\partial \theta_n} \end{bmatrix}$$



**THE GRADIENT CAN BE ESTIMATED USING
PERTURBATIONS, BUT THIS IS INEFFICIENT**

Finite difference approximation of derivatives

Recall the definition of the derivative of $f(x)$,

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

The finite difference approximation is obtained by using a small $h > 0$,

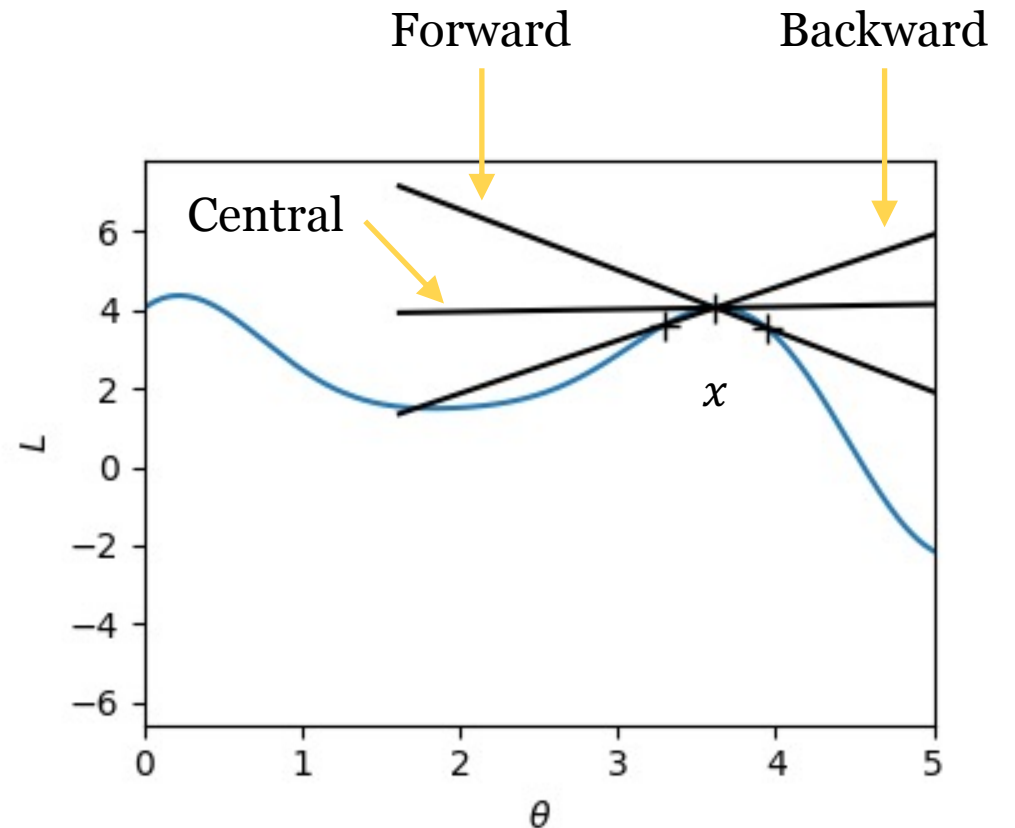
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

An advantage is that this does not require symbolic evaluation of the derivative.

Finite difference approximation of derivatives

- The step h can be positive or negative, leading to the forward and backward differences
- A better approximation is obtained by comparing forward and backward steps, leading to the central difference,

$$f'(x) \approx \frac{f(x + h/2) - f(x - h/2)}{h}.$$

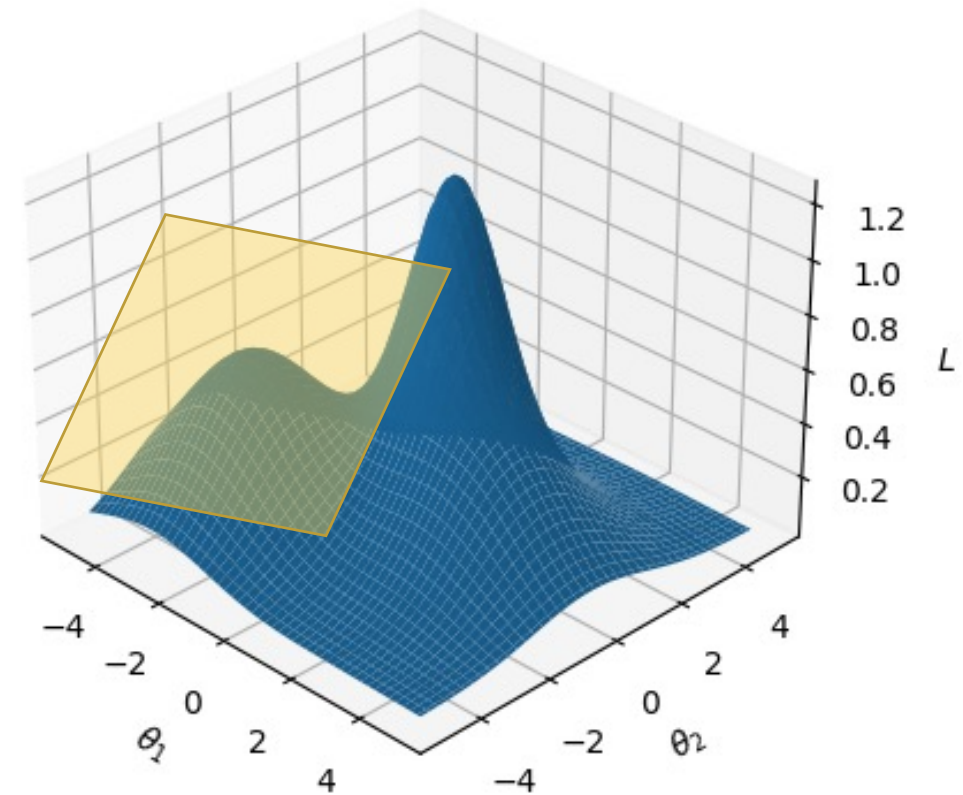


Finite difference approximation of gradient

For functions $f(\mathbf{x})$ of multiple dimensions, the gradient can be approximated by taking steps along each dimension,

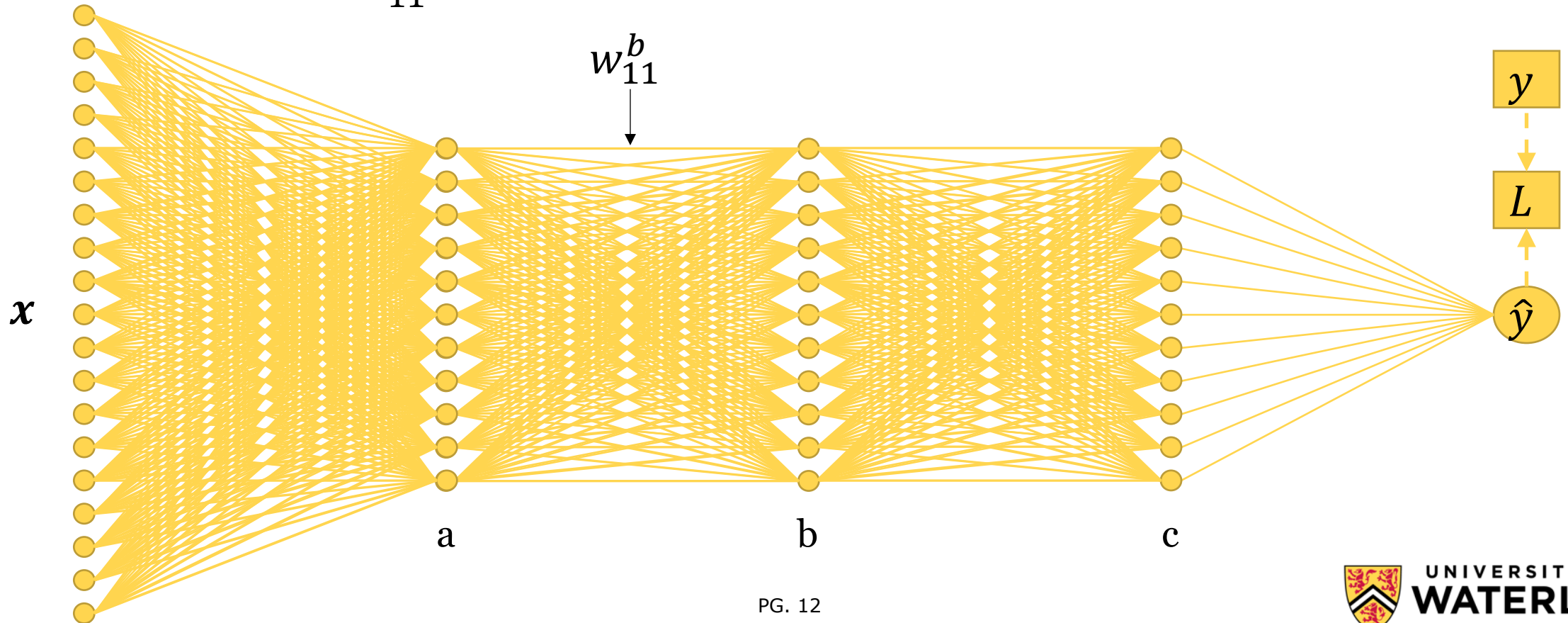
$$\nabla f(\mathbf{x})_i = \frac{f\left(\mathbf{x} + \frac{h}{2}\mathbf{e}_i\right) - f\left(\mathbf{x} - \frac{h}{2}\mathbf{e}_i\right)}{h},$$

Where \mathbf{e}_i is a vector with 1 in the i^{th} entry and 0 elsewhere.



Approximating loss gradients with finite differences

$$\frac{\partial L}{\partial w_{11}^b} \approx \frac{L(x, y, \theta, w_{11}^b + h/2) - L(x, y, \theta, w_{11}^b - h/2)}{h}$$



Requires repeated forward passes through the network

- This is a simple but inefficient way to estimate gradients
- For each training example presented to the network, estimating the derivative of the loss with respect to a certain parameter requires two additional evaluations of parts of the network downstream of that parameter
 - Approximately like doing the same number of forward passes as there are parameters in the network

Good for code verification

- Because it is simple and distinct from the backpropagation algorithm, finite difference approximations can be used to verify that backpropagation code is correct
- Efficiency is less important in this context because
 - Such tests only need to run whenever something changes in code that implements backpropagation
 - In contrast with training, it isn't necessary to calculate the full gradient of a large network many times for this purpose

**THE GRADIENT CAN BE CALCULATED USING
THE CHAIN RULE, BUT THIS CAN BE
INEFFICIENT**

Chain rule of differentiation

Recall the chain rule, if $h(x) = g_2(g_1(x))$ then,

$$h'(x) = g_2'(g_1(x))g_1'(x)$$

This extends to compositions of multiple functions, e.g., if $h(x) = g_3(g_2(g_1(x)))$ then,

$$h'(x) = g_3'(g_2(g_1(x)))g_2'(g_1(x))g_1'(x)$$

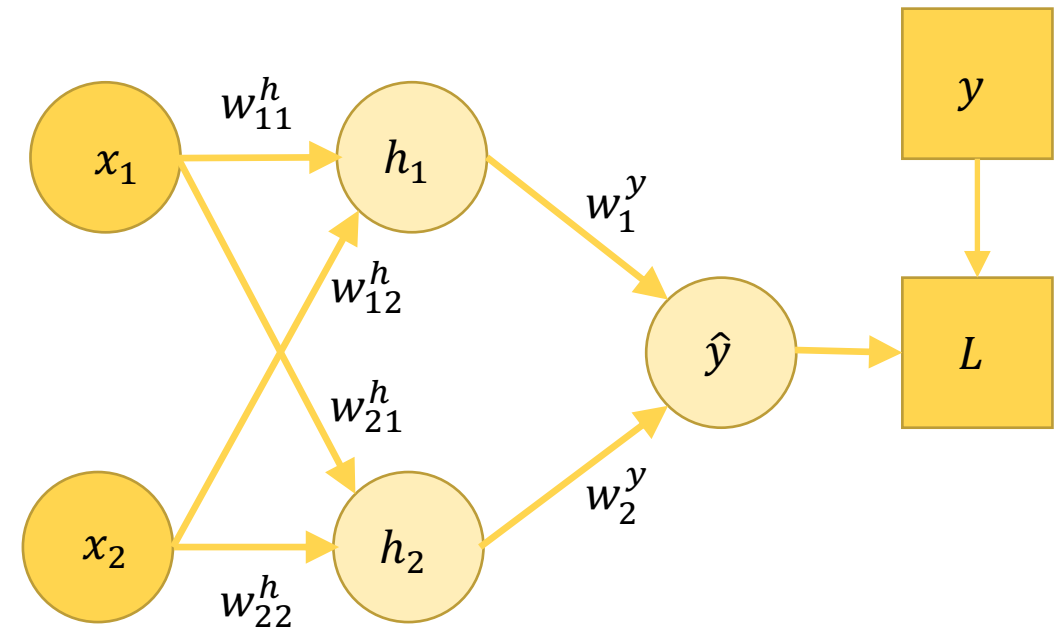
Or (equivalently),

$$\frac{dh}{dx} = \frac{dg_3}{dg_2} \frac{dg_2}{dg_1} \frac{dg_1}{dx}$$

Chain rule of differentiation

- Multi-layer networks are composite functions of their parameters as well as their inputs
- So, we can use the chain rule to find derivatives with respect to parameters
- E.g., hold constant the inputs and all parameters but one, say w_{11}^h . Then the loss is,

$$L(w_{11}^h) = L(\hat{y}(h_1(w_{11}^h))).$$



Chain rule of differentiation

$$L(w_{11}^h) = L(\hat{y}(h_1(w_{11}^h)))$$

Suppose:

$$L(\hat{y}) = \frac{1}{2}(\hat{y} - y)^2$$

$$\hat{y}(h_1) = w_1^y h_1 + w_2^y h_2$$

$$h_1 = \max(0, a_1)$$

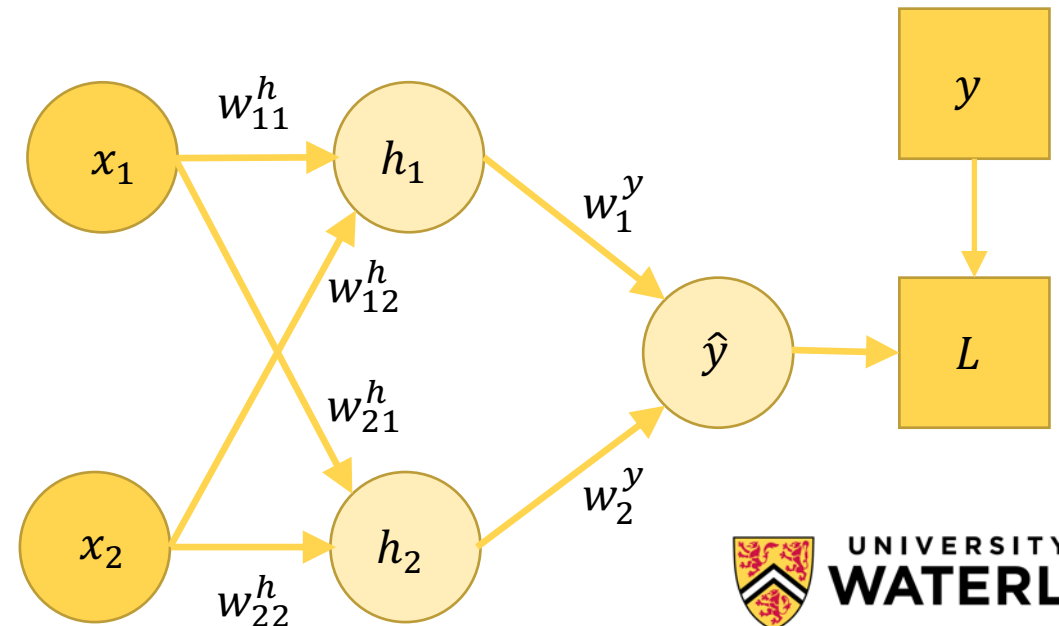
$$a_1 = w_{11}^h x_1 + w_{12}^h x_2$$

Then,

$$\frac{\partial L}{\partial w_{11}^h} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h_1} \frac{\partial h_1}{\partial a_1} \frac{\partial a_1}{\partial w_{11}^h}$$

Note that dh_1/da_1 may be 0 or 1 for different inputs. If g is the nonlinear activation function of this neuron, then

$$\frac{\partial L}{\partial w_{11}^h} = (\hat{y} - y) w_1^y g' x_1$$



Efficiency

- Note the derivative with respect to a given parameter involves function evaluations and multiplication with weights, e.g. (from the last slide),

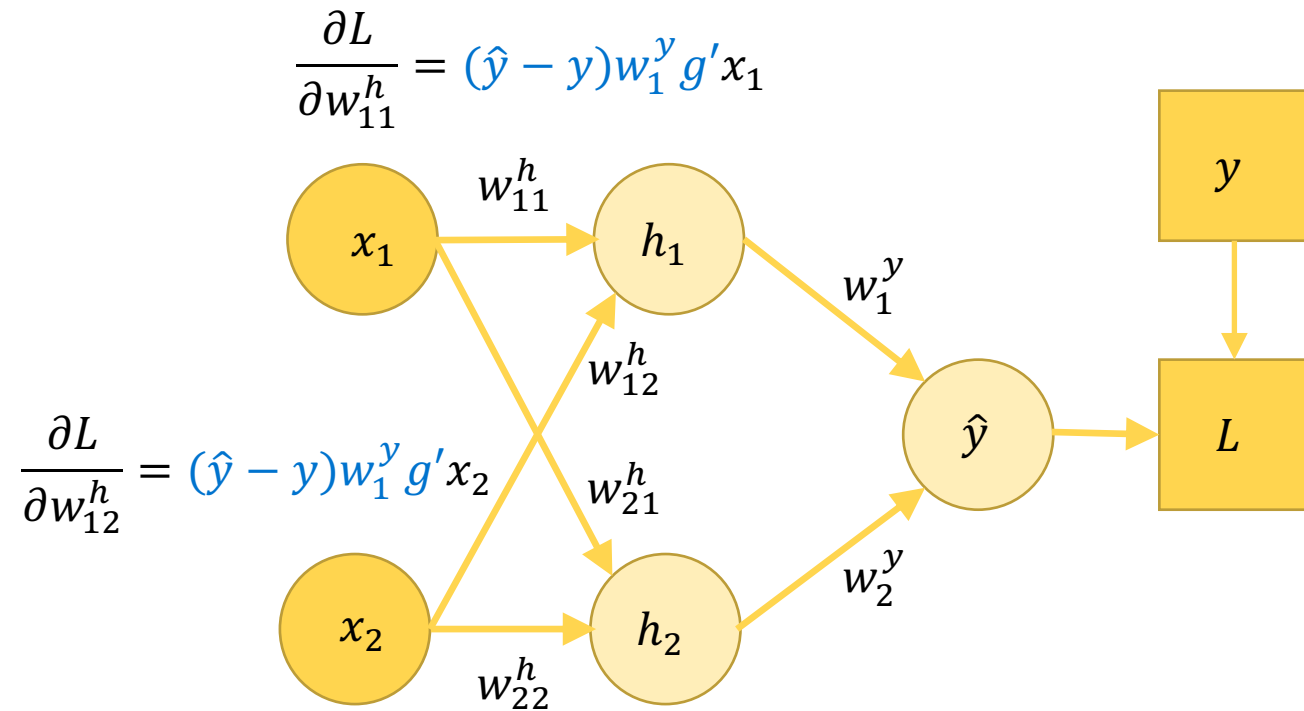
$$\frac{\partial L}{\partial w_{11}^h} = (\hat{y} - y) w_1^y g' x_1$$

- For a parameter near the input, this is about as expensive to evaluate as a forward pass through the network
- So, using the chain rule naively is about as inefficient as using the finite difference approximation

**BACKPROPAGATION IS AN EFFICIENT WAY
TO USE THE CHAIN RULE IN A NEURAL
NETWORK**

Basic idea

- Derivatives of the loss with respect to different parameters **share** many terms
- The backpropagation algorithm calculates the shared terms first, and sends them back through the network
- Calculating each derivative then just requires multiplying a backpropagated value by one unshared term



Backpropagation algorithm

Let the loss associated with the n^{th} item of training data be L_n .

Let the activation (input to the nonlinearity) of the j^{th} unit in a given layer be,

$$a_j = \sum_i w_{ji} z_i ,$$

where z_i is the i^{th} input from the previous layer. Its output is,

$$z_j = g(a_j),$$

where g is the activation function, such as the ReLU function.

Backpropagation algorithm

A weight w_{ji} affects the output only through the activation a_j , so

$$\frac{\partial L_n}{\partial w_{ji}} = \frac{\partial L_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}.$$

$\partial L_n / \partial a_j$ is of great significance in backpropagation, so it has a catchy name,

$$\delta_j = \frac{\partial L_n}{\partial a_j}.$$

Note that $\frac{\partial a_j}{\partial w_{ji}} = z_i$, so we can rewrite,

$$\frac{\partial L_n}{\partial w_{ji}} = \delta_j z_i.$$

Backpropagation algorithm

Recall that δ s of output neurons are typically,

$$\delta_j = \hat{y}_j - y_j$$

To find δ s of non-output neurons, we propagate the output-layer δ s backward through the network. Let j index neurons in a certain layer and k index neurons in the following layer. Then,

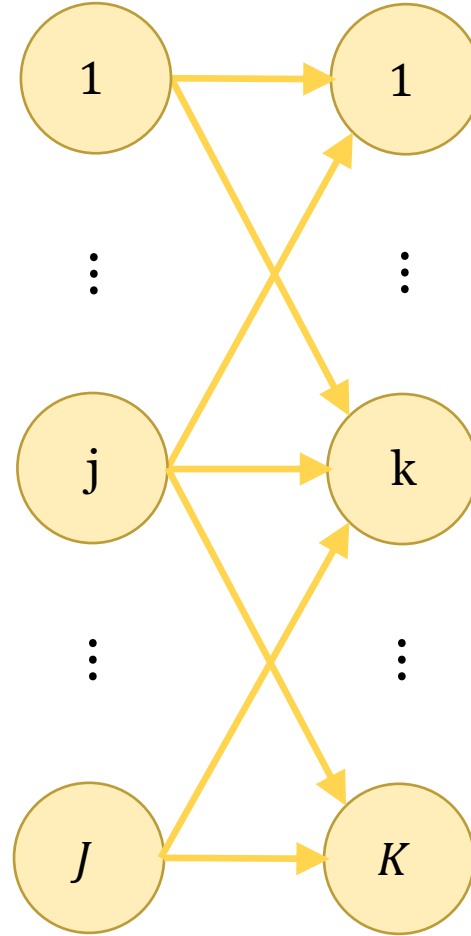
$$\begin{aligned}\delta_j = \frac{\partial L_n}{\partial a_j} &= \sum_k \frac{\partial L_n}{\partial a_k} \frac{\partial a_k}{\partial z_j} \frac{\partial z_j}{\partial a_j} \\ &= \sum_k \delta_k w_{kj} g'(a_j) = g'(a_j) \sum_k w_{kj} \delta_k.\end{aligned}$$

Backpropagation algorithm

Backward propagation

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k$$

This sum is due to z_j affecting the loss through multiple paths.



Forward propagation

$$z_k = g \left(\sum_j w_{kj} z_j \right)$$

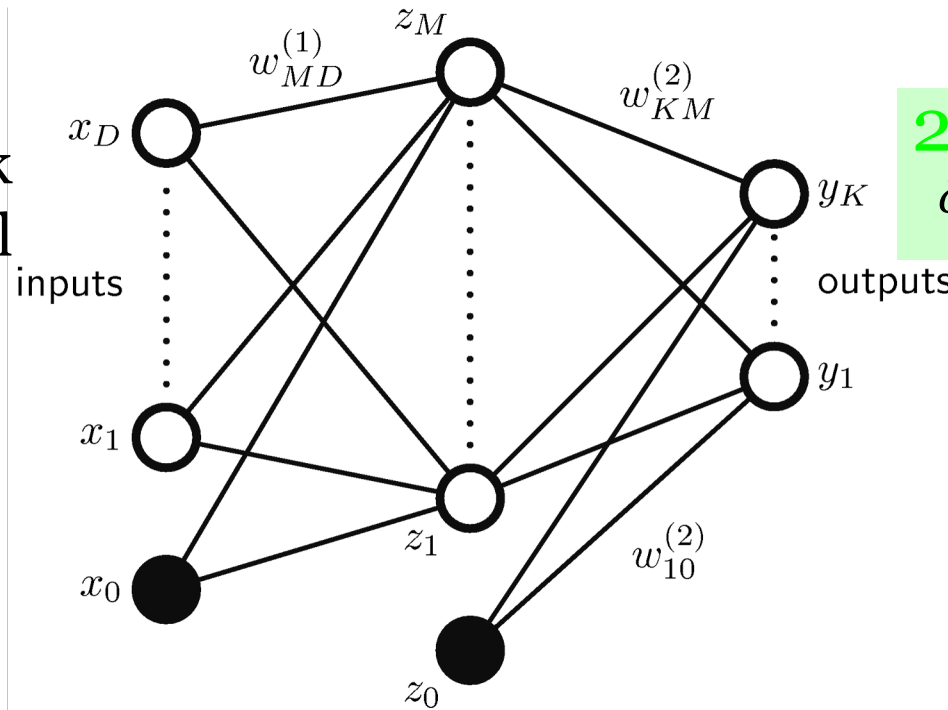
This sum is due to the input affecting z_k through multiple paths.

Summary

To find derivatives of L_n with respect to weights:

1. Apply an input \mathbf{x}_n to the network and propagate forward to find all activations and outputs
2. Find derivative of loss with respect to activations of the output units ($\delta_k = \hat{y}_k - y_k$)
3. Backpropagate to find δ for each hidden unit
4. Use the δ 's to find the derivatives of the loss with respect to the weights

$$1 \quad z_m = h \left(\sum_{d=0}^D w_{md} x_d \right) \quad y_m = g \left(\sum_{m=0}^M w_{km} z_m \right)$$



$$2 \quad \delta_k = \hat{y}_k - y_k$$

$$3 \quad \delta_m = h'(a_m) \sum_{k=1}^K w_{km} \delta_k$$

$$4 \quad \frac{\partial L_n}{\partial w_{md}} = \delta_m x_d$$
$$\frac{\partial L_n}{\partial w_{km}} = \delta_k z_m$$

Efficiency

- Using this approach, calculating the derivatives of the loss with respect to all the parameters in the network is about as expensive as a single forward pass

Group exercise

On paper/whiteboard, forward and backpropagate to calculate the derivatives of the weights of this network for the following dataset:

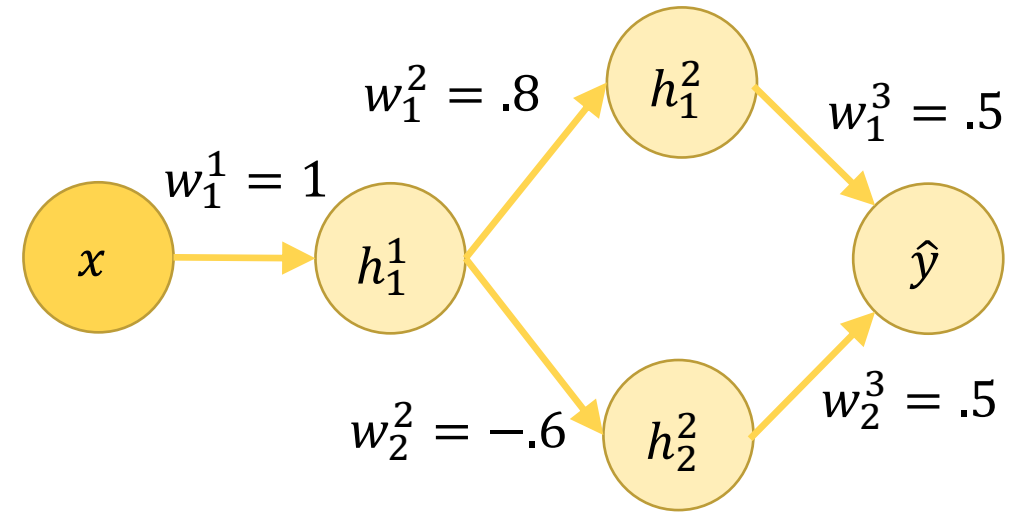
$$X = [-1 \quad 1]$$

$$Y = [.5 \quad .5]$$

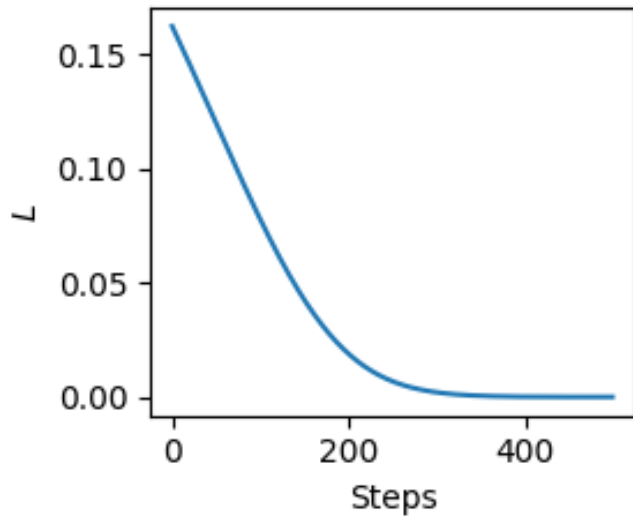
Neurons in the second hidden layer have a ReLU nonlinearity; others do not have a nonlinearity.

The biases of the neurons in the first and third layer are 0; the biases of the hidden neurons in the second layer are -0.5; leave these alone.

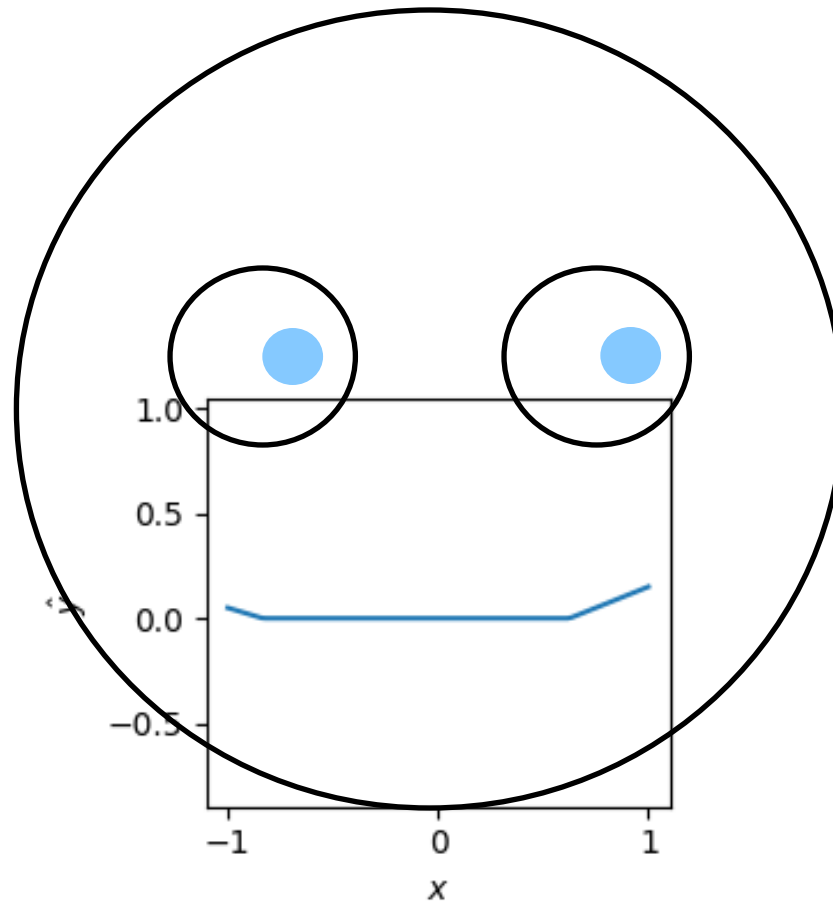
Calculate the weights after a step with example $x=-1$, $y=.5$. Use learning rate 0.1 and loss $L = (\hat{y} - y)^2/2$.



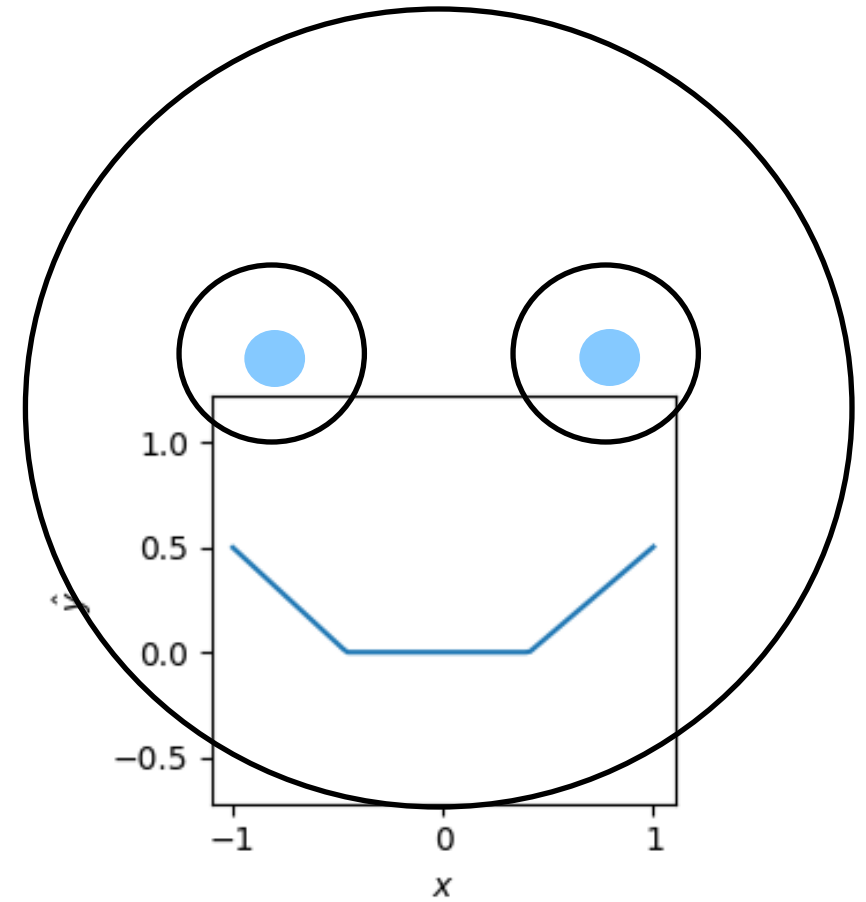
Training curve



Before training



After training



BACKPROPAGATION REQUIRES STORAGE OF ACTIVATIONS THROUGHOUT THE NETWORK

Must keep activations

- Note that δ_j depends not only on parameters and on δ_k , but also on a_j (the activation function g is nonlinear, so its derivative depends on the argument):

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k$$

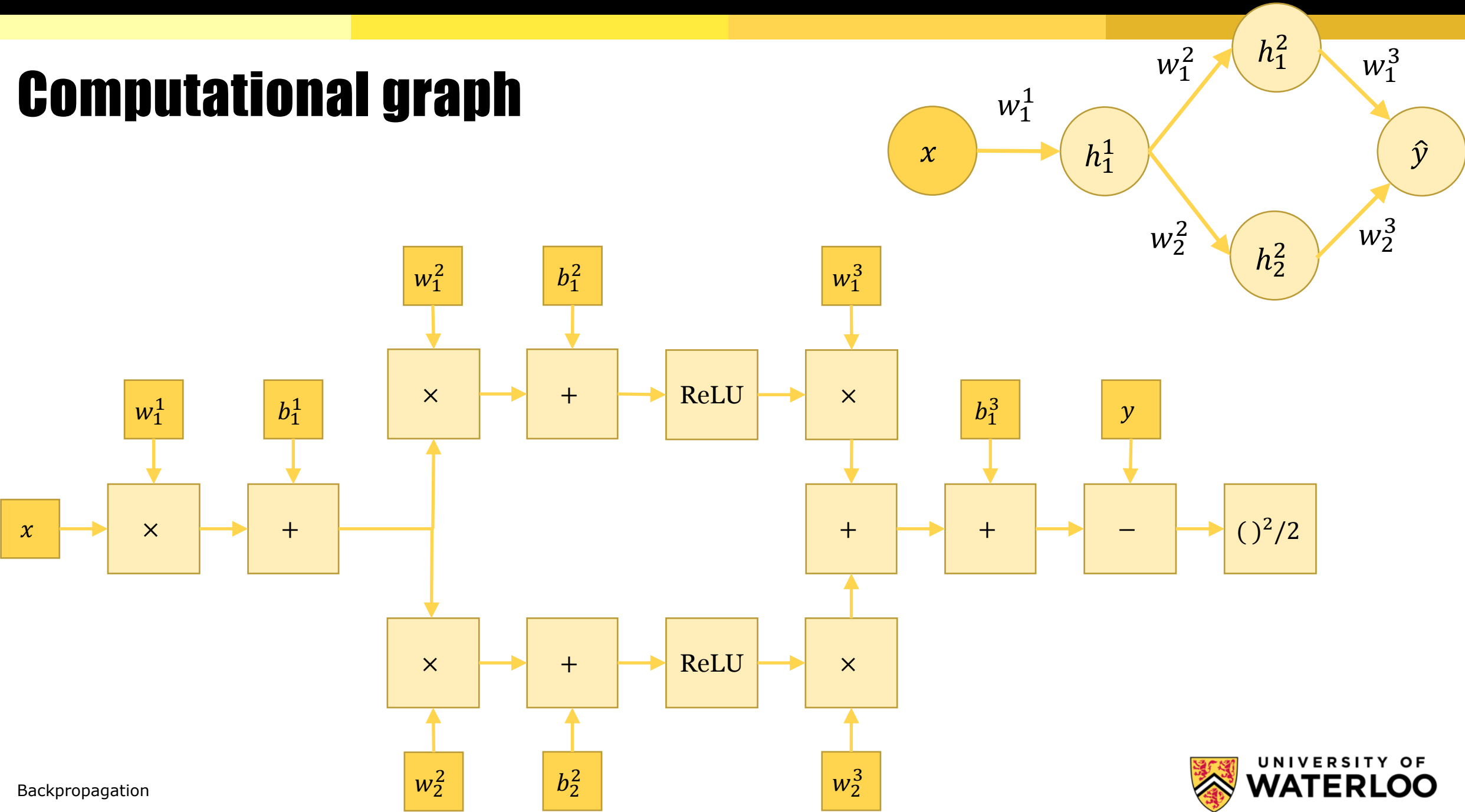
- Also, each weight derivative depends on the corresponding input from the previous layer
- For these reasons, the forward activations must be stored in memory during backpropagation
- This can make up a large fraction of the total memory requirement because
 - In convolutional layers there are usually more activations than parameters
 - We usually process batches of training examples in parallel

**BACKPROPAGATION IS DONE
AUTOMATICALLY BY DEEP LEARNING
SOFTWARE**

Automatic differentiation

- A deep learning package builds a directed acyclic graph called a *computational graph* that keeps track of which operations and arguments lead to each computed value (activations, loss, etc.)
- Each operation has an associated local derivative
- To calculate the derivative of one variable with respect to another, the graph is traversed backward, and the local derivatives are multiplied along the way
- Backpropagation is a special case of autodifferentiation (for neural networks)
- This is not the same as symbolic differentiation
 - A symbolic expression is never produced
 - Each local derivative function is evaluated numerically

Computational graph



Computational graph

- Each forward function knows its associated local derivatives

×

Forward: Multiply(a,b)

$\partial/\partial a: b$

$\partial/\partial b: a$

+

Forward: Add(a,b)

$\partial/\partial a: 1$

$\partial/\partial b: 1$

ReLU

Forward: Max(0,a)

$\partial/\partial a: 1$ if $a > 0$

0 otherwise

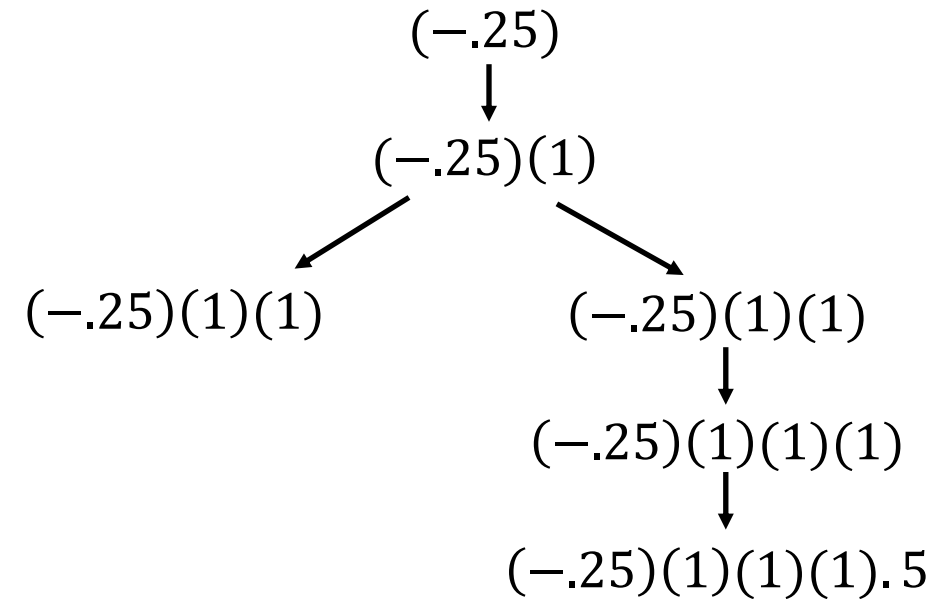
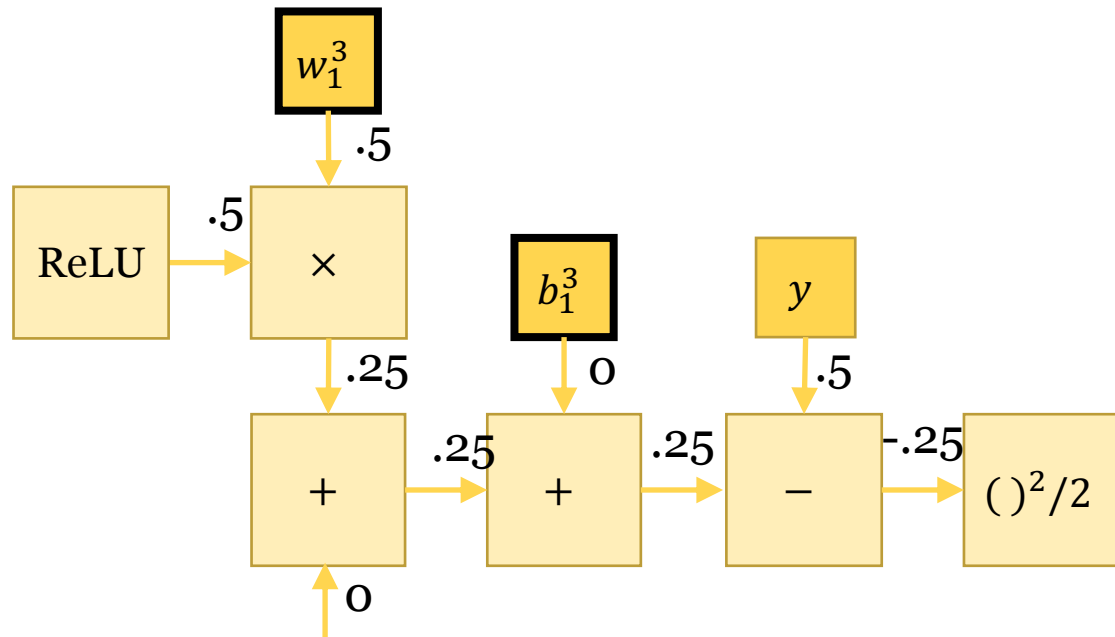
$()^2/2$

Forward: $\frac{1}{2}$ Square(a)

$\partial/\partial a: a$

Autodifferentiation

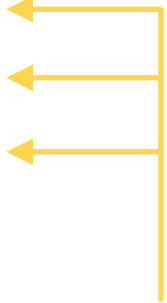
- To find the gradient of the loss with respect to the parameters, the code works backward through the computational graph, multiplying as it goes
- Example: $\left[\frac{\partial L}{\partial w_1^3}, \frac{\partial L}{\partial b_1^3} \right]$



Autograd in PyTorch

- A key data type in PyTorch is Tensor
- Tensors act much like Numpy ndarrays and support similar operations
- Performing an operation on a Tensor creates a new Tensor that keeps references to:
 - The operation that created it
 - The Tensor(s) that went into the operation
- These references make up the computational graph

```
class Network(nn.Module):  
    def __init__(self):  
        super(Network, self).__init__()  
        self.hidden = nn.Linear(784, 100)  
        self.output = nn.Linear(100, 10)  
  
    def forward(self, x):  
        x = torch.flatten(x, 1)  
        x = self.hidden(x)  
        x = F.relu(x)  
        x = self.output(x)  
        return x
```



These lines create new
Tensors with backward
references

PyTorch: Requires Grad

- Tensors have a Boolean attribute called `requires_grad`
- The computational graph only includes Tensors that have `requires_grad=True`
- Tensors derived from at least one other Tensor with `requires_grad=True` have `requires_grad=True`
- Parameters of a neural network have `requires_grad=True` by default
- Set to `requires_grad=False` to freeze a parameter during fine tuning

```
# no computational graph
```

```
a = torch.tensor(1., requires_grad=False)  
b = torch.tensor(1., requires_grad=False)  
c = a+b
```

```
# computational graph includes d, e, f
```

```
d = torch.tensor(1., requires_grad=True)  
e = torch.tensor(1., requires_grad=True)  
f = d+e
```

PyTorch: Backward

- Automatic differentiation is performed by calling a Tensor's backward() method
- This can be called with no arguments if the Tensor is a single number
- Normally we call backward() on the loss (which is a single number)
- This also frees the values calculated during the forward pass

```
for batch_idx, (inputs, targets) in enumerate(train_loader):  
    optimizer.zero_grad()  
    outputs = model(inputs)  
    loss = nn.CrossEntropyLoss()(outputs, targets)  
    loss.backward()  
    optimizer.step()
```

Create loss function

Call loss function

Automatic differentiation backward from loss

PyTorch: Accumulating gradients

- Every time you call `backward()`, the calculated gradients are added to any existing gradients
- This makes sense because a node doesn't know whether multiple backward calls are due to user code or multiple paths through the network
- Once you take a gradient descent step (typically after a single `backward()` call on the loss Tensor) you should call `zero_grad()` on the optimizer to set the gradients to zero

Code:

```
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(2., requires_grad=True)
c = a + b
```

```
c.backward()
print(a.grad)
c.backward()
print(a.grad)
a.grad.zero_()
c.backward()
print(a.grad)
```

Output:

```
tensor(1.)
tensor(2.)
tensor(1.)
```


PYTORCH CREATES COMPUTATIONAL GRAPHS DYNAMICALLY

The graph is built as tensor operations are executed

```
def print_graph(grad_fn, indent=0):
    if isinstance(grad_fn, tuple):
        grad_fn = grad_fn[0]
    if grad_fn is not None:
        print('{}{}'.format(' ' * indent, type(grad_fn)))
    if hasattr(grad_fn, 'next_functions'):
        for nf in grad_fn.next_functions:
            print_graph(nf, indent=indent+2)
```

```
a = torch.tensor([1.], requires_grad=True)
```

```
b = torch.tensor([1.], requires_grad=True)
```

```
c = torch.tensor([2.], requires_grad=True)
```

```
d = c * (a + b) ← Graph built when this line runs
```

```
print_graph(d.grad_fn)
```

Output:

```
<class 'MulBackward0'>
  <class 'AccumulateGrad'>
    <class 'AddBackward0'>
      <class 'AccumulateGrad'>
        <class 'AccumulateGrad'>
```

Leaf nodes

The graph structure is determined at runtime

```
class Network(nn.Module):  
    def __init__(self):  
        super(Network, self).__init__()  
        self.hidden = nn.Linear(784, 100)  
        self.output = nn.Linear(100, 10)  
  
    def forward(self, x):  
        x = torch.flatten(x, 1)  
        x = self.hidden(x)  
        if np.random.rand() < .5:  
            x = F.relu(x)  
        return self.output(x)
```

Depends
on run

Output of one run:

```
<class 'AddmmBackward0'>  
  <class 'AccumulateGrad'>  
    <class 'ReluBackward0'>  
      <class 'AddmmBackward0'>  
        <class 'AccumulateGrad'>  
          <class 'TBackward0'>  
            <class 'AccumulateGrad'>  
              <class 'TBackward0'>  
                <class 'AccumulateGrad'>
```

```
model = Network()  
enum = enumerate(train_loader)  
batch_idx, (inputs, targets) = next(enum)  
outputs = model(inputs)  
print_graph(outputs.grad_fn)
```

Summary

1. The gradient can be estimated using parameter perturbations, but this is inefficient
2. The gradient can be calculated using the chain rule, but this can be inefficient
3. Backpropagation is an efficient way to use the chain rule in a neural network
4. Backpropagation requires storage of activations throughout the network
5. Backpropagation is done automatically by deep learning software
6. PyTorch creates computational graphs dynamically