

CS440/ECE448 Lecture 12: Autograd

Mark Hasegawa-Johnson

Spring 2021

Outline

- Running example: neural net regression
- The same example with autograd
- `forward()` saves the state, `backward()` uses it
- `torch.nn` module: using predefined neural net components
- `torch.optim` module: using predefined second-order optimizers

Running example: neural net regression

- So far, we've studied classification: network outputs $P(Y = y|X = x)$
- A regression network learns a function $\hat{y} = f(x)$ so that the resulting \hat{y} approximates a desired real-valued y as well as possible
- A regression network is trained to minimize the mean-squared error or sum-squared error:

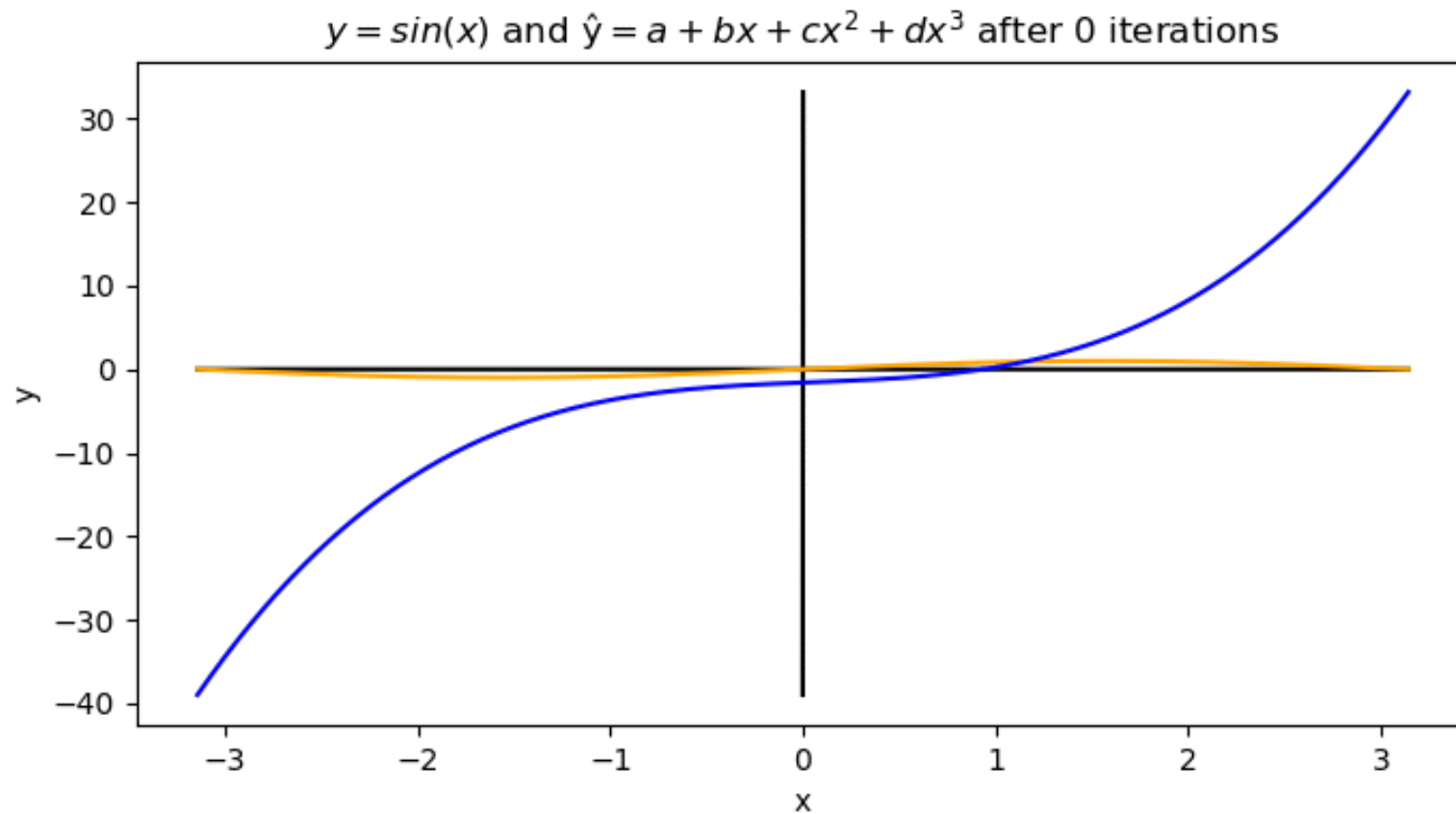
$$\mathcal{L} = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

Where (x_i, y_i) is a training example, and $\hat{y}_i = f(x_i)$.

Running example: neural net regression

- For example, suppose $y = \sin(x)$
- Suppose that the network can only model functions of the form
$$\hat{y} = a + bx + cx^2 + dx^3$$
- We want to learn a, b, c, d so that $\hat{y} \approx y$

Running example: neural net regression



How a neural network is trained

A neural network is trained using gradient descent. In this case, each iteration updates a , b , c , and d using the equations

$$a = a - \eta \frac{d\mathcal{L}}{da}, \quad b = b - \eta \frac{d\mathcal{L}}{db}$$
$$c = c - \eta \frac{d\mathcal{L}}{dc}, \quad d = d - \eta \frac{d\mathcal{L}}{dd}$$

Where, in this case, $\eta = 1e-6$.

How a neural network is trained

Let's work out the derivatives.

$$\hat{y}_i = a + bx_i + cx_i^2 + dx_i^3, \quad \mathcal{L} = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

$$\frac{d\mathcal{L}}{da} = \sum_{i=1}^n \frac{d\mathcal{L}}{d\hat{y}_i} \frac{d\hat{y}_i}{da}$$

$$\frac{d\mathcal{L}}{d\hat{y}_i} = 2(\hat{y}_i - y_i), \quad \frac{d\hat{y}_i}{da} = 1$$

How a neural network is trained

1. Calculate $\frac{d\mathcal{L}}{d\hat{y}_i}$
2. Calculate $\frac{d\mathcal{L}}{da}$ as $\frac{d\mathcal{L}}{da} = \sum_{i=1}^n \frac{d\mathcal{L}}{d\hat{y}_i}$
3. Calculate $\frac{d\mathcal{L}}{db}$ as $\frac{d\mathcal{L}}{db} = \sum_{i=1}^n \frac{d\mathcal{L}}{d\hat{y}_i} x_i$
4. Calculate $\frac{d\mathcal{L}}{dc}$ as $\frac{d\mathcal{L}}{dc} = \sum_{i=1}^n \frac{d\mathcal{L}}{d\hat{y}_i} x_i^2$
5. Calculate $\frac{d\mathcal{L}}{dd}$ as $\frac{d\mathcal{L}}{dd} = \sum_{i=1}^n \frac{d\mathcal{L}}{d\hat{y}_i} x_i^3$
6. Perform the gradient updates: $a = a - \eta \frac{d\mathcal{L}}{da}$, $b = b - \eta \frac{d\mathcal{L}}{db}$, $c = c - \eta \frac{d\mathcal{L}}{dc}$, $d = d - \eta \frac{d\mathcal{L}}{dd}$

How a neural network is trained

Here's Justin Johnson's code for doing those things:

(https://pytorch.org/tutorials/beginner/pytorch_with_examples.html)

```
for t in range(2000):
    # Forward pass: compute predicted y
    #  $y = a + b x + c x^2 + d x^3$ 
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss
    loss = np.square(y_pred - y).sum()
    if t % 100 == 99:
        print(t, loss)

    # Backprop to compute gradients of a, b, c, d with respect to loss
    grad_y_pred = 2.0 * (y_pred - y)
    grad_a = grad_y_pred.sum()
    grad_b = (grad_y_pred * x).sum()
    grad_c = (grad_y_pred * x ** 2).sum()
    grad_d = (grad_y_pred * x ** 3).sum()

    # Update weights
    a -= learning_rate * grad_a
    b -= learning_rate * grad_b
    c -= learning_rate * grad_c
    d -= learning_rate * grad_d
```

© 2017 Pytorch,
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Outline

- Running example: neural net regression
- The same example with autograd
- `forward()` saves the state, `backward()` uses it
- `torch.nn` module: using predefined neural net components
- `torch.optim` module: using predefined second-order optimizers

Autograd

- A neural network is a complicated function $\hat{y} = f(x)$, made up of many simple components
- If we try to take all the derivatives, $d\mathcal{L}/dw_{jk}^{(l)}$, all at once, in a big mass of spaghetti code, then the code will be really ugly.
- HOWEVER: Each of the components is simple to compute. Furthermore, the derivative of its output w.r.t. its input is simple.

Autograd

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{jk}^{(l)}$

Autograd

- In pytorch, variables that take responsibility for their own gradients are called “tensors” (<https://pytorch.org/docs/stable/tensors.html>)
- Here’s how Justin Johnson defines tensors for the polynomial regression problem:

```
# Create random Tensors for weights. For a third order polynomial, we need  
# 4 weights:  $y = a + b x + c x^2 + d x^3$   
# Setting requires_grad=True indicates that we want to compute gradients  
with  
# respect to these Tensors during the backward pass.  
a = torch.randn(), device=device, dtype=dtype, requires_grad=True)  
b = torch.randn(), device=device, dtype=dtype, requires_grad=True)  
c = torch.randn(), device=device, dtype=dtype, requires_grad=True)  
d = torch.randn(), device=device, dtype=dtype, requires_grad=True)
```

Autograd

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{jk}^{(l)}$
- These new objects have overloaded operators, so that any time we use them to compute some output, the input is cached.

Autograd

The operator overload code looks something like this:

```
class Tensor(torch.autograd.Function):
```

```
    def __init__(self, weight):  
        self.weight = weight  
        self.saved_tensors = ()
```

```
    def __mul__(self, other):  
        self.saved_tensors = (self.saved_tensors[:], other)  
        returnvalue = self.weight * other  
        return Tensor(returnvalue)
```

Cache the input, so we
can use it later...



Use numpy's `__mul__`
operator to compute the
return value...

Cast the return value as a
Tensor.

Autograd

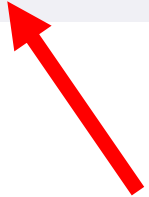
Here's how it gets used:

```
for t in range(2000):  
    # Forward pass: compute predicted y using operations on Tensors.  
    y_pred = a + b * x + c * x ** 2 + d * x ** 3
```

Stores x in b.saved_tensors



Stores x**2 in c.saved_tensors



Autograd

The basic idea of autograd is to create a new kind of object that takes responsibility for its own gradient.

- For example, the object might be a network weight, $w_{jk}^{(l)}$
- These new objects have overloaded operators, so that any time we use them to compute some output, the input is cached.
- During the `backward()` function, the “network weight” is given the loss gradient with respect to its output, $d\mathcal{L}/d(w_{jk}^{(l)} h_k^{(l-1)})$, from which it computes its input gradient, $d\mathcal{L}/dh_k^{(l-1)}$, and returns it to the calling function.
- It also computes its own gradient, $d\mathcal{L}/dw_{jk}^{(l)}$, and stores it internally.

Autograd

loss depends on `y_pred`, which depends on `a`, `b`, `c`, `d`.

Calculates the derivative of the loss w.r.t. each of its input tensors.

Uses the resulting derivatives to update the weights.

```
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

How to turn off autograd

- As you know, every time you add, subtract, multiply or divide a tensor by anything, the tensor stores data in `self.saved_tensors`, so it can use that information later to compute the gradient
- How do you turn this behavior off?

Dynamically turning off Autograd

These weight updates are not part of the neural network forward pass.

```
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
```

Statically turning autograd on and off

```
a.requires_grad = False
```

...

(do some stuff for which you don't want to keep track of gradients)

...

```
a.requires_grad = True
```

...

(go back to doing stuff that requires keeping track of gradients)

How to zero out the gradients

- When you call `backward()` over a tensor, it doesn't zero out any previous gradients
- Instead, it adds the current gradient to the previous gradients
- A very very very common mistake: running 2000 iterations, with the gradient accumulating from each iteration to the next, instead of zeroing it out in between iterations

Manually zeroing out the gradients

Here's the part I didn't show you before.

```
learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad, c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

    # Manually zero the gradients after updating weights
    a.grad = None
    b.grad = None
    c.grad = None
    d.grad = None
```

Outline

- Running example: neural net regression
- The same example with autograd
- `forward()` saves the state, `backward()` uses it
- `torch.nn` module: using predefined neural net components
- `torch.optim` module: using predefined second-order optimizers

Internal state

Being responsible for your own gradient means that you have to cache internal state variables, containing any information you will need, in the future, to compute your own gradient.

Example: Legendre polynomial

- Justin Johnson uses this example. Suppose we want an operation that computes a third-order Legendre polynomial:

$$o = \frac{1}{2} (5i^3 - 3i)$$

...where i is the input to this module, and o is its output.

- The derivative $d\mathcal{L}/di$ can be computed from $d\mathcal{L}/do$ as:

$$\frac{d\mathcal{L}}{di} = \frac{d\mathcal{L}}{do} \frac{do}{di} = \frac{d\mathcal{L}}{do} \frac{3}{2} (5i^2 - 1)$$

Example: Legendre polynomial

$$o = \frac{1}{2} (5i^3 - 3i)$$

$$\frac{d\mathcal{L}}{di} = \frac{d\mathcal{L}}{do} \frac{do}{di} (5i^2 - 1)$$

```
class LegendrePolynomial3(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

Outline

- Running example: neural net regression
- The same example with autograd
- `forward()` saves the state, `backward()` uses it
- `torch.nn` module: using predefined neural net components
- `torch.optim` module: using predefined second-order optimizers

Pytorch nn module

- The autograd feature of pytorch allows you to define only the forward propagation of your neural net. As long as all of the component operations are in pytorch's library, the back-propagation will be computed for you.
- Tensors just do multiplication and addition. What about other types of operations?
- General operations are contained in the nn module, using the formalism of a "layer."

Some types of layers

- `torch.nn.Linear`: a layer that computes $o = iW^T + b$
- `torch.nn.Softmax`: a layer that computes $o_j = \frac{\exp(i_j)}{\sum_k \exp(i_k)}$
- `torch.nn.Sigmoid`: a layer that computes $o_j = \frac{1}{1 + \exp(-i_j)}$
- `torch.nn.ReLU`: a layer that computes $o_j = \max(0, i_j)$
- `torch.nn.Sequential`: a model that takes a sequence of layers as its arguments, and applies them, one after the other, in order

torch.nn.Linear

- To create the layer object, you call: `m=torch.nn.Linear(n_in,n_out)`
 - This creates and initializes a weight matrix in `m.weight`
 - It also creates and initializes a bias vector in `m.bias`
- Once you've created the object, you can apply it in the forward pass to some input data as `output=m(input)`
 - `input` needs to be a tensor of size `(...,n_in)`
 - `output` will be a tensor of size `(...,n_out)`
- To compute backprop, you could then call `output.backward()`
 - ...or, you can call `backward()` on any tensor that is computed from `output`
 - The resulting gradient is stored in `m.weight.grad` and `m.bias.grad`
- Weight update needs to be computed explicitly
 - `m.weight -= learning_rate * m.weight.grad`

`m=torch.nn.Linear(n_1,n_2)`

- This creates a model such that `o=m(i)` performs, on each row of `i`, the operation:

$$o = iW^T + b$$

- `i` can be a tensor of any size, as long as its last dimension (the dimension of each row) is `n_1`
- `o` is then a tensor of the same shape as `i`, except that its last dimension (the row length) is now `n_2`
- `m.weight (W)` is a tensor of size `(n_2,n_1)`
- `m.bias (b)` is a row vector of length `n_2`

Example: Linear, Sigmoid, Softmax

- Here's an example from lecture 10. We could create the layers as:

```
excitation1 = torch.nn.Linear(2,3)
```

```
activation1 = torch.nn.ReLU()
```

```
excitation2 = torch.nn.Linear(3,2)
```

```
activation2 = torch.nn.Softmax(2)
```

- Having created them, we could then run forward pass as:

```
e1 = excitation1(x)
```

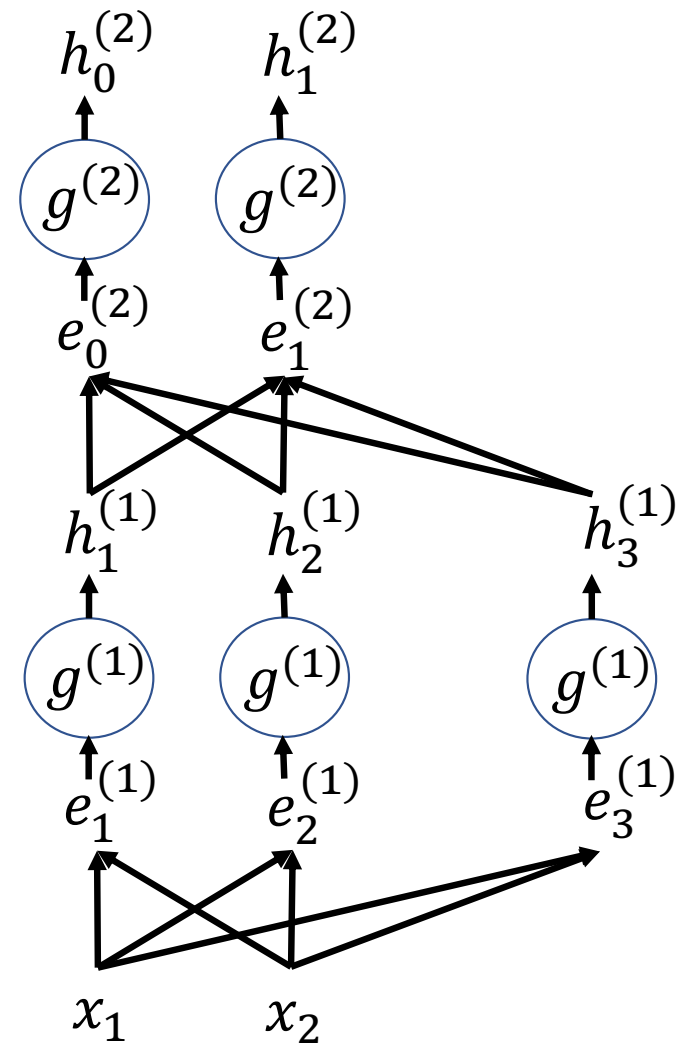
```
h1 = activation1(e1)
```

```
e2 = excitation2(h1)
```

```
h2 = activation2(e2)
```

- Then we could calculate all of the gradients by running

```
h2.backward()
```



torch.nn.Sequential

- `torch.nn.Sequential` is a special module that creates a sequence of layers, where each layer's output is the next layer's input. For example:

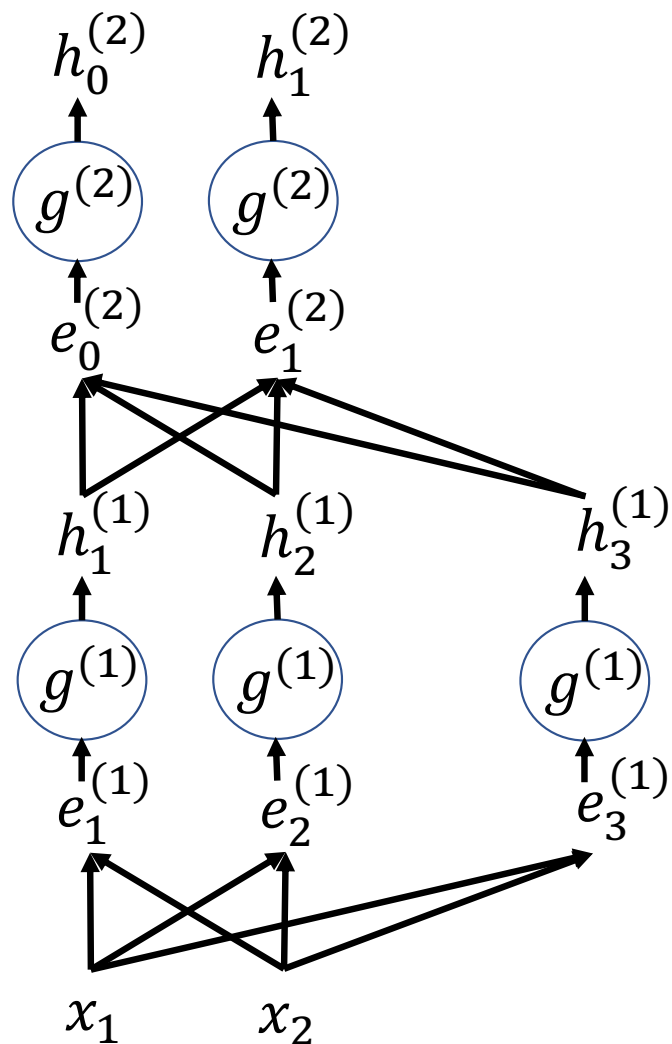
```
model = torch.nn.Sequential(  
    torch.nn.Linear(2,3),  
    torch.nn.ReLU(),  
    torch.nn.Linear(3,2),  
    torch.nn.Softmax(2))
```

- Then you can run forward pass by just typing:

```
h2 = model(x)
```

- You can still calculate all of the gradients by running

```
h2.backward()
```



torch.nn.Sequential: where are the parameters?

- The layers each have their own parameters, for example, a model created using the commands on the previous slide would have

```
model[0].weight
```

```
model[0].bias
```

```
model[2].weight
```

```
model[2].bias
```

- Accessing them that way requires you to know which layers have weights and biases, and which don't. An easier way is to use the function `model.parameters()`, which iterates through all trainable parameters, regardless of where they are actually stored:

```
for param in model.parameters():
```

```
    param -= learning_rate * param.grad
```

Outline

- Running example: neural net regression
- The same example with autograd
- `forward()` saves the state, `backward()` uses it
- `torch.nn` module: using predefined neural net components
- **`torch.optim` module: using predefined optimizers**

torch.optim: let pytorch do your optimization

- Gradient descent is easy to implement yourself
- You should remember from calculus: Newton's method is not just gradient descent. It's actually

$$w \leftarrow w - \frac{d\mathcal{L}/dw}{d^2\mathcal{L}/dw^2}$$

- Putting the second derivative in the denominator saves us from having to guess the learning rate, so the NN converges in far fewer optimization steps.
- Unfortunately, computing the second derivative with respect to a large parameter vector is computationally expensive.
- In practice, people use many different types of approximate second-order methods: Adam, LBFGS, RMSprop.... These are computationally cheap but require complicated code, therefore pytorch wrote the code for us.

How to use torch.optim

1. Instead of computing the loss yourself, use pytorch's implementation:

```
loss_fn = torch.nn.MSELoss(reduction='sum')  
loss = loss_fn(y_pred, y)
```

2. Instead of doing parameter update yourself, use pytorch's implementation:

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)  
loss.backward()  
optimizer.step()
```

... that last step does the parameter update, i.e., all the work that we used to do in:

```
param -= learning_rate * param.grad
```

Complete example, in the recommended style

© 2017 Pytorch,

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

Optimizer 

Forward Prop 

Calculate the loss 

Zero out the gradient 

Calculate the gradient 

Update the model parameters 

```
# Use the nn package to define our model and loss function.
model = torch.nn.Sequential(
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us. Here we will use RMSprop; the optim package contains many other
# optimization algorithms. The first argument to the RMSprop constructor tells the
# optimizer which Tensors it should update.
learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
for t in range(2000):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(t, loss.item())

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()
```

Model

Loss Function

Outline

- Running example: neural net regression
- The same example with autograd
- `forward()` saves the state, `backward()` uses it
- `torch.nn` module: using predefined neural net components
- `torch.optim` module: using predefined second-order optimizers