# Make a forward pass before the backward pass

# Backpropagation: Understanding the implications of the chain rule

Jonathon Hare

Vision, Learning and Control
University of Southampton

A lot of the ideas in this lecture come from Andrej Karpathy's blog post on backprop
(https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b) and his CS231n Lecture Notes
(http://cs231n.github.io/optimization-2/)

# Topics

- A quick look at an MLP again
- The chain rule (again)
- Uninititive gradient effects
- A closer look at basic stochastic gradient descent algorithms

# The unbiased Multilayer Perceptron (again)...
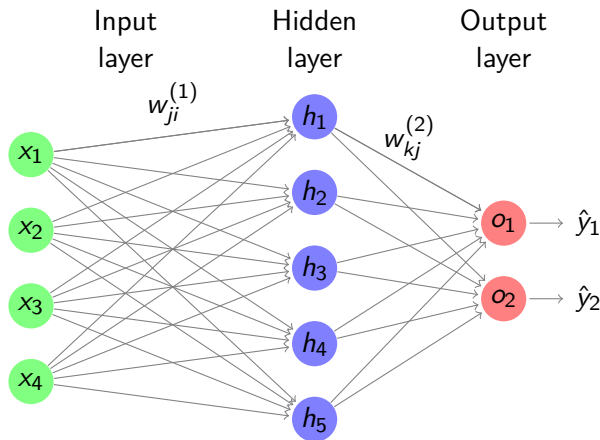


Without loss of generality, we can write the above as:

$$\hat{\boldsymbol{y}} = g(f(\boldsymbol{x}; \boldsymbol{W}^{(1)}); \boldsymbol{W}^{(2)}) = g(\boldsymbol{W}^{(2)}f(\boldsymbol{W}^{(1)}\boldsymbol{x}))$$

where $f$ and $g$ are activation functions.

- Let's assume MSE Loss

$$\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$$

# Gradients of our simple unbiased MLP

- Let's assume MSE Loss

$$\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$$

- What are the gradients?

$$\nabla_{\boldsymbol{W}^*} \ell_{MSE}(g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x})), \boldsymbol{y})$$

# Gradients of our simple unbiased MLP

- Let's assume MSE Loss

$$\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$$

- What are the gradients?

$$\nabla_{\boldsymbol{W}^*} \ell_{MSE}(g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x})), \boldsymbol{y})$$

- Clearly we need to apply the chain rule (vector form) multiple times

- Let's assume MSE Loss

$$\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$$

- What are the gradients?

$$\nabla_{\boldsymbol{w}^*} \ell_{MSE}(g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x})), \boldsymbol{y})$$

- Clearly we need to apply the chain rule (vector form) multiple times
- We could do this by hand

- Let's assume MSE Loss

$$\ell_{MSE}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$$

- What are the gradients?

$$\nabla_{\boldsymbol{w}^*} \ell_{MSE}(g(\boldsymbol{W}^{(2)} f(\boldsymbol{W}^{(1)} \boldsymbol{x})), \boldsymbol{y})$$

- Clearly we need to apply the chain rule (vector form) multiple times
- We could do this by hand
- (But we're not that crazy!)

# Let's go back to a simpler expression

$$f(x, y, z) = (x + y)z$$
$$\equiv qz \text{ where } q = (x + y)$$

## Let's go back to a simpler expression

$$f(x, y, z) = (x + y)z$$
$$\equiv qz \text{ where } q = (x + y)$$

Clearly the partial derivatives of the subexpressions are trivial:

$$\partial f / \partial z = q \qquad \partial f / \partial q = z$$
$$\partial q / \partial x = 1 \qquad \partial q / \partial y = 1$$

# Let's go back to a simpler expression

$$f(x, y, z) = (x + y)z$$
$$\equiv qz \text{ where } q = (x + y)$$

Clearly the partial derivatives of the subexpressions are trivial:

$$\partial f / \partial z = q \qquad \partial f / \partial q = z$$
$$\partial q / \partial x = 1 \qquad \partial q / \partial y = 1$$

and the chain rule tells us how to combine these:

$$\partial f / \partial x = \partial f / \partial q \cdot \partial q / \partial x = z$$
$$\partial f / \partial y = \partial f / \partial q \cdot \partial q / \partial y = z$$

# Let's go back to a simpler expression

$$f(x, y, z) = (x + y)z$$
$$\equiv qz \text{ where } q = (x + y)$$

Clearly the partial derivatives of the subexpressions are trivial:

$$\partial f/\partial z = q \qquad \partial f/\partial q = z$$
$$\partial q/\partial x = 1 \qquad \partial q/\partial y = 1$$

and the chain rule tells us how to combine these:

$$\partial f/\partial x = \partial f/\partial q \cdot \partial q/\partial x = z$$
$$\partial f/\partial y = \partial f/\partial q \cdot \partial q/\partial y = z$$

so $\nabla_{[x,y,z]} f = [z, z, q]$

$$f(x, y, z) = (x + y)z$$

# An intuition of the chain rule

- Notice how every operation in the computational graph given its inputs can immediately compute two things:
  1. its output value
  2. the *local* gradient of its inputs with respect to its output value
- The chain rule tells us literally that each operation should take its local gradients and multiply them by the gradient that *flows* backwards into it

# This is backpropagation

- The backprop algorithm is just the idea that you can perform the forward pass (computing and caching the local gradients as you go),
- and then perform a backward pass to compute the total gradient by applying the chain rule and re-utilising the cached local gradients

# This is backpropagation

- The backprop algorithm is just the idea that you can perform the forward pass (computing and caching the local gradients as you go),
- and then perform a backward pass to compute the total gradient by applying the chain rule and re-utilising the cached local gradients
- Backprop is just another name for 'Reverse Mode Automatic Differentiation'...

- Consider the multiplication operation $f(a, b) = a * b$.

# Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.

# Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)

# Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If $a$ is large and $b$ is tiny the gradient assigned to $b$ will be large, and the gradient to $a$ small.

# Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If $a$ is large and $b$ is tiny the gradient assigned to $b$ will be large, and the gradient to $a$ small.
- This has implications for e.g. linear classifiers ($\mathbf{w}^\top \mathbf{x}_i$) where you perform many multiplications

# Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If $a$ is large and $b$ is tiny the gradient assigned to $b$ will be large, and the gradient to $a$ small.
- This has implications for e.g. linear classifiers ($\mathbf{w}^\top \mathbf{x}_i$) where you perform many multiplications
  - the magnitude of the gradient is directly proportional to the magnitude of the data

## Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If $a$ is large and $b$ is tiny the gradient assigned to $b$ will be large, and the gradient to $a$ small.
- This has implications for e.g. linear classifiers ($\mathbf{w}^\top \mathbf{x}_i$) where you perform many multiplications
  - the magnitude of the gradient is directly proportional to the magnitude of the data
  - multiply $\mathbf{x}_i$ by 1000, and the gradients also increase by 1000

## Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If $a$ is large and $b$ is tiny the gradient assigned to $b$ will be large, and the gradient to $a$ small.
- This has implications for e.g. linear classifiers ($\mathbf{w}^\top \mathbf{x}_i$) where you perform many multiplications
  - the magnitude of the gradient is directly proportional to the magnitude of the data
  - multiply $\mathbf{x}_i$ by 1000, and the gradients also increase by 1000
  - if you don't lower the learning rate to compensate your model might not learn

## Unintuitive effects I: Multiplication

- Consider the multiplication operation $f(a, b) = a * b$.
- The gradients are clearly $\partial f / \partial b = a$ and $\partial f / \partial a = b$.
  - (in a computational graph these would be the local gradients w.r.t the inputs)
- If $a$ is large and $b$ is tiny the gradient assigned to $b$ will be large, and the gradient to $a$ small.
- This has implications for e.g. linear classifiers ($\mathbf{w}^\top \mathbf{x}_i$) where you perform many multiplications
  - the magnitude of the gradient is directly proportional to the magnitude of the data
  - multiply $\mathbf{x}_i$ by 1000, and the gradients also increase by 1000
  - if you don't lower the learning rate to compensate your model might not learn
  - **Hence you need to always pay attention to data normalisation!**

- It used to be popular to use sigmoids (or tanh) in the hidden layers...

# Unintuitive effects II: vanishing gradients of the sigmoid

- It used to be popular to use sigmoids (or tanh) in the hidden layers...
- Gradient of $\sigma(x) = \sigma(x)(1 - \sigma(x))$

# Unintuitive effects II: vanishing gradients of the sigmoid

- It used to be popular to use sigmoids (or tanh) in the hidden layers...
- Gradient of $\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Thus as part of a larger network where this is the local gradient, if $x$ is large (+ve or -ve), then all gradients backwards from this point will be zero due to multiplication of the chain rule
  - Why might $x$ be large?

# Unintuitive effects II: vanishing gradients of the sigmoid

- It used to be popular to use sigmoids (or tanh) in the hidden layers...
- Gradient of $\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Thus as part of a larger network where this is the local gradient, if $x$ is large (+ve or -ve), then all gradients backwards from this point will be zero due to multiplication of the chain rule
    - Why might $x$ be large?
- Maximum gradient is achieved when $x = 0$   ($\sigma(x) = 0.5$, $dx = 0.25$)

# Unintuitive effects II: vanishing gradients of the sigmoid

- It used to be popular to use sigmoids (or tanh) in the hidden layers...
- Gradient of $\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Thus as part of a larger network where this is the local gradient, if $x$ is large (+ve or -ve), then all gradients backwards from this point will be zero due to multiplication of the chain rule
  - Why might $x$ be large?
- Maximum gradient is achieved when $x = 0$  ($\sigma(x) = 0.5$, $dx = 0.25$)
  - This means that the maximum gradient that can flow out of a sigmoid will be a quarter of the input gradient

# Unintuitive effects II: vanishing gradients of the sigmoid

- It used to be popular to use sigmoids (or tanh) in the hidden layers...
- Gradient of $\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Thus as part of a larger network where this is the local gradient, if $x$ is large (+ve or -ve), then all gradients backwards from this point will be zero due to multiplication of the chain rule
  - Why might $x$ be large?
- Maximum gradient is achieved when $x = 0$ ($\sigma(x) = 0.5$, $dx = 0.25$)
  - This means that the maximum gradient that can flow out of a sigmoid will be a quarter of the input gradient
    - What's the implication of this in a deep network with sigmoid activations?

- Modern networks tend to use ReLUs

- Modern networks tend to use ReLUs
- Gradient is 1 for $x > 0$ and 0 otherwise

# Unintuitive effects III: dying ReLUs

- Modern networks tend to use ReLUs
- Gradient is 1 for $x > 0$ and 0 otherwise
- Consider ReLU($\boldsymbol{w}^\top \boldsymbol{x}$)
  - What happens if $\boldsymbol{w}$ is initialised badly?
  - What happens if $\boldsymbol{w}$ receives an update that means that $\boldsymbol{w}^\top \boldsymbol{x} < 0 \, \forall \, \boldsymbol{x}$?

# Unintuitive effects III: dying ReLUs

- Modern networks tend to use ReLUs
- Gradient is 1 for $x > 0$ and 0 otherwise
- Consider ReLU($\boldsymbol{w}^\top \boldsymbol{x}$)
  - What happens if $\boldsymbol{w}$ is initialised badly?
  - What happens if $\boldsymbol{w}$ receives an update that means that $\boldsymbol{w}^\top \boldsymbol{x} < 0 \; \forall \; \boldsymbol{x}$?
- These are dead ReLUs - ones that never fire for all training data
  - Sometimes you can find that you have a large fraction of these
  - if you get them from the beginning, check weight initialisation and data normalisation
  - if they're appearing during training, maybe $\eta$ is too big?

# Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps

# Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps
- Often this recursion involves a multiplication at each timestep, the gradients of which are all multiplied together because of the chain rule...

# Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps
- Often this recursion involves a multiplication at each timestep, the gradients of which are all multiplied together because of the chain rule...
- Consider $z = a \prod_n^\infty b$

# Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps
- Often this recursion involves a multiplication at each timestep, the gradients of which are all multiplied together because of the chain rule...
- Consider $z = a \prod_n^\infty b$
  - $z \to 0$ *if* $|b| < 1$

# Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps
- Often this recursion involves a multiplication at each timestep, the gradients of which are all multiplied together because of the chain rule...
- Consider $z = a \prod_n^\infty b$
  - $z \to 0$ if $|b| < 1$
  - $z \to \infty$ if $|b| > 1$

# Unintuitive effects IV: Exploding gradients in recurrent networks

- Recurrent networks apply a function recursively for some number of timesteps
- Often this recursion involves a multiplication at each timestep, the gradients of which are all multiplied together because of the chain rule...
- Consider $z = a \prod_n^\infty b$
  - $z \to 0$ if $|b| < 1$
  - $z \to \infty$ if $|b| > 1$
- Same thing happens in the backward pass of an RNN (although with matrices rather than scalars, so the reasoning applies to the largest eigenvalue)