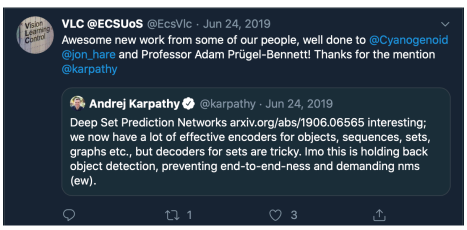
Make a forward pass before the backward pass

Backpropagation: Understanding the implications of the chain rule

Jonathon Hare

Vision, Learning and Control
University of Southampton

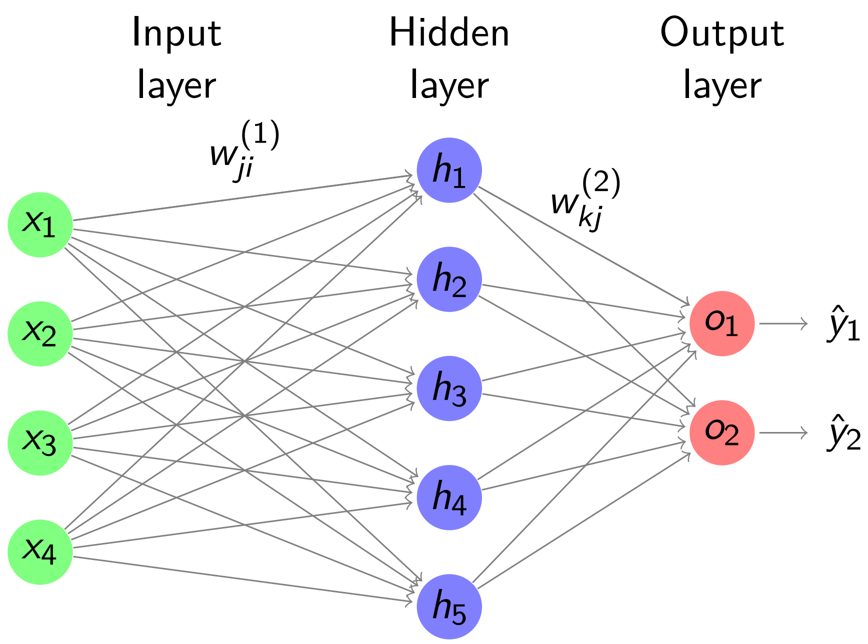
A lot of the ideas in this lecture come from Andrej Karpathy's blog post on backprop (`https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b`) and his CS231n Lecture Notes (`http://cs231n.github.io/optimization-2/`)



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.

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
- We could do this by hand
- (But we're not that crazy!)

Let's go back to a simpler expression

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

Clearly the partial derivatives of the subexpressions are trivial:

$$\begin{aligned} \partial f / \partial z &= q \qquad & \partial f / \partial q &= z \\ \partial q / \partial x &= 1 \qquad & \partial q / \partial y &= 1 \end{aligned}$$

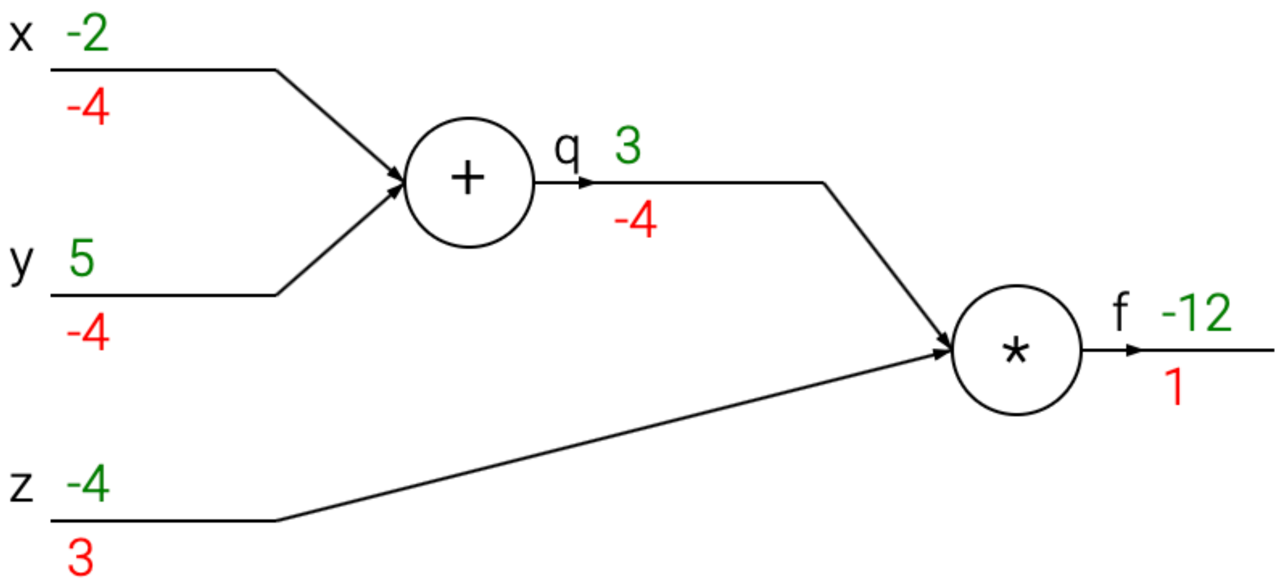
and the chain rule tells us how to combine these:

$$\begin{aligned} \partial f / \partial x &= \partial f / \partial q \cdot \partial q / \partial x = z \\ \partial f / \partial y &= \partial f / \partial q \cdot \partial q / \partial y = z \end{aligned}$$

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

A computational graph perspective

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



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
- Backprop is just another name for ‘Reverse Mode Automatic Differentiation’...

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 ($\boldsymbol{w}^\top \boldsymbol{x}_i$) where you perform many multiplications
  - the magnitude of the gradient is directly proportional to the magnitude of the data
  - multiple $\boldsymbol{x}_i$ 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!**

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 $x = 0$ $(x = 0.5, dx = 0.25)$
  - 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?

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 η is too big?

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 $z = a \prod_n^{\infty} b$
  - $z \rightarrow 0 if |b| < 1$
  - $z \rightarrow \infty if |b| > 1$
- Same thing happens in the backward pass of an RNN (although with matrices rather than scalars, so the reasoning applies to the largest eigenvalue)

Gradient descent and SGD (again), and mini-batch SGD

We'll wrap up by looking again at gradient descent algorithms and their behaviours...

Reminder: Gradient Descent

- Define total loss as $\mathcal{L} = -\sum_{(\boldsymbol{x},y)\in\boldsymbol{D}} \ell(g(\boldsymbol{x},\boldsymbol{\theta}),y)$ for some loss function ℓ, dataset $\boldsymbol{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the **total loss** $\mathcal{L}$ by the learning rate η multiplied by the gradient:

for each Epoch:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}$$

Reminder: Stochastic Gradient Descent

- Define loss function ℓ, dataset $\boldsymbol{D}$ and model g with learnable parameters $\boldsymbol{\theta}$.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Stochastic Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **single item** ℓ by the learning rate η multiplied by the gradient:

for each Epoch:

for each $(\boldsymbol{x}, y) \in \boldsymbol{D}$:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\ell$$

Mini-batch Gradient Descent

- Define a batch size b
- Define batch loss as $\mathcal{L}_b = -\sum_{(\boldsymbol{x},y)\in\boldsymbol{D}_b} \ell(g(\boldsymbol{x},\boldsymbol{\theta}),y)$ for some loss function $\ell$and model g with learnable parameters $\boldsymbol{\theta}$. $\boldsymbol{D}_b$ is a subset of dataset $\boldsymbol{D}$ of cardinality b.
- Define how many passes over the data to make (each one known as an Epoch)
- Define a learning rate η

Mini-batch Gradient Descent updates the parameters $\boldsymbol{\theta}$ by moving them in the direction of the negative gradient with respect to the loss of a **mini-batch** $\boldsymbol{D}_b$, $\mathcal{L}_b$ by the learning rate η multiplied by the gradient:

partition the dataset $\boldsymbol{D}$ into an array of subsets of size b
`for each Epoch:`
    `for each` $\boldsymbol{D}_b \in \mathit{partitioned}(\boldsymbol{D})$ `:`
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\nabla_{\boldsymbol{\theta}}\mathcal{L}_b$