COMP 451 – Fundamentals of Machine Learning Lecture 22 – Backpropagation

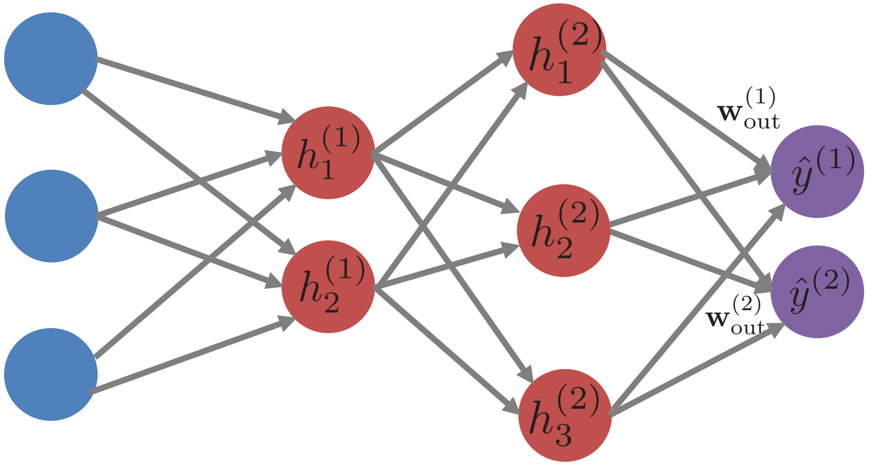
William L. Hamilton

Why neural networks?

- Key idea: Learn good features/representations rather than doing manual feature engineering.
- Hidden layers correspond to “higher-level” learned features.
- Critical in domains where manual feature engineering is difficult/impossible (e.g., images, raw audio) and empirically state-of-the-art in most domains.
- Allow for “end-to-end” learning where manual feature engineering/selection is no longer necessary.

Generalizing the feed-forward NN

- Can use arbitrary output activation functions.
- In practice, we do not necessarily need to use a sigmoid activation in the hidden layer.
- We can make networks as deep as we want.
- We can add regularization.



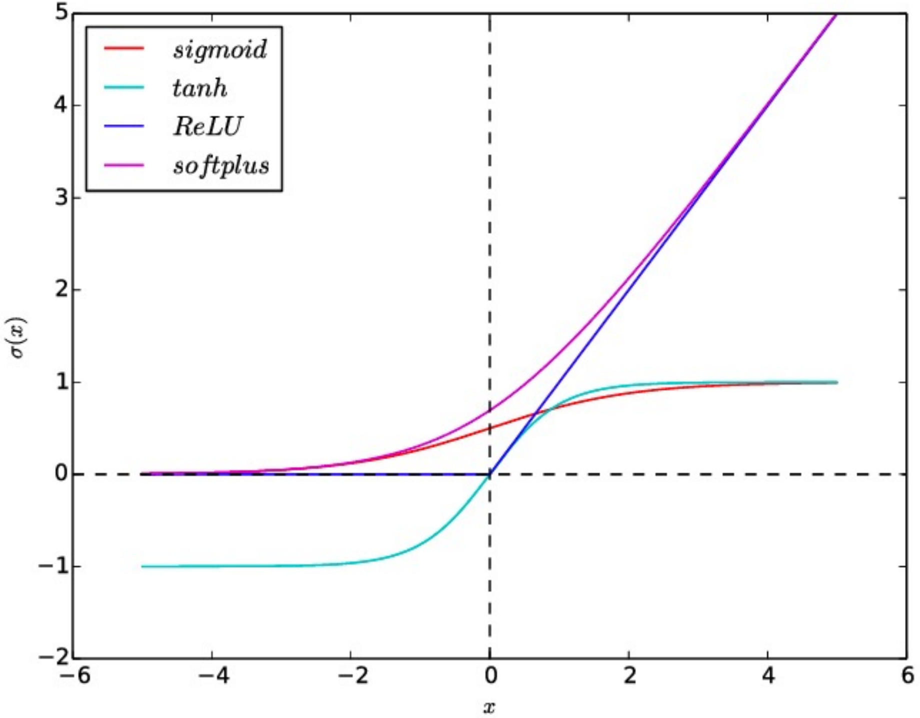
$$\mathbf{h}^{(i)} = \boxed{\phi_i}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

Can be an arbitrary **non-linear** activation function

Activation functions

$$\mathbf{h}^{(i)} = \boxed{\phi_i}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

- There are many choices for the activation function!
- It must be **non-linear**
  - Stacking multiple linear functions is equivalent to a single linear function, so there would be no gain…..



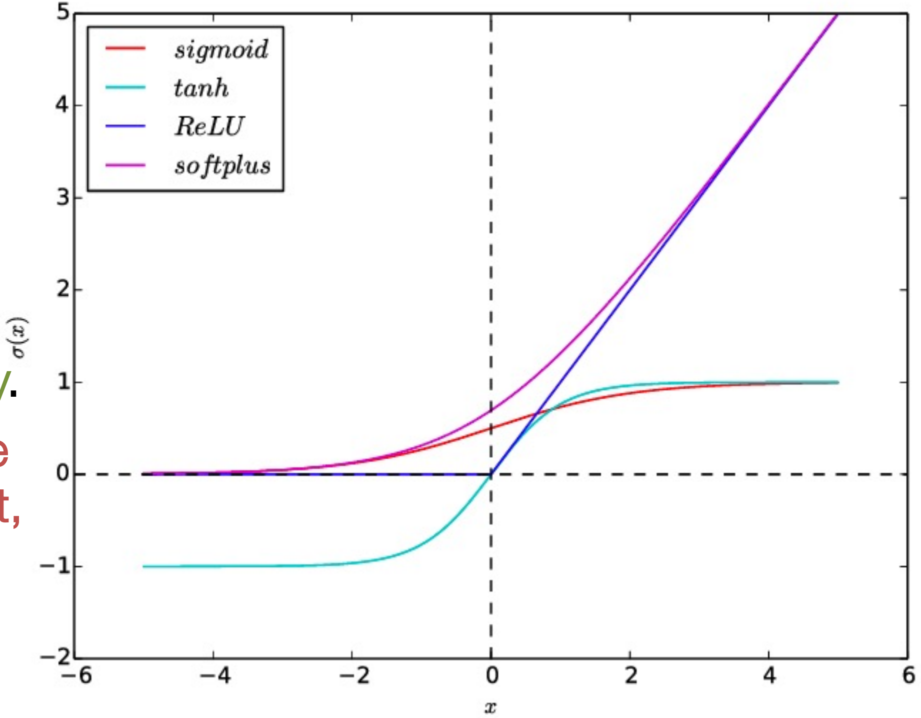
Activation functions: Sigmoid

$$\mathbf{h}^{(i)} = \boxed{\phi_i}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

- The sigmoid activation is the "classic"/ "original" activation function:

$$\phi(z) = \frac{1}{1+e^{-z}}$$

- Easy to differentiate.
- Can often be interpreted as a probability.
- Easily "saturates", i.e., for inputs outside the range [-4, 4] it is essentially constant, which can make it hard to train.



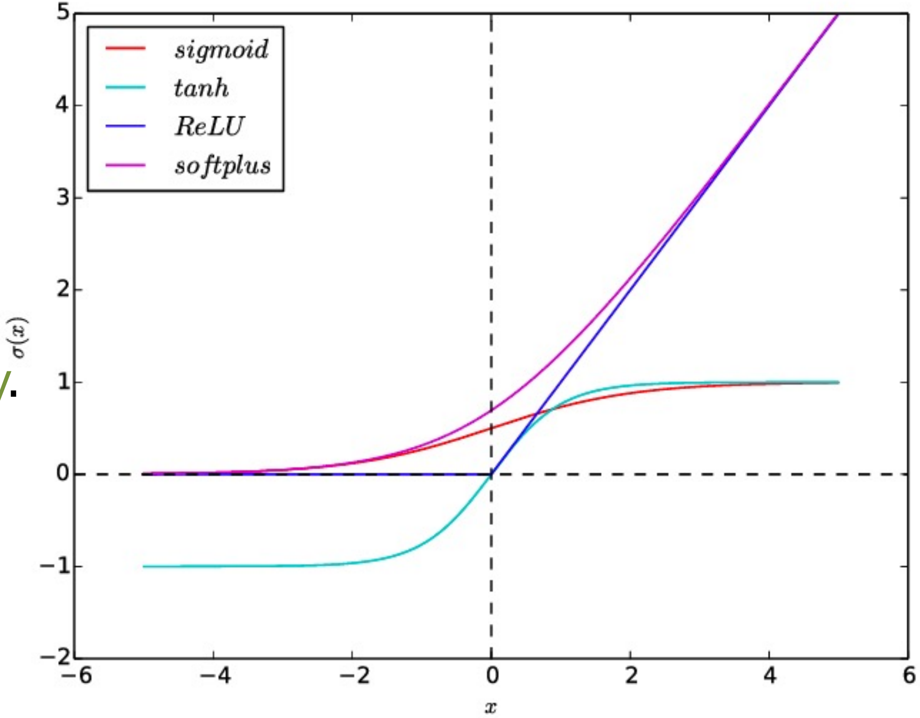
Activation functions: Tanh

$$\mathbf{h}^{(i)} = \boxed{\phi_i}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

- The tanh activation is another popular and traditional activation function:

$$\phi(z) = \tanh z = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

- Easy to differentiate: $\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z)$
- Can often be interpreted as a probability.
- Slightly less prone to "saturation" than the sigmoid but still has a fixed range.



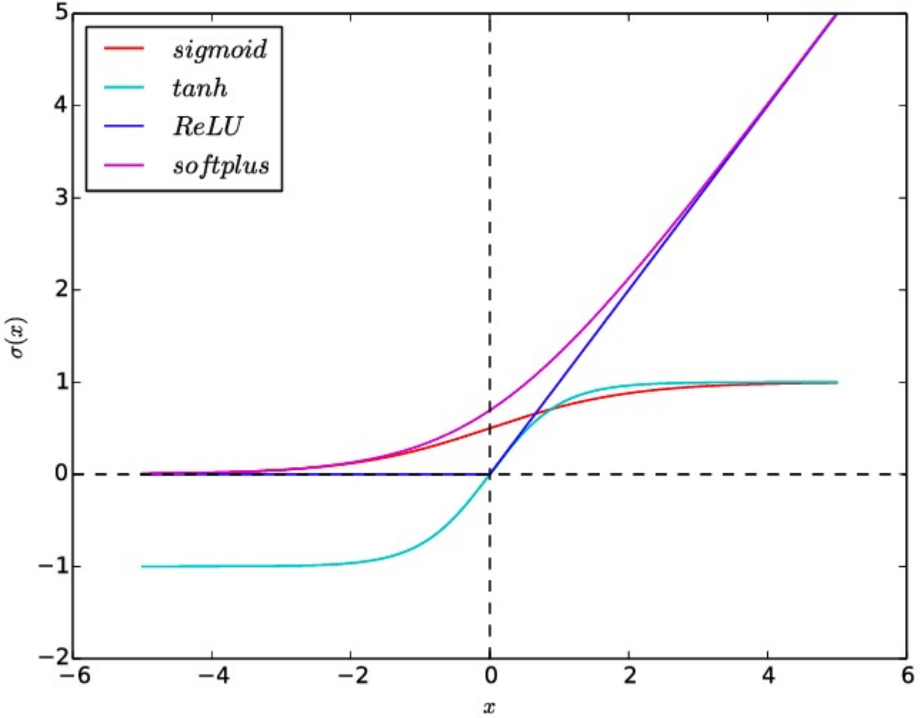
Activation functions: ReLU

$$\mathbf{h}^{(i)} = \boxed{\phi_i}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

- Rectified Linear Unit (ReLU) is the *de facto* standard in deep learning:

$$\text{ReLU}(z) = \max(z, 0)$$

- Unbounded range so it never saturates (i.e., activation can increase indefinitely).
- Very strong empirical results.



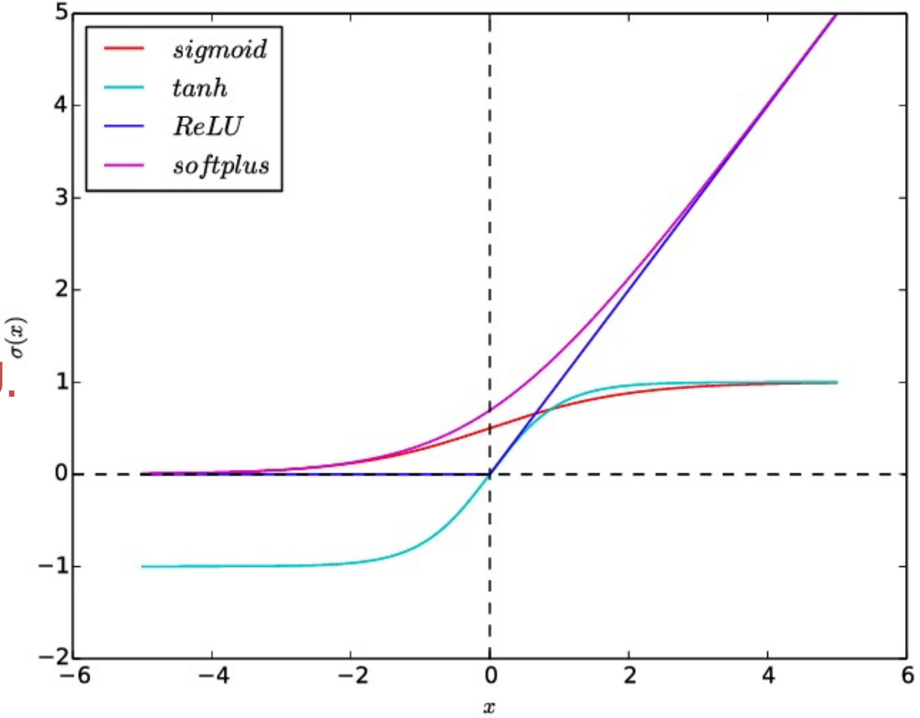
Activation functions: softplus

- There have been many variants of ReLUs proposed in recent years, e.g.,

$$\text{softplus}(z) = \ln(1 + e^x)$$

- Similar to ReLU but smoother.
- Expensive derivative compared to ReLU.

$$\mathbf{h}^{(i)} = \boxed{\phi_i}(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$



Regularization

- We can combat overfitting in neural networks by adding regularization.
- Standard approach is to apply L2-regularization to certain layers:

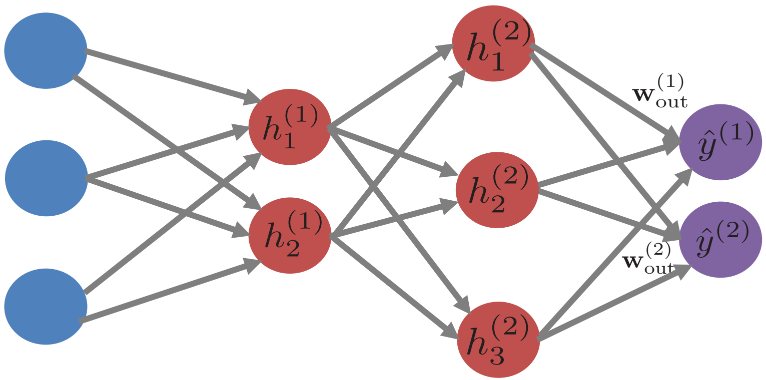
$$J_{\text{reg}} = J + \lambda\left(\sum_{i=1}^{H} \|\mathbf{W}^{(i)}\|_F^2 + \|\mathbf{w}_{\text{out}}\|_2^2\right)$$

Regularized loss

Original loss

Frobenius norm of hidden layer weights (“Frobenius” = 2-norm for matrices)

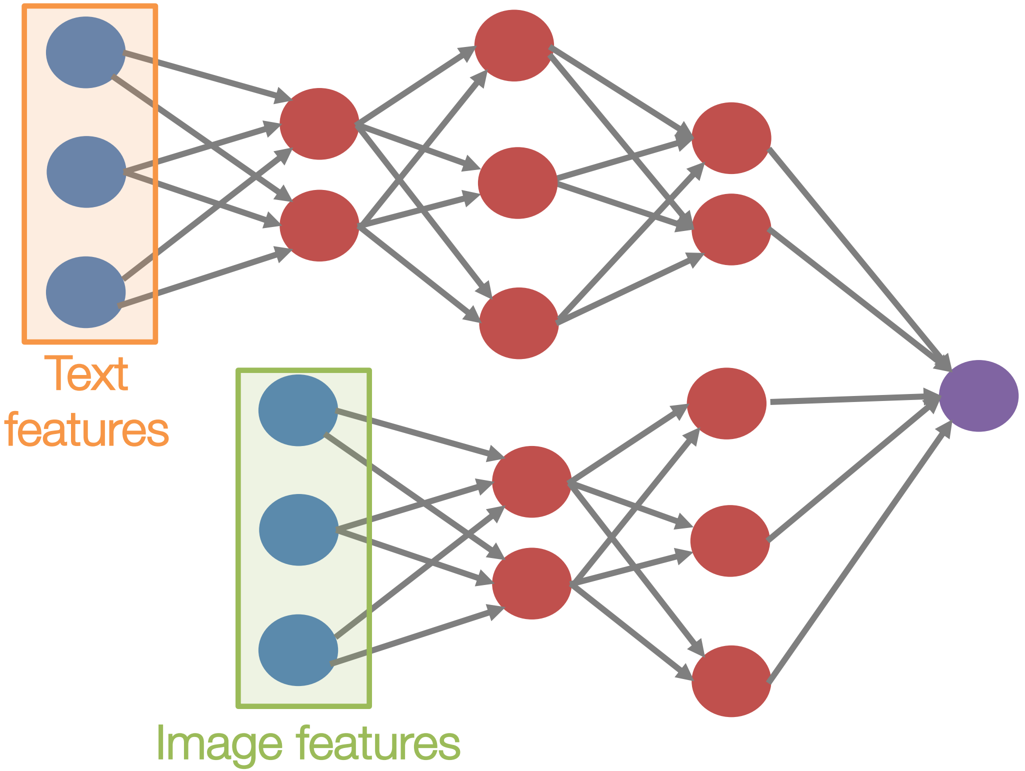
2-norm of output weights



$$\mathbf{h}^{(i)} = \phi_i(\mathbf{W}^{(i)}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

Complex architectures

- Neural network architectures can get very complicated!
- E.g., might want to combine feature information from different modalities (e.g., text and images) using different networks.
- Many more complex architectures will be covered in upcoming lectures!



How are we going to train these models?

- As long as all the operations are differentiable (or "sub-differentiable") we can always just apply the chain rule and compute the derivatives.
- But manually computing the gradient would be very painful/tedious!
  - Need to recompute every time we modify the architecture…
  - Lots of room for minor bugs in the code…
- Solution: Automatically compute the gradient.

Computation graphs

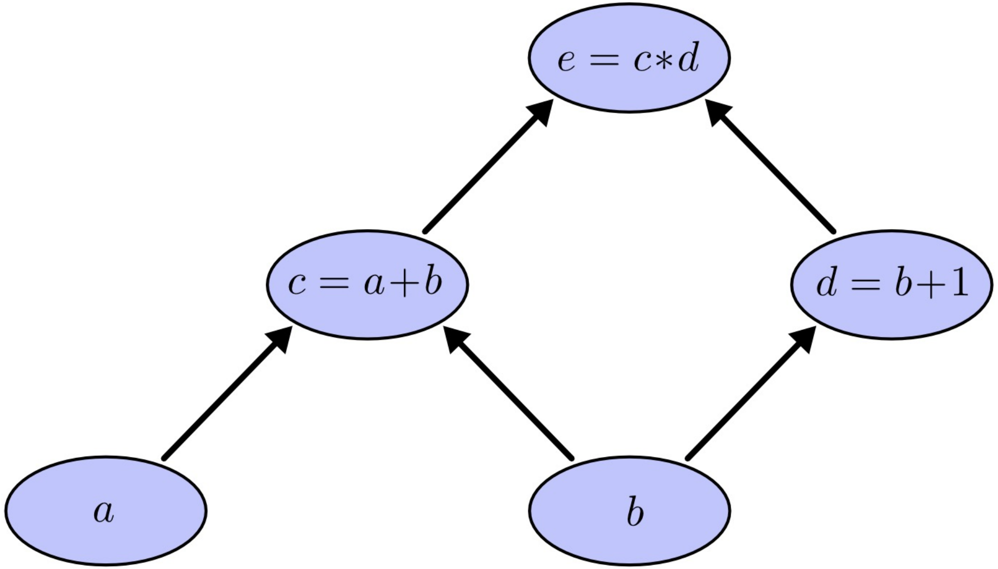
- A very simple "neural network" (i.e., computation graph):

$$c = a + b$$

$$d = b + 1$$

$$e = c * d$$

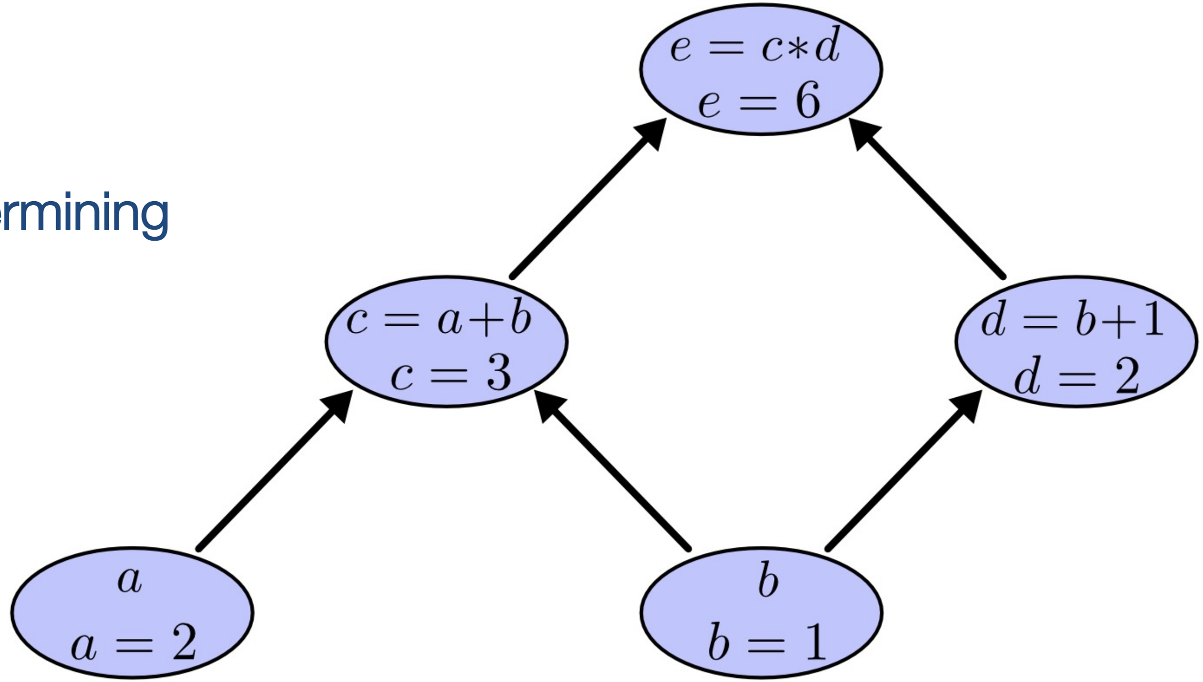
- Data is transformed at the nodes and "flows" allow the arrows.
- a and b are source nodes (i.e., the input) and e is the sink (i.e., the output).



[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Computation graphs

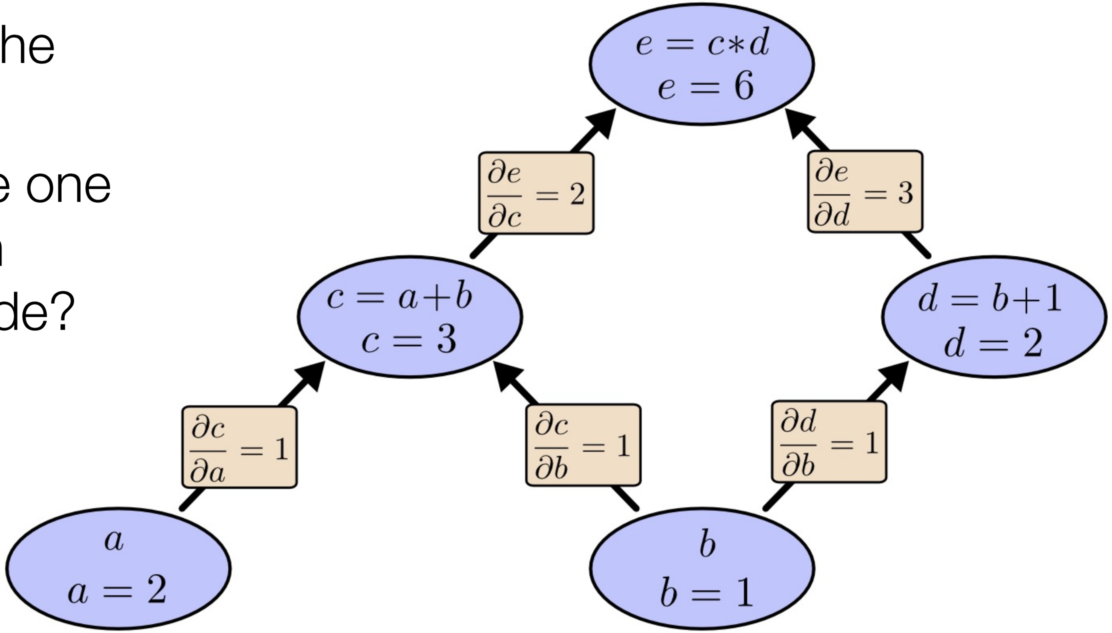
- The forward pass in a computational graph = setting the source variables/nodes and determining the sink/output value.



[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Derivatives on computation graphs

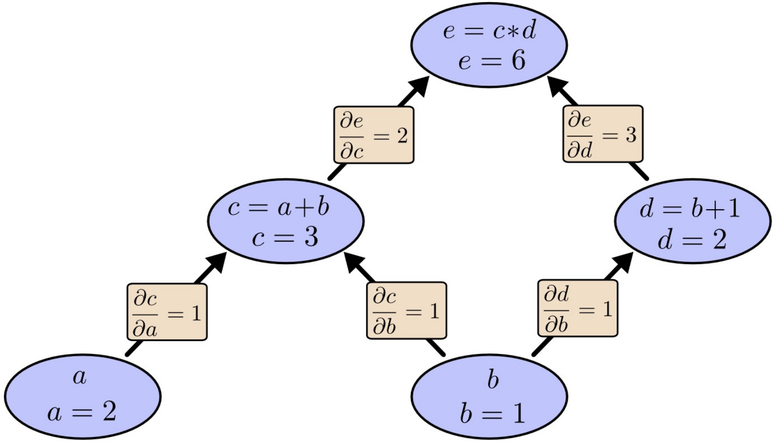
- We can use basic calculus to compute the derivatives on the edges.
- **Derivative tells us:** If I change one node by one unit, how much does this impact another node?



[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Derivatives on computation graphs

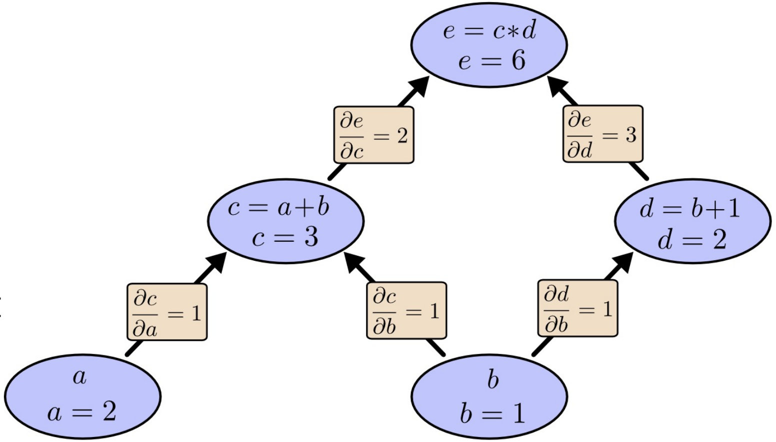
- But what if we want to compute the derivative between distant nodes?



[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Derivatives on computation graphs

- But what if we want to compute the derivative between distant nodes?
- E.g., consider how e is affected by a.
  - Change a at a speed of 1.
  - Then c also changes at a speed of 1.
  - And, c changing at a speed of 1 causes e to change at a speed of 2.
  - So e changes at a rate of 1x2 with respect to a.

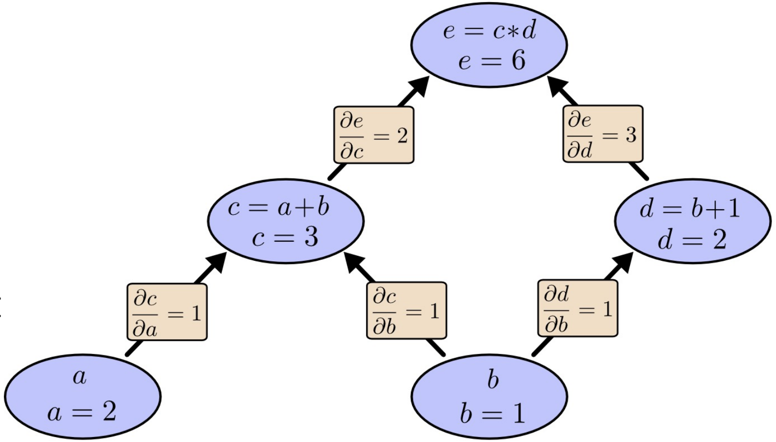


[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Derivatives on computation graphs

- But what if we want to compute the derivative between distant nodes?
- E.g., consider how e is affected by a.
  - Change a at a speed of 1.
  - Then c also changes at a speed of 1.
  - And, c changing at a speed of 1 causes e to change at a speed of 2.
  - So e changes at a rate of 1x2 with respect to a.
- **General rule:** sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path, e.g.

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$

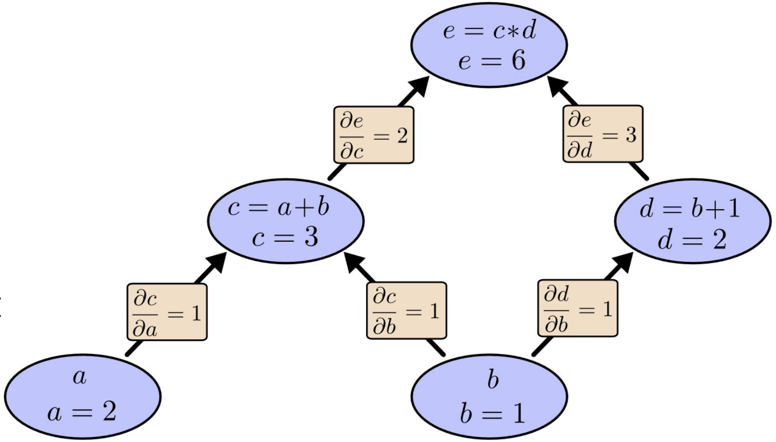


[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Derivatives on computation graphs

- But what if we want to compute the derivative between distant nodes?
- E.g., consider how e is affected by a.
  - Change a at a speed of 1.
  - Then c also changes at a speed of 1.
  - And, c changing at a speed of 1 causes e to change at a speed of 2.
  - So e changes at a rate of 1x2 with respect to a.
- **General rule:** sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path, e.g.

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$



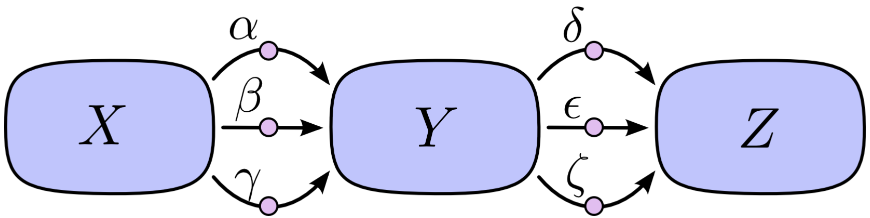
"Sum over paths" idea is just chain rule!

[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Challenge: combinatorial explosion

- Naively summing over paths (i.e. naively applying the chain rule) can lead to combinatorial explosion!

- Number of paths between two nodes can be exponential in the number of graph edges!



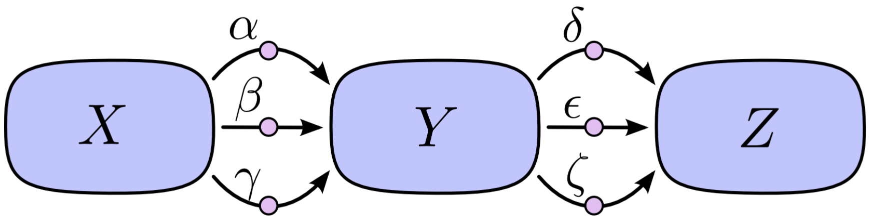
$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

3x3=9 paths between X and Z….

[Image and inspiration credit: http://colah.github.io/posts/2015-08-Backprop/]

Challenge: combinatorial explosion

- Naively summing over paths (i.e. naively applying the chain rule) can lead to combinatorial explosion!
- Number of paths between two nodes can be exponential in the number of graph edges!
- Idea: Factor the paths!

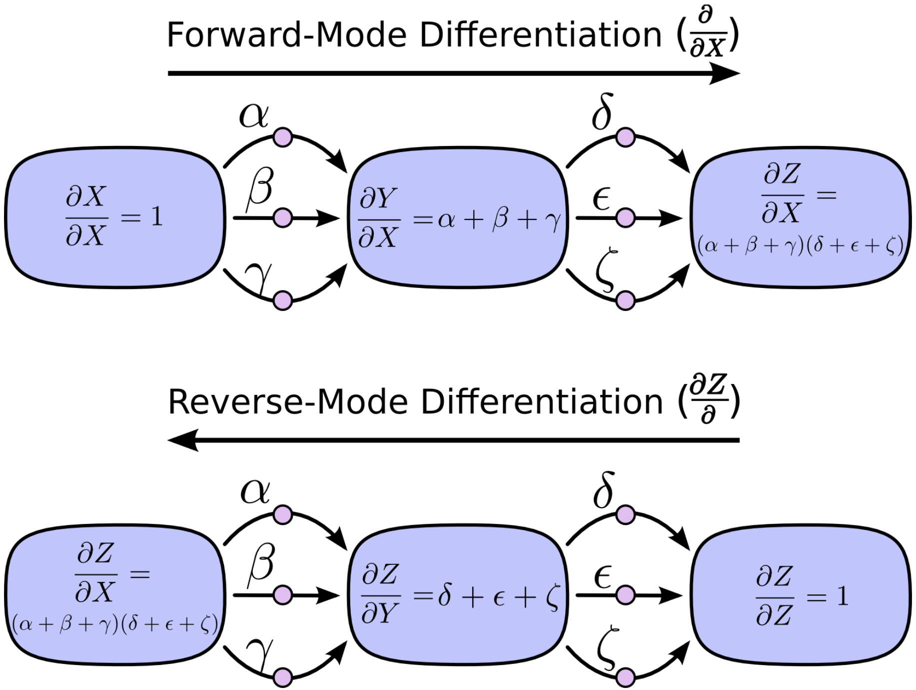


$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$$

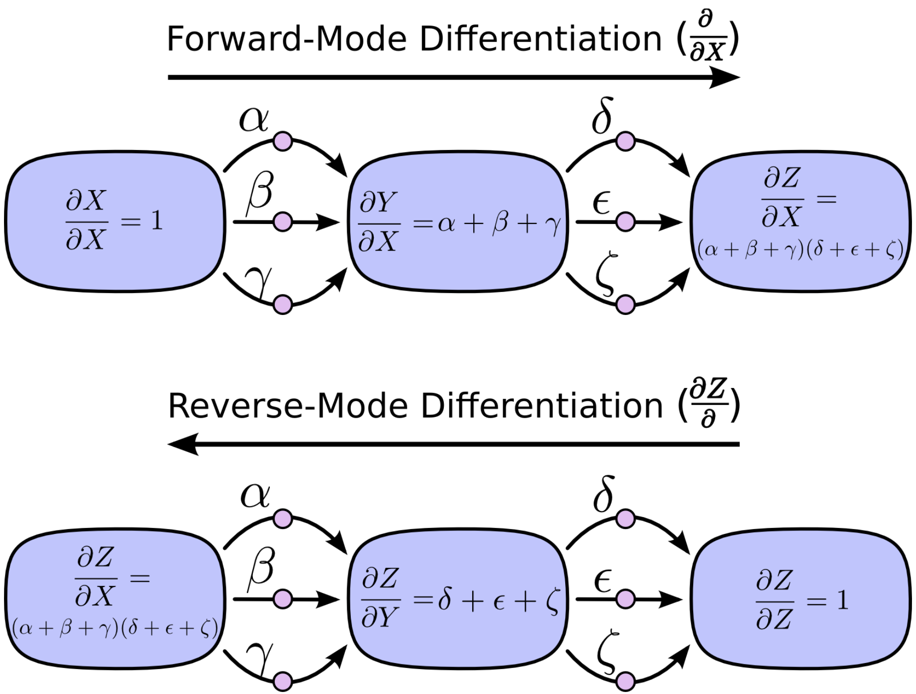
Factoring paths

- Forward mode
  - Goes from source(s) to sink.
  - At each node, sum all the incoming edges/derivatives.



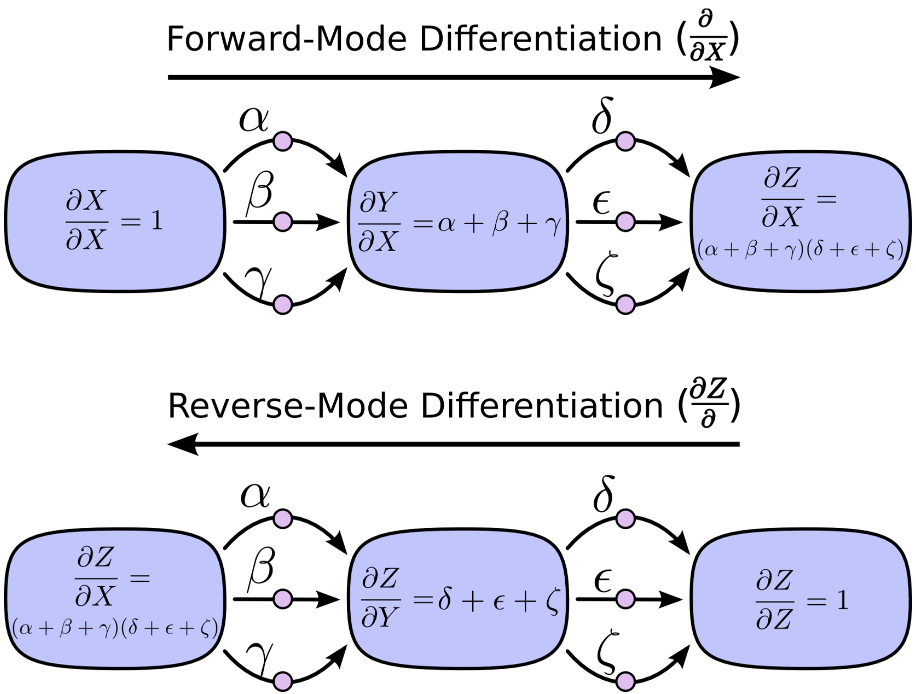
Factoring paths

- Forward mode
  - Goes from source(s) to sink.
  - At each node, sum all the incoming edges/derivatives.
- Reverse mode:
  - Goes from sink to source(s).
  - At each node, sum all the outgoing edges/derivatives.



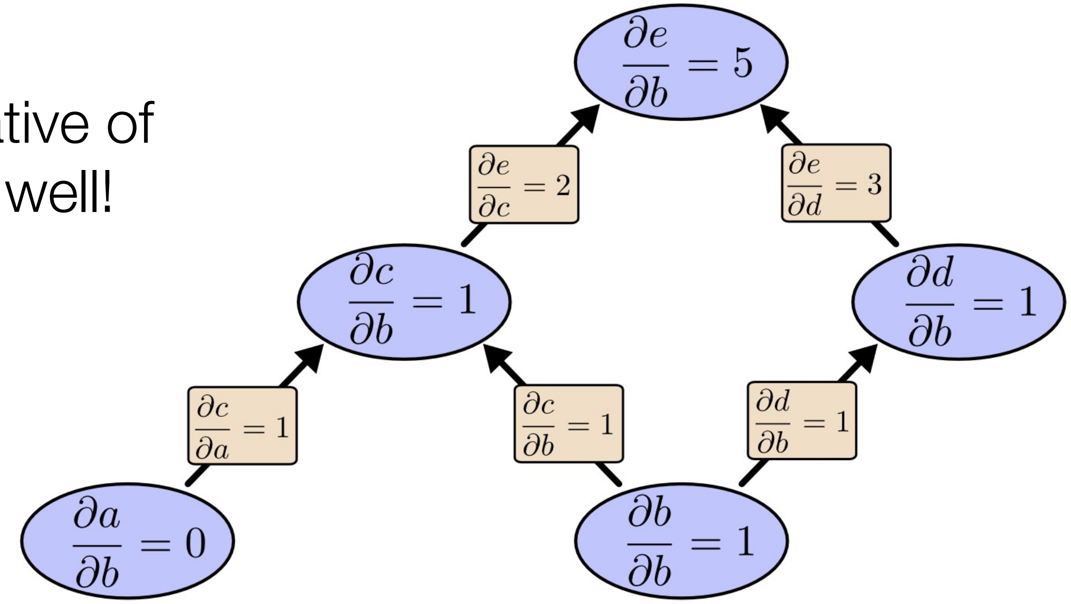
Factoring paths

- Forward mode
  - Goes from source(s) to sink.
  - At each node, sum all the incoming edges/derivatives.
- Reverse mode:
  - Goes from sink to source(s).
  - At each node, sum all the outgoing edges/derivatives.
- Both only touch each edge once!



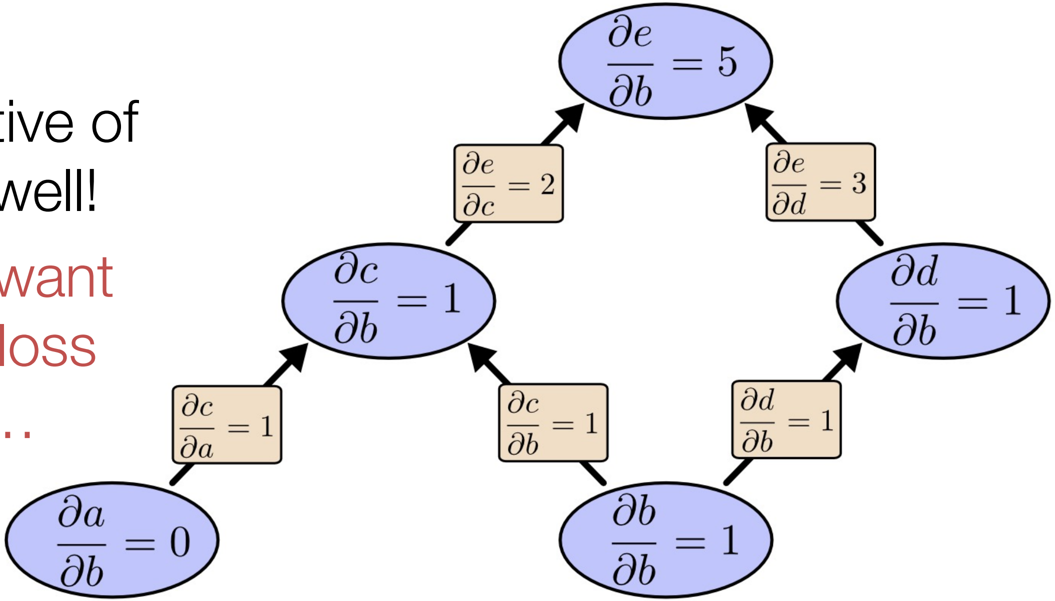
Computational savings: Forward-mode

- Run forward mode from the source b to the sink e.
- Note that we get the derivative of each intermediate edge as well!



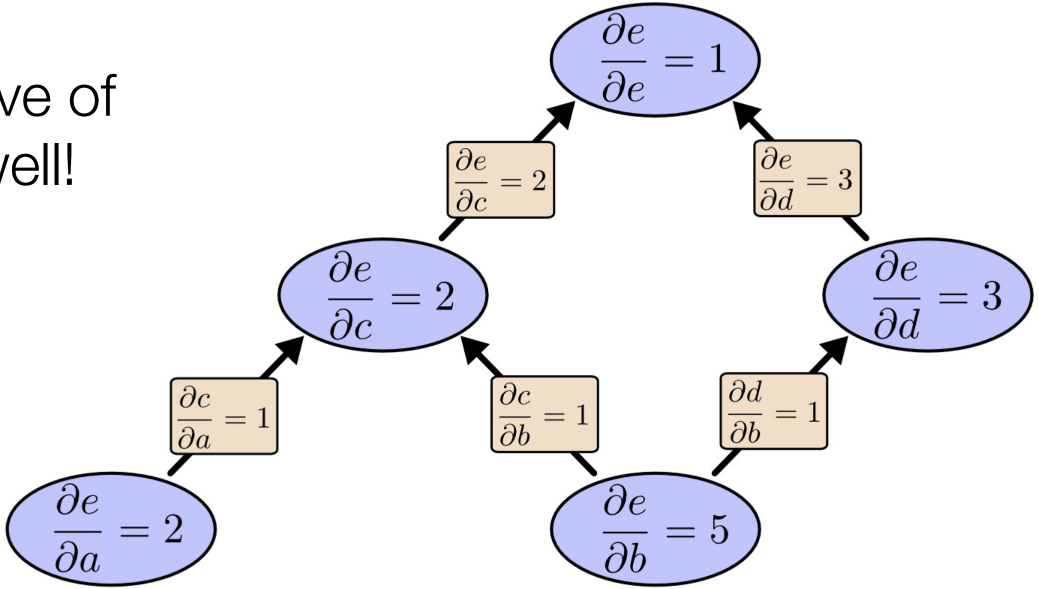
Computational savings: Forward-mode

- Run forward mode from the source b to the sink e.
- Note that we get the derivative of each intermediate edge as well!
- But in neural networks, we want the derivative of the output/loss w.r.t. all the previous layers…



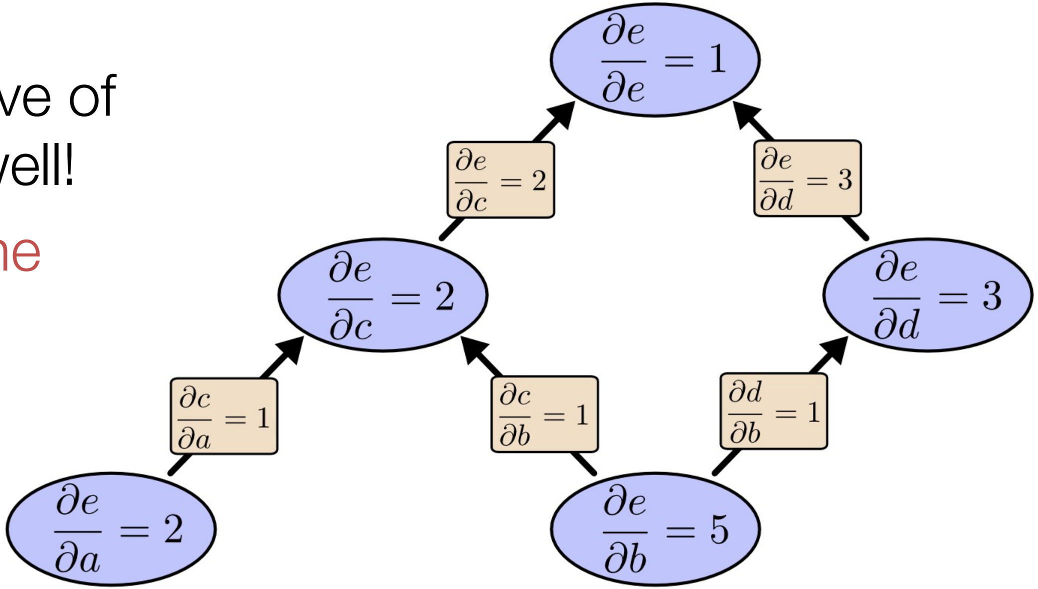
Computational savings: Backward mode

- Run backward mode from the sink e to the sources a and b.
- Note that we get the derivative of each intermediate edge as well!



Computational savings: Backward mode

- Run backward mode from the sink e to the sources a and b.
- Note that we get the derivative of each intermediate edge as well!
- We get the derivative w.r.t. the output in one pass!



Automated differentiation vs. backpropagation

- Reverse-mode automatic differentiation (RV-AD) can efficiently compute the derivative of every node in a computation graph.

- Neural networks are just computation graphs!

- We generally call RV-AD "backpropagation" in the context of neural networks and deep learning, since we are propagating the derivative/error backwards from the output node.

Automated differentiation in practice

- Many automated differentiation frameworks exist.
  - Basic derivatives are hard-coded (often called “kernels”) and everything else computed via automated differentiation.
- Nearly all based on Python/NumPy.
- You specify:
  - The forward model (i.e., the computation graph).
  - The training examples and training schedule (e.g., how the points are batched together).
  - The optimization details (e.g., loss function, learning rate).
- The framework takes care of the derivative computations!

TensorFlow

PyTorch

mxnet

theano

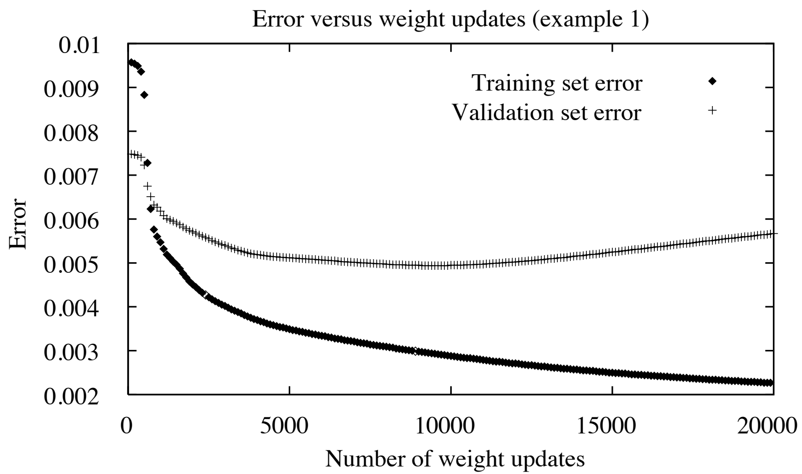
Convergence of backpropagation

- If the learning rate is appropriate, the algorithm is guaranteed to converge to a local minimum of the cost function.
  - NOT the global minimum. (Can be much worse.)
  - There can be MANY local minimum.
  - Use random restarts = train multiple nets with different initial weights.
  - In practice, the solution found is often good (i.e., local minima are not a big problem in practice)…

- Training can take thousands of iterations – CAN BE VERY SLOW! But using network after training is generally fast.

Overtraining

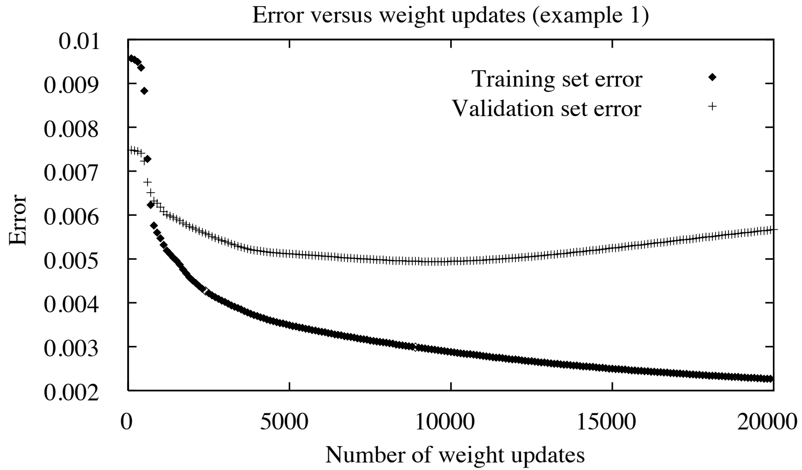
- Traditional overfitting is concerned with the number of parameters vs. the number of instances
- In neural networks: related phenomenon called overtraining occurs when weights take on large magnitudes, i.e. unit saturation
  - As learning progresses, the network has more active parameters.



Overtraining

- Traditional overfitting is concerned with the number of parameters vs. the number of instances
- In neural networks: related phenomenon called overtraining occurs when weights take on large magnitudes, i.e. unit saturation
  - As learning progresses, the network has more active parameters.

- Use validation set to decide when to stop.
- # training updates is a hyper-parameter.



Choosing the learning rate

- Backprop is **very sensitive** to the choice of learning rate.
  - Too large ⇒ divergence.
  - Too small ⇒ VERY slow learning.
  - The learning rate also influences the ability to escape local optima.

- The learning rate is a critical hyperparameter.

Adaptive optimization algorithms

- It is now standard to use “adaptive” optimization algorithms.
- These approaches modify the learning rate adaptively depending on how the training is progressing.
- Adam, RMSProp, and AdaGrad are popular approaches, with Adam being the de facto standard in deep learning.
  - All of these approaches scale the learning rate for each parameter based on statistics of the history of gradient updates for that parameter.
  - **Intuition:** increase the update strength for parameters that have had smaller updates in the past.

Adding momentum

- At each iteration of gradient descent, we are computing an update based based on the derivative of the current (mini)batch of training examples:

Update for weight vector

$$\boxed{\Delta_i \mathbf{w}} = \alpha \boxed{\frac{\partial J}{\partial \mathbf{w}}}$$

Derivative of error w.r.t. weights for current minibatch

$$\mathbf{w} \leftarrow \Delta_i \mathbf{w}$$

Adding momentum

- On i'th gradient descent update, instead of:

$$\Delta_i \mathbf{w} = \alpha \frac{\partial J}{\partial \mathbf{w}}$$

We do:

$$\Delta_i \mathbf{w} = \alpha \frac{\partial J}{\partial \mathbf{w}} + \beta \Delta_{i-1} \mathbf{w}$$

The <u>second term</u> is called <u>momentum</u>

Adding momentum

- On i'th gradient descent update, instead of:

$$\Delta_i \mathbf{w} = \alpha \frac{\partial J}{\partial \mathbf{w}}$$

We do:

$$\Delta_i \mathbf{w} = \alpha \frac{\partial J}{\partial \mathbf{w}} + \beta \Delta_{i-1} \mathbf{w}$$

The second term is called momentum

Advantages:

- Easy to pass small local minima.
- Keeps the weights moving in areas where the error is flat.

Disadvantages:

- With too much momentum, it can get out of a global maximum!
- One more parameter to tune, and more chances of divergence

What you should know

- Basic generalizations of neural networks: activation functions, regularization, and complex architectures
- The concept of a computation graph.
- The basics of automated differentiation (reverse and forward mode) and backpropagation.
- The issue of overtraining.
- Optimization hyperparameters for backpropagation: learning rate, momentum, adaptive optimization algorithms