Neural networks

Fraida Fund

Contents

Recap

- Last week: neural networks
- Many parameters
- Train using gradient descent
- How to compute gradients?

Backpropagation

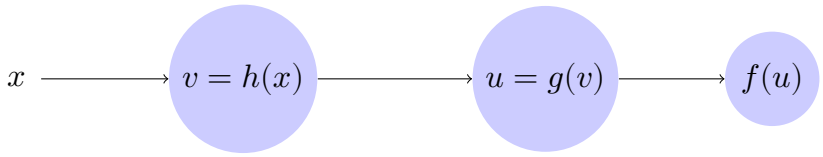
How to compute gradients?

- Gradient descent requires computation of the gradient $\nabla L(\theta)$
- Backpropagation is key to efficient computation of gradients

Composite functions and computation graphs

Suppose we have a composite function $f(g(h(x)))$

We can represent it as a computational graph, where each connection is an input and each node performs a function or operation:



Forward pass on computational graph

To compute the output $f(g(h(x)))$, we do a *forward pass* on the computational graph:

- Compute $v = h(x)$
- Compute $u = g(v)$
- Compute $f(u)$

Derivative of composite function

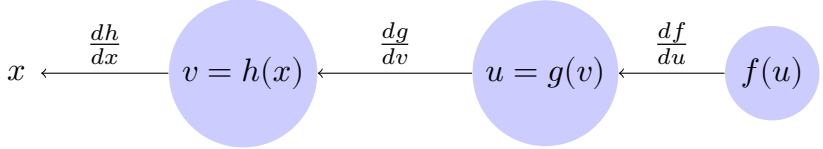
- Suppose need to compute the derivative of the composite function $f(g(h(x)))$ with respect to x
- We will use the chain rule.

Backward pass on computational graph

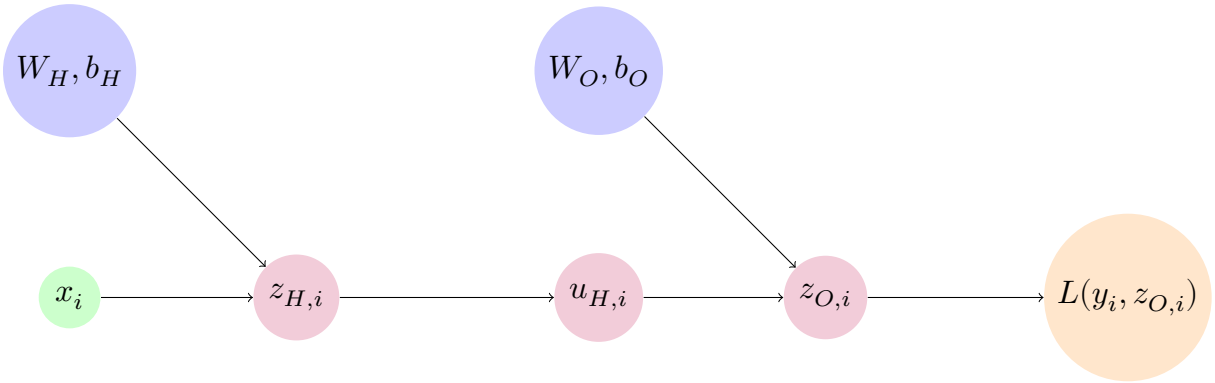
We can compute this chain rule derivative by doing a *backward pass* on the computational graph:

We just need to get the derivative of each node with respect to its inputs:

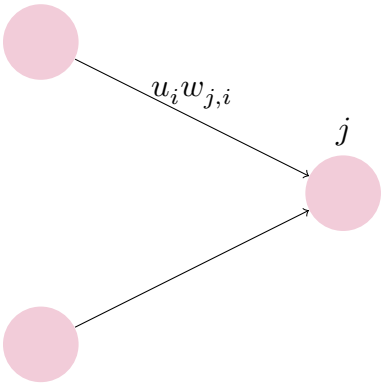
$$\frac{df}{dx} = \frac{df}{du}\frac{dg}{dv}\frac{dh}{dx}$$



Neural network computation graph



Backpropagation error: illustration



At a node j,

- $z_j = \sum_i w_{j,i} u_i$
- $u_j = g(z_j)$

Backpropagation error: definition

- Chain rule: $\frac{\partial L}{\partial w_{j,i}} = \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial w_{j,i}}$
- Denote the backpropagation error of the node j as $\delta_j = \frac{\partial L}{\partial z_j}$
- Since $z_j = \sum_i w_{j,i} u_i$, $\frac{\partial z_j}{\partial w_{j,i}} = u_i$
- Then $\frac{\partial L}{\partial w_{j,i}} = \delta_j u_i$

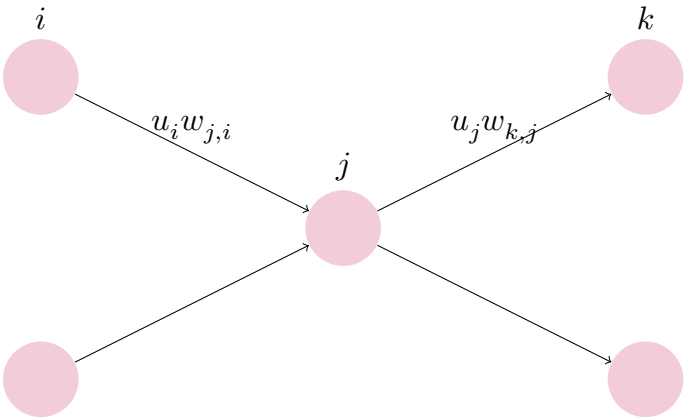
Backpropagation error: output unit

For output unit in regression network, with

$$L = \frac{1}{2} \sum_n (y_n - z_{O,n})^2$$

Then $\delta_O = \frac{\partial L}{\partial z_O} = -(y_n - z_O)$

Backpropagation error: hidden unit illustration



Backpropagation error: hidden unit

For a hidden unit,

$$\delta_j = \frac{\partial L}{\partial z_j} = \sum_k \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial z_j}$$

$$\delta_j = \sum_k \delta_k \frac{\partial z_k}{\partial z_j} = \sum_k \delta_k w_{k,j} g'(z_j) = g'(z_j) \sum_k \delta_k w_{k,j}$$

using $\delta_k = \frac{\partial L}{\partial z_k}$. And because $z_k = \sum_l w_{k,l} u_l = \sum_l w_{k,l} g(z_l)$, $\frac{\partial z_k}{\partial z_j} = w_{k,j} g'(z_j)$.

Derivatives for common loss functions

Squared/L2 loss:

$$L = \sum_i (y_i - z_{O,i})^2, \quad \frac{\partial L}{\partial z_{O,i}} = \sum_i -(y_i - z_{O,i})$$

Binary cross entropy loss:

$$L = \sum_i -y_i z_{O,i} + \ln(1 + e^{y_i z_{O,i}}), \quad \frac{\partial L}{\partial z_{O,i}} = y_i - \frac{e^{y_i z_{O,i}}}{1 + e^{y_i z_{O,i}}}$$

Derivatives for common activation functions

- Sigmoid activation: $g'(x) = \sigma(x)(1 - \sigma(x))$
- Tanh activation: $g'(x) = \frac{1}{\cosh^2(x)}$

Backpropagation + gradient descent algorithm

1. Start with random (small) weights. Apply input x_n to network and propagate values forward using $z_j = \sum_i w_{j,i} u_i$ and $u_j = g(z_j)$. (Sum is over all inputs to node j.)
2. Evaluate δ_k for all output units.
3. Backpropagate the δs to get δ_j for each hidden unit. (Sum is over all outputs of node j.)

$$\delta_j = g'(z_j) \sum_k w_{k,j} \delta_k$$

4. Use $\frac{\partial L_n}{\partial w_{j,i}} = \delta_j u_i$ to evaluate derivatives.
5. Update weights using gradient descent.

Backpropagation demo notebook

Link to demo notebook

Why backpropagation?

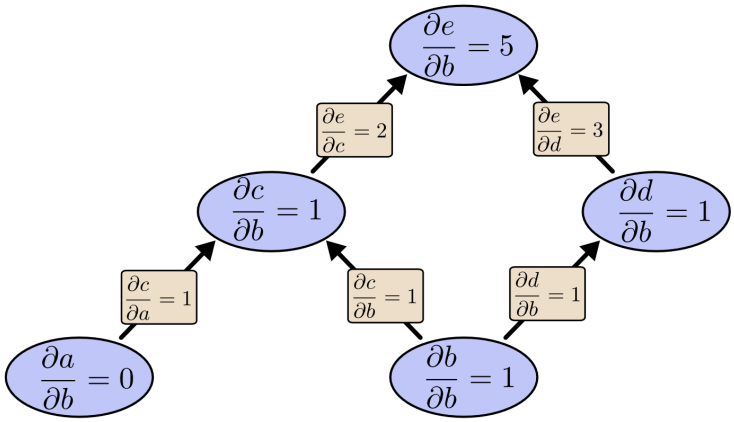


Figure 1: Forward-mode differentiation from input to output gives us derivative of every node with respect to each input. Then we can compute the derivative of output with respect to input. Image via https://colah.github.io/posts/2015-08-Backprop/.

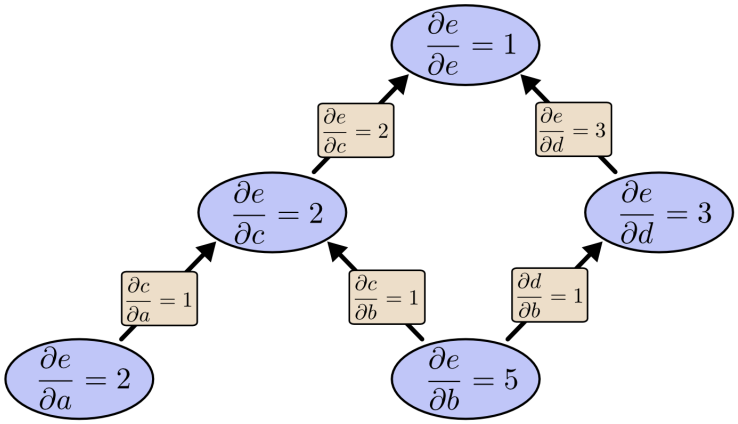


Figure 2: Reverse-mode differentiation from output to input gives us derivative of output with respect to every node. Image via https://colah.github.io/posts/2015-08-Backprop/.

Training challenges

Some models may not "converge" - why?

Learning rate

- If learning rate is too high, weights can oscillate
- Can use adaptive learning rate algorithm like: momentum, RMSProp, Adam

Local minima

- Error surface may have local minima
- Gradient descent can get "trapped"
- "Noise" can help get out of local minima: using stochastic gradient descent with one sample at a time, adding noise to data or weights, etc.

Unstable gradients

- Backprop in a neural network involves multiplication of terms of the form $w_j g'(z_j)$
- When this term tends to be small: gradients get smaller and smaller as we move from output to input
- When this term tends to be large: gradients get larger and larger as we move from output to input

Vanishing gradient problem: illustration

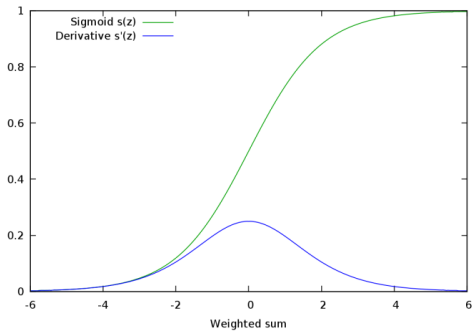


Figure 3: Note the "flat spot" on the sigmoid, where the derivative is close to zero. Ideally, we want to operate in "linear" region of activation function

"Herd effect"

- Hidden units all move in the same direction at once, instead of "specializing"
- Solution: use initial (small!) random weights
- Use small initial learning rate so that hidden units can find "specialization" before taking large steps

Many factors affect training efficiency

- Number of layers/hidden units
- Choice of activation function
- Choice of loss function

Classic paper: "Efficient BackProp", Yann LeCun et al, 1998

Training a neural network in Python

Keras

- High-level Python library for building and fitting neural networks
- Runs on top of a *backend*

Backends for deep learning

Keras-compatible backends:

- TensorFlow (Google)
- CNTK (Microsoft)
- Theano (LISA Lab at Université de Montréal)

PyTorch

- Also a high-level Python library for neural networks
- Developed by Facebook

Keras recipe

1. Describe model architecture
2. Select optimizer
3. Select loss function, compile model
4. Fit model
5. Test/use model

Demo notebook

Link to demo notebook (by Sundeep Rangan)

Networks with multiple hidden layers

Networks of linear units

- Recall: hidden layer doesn't do anything with linear activation function
- Equivalent to a single layer of weights

Non-linear units and one layer of weights

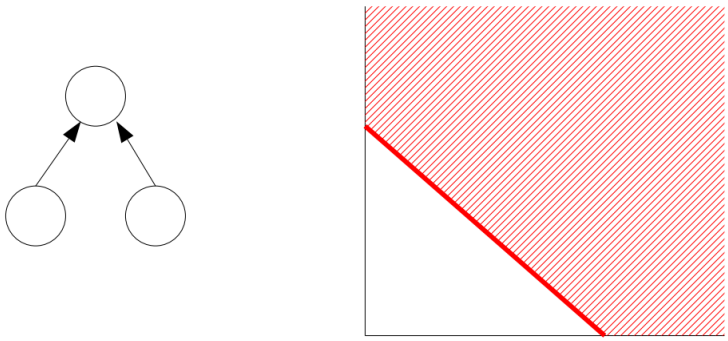


Figure 4: One layer of weights with non-linear activation creates a separating hyperplane.

TensorFlow Playground Link: Logistic regression on linearly separable data

Non-linear units and two layers of weights

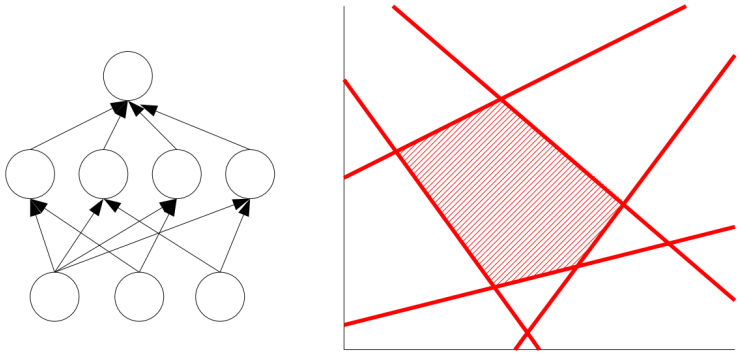


Figure 5: Two layers of weights with non-linear activation create a convex polygon region.

TensorFlow Playground Link: One hidden layer on circles

Non-linear units and many layers of weights

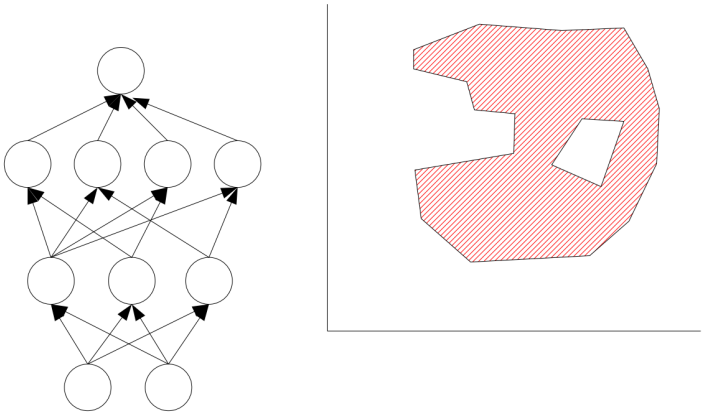


Figure 6: Three layers of weights with non-linear activation create a composition of polygon regions.

Deep networks

Networks with many hidden layers are challenging -

- Computationally expensive to train
- Many parameters - at risk of overfitting
- Vanishing gradient problem

Breakthroughs

In early 2010s, some breakthroughts

- Efficient training with GPU
- Huge data sets
- ReLu activation function