CHL7001H S1 Applied Deep Learning

Lecture 3: Neural Networks

Neural Networks

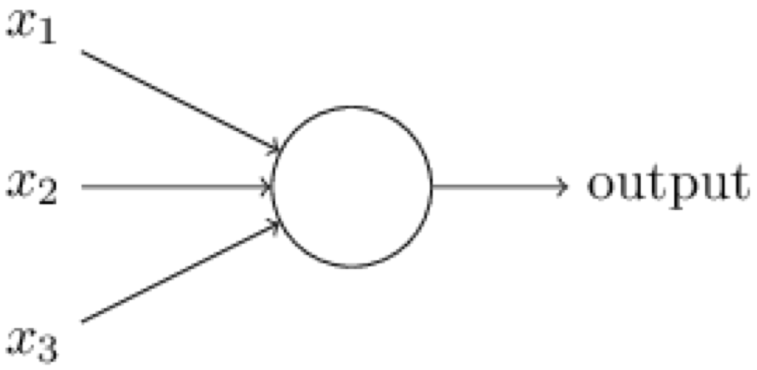
Neural Networks are a type of model originally inspired by how the human brain is thought to “work”

Deep Learning is sometimes defined as neural networks once you have multiple “layers”, but in general the field of modern techniques using neural networks are collectively referred to as “deep learning”

Early idea: Perceptron

All inputs are binary, and the output is binary

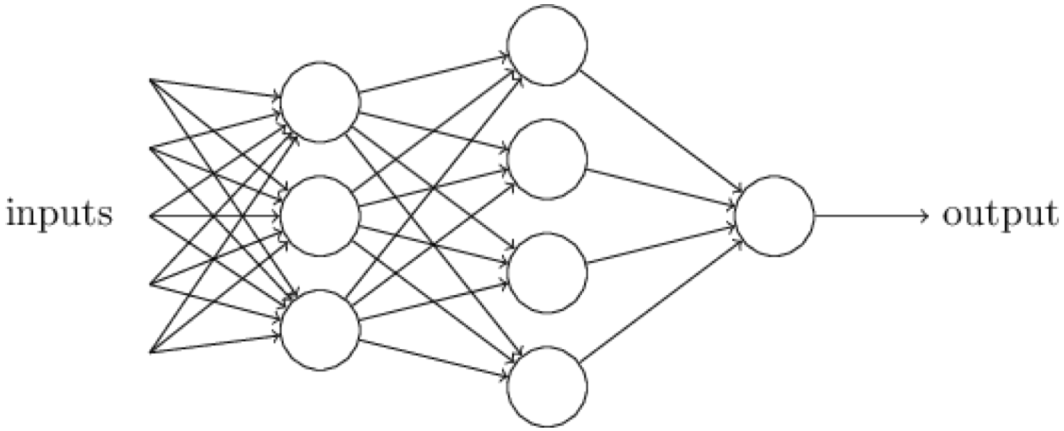
$$Y = \begin{cases} 1 & \Sigma_i w_i x_i \geq \text{threshold} \\ 0 & \Sigma_i w_i x_i < \text{threshold} \end{cases}$$



(for simplicity, we'll later set all thresholds to 0 and add a bias term instead...it's equivalent)

Data source: Neilson

Perceptrons



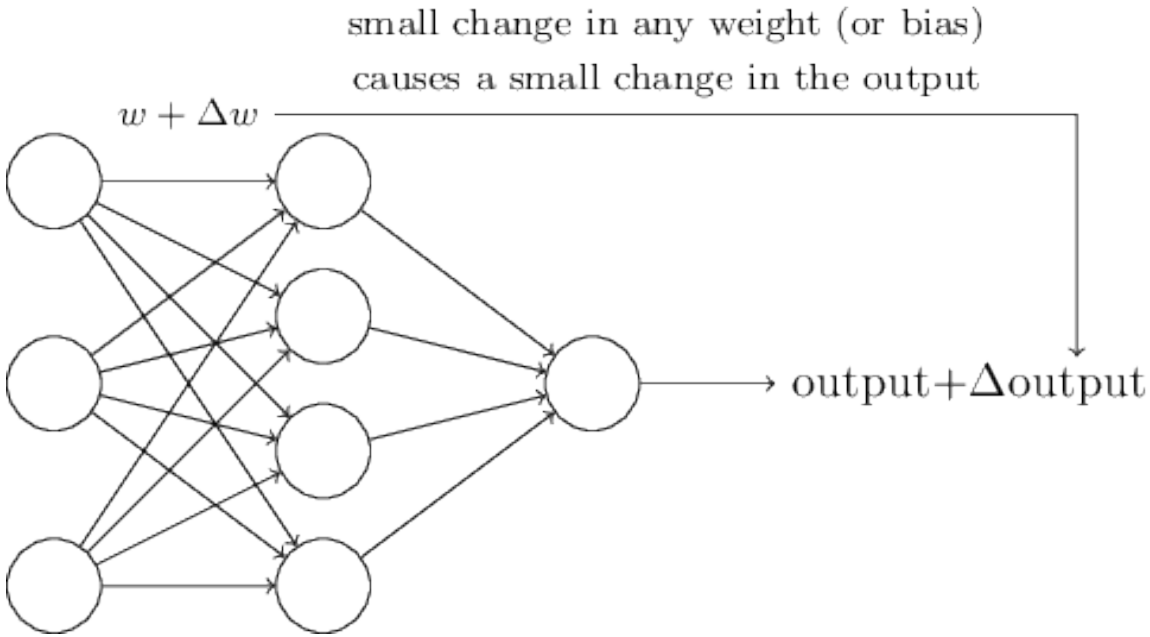
Perceptrons

NAND gate – since NAND gates are universal (you can build any Boolean logic circuit), if perceptron networks can model NAND, a large enough network can compute anything

Training (or why perceptrons won't cut it)

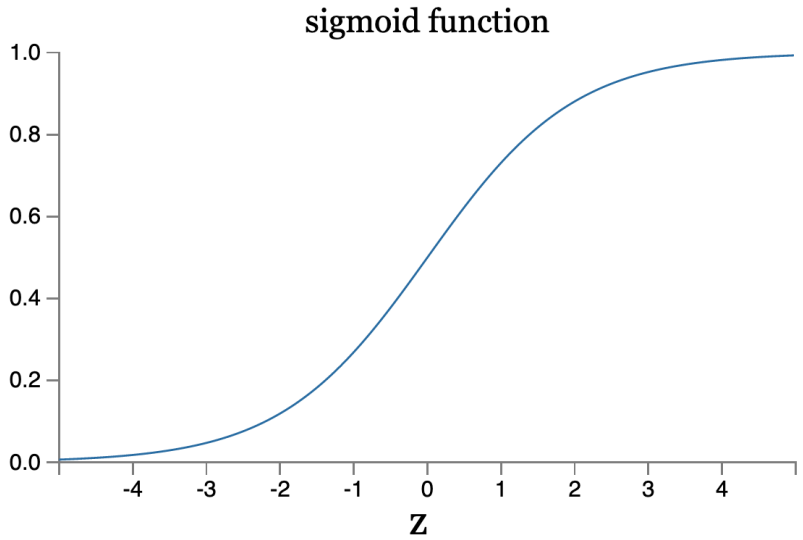
- Recall the delta weight adjustments from gradient descent when training linear models
- Linear models have an optimal solution that can be solved
- Neural networks (excepting trivially small ones) are intractable
    - You also don't want to (an optimal model is almost certainly overfitted)

Training (or why perceptrons won't cut it)



Activation functions (sigmoid)

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Activation functions (sigmoid)

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b$$

Activation functions (important aside)

Rectified Linear Unit (ReLU)

$$\max(0, x)$$

- Sigmoids saturate and can stop learning
- Empirically ReLU works well a lot of the time
- Others: Leaky ReLU, SeLU, Tanh

Recall gradient descent algorithm

Repeat until convergence:

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

For j=0, 1, **correct simultaneous update**:

$$temp_0 = \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta)$$
$$temp_1 = \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta)$$
$$\theta_0 = temp_0$$
$$\theta_1 = temp_1$$

Recall gradient descent algorithm

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

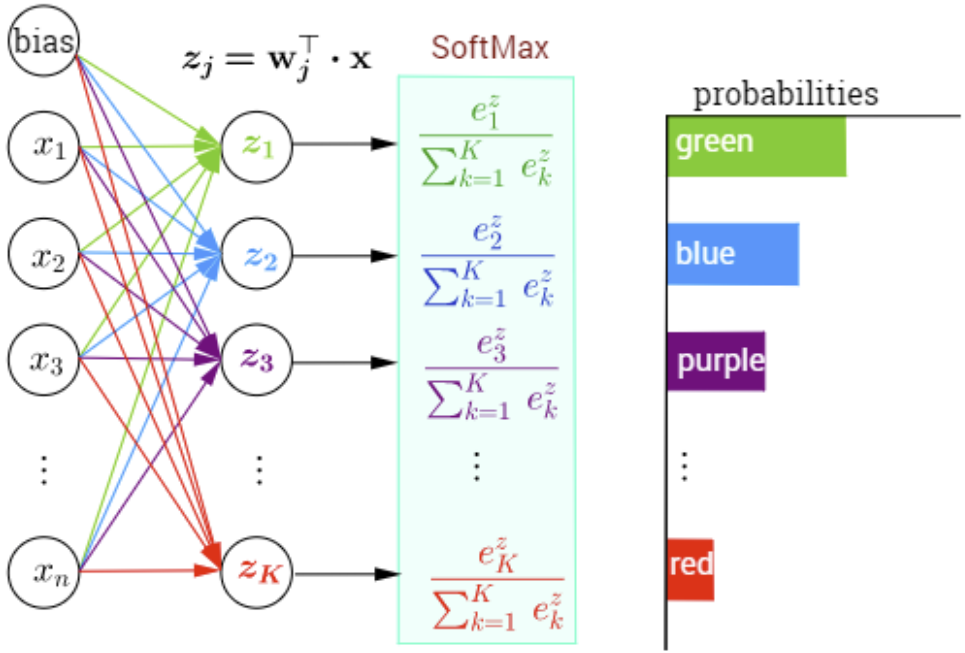
- How do you get the gradient of a weight for a neural network?
- Backprop!
- Work through Chapter 2 of neuralnetworksanddeeplearning.com
- We'll do this in code later

Minibatch training

- Standard way of training neural networks
- Take a random batch of training data
- Calculate the error for that batch in forward pass
- Update the weights
- Take the next batch
- When all batches are complete, one “epoch” has been trained
- Repeat! (usually)

Softmax

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \mathbf{w}_3^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$



Source: http://rinterested.github.io

Cross Entropy

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)]$$

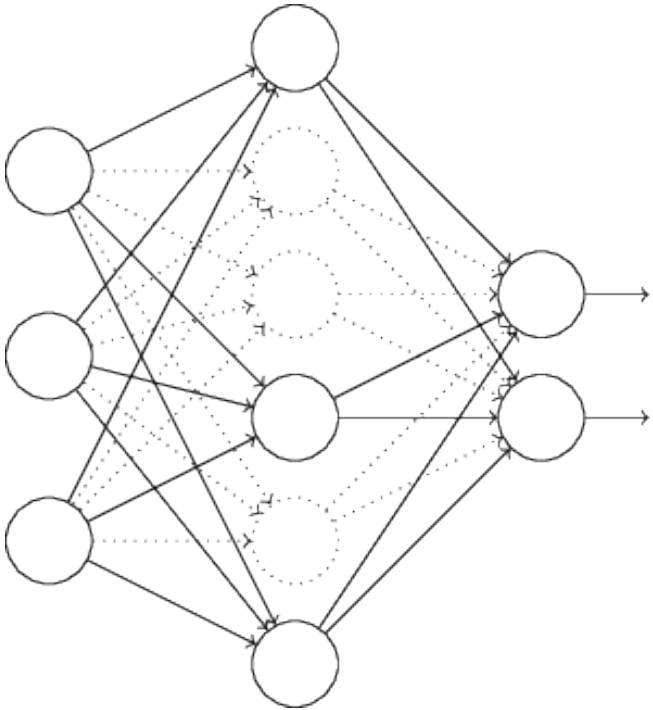
- A better loss function for learning in categorical problems
- For intuition: https://colah.github.io/posts/2015-09-Visual-Information/

Overfitting

- Neural networks, especially large ones, are extremely prone to overfitting because they have enough parameters to essentially memorize datasets
- Early stopping – ending training before validation loss starts to increase
- L1 and L2 (much as in last lecture)
- Dropout – randomly ”turning off” some portion of neurons during training
- More data!
- And much more
- Careful about hyperparameter-overfitting (retain separate validation and tests sets)

Dropout

Turn off some portion *p* of neurons
Train your mini



Resources

https://playground.tensorflow.org – an in-browser neural network builder that's extremely useful for building intuitions

http://neuralnetworksanddeeplearning.com/
(unspecified image sources come from here)

https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

http://cs231n.github.io/optimization-2/