CS 486 — Lecture 9: Artificial Neural Networks, Part 2

1 Gradient Descent

- Like walking downhill and taking a step in the direction that goes down the most.
- A local search algorithm to find the minimum of a function.
- Initialize weights randomly; change them in proportion to the negative of the partial derivative of the error with respect to the weight, or $w = w - \sum \eta \frac{\partial error}{\partial w}$.
- η is the learning rate.
- Set a max termination number of steps to prevent getting stuck forever!
- Two types of GD are:
  - Incremental gradient descent — update weights after each example.
  - Stochastic gradient descent — each example is chosen randomly.
- The problem with those is that while there are cheaper steps and the weights become accurate faster, it is not guaranteed to converge!
- There is a trade-off between learning speed and convergence — we can do batched GD, where we only update the weights after a *batch* of examples.
- So why do we update the weight proportionally to the negative of the partial derivative?
- When the partial derivative is positive, we want to head down towards the centre, so we would want the negative, and VV.
- In terms of magnitude, if it is large, then we want to take larger steps as it means we are still far. Likewise, if it is small, we want to take smaller steps to avoid overshooting.

2 Back-propagation

- Aims to learn the weights in a neural network by using GD.
- We'll refer to the XOR 3-layer NN for now.
- For each training example, (x_1, x_2, y_1, y_2), perform 2 passes:
  - Forward pass — compute o_1, o_2 given x_1, x_2 and the weights.
  - Backwards pass — calculate the partial derivative of the error with respect to each weight.
- Then update each weight by the sum of changes for all training examples.
- So what is the partial derivative, $\frac{\partial E}{\partial w2_{12}}$?
- $w2_{12}$ links to b_2 which links to o_2 which links to E.
- It's equal to:

$$\frac{\partial E}{\partial o_2} \times \frac{\partial o_2}{\partial b_2} \times \frac{\partial b_2}{\partial w2_{12}}$$

- And $E = (o_2 - y_2)^2 + (o_1 - y_1)^2$.

- $o_2 = g(b_w)$
- $g(x) = \frac{1}{1+e^{-x}} \implies g'(x) = g(x)(1 - g(x))$
- So, $\frac{\partial E}{\partial o_2} = 2(o_2 - y_2)$.
- And $\frac{\partial o_2}{\partial b_2} = o_2(1 - o)2 = g(b_2)(1 - g(b_2))$.
- And lastly $\frac{b_2}{w2_{12}} = h_1$.
- So, the original partial is $\frac{\partial E}{\partial w2_{12}} = 2(o_2 - y_2)o_2(1 - o_2)h_1$.
- Note we did $w2$ first — this is why it's a backwards pass, because doing so makes calculating $w1$ easier!
- Now, let's calculate $w1_{12}$.
- $w1_{12}$ affects a_2, then h_2, then b_1 *and* b_2, which go to o_1 and o_2 respectively which both then feed into E.
- This is equal to...

$$\frac{\partial E}{\partial W1_{12}} = \frac{\partial E}{\partial o_1}\frac{\partial o_1}{\partial b_1}\frac{\partial b_1}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial w1_{12}} + \frac{\partial E}{\partial o_2}\frac{\partial o_2}{\partial b_2}\frac{\partial b_2}{\partial h_2}\frac{\partial h_2}{\partial a_2}\frac{\partial a_2}{\partial w1_{12}}$$

- We actually have many parts of this due to calculating backwards!

3 NN vs. Decision Trees

- So... when do we use DT vs. NN?
- A NN is good if there is:
  - A high dimensional or real-valued inputs
  - Noisy data
  - Form of target function is not known
  - Not important for humans to explain the learned function
- Some disadvantages of NNs:
  - Difficult to determine the network structure (no. of layers, no. of neurons)
  - Difficult to interpret the weights, especially with multiple layers
  - Tendency to overfit in practice
- So which model we use depends on a few factors:
  - What data type do we have? NNs are good for complex/high dimensional data, like images, audio, text. DTs are better for *tabular* data.
  - The size of the dataset. DTs could work better with less data (think of the in-class example), but to train a NN that works well we need a lot of data.
  - What form does the target function take? A NN can express many functions, while a DT is basically just a bunch of nested if-else statements.
  - What is the architecture? For a DT, this is simple — it's just a tree, with few parameters unless you add things like pruning at a specific depth. But for a NN, there are *many* parameters — initial weights, learning rates, etc.
  - Do we need to interpret the learning function? DTs are simple, NNs can give learned functions that are impossible for people to explain.
  - How much time do we have for training and classification? A NN is slower to train and test in comparison to a DT.