Neural Nets

Chapter 6 Machine Learning

- The whole point is that previous methods (Linear Discriminant Functions, Support Vector Machines, Regression, etc.) all require the form of the fitting function to be well crafted by the designer (e.g. what coefficients should we use, should we use a linear, polynomial, exponential model, etc.)
  - With NNs, you can leave the optimal non-linear function finding to the model, as well as the coefficients (with the coeffs being the only thing you could leave to the model in the previous cases)
- The power of neural nets comes from their usage of non-linear functions after each layer's linear discriminant function (tanh, ReLU, sigmoid, etc.)
  - If you just kept things linear, you aren't able to model complex relationships and all your functions could be condensed into a one layer linear model (you're merely adding/ subtracting at that point)
- So at each node you have a net activation value that denotes the (pre-non-linearised) value of the linear discriminant, where the superscript on w denotes the layer in question which node j lives:

$$net_j = \sum_{i=0}^{n} w_{j,i}^{1} x_i$$

  - This translates to: "The activation at node j on the current layer is the sum of the n inputs x their respective weights from the previous layer"

Some different types of non-linear activation functions

- **ReLU** (Rectified Linear Unit): sets activations < 0 to 0. Most popular in the state of the art
- **Softplus**: A smoothed ReLU, differentiable across all values of x unlike ReLU but more expensive computationally
- **Step Function**: 1 if >0, -1 if < 0. (has an undefined gradient for x = 0 though so makes training trickier)
- **Sigmoid**: A smoothed step function bounding a curve from -1 to 1.
- **Softmax**: A normalising function that converts all outputs to a probability dist through normalisation. Very often used in the last layer.

Training

- Same principle of calculating some loss J, using l2 or whatever, difference is we propagate weights gradients in gradient descent.

- Important to note that derivative of the output of a particular node is only non-zero when done with respect to a weight that actually goes into that node.
- A lot of terms in back-prop can be reused and so are saved to reduce computation
- There are a couple of training types:
    - Stochastic Gradient Descent updates weights based on a particular data point in the training set
    - Batch Gradient Descent updates weights based on all data points at once.
    - You can compromise these two by choosing a batch size < training data set size
- An epoch denotes how many times the entire training dataset has been fed through the model.

Week 2 of the Stanford NLP & Deep Learning course

- NNs are able to create non-linear decision boundaries in hyperspace
- back prop works by assigning a loss function for the network, then using the weightings of each pre-descendent node, calculating how much each predecessor contributes to the loss
- Philosophically, the Bayesian school of thoughts believe that weights should be around 0.
- There is a back prop technique called like sparse sampling, wherein you randomly sample nodes to back prop each pass. Intuitively, its as though you are training many simple networks, whose average judgement is usually better than one monolith.
- A key principle in back prop is the idea that gradients for a layer are only dependent on the weights they affect (sumn like that)
- When training a model, it is computationally efficient to store the gradients for layer k-1 as these are to be used in layer k, making it computationally viable.