Lesson 3: Neural Networks

Artificial neural networks

Based on biological neuron (cell body, axon, and synapses – cell bodies fire signals down axons to synapses if their activation threshold is surpassed).

Perceptron

The perceptron hypothesis boils down to being the sign of the dot of the weights and the input. Let's say all inputs have a zeroth component which is 1 and the zeroth component of the weight is the negative threshold.

$$h(\mathbf{w})(\mathbf{x}) = \{\mathbf{w} \cdot \mathbf{x} \geq 0\}$$

The output is 0 if the dot product is negative, 1 if positive.

How powerful is a perceptron unit

Perceptrons **define a discriminant line which splits the (maybe high dimensional) plane in half**.

When the components of the input are boolean, then the perceptron can define **boolean logic** such as AND and OR, but a single perceptron can not discover XOR and others.

Given a vector $< -\theta, w_0, w_1 >$ be able to visualize a half-plane and given a half-plane write the weights vector.

Perceptron training

Two ways to learn weights

- Perceptron learning rule: use thresholded output
- Gradient descent/delta rule: use unthresholded output

The perceptron learning rule defines how to update the weights in iterative stages. Over time, it is **capable of converging on the best classifier** on the training set is the training set if **linearly separable**. However, it will not converge if the data is not linearly separable.

algorithm:

- Repeat for x_i, y_i while there is some error:

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta(y_i - \hat{y}_i)x_i$$

$$\hat{y}_i = (\sum_i x_i w_i \geq 0)$$

Where: y_i is the target, $\hat{y}_i$ is the output, x_i is the input, η is the learning rate.

This is online learning and you can stop when average error on the training set is below some threshold.

If the data is "nearly" linearly separable, perhaps reduce the learning rate over time such that the discriminant line at least appears to converge?

Perceptron learning can not be applied in the conventional sense to a multiple-layer neural network. It is only defined for a **single layer**.

Gradient descent

Is there a learning algorithm which is robust to non-(linear separability)?

We would like to perform gradient descent on the total error on the training set such that we can arrive at the weights which minimize the error.

Let D be the training set with ordered pairs (x, y). Let y be the target. Instead of taking the sign, we will **simply take the dot product of** w **and** x (this is essentially simple linear regression). We want to minimize:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{(\mathbf{x},y)\in D} (y - \mathbf{w} \cdot \mathbf{x})^2$$

The partials with respect to components of w:

$$\frac{\partial E}{\partial w_i} = - \sum_{(\mathbf{x},y)\in D} x_i(y - \mathbf{w} \cdot \mathbf{x})$$

Note that $\hat{y} = w \cdot x \geq 0$, but we are not using the $\hat{y}$ here.

The error decreases fastest in the direction opposite to the gradient. Walk in small steps opposite to the gradient.

Comparison of learning rules

The **perceptron learning** rule uses the following weight update. This rule guarantee the finite converge in linearly separable case.

$$\Delta w_i = \eta(y - \{\mathbf{w} \cdot \mathbf{x} \geq 0\})x_i$$

Learning a classifier using **gradient descent** on a linear regression model uses the following weight update. This is more robust to non-(linear separable) case but only to a local optimal.

$$\Delta w_i = \eta(y - \mathbf{w} \cdot \mathbf{x})x_i$$

Sigmoid

Why not do gradient descent on the original perceptron itself rather than remodeling it as linear regression? The step function is not **differentiable**. We can get around this with the sigmoid function (hyperbolic tan, arctan, logistic sigmoid).

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

Note that the derivative has a nice form:

$$\frac{d}{da}\sigma(a) = \sigma(a)(1 - \sigma(a))$$

Now we can use the chain rule to extend gradient descent to neurons with sigmoid activation functions.

Neural Networks

Using the sigmoid and the dot product, we can get the data flow:

Forward: input layer -> hidden layer -> output layer

Backward: Since all of the computation is differentiable, then we can compute the gradient using chain rule of the output with respect to all the parameters. So we can use the **backpropagation** to update the parameters.

With networks, due to the combination of lots of non-linear functions, there may be many local optima. We need to use clever techniques to arrive at a suitable set of weights.

Optimizing weights

How do arrive at a suitable set of weights? Optimization problem.

- Use momentum terms in the gradient (simulated annealing?)
- higher order derivatives (hamiltonians, etc.)
- randomized optimization (later on in the course)
- penalty for "complexity" (overfitting) – too many hidden layers? too many nodes in a layer? large valued weights (interesting)

Restriction bias

Restriction bias – the representational power and the set of hypotheses we will consider.

Perceptrons are linear models (discriminant lines splitting planes). Networks of perceptrons can approximate more complex functions. Sigmoids allow even better learning and can fit more interesting functions. -> **not much restriction bias at all**.

- Boolean functions – network of threshold-like units.
- Continuous functions – use sigmoid activation and a hidden layer.
- Arbitrary functions – stitch together two or more functions with discontinuitives with two hidden layers.

Overfitting problem with more complex networks.

- We can bound the architecture of the network.
- Cross validate to better bound the architecture or bound the weights.
- We should also find the number of iterations at which the network best generalizes.

Not really much of a restriction bias.

Preference bias

Preference bias – which representation is preferred by the algorithm.

Generally we initialize the network with small random weights – we prefer "simpler"/less complex representations. (This can hit a local minimum. We generally run the training multiple times to avoid local minimum.)

Practically, (Occam's Razor) produces **simpler and generalizable representations**.

Summary

- Perceptron (linear and thresholded)
- Perceptron learning rule
- Gradient descent
- Sigmoid function
- Restriction bias
- Preference bias