Chapter 6 Notes

Jonathan D Lima

1 Machine Learning Basics

Here we'll quickly recap a few basic ideas from the previous section, so that we can observe the role that they play in the design of Artificial Neural Networks (ANN).

- Parametrize a Family of Functions: $f_\theta(x)$.
  - θ: An (often vectorized) parametrization of the family.
  - x: An element from our input space.
- Define a Loss Function: $L(y, \hat{y})$.
  - L: Often the Negative Log-Likelihood (NLL) for some probabilistic model.
  - $\hat{y}$: The predicted value $f_\theta(x)$.
  - y: An element from our output space (corresponding to x).
- Define a Training Criterion (plus Regularizer): $C(\theta)$
  - First term approximates $\mathbb{E}_{P(x,y)}[L(y, f_\theta(x))] \approx \frac{1}{N} \sum_i L(y_i, f_\theta(x_i))$.
  - $\Omega(\theta)$: Regularizer enforces prior belief or preferences about θ, (e.g. linear combination of L1 and L2 norm).
- Define an Optimization Procedure:
  - Procedure to determine $\theta^* = argmin_\theta C(\theta)$
  - Optimization can be analytic (OLS), convex (Logistic Regression), or non-convex (most ANNs).

2 Feedforward Deep Networks

A graphical network of **nodes**, representing a computation from input nodes (x) to output nodes (y). Each node is a simple function most often comprised of an **affine transformation** ($\mathbf{a} \rightarrow b + \mathbf{w}^T \cdot \mathbf{a}$) composed with a **nonlinear transformation** (examples to follow). A whole **layer** of nodes can be represented as

$$\mathbf{h}^k = g^k(\mathbf{b}^k + W^k \mathbf{h}^{(k-1)}), \tag{1}$$

where h^k is the output of the k^{th} layer, g^k is the nonlinear transformation occuring at the k^{th} layer applied elementwise, and $\mathbf{b}^k$ and W^k are weights which parameterize the k^{th} layer of the learned predictor. The layers between input and output are referred to as **hidden**. The **depth** of the network (which coincides with the notion for partially ordered sets) is equal to the number of hidden layers plus one.

- Advantages of Deep Networks
  - A general framework (such as GLM) with flexibility to handle many different types of data generating mechanics, using different node functions and loss functions (discussed below).
  - Comes with a similarly general gradient computation procedure (back-propogation) to assist gradient optimization methods.
  - As a function, can theoretically approximate arbitrarily complex smooth functions with only one hidden layer.
  - In practice, can efficiently (in data and computation) represent complex relationships using more hidden layers.
- Disadvantages of Deep Networks
  - Typically non-convex optimization makes result highly dependent on initialization of parameters.
  - Somewhat limited interpretability.
- Kicker
  - Can do better than random initialization! (Future chapter)

3 Node Functions

Below we introduce several common node functions, giving definition for future discussion or reference. It should be noted that there is ambiguity regarding what is considered the work of one node, as opposed to the composition of two or more. Below we simply discuss building blocks.

- Many common node functions are evaluated on a scalar which is often the result of an affine transformation: $h(\mathbf{a}) = g(b + \mathbf{w}^T \cdot \mathbf{a}) = g(z)$.
  - **Rectifier, Rectified Linear Unit**: $g(z) = \max(0, z)$.
  - **Hyperbolic Tangent**: $g(z) = \tanh(z) = \frac{e^{2z}-1}{e^{2z}+1}$.
  - **Sigmoid**: $g(z) = \frac{1}{1+e^{-z}}$.
  - **Hard Tanh**: $g(z) = \max(-1, \min(1, z))$.
  - **Absolute Value Rectification**: $g(z) = |z|$.
  - **Soft-Plus**: Smooth version of rectifier
  - **Max-Out**: $g(z) = \max_i(b_i + w_i \cdot z)$
- Other node functions require a full input vector, $\mathbf{a}$, rather than just a scalar (such as z above).
  - **Soft-Max**: $h(\mathbf{a}) = \mathbf{p}$, if $p_i = \frac{e^{a_i}}{\sum e^{a_j}}$
  - **Radial Basis Function**: $h(a) = e^{-\|w-a\|^2/\sigma^2}$

The factors that effect the choice of hidden node functions are different from those that effect the decision of output node functions. Hidden node functions are often bounded (tanh, sigmoid, ...) which give an internal clustering or classification behaviour even if the end goal is regression. Some asymetrically take on a value of 0 part of the time and a positive value the other (rectifier), simulating the biological phenomenon of **sparse activation**, allowing certain mechanisms to act on some inputs, x, and not on others.

On the other hand output nodes must be constrained to fit the nature of the desired result, which differs in classification, regression, and other applications. A good way to think about this is through the lens of generalized linear models (GLM), which is essentially depth one ANN. GLM is general regression technique that applies to exponential families. The output layer of ANN roughly corresponds with the (inverse) link in GLM, which has a canonical choice for each distribution in the exponential family.

4 Training Criterion and Regularizer

4.1 Loss Function

Just as we discussed above with the output layer, the loss function should be taylored to the machine learning task, and accompanying probabilistic model. It is mentioned in 6.0.4 that using mean-squared-error (MSE) yields an estimator for $\mathbb{E}_{P(x,y)}[y|x]$, (if this function can be approximated by a function in the parametrized family).

That being said, with data which is conditionally Bernoulli with rare events MSE can overstate accuracy, whereas the negative-log-likelihood (NLL) of the Bernoulli model will correctly handle such errors. For this reason the appropriate NLL is often the loss function used. This again, is reminiscent of GLM, in which the loss function varies canonically with the probabilistic model.

4.2 Regularizers

Just as in most Bayesian methods, regularization of ANN can be viewed as prior on the effects of (input or internal) variables, the most common prior being that the variables have no effect. Common and simple forms of regularization for ANN are linear combinations of L1 and L2 norms. The L2 norm penalizes substantial deviation from 0, and the L1 norm penalizes all deviation from 0 equally. For this reason the L1 norm tends to lead to a sparse network, with many of the internal weights approximately equal to 0. Regularization can also be achieved by limiting the edges between layers, as opposed to allowing full connectivity between layers.

5 Optimization Procedure

As mentioned above ANN are equipped with a general gradient procedure, which make them suitable for gradient based optimization protocol's such as gradient descent. While the number of variables typically makes Newton's method (which requires a Hessian) infeasible, quasi-Newton methods such as BFGS (more prominently limited memory LBFGS), which are known to exploit the curvature of the function without requiring a Hessian, are commonly used. Because batch learning requires using the full set of data, which is often infeasible due to size, or impractical in the case of many applications in which the data is constantly growing, most training of ANN is done online in mini-batches (sometimes indicidual training examples).

5.1 Gradient Procedure: Backpropogation

The gradient procedure mentioned above is referred to as **backpropogation**. In its most general form, backpropogation is referred to as **automatic differentiation** and applies to any acyclic **flow graph**, a directed graphical decomposition of a computation into simpler node computations, and edges carrying results. Equipped with partial derivatives formulae for each (node function, node input), as one executes the flow graph, one can simultaneously compute partial derivatives of node functions with respect to its inputs (parent nodes).

Applications of the chain rules, allow one to use these partial derivatives to compute the partial derivative of any node function with respect to any ancestor. With this in mind, we can add one additional layer to the end of our ANN which takes the output, $\hat{y}$, and the training output, y, and executes the loss $L(y, \hat{y})$ (because the regularizer is typically independent of training data and has a simple gradient calculated using the weight parameters alone, it is left out here). With this addition the network is now a flow graph for the computation of $L(y, \hat{y})$. The procedure of backpropogation can be applied to obtain $\nabla_\theta L(y, \hat{y})$, or more frequently $\frac{1}{n}\sum_i \nabla_\theta L(y_i, \hat{y}_i)$ (over some minibatch).