CS 5785 – Applied Machine Learning – Lec. 19

Prof. Serge Belongie, Cornell Tech
Scribe: Jared W. , Luke A.

November 12, 2019

1 Neural Networks

Neural networks were developed separately from the statistics community, but they are nearly identical models to something from statistics called Projection Pursuit Regression (PPR). PPR was introduced in 1981, but didn't flourish, probably due to computational demands, but it has blossomed in its reincarnation in the field of neural networks. Neural networks are sometimes called "artificial neural networks" since actual neurons have numerous subtle properties that are not captured by the commonly used simple model.

1.1 "Vanilla" Neural Networks

We start with what HTF call the "vanilla" neural network. This is also referred to as the *single hidden layer back-propagation network* or *single layer perceptron.*

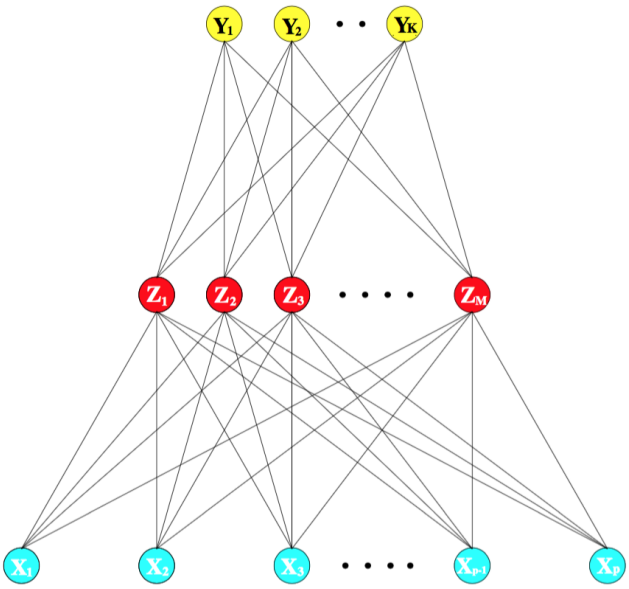

FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Neural networks can be used for both classification and regression. For classification, we use K units at the top, one for each class. For the case of MNIST digits, we have $K = 10$. This gives us K target measurements y_k, $k = 1, ..., K$, each coded as 0 or 1 for the kth class. This is also known as a *one-hot encoding*. The K units in the output layer model the probabilities of each class.

The computation effected by this network is as follows:

- $Z_m = \sigma(\alpha_{om} + \alpha_m^\top X),\ m = 1, ..., M$
- $T_k = \beta_{0k} + \beta_k^\top Z,\ k = 1, ..., K$
- $f_k(X) = g_k(T),\ k = 1, ..., K$
- where $Z = (Z_1, ..., Z_M)$ and $T = (T_1, ..., T_K)$

The activation function $\sigma(v)$ is usually chose to be sigmoid, $\sigma(v) = \frac{1}{1+e^{-v}}$, which we've seen before. It is nicely differentiable – a property we will soon exploit.

394 Neural Networks

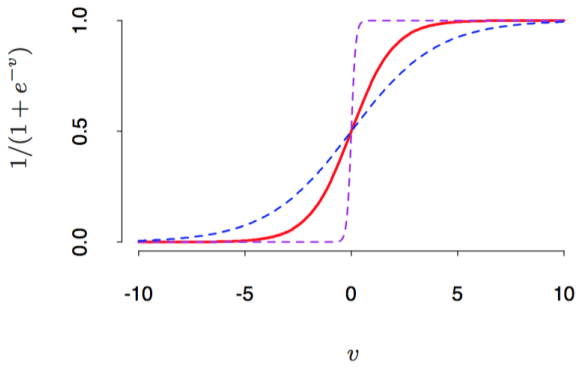

FIGURE 11.3. *Plot of the sigmoid function $\sigma(v) = 1/(1+\exp(-v))$ (red curve), commonly used in the hidden layer of a neural network. Included are $\sigma(sv)$ for $s = \frac{1}{2}$ (blue curve) and $s = 10$ (purple curve). The scale parameter s controls the activation rate, and we can see that large s amounts to a hard activation at $v = 0$. Note that $\sigma(s(v - v_0))$ shifts the activation threshold from 0 to v_0.*

As we saw before with linear regression, we can absorb the offset terms α_{0k} and β_{0k} by considering the constant 1 as an additional input feature. This makes notation cleaner.

The output of the function $g_k(T)$ allows a final transformation of the vector of outputs T; typically this is a softmax function:

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$

Softmax ensures that the density output is non-negative and sums to 1. It is the same transformation used in the multilogit model (Sec. 4.4). We only studied the 2-class case at the time, when we covered logistic regression.

Logistic regression is similar to using just one of the hidden layers. The idea is that you could have some raw inputs coming in, but the inputs might benefit from a "basis transformation" after which simple linear regression could get us to the result we want.

The units in the middle, $Z_m, m = 1, ..., M$ are called hidden units since we don't observe them directly. In general, one can have more than a single hidden layer. We can think of the Z_m as a "basis expansion" of the raw inputs X – the neural net is then a linear multilogit model using these transformations as inputs.

1.2 Fitting a Neural Network

Fitting a neural network means solving for the weights Θ consisting of:

$$\{\alpha_{0m}, \alpha_m; m = 1, 2, ..., M\} \qquad M(p+1) \quad \text{weights}$$

$$\{\beta_{0k}, \beta_k; k = 1, 2, ..., K\} \qquad K(M+1) \quad \text{weights}$$

The appropriate measure of fit (i.e., error function) depends on the task. Regression uses:

$$R(\Theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2$$

Note that this is sum of squares, with f_k representing the neural net. For classification we usually use cross-entropy:

$$R(\Theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

with classifier

$$G(X) = \arg\max_k \; f_k(x)$$

In the case of softmax activation function and the cross-entropy error function, this is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood.

2 Back-Propagation

Finding the global minimum of $R(\Theta)$ will probably result in overfitting to the training data. In practice we will need some form of regularization, either via penalty term or indirectly via early stopping. Applying gradient descent to $R(\Theta)$ has a special name: **back-propagation**. In purely mathematical terms, back-propagation is "simply the chain rule" of differentiation. It is implemented as a forward and backward sweep over the network, keeping track only of quantities local to each unit.

2.1 The Chain Rule

Recall the chain rule: $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$ when z is a function of y which is itself a function of x. This is not to be confused with chain rule of probability, which is useful in the study of Bayesian networks: $P(A, B, C) = P(A|B, C)P(B|C)P(C)$.

Let's examine back-propagation for the case of *squared error* loss. Let $z_{mi} = \sigma(\alpha_{0m} + \alpha_m^\top x_i)$ and let $z_i = (z_{1i}, z_{2i}, ..., z_{Mi})$. Recall that i ranges over observations ($i = 1, ..., N$) and k ranges over output units ($k = 1, ..., K$). Then we can write the cost function as

$$\begin{aligned} R(\theta) &= \sum_{i=1}^{N} R_i \\ &= \sum_{i=1}^{N}\sum_{k=1}^{K}(y_{ik} - f_k(x_i))^2 \end{aligned}$$

Now we look at the derivatives with respect to the βs and αs. Remember that the αs act on the input features and the βs act on the hidden nodes.

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_k^\top z_i)z_{mi} \tag{1}$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} \underline{2(y_{ik} - f_k(x_i))g_k'(\beta_k^\top z_i)}\beta_{km}\sigma'(\alpha_m^\top x_i)x_{il} \tag{2}$$

Next we apply a gradient descent update, expressed as:

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \beta_{km}^{(r)}} \\ \alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{N} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}} \end{aligned}$$

where r is the iteration and γ_r is the *learning rate*. This can be a constant or something more interesting as a function of r which we will discuss later.

To get more insight into what is happening in the gradient descent steps, let's use the following notation.

$$\begin{aligned} \frac{\partial R_i}{\partial \beta_{km}} &= \delta_{ki} z_{mi} \\ \frac{\partial R_i}{\partial \alpha_{ml}} &= s_{mi} x_{il} \end{aligned}$$

We can think of δ_{ki} and s_{mi} as "errors" arising from the current model at the output and hidden layer units respectively. Here is where the *back-propagation* idea emerges. These errors satisfy:

$$s_{mi} = \sigma'(\alpha_m^\top x_i)\sum_{k=1}^{K}\beta_{km}\delta_{ki} \tag{3}$$

How do we get this? Recall that the big portion of (2) comes from (1) (see above equations).

- We call the underlined quantity multiplying x_{il} in (2) s_{mi} and that, in turn, is a sum over the quantity multiplying z_{mi} in (1) over all K.
- The term $\sigma'(\alpha_m^\top x_i)$ doesn't depend on k, so we can pull it out front.

$\Rightarrow$ This says how these two errors are related to one another.
Using this formalism, we can do the gradient descent updates using a 2-pass algorithm:

1. Forward pass: fix the weights (the αs and βs) and compute the predicted values which we denote $\hat{f}_k(x_i)$, i.e., the output of the neural net.
2. Backward pass: compute the errors δ_{ki} and back propagate them using (3) to obtain s_{mi}.
3. With δ_{ki} and s_{mi} in hand, we can compute the gradients for the updates.

Further explanation available in this video: `https://www.youtube.com/watch?v=Ilg3gGewQ5U` (14 min.).[1]

2.2 Discussion on Back-Propagation

The nice feature of back-propagation is its simple, local nature. Each hidden unit passes and receives information only to and from units that share a connection. This makes it particularly well suited for parallel computation.

The gradient descent update we derived is a form of batch learning, with parameter updates being a sum over all training cases. As we saw in the discussion on IRLS, we can drop the sum and update the gradient after each training case (Stochastic Gradient Descent).

- Whereas γ_r is usually chosen (after some simple optimization) to be a constant in the batch case, in the stochastic case $\gamma_r \to 0$ as $r \to 0$.
- Second order techniques (like Newton-Raphson) are problematic since the Hessian of k is too big.

There are many, many tricks of the trade that exist for speeding up back-propagation, however we will not be covering them in this class.

We need to understand that these models are generally overparameterized (parameters > input) and sometimes dramatically so. This makes them susceptible to overfitting, and the optimization problem is nonconvex. Other issues include: input scaling, weight decay (and other forms of regularization), number of hidden layers, and hidden units. As HTF says, "There is quite an art in training neural networks". $\ddot{\smile}$

[1] 3Blue1Brown S3 E3, "What is backpropagation really doing?" Deep Learning, Ch. 3.