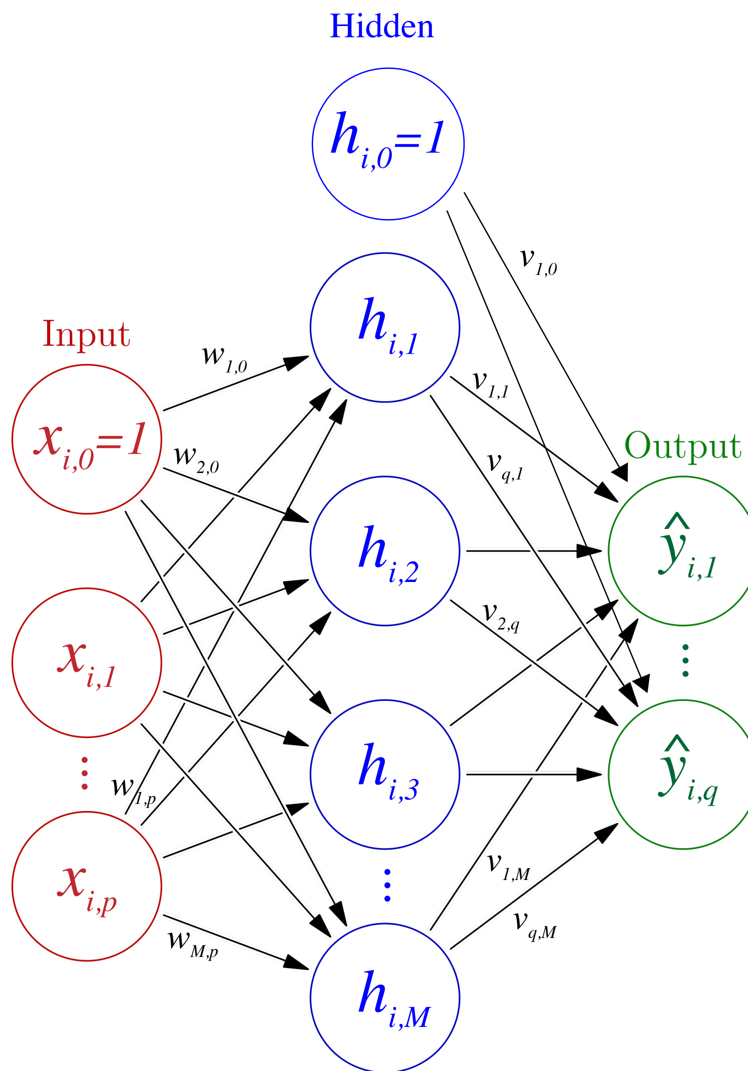
© 2016 Rebecca Willett

Backpropagation in Neural Networks

Artificial neural networks can be used to learn predictors in a wide variety of machine learning settings. The basic idea is take a feature vector $\boldsymbol{x} \in \mathbb{R}^p$, compute different weighted combinations of the p covariates, apply a nonlinear rectification function σ to each weighted combination, and repeat this process for each layer in the network. The idea is illustrated for a three-level network with input $\boldsymbol{x}_i$ below.



In this figure,

$$h_{i,m} = \sigma\left(\sum_{j=0}^{p} w_{m,j} x_{i,j}\right) \qquad \text{for } m = 1, \ldots, M$$

and

$$\hat{y}_{i,k} = \sigma\left(\sum_{m=0}^{M} v_{k,m} h_{i,m}\right) = \sigma\left(\sum_{m=0}^{M} v_{k,m}\sigma\left[\sum_{j=0}^{p} w_{m,j} x_{i,j}\right]\right) \qquad \text{for } k = 1, \ldots, q.$$

Common choices for σ include

- $\sigma(z) = \text{sign}(z) \mapsto -1, 0, 1$
- $\sigma(z) = \frac{1}{1+e^{-z}} \mapsto [0, 1]$
- $\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \mapsto [-1, 1]$

Given training samples $(\boldsymbol{x}_i, y_i) \in [-1, 1]^p \times \{0, 1\}$ for $i = 1, \ldots, n$, we want to learn weights $\{w_{m,j}\}_{m,j}$ and $\{v_{k,m}\}_{k,m}$ so that if we input $\boldsymbol{x}_i$ into the network, the output $\hat{y}_i$ is close to the training label y_i.

Much current ML research is focused on "deep" networks with many more than the three levels depicted here. However, even the network with one hidden layer shown above is very flexible. In particular, let $\sigma(\cdot)$ be a nonconstant, bounded, and monotonically-increasing continuous function. Let $g(\boldsymbol{x})$ be a continuous function mapping $[0, 1]^p \mapsto \mathbb{R}$. (Think of this an ideal predictor.) For any $\epsilon > 0$, there exists an integer M and weights $\{w_{m,j}\}_{m,j}$ and $\{v_m\}_{m=0}^M$ so that if we define

$$G(\boldsymbol{x}) = \sum_{m=0}^{M} v_m \sigma\left(\sum_{j=0}^{p} w_{j,m} x_j\right)$$

then

$$|G(\boldsymbol{x}) - g(x)| < \epsilon \qquad \forall\, x \in [0, 1]^p.$$

That is, this simple three-layer network can approximate any predictor function g arbitrarily well given enough hidden nodes and the right set of weights. This is known as the *universal approximation theorem*.[1]

Deep learning research is generally focused on two goals:

- Constructing the network – i.e. deciding how many layers to use, how many nodes in each layer, and the choice of σ;
- Estimating network weights / parameters using training data.

At this point, we have limited theory telling us the "right" way to perform the first task. For the second task, we can use SGD within a backpropagation framework to estimate weights.

1 Learning weights in two-layer network

We first consider a simpler, two-layer network with a single output in which

$$\hat{y}_i = \sigma\left(\sum_{j=0}^{p} w_j x_{i,j}\right).$$

[1] Hornik, Kurt. "Approximation capabilities of multilayer feedforward networks." Neural networks 4.2 (1991): 251-257.

We can learn weights via SGD! Our objective function is

$$f(\boldsymbol{w}) = \sum_{i=1}^{n} \frac{1}{2}(\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} \underbrace{\frac{1}{2}\left[\sigma\left(\sum_{j=0}^{p} w_j x_{i,j}\right) - y_i\right]^2}_{=:f_i(\boldsymbol{w})}$$

where $\boldsymbol{w} = [w_0, w_1, \ldots, w_p]^T$.

When $\sigma(z) = \text{sign}(z)$, SGD on this objective will yield the *perceptron algorithm*. Today we will consider $\sigma(z) = \frac{1}{1+e^{-z}}$. Note that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Here is SGD applied to this setting:

- Choose i_t uniformly at random from $\{1, 2, \ldots, n\}$.
- Set $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \alpha_t \nabla f_{i_t}(\boldsymbol{w}^{(t)})$.

Let's characterize $\nabla f_{i_t}(\boldsymbol{w}^{(t)})$. Let's consider the jth element of $\boldsymbol{w}^{(t)}$, and use i for i_t:

$$\begin{aligned}
\frac{df_i}{dw_j}\bigg|_{\boldsymbol{w}^{(t)}} &= \frac{df_i}{d\hat{y}_i}\frac{d\hat{y}_i}{dw_j}\bigg|_{\boldsymbol{w}^{(t)}} \\
&= (\hat{y}_i - y_i)\sigma'\left(\sum_{j'=0}^{p} w_{j'} x_{i,j'}\right) x_{i,j} \\
&= (\hat{y}_i - y_i)\sigma\left(\sum_{j'=0}^{p} w_{j'} x_{i,j'}\right)\left[1 - \sigma\left(\sum_{j'=0}^{p} w_{j'} x_{i,j'}\right)\right] x_{i,j} \\
&= \underbrace{(\hat{y}_i - y_i)\hat{y}_i(1 - \hat{y}_i)}_{\delta_i} x_{i,j}
\end{aligned}$$

Note δ_i is a scalar independent of j. Thus

$$\nabla f_{i_t}(\boldsymbol{w}^{(t)}) = \delta_{i_t}\boldsymbol{x}_{i_t}$$
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \alpha_t \delta_{i_t}\boldsymbol{x}_{i_t}$$

This is called the *delta rule* in the neural network community.

2 Learning weights in a 3-layer network

Now we return to the three-layer network presented at the beginning of this note. Note that at iteration t the network predicts label

$$\hat{y}_{i_t,k} = \sigma\left(\sum_{m=0}^{M} v_{k,m}^{(t)} h_{i_t,m}\right) = \sigma\left(\sum_{m=0}^{M} v_{k,m}^{(t)} \sigma\left[\sum_{j=0}^{p} w_{m,j}^{(t)} x_{i_t,j}\right]\right) \qquad \text{for } k = 1, \ldots, q.$$

In the following, we will drop the t notation for simplicity of presentation and simply write

$$\hat{y}_{i,k} = \sigma\left(\sum_{m=0}^{M} v_{k,m} h_{i,m}\right) = \sigma\left(\sum_{m=0}^{M} v_{k,m} \sigma\left[\sum_{j=0}^{p} w_{m,j} x_{i,j}\right]\right) \qquad \text{for } k = 1, \dots, q$$

with t understood. To estimate the parameters of this network, we use the *backpropagation algorithm*:

1. Choose initial weights $\{v_{k,m}^{(1)}\}_{k,m}$ and $\{w_{m,j}^{(1)}\}_{m,j}$.
2. For $t = 1, 2, 3, \dots$
   (a) Choose i_t uniformly at random from $\{1, \dots, n\}$.
   (b) Calculate values of $\hat{y}_{i_t,k}$ and $h_{i_t,m}$ using weights $\{v_{k,m}^{(t)}\}_{k,m}$ and $\{w_{m,j}^{(t)}\}_{m,j}$.
   (c) Optimize the weights for each layer, starting with the deepest layer (closest to outputs) and working back to the shallowest layer (closest to inputs).

To update the $v_{k,m}$s, we note

$$\begin{aligned}
\frac{df_i}{dv_{k,m}} &= \frac{df_i}{d\hat{y}_i} \frac{d\hat{y}_i}{dv_{k,m}} \\
&= (\hat{y}_{i,k} - y_{i,k}) \sigma'\left(\sum_{m'=0}^{M} v_{k,m'} h_{i,m'}\right) h_{i,m} \\
&= \underbrace{(\hat{y}_{i,k} - y_{i,k}) \hat{y}_{i,k} (1 - \hat{y}_{i,k})}_{=:\delta_{i,k}} h_{i,m} \\
&= \delta_{i,k} h_{i,m}.
\end{aligned}$$

Thus we can update the v's via SGD as follows:

$$\begin{gathered}
v_{k,m}^{(t+1)} = v_{k,m}^{(t)} - \alpha_t \delta_{i_t,k} h_{i_t,m} \qquad \text{for} \qquad k = 1, \dots q, \qquad m = 0, \dots, M \\
\boldsymbol{v}_k^{(t+1)} = \boldsymbol{v}_k^{(t)} - \alpha_t \delta_{i_t,k} h_{i_t} \qquad \text{for} \qquad k = 1, \dots q \\
\boldsymbol{V}^{(t+1)} = \boldsymbol{V}^{(t)} - \alpha_t h_{i_t} \delta_{i_t}^T
\end{gathered}$$

where $h_{i_t} = [h_{i_t,0}, h_{i_t,1}, \dots, h_{i_t,M}]^T \in \mathbb{R}^{M+1}$, $\delta_{i_t} = [\delta_{i_t,1}, \dots, \delta_{i_t,q}]^T \in \mathbb{R}^q$, $\boldsymbol{V}^{(t)} = (v_{k,m}^{(t)})^T \in \mathbb{R}^{(M+1)\times q}$.

To update the $w_{m,j}$s, we note by the chain rule

$$\begin{aligned}
\frac{df_i}{dw_{m,j}} &= \sum_{k=1}^{q} \frac{df_i}{d\hat{y}_{i,k}} \frac{d\hat{y}_{i,k}}{dh_{i,m}} \frac{dh_{i,m}}{dw_{m,j}} \\
&= \sum_{k=1}^{q} (\hat{y}_{i,k} - y_{i,k}) \sigma' \left(\sum_{m'=0}^{M} v_{k,m'} h_{i,m'} \right) v_{k,m} \sigma' \left(\sum_{j'=0}^{p} w_{m,j'} x_{i,j'} \right) x_{i,j} \\
&= \sum_{k=1}^{q} \underbrace{(\hat{y}_{i,k} - y_{i,k}) \hat{y}_{i,k} (1 - \hat{y}_{i,k})}_{=:\delta_{i,k}} v_{k,m} h_{i,m} (1 - h_{i,m}) x_{i,j} \\
&= \underbrace{\sum_{k=1}^{q} \delta_{i,k} v_{k,m} h_{i,m} (1 - h_{i,m})}_{\gamma_{i,m}} x_{i,j} \\
&= \gamma_{m,i} x_{i,j}.
\end{aligned}$$

Thus we can update the w's via SGD as follows:

$$\begin{aligned}
w_{m,j}^{(t+1)} &= w_{m,j}^{(t)} - \alpha_t \gamma_{i_t,m} x_{i_t,j} \qquad \text{for} \qquad j = 0, \ldots p, \qquad m = 0, \ldots, M \\
\boldsymbol{w}_m^{(t+1)} &= \boldsymbol{w}_m^{(t)} - \alpha_t \gamma_{i_t,m} x_{i_t} \qquad \text{for} \qquad m = 0, \ldots M \\
\boldsymbol{W}^{(t+1)} &= \boldsymbol{W}^{(t)} - \alpha_t x_{i_t} \gamma_{i_t}^T
\end{aligned}$$

where $\gamma_{i_t} = [\gamma_{i_t,0}, \ldots, \gamma_{i_t,M}]^T \in \mathbb{R}^q$, $\boldsymbol{W}^{(t)} = (w_{m,j}^{(t)})^T \in \mathbb{R}^{(p+1)\times(M+1)}$.

3 Deeper networks

For deeper networks this trend continues; for each layer, compute the gradient of f_{i_t} with respect to the weights at that layer using the chain rule and then use that gradient to update the weights. At each layer, if we need to use weights from other layers to compute our gradients, then we always use the weights from the previous iteration (i.e. we always use the node values calculated in step (b) above).

Note that our overall objective function is highly nonconvex in the weight parameters. Thus this method, while based on SGD, will only converge to a local minimum, and not necessarily to the global minimum. In practice, people run this algorithm for many different initial values of the weight parameters and choose the weights corresponding to the lowest local minimum among all these runs.

4 Minibatch

Sometimes practitioners use *minibatch* methods instead of vanilla SGD. Essentially, at each round t, instead of choosing a single sample i_t, we choose a set of $B \ll n$ samples $\mathcal{I}_t \subseteq \{1, \ldots, n\}$ with $|\mathcal{I}_t| = B$. Then instead of computing gradient $\nabla f_{i_t}(\boldsymbol{w}^{(t)})$ to update $\boldsymbol{w}^{(t)}$, we instead compute the minibatch gradient $\sum_{i \in \mathcal{I}_t} \nabla f_i(\boldsymbol{w}^{(t)})$.

5 Batch normalization

Consider updating a single layer of a neural network with minibatch inputs $h_{i,m},\ m = 0, \ldots, M$ and outputs $\hat{y}_{i,k},\ k = 1, \ldots, q$ for $i \in \mathcal{I}_t$. Empirically it has been noted that the backpropagation methods converges most quickly when the inputs are approximately normal with zero mean and unit variance. Unfortunately, this generally does not occur. Furthermore, every update of the weights in previous layers can change the distribution of the weights in the current layer. *Batch normalization* attempts to improve convergence by renormalizing the minibatch inputs.[2]

For this layer of the network, we have

$$\hat{y}_{i,q} = \sigma\left(\underbrace{\sum_{m=0}^{M} w_{q,m} h_{i,m}}_{=:z_{i,q}}\right).$$

The basic idea of batch normalization is to "whiten" the zs corresponding to a minibatch $\mathcal{I}_t$: for some small $\epsilon > 0$, and parameters γ_q and β_q that will be learned, we set

$$\begin{aligned}
\mu_q &:= \frac{1}{B} \sum_{i \in \mathcal{I}_t} z_{i,q} \\
\sigma_q^2 &:= \frac{1}{B} \sum_{i \in \mathcal{I}_t} (z_{i,q} - \mu_q)^2 \\
\tilde{z}_{i,q} &:= \frac{z_{i,q} - \mu_q}{\sqrt{\sigma_q^2 + \epsilon}} \\
\hat{z}_{i,q} &:= \gamma_q \tilde{z}_{i,q} + \beta_q \\
\hat{y}_{i,q} &= \sigma(\hat{z}_{i,q}).
\end{aligned}$$

Within this framework, to update a layer we (a) optimize γ_q and β_q using the $z_{i,q}$s from the previous round, and then (b) update the $w_{q,m}$ using the above relations, the chain rule, and SGD.

[2] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).