Neuron-Inspired Classifiers

\[
\text{weights} \times x_n \sum \text{tanh} \sum v \times \left( y_n - L_n \right) \text{loss}
\]
Two-Layer Neural Network

\[ f(x) = \text{sign}\left( \sum_{h=1}^{H} v_h \cdot \tanh (w_h \cdot x + b_h) \right) \]

\[ = \text{sign}\left( v \cdot \tanh (Wx + b) \right) \]

- Two-layer networks allow decision boundaries that are nonlinear.
- It’s fairly easy to show that “XOR” can be simulated (recall conjunction features from the “practical issues” lecture on 10/18).
- Theoretical result: any continuous function on a bounded region in \( \mathbb{R}^d \) can be approximated arbitrarily well, with a finite number of hidden units.
- The number of hidden units affects how complicated your decision boundary can be and how easily you will overfit.
Learning with a Two-Layer Network

Parameters: $W \in \mathbb{R}^{H \times d}$, $b \in \mathbb{R}^H$, and $v \in \mathbb{R}^H$

- If we choose a differentiable loss, then the whole function will be differentiable with respect to all parameters.
- Because of the squashing function, which is not convex, the overall learning problem is not convex.
- What does (stochastic) (sub)gradient descent do with non-convex functions? It finds a local minimum.
- To calculate gradients, we need to use the chain rule from calculus.
- Special name for (S)GD with chain rule invocations: backpropagation.
Backpropagation

For every node in the computation graph, we wish to calculate the first derivative of $L_n$ with respect to that node. For any node $a$, let:

$$\bar{a} = \frac{\partial L_n}{\partial a}$$
Backpropagation

For every node in the computation graph, we wish to calculate the first derivative of $L_n$ with respect to that node. For any node $a$, let:

$$\bar{a} = \frac{\partial L_n}{\partial a}$$

Base case:

$$\bar{L}_n = \frac{\partial L_n}{\partial L_n} = 1$$
Backpropagation

For every node in the computation graph, we wish to calculate the first derivative of $L_n$ with respect to that node. For any node $a$, let:

$$\bar{a} = \frac{\partial L_n}{\partial a}$$

Base case:

$$\bar{L}_n = \frac{\partial L_n}{\partial L_n} = 1$$

From here on, we overload notation and let $a$ and $b$ refer to nodes and to their values.
Backpropagation

For every node in the computation graph, we wish to calculate the first derivative of \( L_n \) with respect to that node. For any node \( a \), let:

\[
\bar{a} = \frac{\partial L_n}{\partial a}
\]

After working forwards through the computation graph to obtain the loss \( L_n \), we work \textit{backwards} through the computation graph, using the chain rule to calculate \( \bar{a} \) for every node \( a \), making use of the work already done for nodes that depend on \( a \).

\[
\frac{\partial L_n}{\partial a} = \sum_{b:a \to b} \frac{\partial L_n}{\partial b} \cdot \frac{\partial b}{\partial a}
\]

\[
\bar{a} = \sum_{b:a \to b} \bar{b} \cdot \frac{\partial b}{\partial a}
\]

\[
= \sum_{b:a \to b} \bar{b} \cdot \begin{cases} 
1 & \text{if } b = a + c \text{ for some } c \\
 c & \text{if } b = a \cdot c \text{ for some } c \\
 1 - b^2 & \text{if } b = \tanh(a)
\end{cases}
\]
Backpropagation with Vectors

Pointwise ("Hadamard") product for vectors in $\mathbb{R}^n$:

$$a \odot b = \begin{bmatrix} a[1] \cdot b[1] \\ a[2] \cdot b[2] \\ \vdots \\ a[n] \cdot b[n] \end{bmatrix}$$

$$\bar{a} = \sum_{b:a \rightarrow b} \sum_{i=1}^{\|b\|} \bar{b}[i] \cdot \frac{\partial b[i]}{\partial a}$$

$$\begin{cases} 
\bar{b} & \text{if } b = a + c \text{ for some } c \\
\bar{b} \odot c & \text{if } b = a \odot c \text{ for some } c \\
\bar{b} \odot (1 - b \odot b) & \text{if } b = \tanh(a)
\end{cases}$$
Backpropagation, Illustrated

Intermediate nodes are de-anonymized, to make notation easier.
Backpropagation, Illustrated

\[ \frac{\partial L_n}{\partial L_n} = 1 \]
The form of $\bar{g}$ will be loss-function specific (e.g., $-2(y_n - g)$ for squared loss).
Backpropagation, Illustrated

\[ y_n = L_n \]

\[ e = \tanh(a) \]

\[ f = v \odot e \]

\[ g = \sum_h f[h] \]

\[ d = Wx_n \]

\[ a = b + d \]

\[ x_n \]

\[ W \]

\[ b \]

\[ v \]

\[ y_n \]

\[ L_n \]

Sum.
Backpropagation, Illustrated

Product.
Backpropagation, Illustrated

\[ d = Wx_n \]

\[ a = b + d \]

\[ e = \text{tanh} \ a \]

\[ f = v \odot e \]

\[ g = \sum_h f[h] \]

\[ \bar{a} = \bar{g} \cdot v \odot (1 - e \odot e) \]

\[ \bar{g} \cdot v \]

\[ \bar{g} \cdot 1 \]

\[ \bar{g} \cdot e \]

\[ y_n \]

\[ L_n \]

\[ 1 \]

Hyperbolic tangent.
Backpropagation, Illustrated

\[
\begin{align*}
    x_n W d &= W x_n a = b + d \\
    b \\
    \tilde{a} \leftarrow \tilde{a} = \tilde{g} \cdot v \odot (1 - e \odot e) \\
    \tilde{g} \cdot v \\
    y_n \rightarrow L_n \rightarrow 1 \\
    v = v \odot e \\
    g = \sum_h f[h] \\
    f = v \odot e \\
    \tilde{g} \cdot 1 \\
    \tilde{g} \cdot e \\
    \bar{a} = \tilde{g} \cdot v \odot (1 - e \odot e) \\
\end{align*}
\]
Backpropagation, Illustrated

\[
\tilde{a} = \tilde{g} \cdot v \odot (1 - e \odot e)
\]

\[
\tilde{g} \cdot v
\]

\[
x_n^T \bar{a} = \bar{a}
\]

\[
d = Wx_n
\]

\[
a = b + d
\]

\[
e = \tanh a
\]

\[
f = v \odot e
\]

\[
g = \sum_h f[h]
\]

\[
\bar{a} = \bar{g} \cdot v
\]

\[
\bar{g} \cdot 1
\]

Product.
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
- Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
- Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
- Hyperparameters?
Practical Notes

▶ Don’t initialize all parameters to zero; add some random noise.
▶ Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
▶ Hyperparameters?
  ▶ number of training iterations
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
- Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
- Hyperparameters?
  - number of training iterations
  - learning rate for SGD
Practical Notes

▶ Don’t initialize all parameters to zero; add some random noise.
▶ Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
▶ Hyperparameters?
  ▶ number of training iterations
  ▶ learning rate for SGD
  ▶ number of hidden units ($H$)
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
- Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
- Hyperparameters?
  - number of training iterations
  - learning rate for SGD
  - number of hidden units ($H$)
  - number of “layers” (and number of hidden units in each layer)
Practical Notes

▶ Don’t initialize all parameters to zero; add some random noise.
▶ Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
▶ Hyperparameters?
  ▶ number of training iterations
  ▶ learning rate for SGD
  ▶ number of hidden units ($H$)
  ▶ number of “layers” (and number of hidden units in each layer)
  ▶ amount of randomness in initialization
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
- Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
- Hyperparameters?
  - number of training iterations
  - learning rate for SGD
  - number of hidden units $(H)$
  - number of “layers” (and number of hidden units in each layer)
  - amount of randomness in initialization
  - regularization
Practical Notes

- Don’t initialize all parameters to zero; add some random noise.
- Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
- Hyperparameters?
  - number of training iterations
  - learning rate for SGD
  - number of hidden units ($H$)
  - number of “layers” (and number of hidden units in each layer)
  - amount of randomness in initialization
  - regularization
- Interpretability?
Practical Notes

▶ Don’t initialize all parameters to zero; add some random noise.
▶ Random restarts: train $K$ networks with different initializers, and you’ll get $K$ different classifiers of varying quality.
▶ Hyperparameters?
  ▶ number of training iterations
  ▶ learning rate for SGD
  ▶ number of hidden units ($H$)
  ▶ number of “layers” (and number of hidden units in each layer)
  ▶ amount of randomness in initialization
  ▶ regularization
▶ Interpretability? 😐
Challenge of Deeper Networks

Backpropagation aims to assign “credit” (or “blame”) to each parameter. In a deep network, credit/blame is shared across all layers. So parameters at early layers tend to have very small gradients. One solution is to train a shallow network, then use it to initialize a deeper network, perhaps gradually increasing network depth. This is called **layer-wise** training.
Radial Basis Function Networks

In the diagram, \( \text{sqd}(x, w) = \|x - w\|_2^2 \).
Radial Basis Function Networks

Generalizing to $H$ hidden units:

$$f(x) = \text{sign} \left( \sum_{h=1}^{H} v[h] \cdot \exp \left( -\gamma_h \cdot \| x - w_h \|^2_2 \right) \right)$$

Each hidden unit is like a little “bump” in data space. $w_h$ is the position of the bump, and $\gamma_h$ is inversely proportional to its width.
A Gentle Reading on Backpropagation

http://colah.github.io/posts/2015-08-Backprop/