Neural Networks: Backpropagation

Adopted from slides by Alexander Ihler and Andrew Ng
Neural networks

\[ z_1 = W^1 x \]
\[ z_2 = W^2 h^1 \]
\[ \vdots \]
\[ z_{L-1} = W^{L-1} h_{L-2} \]
\[ z_L = W^L h_{L-1} \]
\[ \hat{y} = \sigma(z_L) \]

Forward propagation

\[ h^1 = \sigma(z^1) \]
\[ h^2 = \sigma(z^2) \]
\[ \vdots \]
\[ h_{L-1} = \sigma(z_{L-1}) \]

Overall, we can denote \( \hat{y} = h_W(x) \)

\[
W^l = \begin{pmatrix}
  w_{1,1}^l & \cdots & w_{1,n_{l-1}}^l \\
  \vdots & \ddots & \vdots \\
  w_{n_l,1}^l & \cdots & w_{n_l,n_{l-1}}^l
\end{pmatrix}
\]
Training MLPs

• Observe features “x” with target “y”
• Push “x” through NN = output is “ŷ”
• Error: \((y - ŷ)^2\)  

(Can use different loss functions if desired...)
• How should we update the weights to improve?

• Single layer
  • Logistic sigmoid function
  • Smooth, differentiable

• Optimize using:
  • Batch gradient descent
  • Stochastic gradient descent
Loss function (single output node)

- Regularized logistic regression:
  \[ J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left( y^{(i)} \log h_{\theta}(x^{(i)}) + (1 - y^{(i)}) \log \left(1 - h_{\theta}(x^{(i)})\right) \right) + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2 \]

- Neural network with cross entropy loss:
  \[ J(W) = -\frac{1}{m} \sum_{i=1}^{m} \left( y^{(i)} \log h_{W}(x^{(i)}) + (1 - y^{(i)}) \log \left(1 - h_{W}(x^{(i)})\right) \right) + \frac{\lambda}{2m} \sum_{l=1}^{L} \sum_{j=1}^{n_l} \sum_{i=1}^{n_{l-1}} (w_{j,i}^l)^2 \]
Loss function (multiple output nodes)

\[ J(W) = \]
\[ -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} (y_k^{(i)} \log h_w(x^{(i)}_k) + (1 - y_k^{(i)}) \log (1 - h_w(x^{(i)}_k))) \]
\[ + \frac{\lambda}{2m} \sum_{l=1}^{L} \sum_{j=1}^{n_l} \sum_{i=1}^{n_{l-1}} (w_{j,i}^l)^2 \]
Loss function

\[ J(W) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, h_W(x^{(i)})) + \frac{\lambda}{2m} \mathcal{R}(W) \]

Cross entropy loss:

\[ \mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \]

\[ \mathcal{L}(y, \hat{y}) = \sum_{k} (-y_k \log \hat{y}_k - (1 - y_k) \log(1 - \hat{y}_k)) \]
Optimization - gradient descent

- Compute \( \frac{\partial J(W)}{\partial w_{j,i}} \)

- Different from logistic regression (or SVM), where all parameters are equivalent in terms of positions in the loss function, parameters in neural network are different due to their locations in the network structure.

Initialize \( W \)
Do {
    \( W \leftarrow W - \alpha \nabla J(W) \)
} while (stop condition)
Gradient calculations

\[ J(W) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(y^{(i)}, h_W(x^{(i)})) + \frac{\lambda}{2m} \mathcal{R}(W) \]

- Compute \( \frac{\partial \mathcal{L}(y, \hat{y})}{\partial w^l_{j,i}} \) where \( \hat{y} = h_W(x) \)
Gradient calculations

• Think of NNs as “schematics” made of smaller functions
  • Building blocks: summations & nonlinearities
  • For derivatives, just apply the chain rule, etc!

\[ f(g, h) \rightarrow \ldots \rightarrow J(\ldots) \]
Gradient calculations

• Chain rule:
  • If \( F(x) = f(g(x)) \), then \( F'(x) = f'(g(x))g'(x) \)
  • Alternative form, if \( z \) depends on \( y \), and \( y \) depends on \( x \), then \( \frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \)

Ex: \( f(g,h) = g^2 h \)

\[
\frac{\partial J}{\partial g} = \frac{\partial J}{\partial f} \cdot 2g(\cdot) h(\cdot) \\
\frac{\partial J}{\partial h} = \frac{\partial J}{\partial f} \cdot g^2(\cdot)
\]

save & reuse info \((g,h)\) from forward computation!
Backpropagation

Forward propagation

\[ z^1 = W^1 x \]
\[ z^2 = W^2 h^1 \]
\[ \cdots \]
\[ z^{L-1} = W^{L-1} h^{L-2} \]
\[ z^L = W^L h^{L-1} \]
\[ h^1 = \sigma(z^1) \]
\[ h^2 = \sigma(z^2) \]
\[ \cdots \]
\[ h^{L-1} = \sigma(z^{L-1}) \]
\[ \hat{y} = \sigma(z^L) \]

Loss function

\[ J = \mathcal{L}(y, \hat{y}) \text{ or } \sum_k \mathcal{L}(y_k, \hat{y}_k) \]
Backpropagation

\[ \frac{\partial J}{\partial w_{k,j}^L} = \]

\[ \frac{\partial J}{\partial w_{j,i}^{L-1}} = \]

\[ z^1 = W^1 x \quad h^1 = \sigma(z^1) \]
\[ z^2 = W^2 h^1 \quad h^2 = \sigma(z^2) \]
\[ \ldots \]
\[ z^{L-1} = W^{L-1} h^{L-2} \quad h^{L-1} = \sigma(z^{L-1}) \]
\[ z^L = W^L h^{L-1} \quad \hat{y} = \sigma(z^L) \]

\[ J = \mathcal{L}(y, \hat{y}) \text{ or } \sum_k \mathcal{L}(y_k, \hat{y}_k) \]
Backpropagation

\[
\frac{\partial J}{\partial w_{k,i}} = \frac{\partial J}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{k,i}} = \mathcal{L}'(y_k, \hat{y}_k) \cdot \sigma'(z_k) \cdot h_j^{L-1}
\]

\[
\frac{\partial J}{\partial w_{j,i}} = \sum_k \frac{\partial J}{\partial \hat{y}_k} \cdot \frac{\partial \hat{y}_k}{\partial z_k} \cdot \frac{\partial z_k}{\partial h_j^{L-1}} \cdot \frac{\partial h_j^{L-1}}{\partial z_j^{L-1}} \cdot \frac{\partial z_j^{L-1}}{\partial w_{j,i}} = \sum_k \mathcal{L}'(y_k, \hat{y}_k) \cdot \sigma'(z_k) \cdot w_{k,i} \cdot \sigma'(z_j^{L-1}) \cdot h_i^{L-2}
\]

\[
z^1 = W^1 x \quad h^1 = \sigma(z^1)
\]
\[
z^2 = W^2 h^1 \quad h^2 = \sigma(z^2)
\]
\[\vdots\]
\[
z^{L-1} = W^{L-1} h^{L-2} \quad h^{L-1} = \sigma(z^{L-1})
\]
\[
z^L = W^L h^{L-1} \quad \hat{y} = \sigma(z^L)
\]
\[
J = \mathcal{L}(y, \hat{y}) \text{ or } \sum_k \mathcal{L}(y_k, \hat{y}_k)
\]
Backpropagation

\[ \frac{\partial J}{\partial w_{j,i}^l} = \beta_j^l \cdot h_{i}^{l-1} \]

\[ x_i \text{ if } l = 1 \]

\[ \beta_j^l = \begin{cases} 
\mathcal{L}_j'(y_j, \hat{y}_j) \cdot \sigma'(z_j^l) & \text{if } l = L \\
\left( \sum_{k=1}^{n_{l+1}} \beta_{k}^{l+1} \cdot w_{k,j}^{l+1} \right) \cdot \sigma'(z_j^l) & \text{otherwise}
\end{cases} \]
Backpropagation

\[ \nabla_{W^l} J = (\beta^l)^T \cdot h^{l-1} \]

\[ \beta^l = \begin{cases} 
\mathcal{L}'(y, \hat{y}) \ast \sigma'(z^L) & \text{if } l = L \\
(\beta^{l+1} \cdot W^{l+1}) \ast \sigma'(z^l) & \text{otherwise} 
\end{cases} \]
Backpropagation for cross entropy loss

Cross entropy loss:
\[ \mathcal{L}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \]

\[ \beta^L = \mathcal{L}'(y, \hat{y}) \ast \sigma'(z^L) = ? \]

\[ \hat{y} = \sigma(z^L) \]
\[ \sigma'(z) = \sigma(z)(1 - \sigma(z)) \]
Forward and back propagation

- $x = h^0$
- $z^l = W^l \cdot h^{l-1}$
- $h^l = \sigma(z^l)$
- $\hat{y} = \sigma(z^L), \mathcal{L}(y, \hat{y})$

- $\beta^L = \mathcal{L}'(y, \hat{y}) \cdot \sigma'(z^L)$
- $\beta^l = (\beta^{l+1} \cdot W^{l+1}) \cdot \sigma'(z^l)$
- $\nabla_{W^l} J = (\beta^l)^T \cdot h^{l-1}$
Gradient checking

- Numerical estimation of gradients
- It is similar for partial derivatives

\[
\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\epsilon, \theta_2, \theta_3, ..., \theta_n) - J(\theta_1-\epsilon, \theta_2, \theta_3, ..., \theta_n)}{2\epsilon}
\]

\[
\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\epsilon, \theta_3, ..., \theta_n) - J(\theta_1, \theta_2-\epsilon, \theta_3, ..., \theta_n)}{2\epsilon}
\]

... 

\[
\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, ..., \theta_n+\epsilon) - J(\theta_1, \theta_2, \theta_3, ..., \theta_n-\epsilon)}{2\epsilon}
\]
Gradient checking

• Implement back propagation to compute DVec
• Implement numerical gradient checking to compute gradApprox
• Check they're basically the same (up to a few decimal places)
• Before using the code for learning turn off gradient checking
  • Why?
    • GradApprox is very computationally expensive
Training a neural network

- Pick a network architecture

- No. of input nodes: dimension of features $x$
- No. of output nodes: number of classes
- Reasonable default: 1 hidden layer, or if $>1$ hidden layer, have same no. of hidden nodes in every layer (usually the more the better)
Training a neural network

1. Randomly initialize weights to small values
2. Implement forward propagation to get $h_w(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute loss function $J(w)$
4. Implement backprop to compute partial derivatives $\frac{\partial J}{\partial w_{j,i}}$

for $i=1:m$

Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$
Training a neural network

5. Use gradient checking to compare $\frac{\partial J}{\partial w_{j,i}}$ computed using backpropagation vs. using numerical estimate of gradient of $J(w)$
   • Then disable gradient checking code

6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(w)$
Example: Regression, MCycle data

- Train NN model, 2 layer
  - 1 input feature => 1 input node
  - 10 hidden nodes
  - 1 target => 1 output node
  - Logistic sigmoid activation for hidden layer, linear for output layer

Data:

+ learned prediction f’n:

Responses of hidden nodes (= features of linear regression):
select out useful regions of “x”

(c) Alexander Ihler
Example: Classification, Iris data

• Train NN model, 2 layer
  • 2 input features => 2 input nodes
  • 10 hidden nodes
  • 3 classes => 3 output nodes (y = [0 0 1], etc.)
  • Logistic sigmoid activation functions

(c) Alexander Ihler
Summary

• Neural networks, multi-layer perceptrons

• Cascade of simple perceptrons
  • Each just a linear classifier
  • Hidden units used to create new features

• Together, general function approximators
  • Enough hidden units (features) = any function
  • Can create nonlinear classifiers
  • Also used for function approximation, regression, …

• Training via backprop
  • Gradient descent; logistic; apply chain rule. Building block view.

• Advanced: deep nets, conv nets, …