Lecture 18: Backpropagation

Mark Hasegawa-Johnson

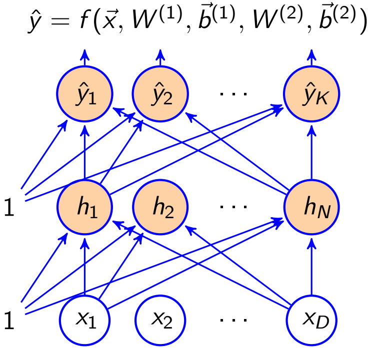
ECE 417: Multimedia Signal Processing, Fall 2021

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Two-Layer Feedforward Neural Network



$$\hat{y}_k = \xi_k^{(2)}$$

$$\xi_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(\xi_k^{(1)})$$

$$\xi_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

Review: Second Layer = Piece-Wise Approximation

The second layer of the network approximates $\hat{y}$ using a bias term $\vec{b}$, plus correction vectors $\vec{w}_j^{(2)}$, each scaled by its activation h_j:

$$\hat{y} = \vec{b}^{(2)} + \sum_j \vec{w}_j^{(2)} h_j$$

The activation, h_j, is a number between 0 and 1. For example, we could use the logistic sigmoid function:

$$h_k = \sigma\left(\xi_k^{(1)}\right) = \frac{1}{1+\exp(-\xi_k^{(1)})} \in (0,1)$$

The logistic sigmoid is a differentiable approximation to a unit step function.

Review: First Layer = A Series of Decisions

The first layer of the network decides whether or not to "turn on" each of the h_j's. It does this by comparing $\vec{x}$ to a series of linear threshold vectors:

$$h_k = \sigma\left(\bar{w}_k^{(1)}\vec{x}\right) \approx \begin{cases} 1 & \bar{w}_k^{(1)}\vec{x} > 0 \\ 0 & \bar{w}_k^{(1)}\vec{x} < 0 \end{cases}$$

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

How to train a neural network

1. Find a **training dataset** that contains n examples showing the desired output, $\vec{y}_i$, that the NN should compute in response to input vector $\vec{x}_i$:

$$\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$$

2. Randomly **initialize** $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
3. Perform **forward propagation**: find out what the neural net computes as $\hat{y}_i$ for each $\vec{x}_i$.
4. Define a **loss function** that measures how badly $\hat{y}$ differs from $\vec{y}$.
5. Perform **back propagation** to find the derivative of the loss w.r.t. $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
6. Perform **gradient descent** to improve $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
7. Repeat steps 3-6 until convergence.

Loss Function: How should y be "similar to" $\hat{y}$?

Minimum Mean Squared Error (MMSE)

$$W^*, b^* = \arg\min \mathcal{L} = \arg\min \frac{1}{2n}\sum_{i=1}^{n} \|\vec{y}_i - \hat{y}(\vec{x}_i)\|^2$$

MMSE Solution: $\hat{y} \rightarrow E\left[\vec{y}|\vec{x}\right]$

If the training samples $(\vec{x}_i, \vec{y}_i)$ are i.i.d., then

$$\lim_{n\to\infty} \mathcal{L} = \frac{1}{2}E\left[\|\vec{y} - \hat{y}\|^2\right]$$

which is minimized by

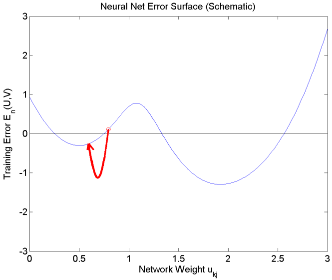
$$\hat{y}_{MMSE}(\vec{x}) = E\left[\vec{y}|\vec{x}\right]$$

Gradient Descent: How do we improve W and b?

Given some initial neural net parameter (called u_{kj} in this figure), we want to find a better value of the same parameter. We do that using gradient descent:

$$u_{kj} \leftarrow u_{kj} - \eta \frac{d\mathcal{L}}{du_{kj}},$$

where η is a learning rate (some small constant, e.g., $\eta = 0.02$ or so).



Gradient Descent = Local Optimization

Given an initial W, b, find new values of W, b with lower error.

$$w_{kj}^{(1)} \leftarrow w_{kj}^{(1)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(1)}}$$
$$w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(2)}}$$

η =Learning Rate

- If η too large, gradient descent won't converge. If too small, convergence is slow.
- Second-order methods like Newton's method, L-BFGS and Adam choose an optimal η at each step, so they're MUCH faster.

Outline

1 Review: Neural Network
2 Learning the Parameters of a Neural Network
3 Definitions of Gradient, Partial Derivative, and Flow Graph
4 Back-Propagation
5 Computing the Weight Derivatives
6 Backprop Example: Semicircle $\rightarrow$ Parabola
7 Binary Cross Entropy Loss
8 Multinomial Classifier: Cross-Entropy Loss
9 Summary

Digression: What is a Gradient?

The gradient of a scalar function, $f(\vec{w})$, with respect to the vector $\vec{w}$ can be usefully defined as

$$\nabla_{\vec{w}} f = \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \vdots \\ \frac{\partial f}{\partial w_N} \end{bmatrix}, \quad \text{where} \quad \vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix}.$$

Here the **partial derivative** sign, ∂, means "the derivative while all other elements of $\vec{w}$ are held constant."

Digression: Total Derivative vs. Partial Derivative

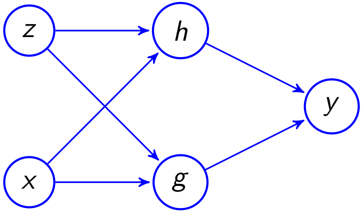
- The notation $\frac{d\mathcal{L}}{dw_{kj}^{(2)}}$ means "the total derivative of $\mathcal{L}$ with respect to $w_{kj}^{(2)}$." It implies that we have to add up several different ways in which $\mathcal{L}$ depends on $w_{kj}^{(2)}$, for example,

$$\frac{d\mathcal{L}}{dw_{kj}^{(2)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{d\hat{y}_{ki}}\right)\left(\frac{\partial \hat{y}_{ki}}{\partial w_{kj}^{(2)}}\right)$$

- The notation $\frac{\partial \mathcal{L}}{\partial \hat{y}_{ki}}$ means "partial derivative." It means "hold other variables constant while calculating this derivative."
- The next obvious question to ask is: **which** other variables should I hold constant?

Flow Graph

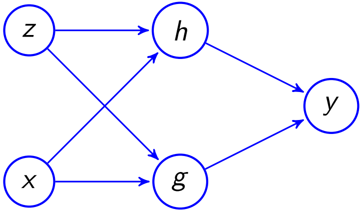
A signal flow graph shows the flow of computations in a system. For example, the following graph shows that y is a function of $\{g, h\}$, g is a function of $\{x, z\}$, and h is a function of $\{x, z\}$:



The Total Derivative Rule

The **total derivative rule** says that the derivative of the output with respect to any one input can be computed as the sum of partial times total, summed across all paths from input to output:

$$\frac{\partial y}{\partial x} = \left(\frac{\partial y}{\partial g}\right)\left(\frac{dg}{dx}\right) + \left(\frac{\partial y}{\partial h}\right)\left(\frac{dh}{dx}\right)$$



Partial with respect to What?

The difference between **partial derivative** and **total derivative** only makes sense in light of the total derivative rule. For example, in this equation

$$\frac{\partial y}{\partial x} = \left(\frac{\partial y}{\partial g}\right)\left(\frac{dg}{dx}\right) + \left(\frac{\partial y}{\partial h}\right)\left(\frac{dh}{dx}\right)$$

- The symbol $\frac{\partial y}{\partial g}$ means "the derivative of y with respect to g while holding h constant."
- The symbol $\frac{dg}{dx}$ means "the derivative of g with respect to x, **without** holding h constant."
- For today's lecture, the difference between partial and total derivative doesn't matter much, because it doesn't matter whether you hold h constant or not. When we get into recurrent neural networks, later, such things will start to matter, so we'll discuss this point again at that time.

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

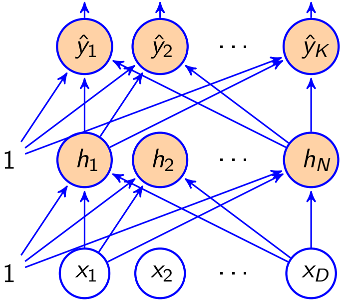
Computing the Gradient: Notation

- $\vec{x}_i = [x_{1i}, \ldots, x_{Di}]^T$ is the i^{th} input vector.
- $\vec{y}_i = [y_{1i}, \ldots, y_{Ki}]^T$ is the i^{th} target vector (desired output).
- $\hat{y}_i = [\hat{y}_{1i}, \ldots, \hat{y}_{Ki}]^T$ is the i^{th} hypothesis vector (computed output).
- $\vec{\xi}_i^{(l)} = [\xi_{1i}^{(l)}, \ldots, \xi_{Ni}^{(l)}]^T$ is the excitation vector after the l^{th} layer, in response to the i^{th} input.
- $\vec{h}_i = [h_{1i}, \ldots, h_{Ni}]^T$ is the hidden nodes activation vector in response to the i^{th} input. (No superscript necessary if there's only one hidden layer).
- The weight matrix for the l^{th} layer is

$$W^{(l)} = \left[\vec{w}_1^{(l)}, \ldots, \vec{w}_j^{(l)}, \ldots\right] = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1j}^{(l)} & \cdots \\ \vdots & \ddots & \vdots & \ddots \\ w_{k1}^{(l)} & \cdots & w_{kj}^{(l)} & \cdots \\ \vdots & \ddots & \vdots & \ddots \end{bmatrix}$$

Two-Layer Feedforward Neural Network

$$\hat{y} = f(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = \xi_k^{(2)}$$

$$\xi_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

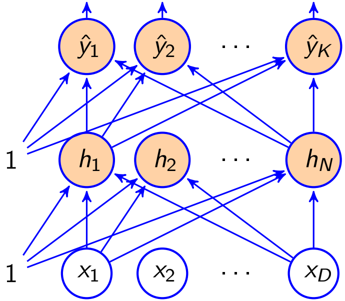
$$h_k = \sigma(\xi_k^{(1)})$$

$$\xi_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

Two-Layer Feedforward Neural Network

$$\hat{y} = f(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y} = \vec{\xi}^{(2)}$$

$$\vec{\xi}^{(2)} = \vec{b}^{(2)} + W^{(2)}\vec{h}$$

$$\vec{h} = \sigma(\vec{\xi}^{(1)})$$

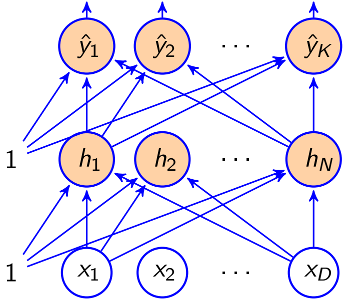
$$\vec{\xi}^{(1)} = \vec{b}^{(1)} + W^{(1)}\vec{x}$$

$\vec{x}$ is the input vector

Back-Propagation

Back-propagation just works backward through this network, calculating the derivative of $\mathcal{L}$ with respect to $\hat{y}$, then $\vec{\xi}^{(2)}$, then $\vec{h}$, then $\vec{\xi}^{(1)}$:

$$\hat{y} = f(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y} = \vec{\xi}^{(2)}$$

$$\vec{\xi}^{(2)} = \vec{b}^{(2)} + W^{(2)}\vec{h}$$

$$\vec{h} = \sigma(\vec{\xi}^{(1)})$$

$$\vec{\xi}^{(1)} = \vec{b}^{(1)} + W^{(1)}\vec{x}$$

$\vec{x}$ is the input vector

Back-Propagation: Derivative w.r.t Output-Layer Activations

Remember that the loss function is mean-squared error:

$$\mathcal{L} = \frac{1}{2n}\sum_{i=1}^{n} \|\vec{y}_i - \hat{y}_i\|^2$$

$$= \frac{1}{2n}\sum_{i=1}^{n}\sum_{k=1}^{K}(\vec{y}_{i,k} - \hat{y}_{i,k})^2$$

So:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_{i,k}} = \frac{1}{n}(\hat{y}_{i,k} - y_{i,k})$$

Back-Propagation: Derivative w.r.t Output-Layer Excitations

In our network, the output layer is linear, so

$$\hat{y}_{i,k} = \xi^{(2)}_{i,k}$$

Therefore:

$$\frac{\partial \mathcal{L}}{\partial \xi^{(2)}_{i,k}} = \frac{1}{n}\left(\hat{y}_{i,k} - y_{i,k}\right)$$

The back-prop deltas: derivative w.r.t. excitations

In order to keep going systematically, it's useful to define a **back-propagation partial derivative**:

$$\delta_{i,k}^{(2)} \equiv \frac{\partial \mathcal{L}}{\partial \xi_{i,k}^{(2)}}$$

These deltas are sometimes called the **excitation gradients**:

$$\vec{\delta}_i^{(2)} = \nabla_{\vec{\xi}_i^{(2)}} \mathcal{L}$$

Back-Propagation: Derivative w.r.t Hidden-Layer Activations

The mapping from **hidden-layer activations** to **output-layer excitations** is

$$\xi_{i,k}^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{k,j}^{(2)} h_{i,j}$$

Notice that the loss, $\mathcal{L}$, depends on $h_{i,j}$ via all of the different paths through all of the different $\xi_{i,k}^{(2)}$ output excitations. The **total derivative rule** therefore gives us:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial h_{i,j}} &= \sum_{k=1}^{K} \left(\frac{\partial \mathcal{L}}{\partial \xi_{i,k}^{(2)}}\right)\left(\frac{\partial \xi_{i,k}^{(2)}}{\partial h_{i,j}}\right)\\ &= \sum_{k=1}^{K} \delta_{i,k}^{(2)} w_{k,j}^{(2)}\end{aligned}$$

Back-Propagation: Derivative w.r.t Hidden-Layer Activations

The mapping from **hidden-layer activations** to **output-layer excitations** is

$$\vec{\xi}_i^{(2)} = \vec{b}^{(2)} + W^{(2)}\vec{h}_i$$

Notice that the loss, $\mathcal{L}$, depends on $h_{j,i}$ via all of the different paths through all of the different $\xi_{k,i}^{(2)}$ output excitations. The **total derivative rule** therefore gives us the following surprising rule:

$$\begin{aligned}\nabla_{\vec{h}_i}\mathcal{L} &= W^{(2),T}\nabla_{\vec{\xi}_i^{(2)}}\mathcal{L} \\ &= W^{(2),T}\vec{\delta}_i^{(2)}\end{aligned}$$

Back-Propagation: Derivative w.r.t Hidden-Layer Excitations

The mapping from **hidden-layer excitations** to **hidden-layer activations** is much simpler:

$$h_{i,j} = \sigma(\xi_{i,j}^{(1)})$$

So

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \xi_{i,j}^{(1)}} &= \left(\frac{\partial \mathcal{L}}{\partial h_{i,j}}\right)\left(\frac{\partial h_{i,j}}{\partial \xi_{i,j}^{(1)}}\right) \\
&= \left(\sum_{k=1}^{K} w_{k,j}^{(2)}\delta_{i,j}^{(2)}\right)\left(\dot{\sigma}\left(\xi_{i,j}^{(1)}\right)\right)
\end{aligned}$$

where $\dot{\sigma}(\cdot)$, the derivative of the scalar nonlinearity $\sigma(\cdot)$, is something you can calculate in advance, and store.

Back-Propagation: Derivative w.r.t Hidden-Layer Excitations

We can define **excitation gradients** at the hidden layer to be

$$\vec{\delta}_i^{(1)} = \nabla_{\vec{\xi}_i^{(1)}} \mathcal{L}$$

Putting together the last two steps, we have that

$$\vec{\delta}_i^{(1)} = \left(W^{(2),T} \vec{\delta}_i^{(2)} \right) \odot \dot{\sigma} \left(\xi_i^{(1)} \right)$$

where $\odot$ means Hadamard product (element-wise multiplication of the two vectors).

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle → Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Computing the Derivative

OK, let's compute the derivative of $\mathcal{L}$ with respect to the $W^{(2)}$ matrix. Remember that $W^{(2)}$ enters the neural net computation as $\xi_{ki}^{(2)} = \sum_k w_{kj}^{(2)} h_{ji}$. So. . .

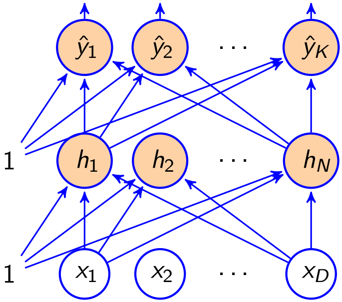
$$\frac{d\mathcal{L}}{dw_{k,j}^{(2)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{d\xi_{i,k}^{(2)}}\right)\left(\frac{\partial \xi_{i,k}^{(2)}}{\partial w_{k,j}^{(2)}}\right)$$

$$= \sum_{i=1}^{n} \delta_{k,i}^{(2)} h_{i,j}$$

If we define the gradient of a matrix as a matrix of partial derivatives, we can write:

$$\nabla_{W^{(2)}}\mathcal{L} = \sum_{i=1}^{n} \vec{\delta}_i^{(2)} \vec{h}_i^T = \sum_{i=1}^{n} \begin{bmatrix} \vdots \\ \frac{\partial \mathcal{L}}{\partial \xi_{i,k}^{(2)}} \\ \vdots \end{bmatrix} [\cdots, h_{i,j}, \cdots]$$

$$\hat{y} = f(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = \xi_k^{(2)}$$

$$\xi_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(\xi_k^{(1)})$$

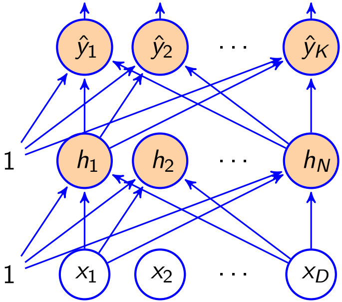
$$\xi_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

Back-Propagating to the First Layer

$$\frac{d\mathcal{L}}{dw_{k,j}^{(1)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{d\xi_{i,k}^{(1)}}\right)\left(\frac{\partial \xi_{i,k}^{(1)}}{\partial w_{k,j}^{(1)}}\right) = \sum_{i=1}^{n} \delta_{i,k}^{(1)} x_{i,j}$$

$$\hat{y} = f(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = \xi_k^{(2)}$$

$$\xi_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(\xi_k^{(1)})$$

$$\xi_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

Back-Propagating to the First Layer

$$\nabla_{W^{(1)}} \mathcal{L} = \sum_{i=1}^{n} \vec{\delta}_i^{(1)} \vec{x}_i^T$$

The Back-Propagation Algorithm

$$W^{(2)} \leftarrow W^{(2)} - \eta \nabla_{W^{(2)}} \mathcal{L}, \qquad W^{(1)} \leftarrow W^{(1)} - \eta \nabla_{W^{(1)}} \mathcal{L}$$

$$\nabla_{W^{(2)}} \mathcal{L} = \sum_{i=1}^{n} \vec{\delta}_i^{(2)} \vec{h}_i^T, \qquad \nabla_{W^{(1)}} \mathcal{L} = \sum_{i=1}^{n} \vec{\delta}_i^{(1)} \vec{x}_i^T$$

$$\delta_{i,k}^{(2)} = \frac{1}{n}(\hat{y}_{ki} - y_{ki}), \qquad \delta_{i,k}^{(1)} = \sum_{\ell=1}^{K} \delta_{i,\ell}^{(2)} w_{\ell,k}^{(2)} \dot{\sigma}(\xi_{i,k}^{(1)})$$

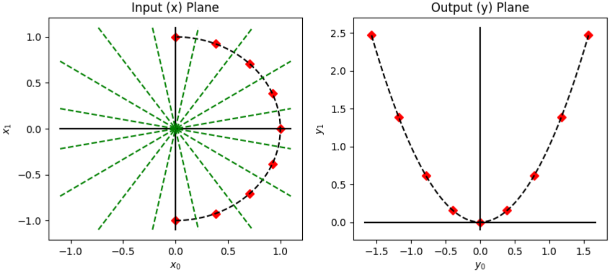
$$\vec{\delta}_i^{(2)} = \frac{1}{n}\left(\hat{\vec{y}}_i - \vec{y}_i\right), \qquad \vec{\delta}_i^{(1)} = \left(W^{(2),T} \vec{\delta}_i^{(2)}\right) \odot \dot{\sigma}(\vec{\xi}_i^{(1)})$$

...where $\odot$ means element-wise multiplication of two vectors; $\dot{\sigma}(\vec{\xi})$ is the element-wise derivative of $\sigma(\vec{\xi})$.

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Backprop Example: Semicircle $\rightarrow$ Parabola

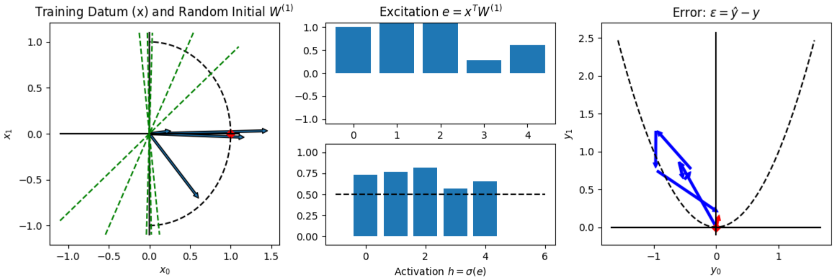


Remember, we are going to try to approximate this using:

$$\hat{y} = \vec{b} + \sum_j \vec{w}_j^{(2)} \sigma\left(\bar{w}_k^{(1)} \vec{x}\right)$$

Randomly Initialized Weights

Here's what we get if we randomly initialize $\bar{w}_k^{(1)}$, $\vec{b}$, and $\vec{w}_j^{(2)}$. The red vector on the right is the estimation error for this training token, $\vec{\delta}^{(2)} = \hat{y} - \vec{y}$. It's huge!



Back-Prop: Layer 2

Remember

$$W^{(2)} \leftarrow W^{(2)} - \eta \nabla_{W^{(2)}} \mathcal{L} = W^{(2)} - \eta \sum_{i=1}^{n} \vec{\delta}_i^{(2)} \vec{h}_i^T$$

$$= W^{(2)} - \frac{\eta}{n} \sum_{i=1}^{n} (\hat{y}_i - \vec{y}_i) \vec{h}_i^T$$

Thinking in terms of the columns of $W^{(2)}$, we have

$$\vec{w}_j^{(2)} \leftarrow \vec{w}_j^{(2)} - \frac{\eta}{n} \sum_{i=1}^{n} (\hat{y}_i - \vec{y}_i) h_{ji}$$

So, in words, layer-2 backprop means

- Each column, $\vec{w}_j^{(2)}$, gets updated in the direction $\vec{y} - \hat{y}$.
- The update for the j^{th} column, in response to the i^{th} training token, is scaled by its activation h_{ji}.

Back-Prop: Layer 1

Remember

$$W^{(1)} \leftarrow W^{(1)} - \eta \nabla_{W^{(1)}} \mathcal{L} = W^{(1)} - \eta \sum_{i=1}^{n} \vec{\delta}_i^{(1)} \vec{x}_i^T$$

$$= W^{(1)} - \eta \sum_{i=1}^{n} \left(\dot{\sigma}(\vec{\xi}_i^{(1)}) \odot W^{(2),T} \vec{\delta}_i^{(2)} \right) \vec{x}_i^T$$

Thinking in terms of the rows of $W^{(1)}$, we have

$$\bar{w}_k^{(1)} \leftarrow \bar{w}_k^{(1)} - \eta \sum_{i=1}^{n} \delta_{ki}^{(1)} \vec{x}_i^T$$

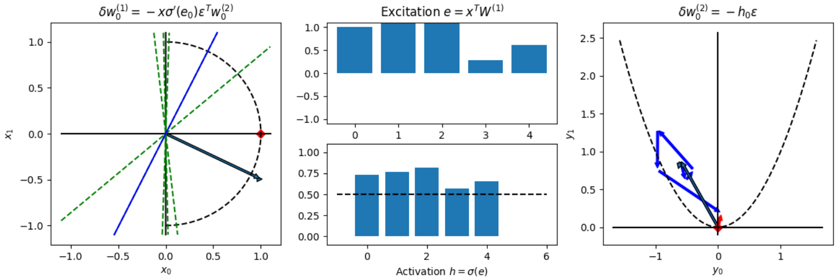
In words, layer-1 backprop means

- Each row, $\bar{w}_k^{(1)}$, gets updated in the direction $-\vec{x}$.
- The update for the k^{th} row, in response to the i^{th} training token, is scaled by its back-propagated error term $\delta_{ki}^{(1)}$.

Back-Prop Example: Semicircle $\rightarrow$ Parabola

For each column $\vec{w}_j^{(2)}$ and the corresponding row $\bar{w}_k^{(1)}$,

$$\vec{w}_j^{(2)} \leftarrow \vec{w}_j^{(2)} - \frac{\eta}{n}\sum_{i=1}^{n}\left(\hat{y}_i - \vec{y}_i\right) h_{ji}, \qquad \bar{w}_k^{(1)} \leftarrow \bar{w}_k^{(1)} - \eta\sum_{i=1}^{n}\delta_{ki}^{(1)}\vec{x}_i^T$$



Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Review: MSE

Until now, we've assumed that the loss function is MSE:

$$\mathcal{L} = \frac{1}{2n}\sum_{i=1}^{n} \left\|\vec{y}_i - \hat{y}(\vec{x}_i)\right\|^2$$

- MSE makes sense if $\vec{y}$ and $\hat{y}$ are both real-valued vectors, and we want to compute $\hat{y}_{MMSE}(\vec{x}) = E\left[\vec{y}|\vec{x}\right]$. But what if $\hat{y}$ and $\vec{y}$ are discrete-valued (i.e., classifiers?)
- Surprise: MSE works surprisingly well, even with discrete $\vec{y}$!
- But a different metric, binary cross-entropy (BCE) works slightly better.

MSE with a binary target vector

- Suppose y is just a scalar binary classifier label, $y \in \{0,1\}$ (for example: "is it a dog or a cat?")
- Suppose that the input vector, $\vec{x}$, is not quite enough information to tell us what y should be. Instead, $\vec{x}$ only tells us the probability of $y=1$:

$$y = \begin{cases} 1 & \text{with probability } p_{Y|\vec{X}}(1|\vec{x}) \\ 0 & \text{with probability } p_{Y|\vec{X}}(0|\vec{x}) \end{cases}$$

- In the limit as $n \to \infty$, assuming that the gradient descent finds the global optimum, the MMSE solution gives us:

$$\begin{aligned} \hat{y}(\vec{x}) &\to_{n\to\infty} E\left[y|\vec{x}\right] \\ &= \left(1 \times p_{Y|\vec{X}}(1|\vec{x})\right) + \left(0 \times p_{Y|\vec{X}}(0|\vec{x})\right) \\ &= p_{Y|\vec{X}}(1|\vec{x}) \end{aligned}$$

Pros and Cons of MMSE for Binary Classifiers

- **Pro:** In the limit as $n \to \infty$, the global optimum is $\hat{y}(\vec{x}) \to p_{Y|\vec{X}}(1|\vec{x})$.
- **Con:** The sigmoid nonlinearity is hard to train using MMSE. Remember the vanishing gradient problem: $\sigma'(wx) \to 0$ as $w \to \infty$, so after a few epochs of training, the neural net just stops learning.
- **Solution:** Can we devise a different loss function (not MMSE) that will give us the same solution ($\hat{y}(\vec{x}) \to p_{Y|\vec{X}}(1|\vec{x})$), but without suffering from the vanishing gradient problem?

Binary Cross Entropy

Suppose we treat the neural net output as a noisy estimator, $\hat{p}_{Y|\vec{X}}(y|\vec{x})$, of the unknown true pmf $p_{Y|\vec{X}}(y|\vec{x})$:

$$\hat{y}_i = \hat{p}_{Y|\vec{X}}(1|\vec{x}),$$

so that

$$\hat{p}_{Y|\vec{X}}(y_i|\vec{x}_i) = \begin{cases} \hat{y}_i & y_i = 1 \\ 1-\hat{y}_i & y_i = 0 \end{cases}$$

The binary cross-entropy loss is the negative log probability of the training data, assuming i.i.d. training examples:

$$\begin{aligned} \mathcal{L}_{BCE} &= -\frac{1}{n}\sum_{i=1}^{n} \ln \hat{p}_{Y|\vec{X}}(y_i|\vec{x}_i) \\ &= -\frac{1}{n}\sum_{i=1}^{n} y_i\left(\ln \hat{y}_i\right) + (1-y_i)\left(\ln(1-\hat{y}_i)\right) \end{aligned}$$

The Derivative of BCE

BCE is useful because it has the same solution as MSE, without allowing the sigmoid to suffer from vanishing gradients. Suppose $\hat{y}_i = \sigma(wh_i)$.

$$\begin{aligned}
\nabla_w \mathcal{L} &= -\frac{1}{n}\left(\sum_{i:y_i=1} \nabla_w \ln \sigma(wh_i) + \sum_{i:y_i=0} \nabla_w \ln(1-\sigma(wh_i))\right) \\
&= -\frac{1}{n}\left(\sum_{i:y_i=1} \frac{\nabla_w \sigma(wh_i)}{\sigma(wh_i)} + \sum_{i:y_i=0} \frac{\nabla_w (1-\sigma(wh_i))}{1-\sigma(wh_i)}\right) \\
&= -\frac{1}{n}\left(\sum_{i:y_i=1} \frac{\hat{y}_i(1-\hat{y}_i)h_i}{\hat{y}_i} + \sum_{i:y_i=0} \frac{-\hat{y}_i(1-\hat{y}_i)h_i}{1-\hat{y}_i}\right) \\
&= -\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)\, h_i
\end{aligned}$$

Why Cross-Entropy is Useful for Machine Learning

Binary cross-entropy is useful for machine learning because:

1. **Just like MSE, it estimates the true class probability:** in the limit as $n \rightarrow \infty$, $\nabla_W \mathcal{L} \rightarrow E\left[(Y - \hat{Y})H\right]$, which is zero only if
$$\hat{Y} = E\left[Y|\vec{X}\right] = p_{Y|\vec{X}}(1|\vec{x})$$
2. **Unlike MSE, it does not suffer from the vanishing gradient problem of the sigmoid.**

Unlike MSE, BCE does not suffer from the vanishing gradient problem of the sigmoid.

The vanishing gradient problem was caused by $\sigma' = \sigma(1-\sigma)$, which goes to zero when its input is either plus or minus infinity.

- If $y_i = 1$, then differentiating $\ln\sigma$ cancels the σ term in the numerator, leaving only the $(1-\sigma)$ term, which is large if and only if the neural net is wrong.
- If $y_i = 0$, then differentiating $\ln(1-\sigma)$ cancels the $(1-\sigma)$ term in the numerator, leaving only the σ term, which is large if and only if the neural net is wrong.

So binary cross-entropy ignores training tokens only if the neural net guesses them right. If it guesses wrong, then back-propagation happens.

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Multinomial Classifier

Suppose, instead of just a 2-class classifier, we want the neural network to classify $\vec{x}$ as being one of K different classes. There are many ways to encode this, but one of the best is

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_K \end{bmatrix}, \quad y_k = \begin{cases} 1 & k = k^* \quad (k \text{ is the correct class}) \\ 0 & \text{otherwise} \end{cases}$$

A vector $\vec{y}$ like this is called a "one-hot vector," because it is a binary vector in which only one of the elements is nonzero ("hot"). This is useful because minimizing the MSE loss gives:

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_K \end{bmatrix} = \begin{bmatrix} \hat{p}_{Y_1|\vec{X}}(1|\vec{x}) \\ \hat{p}_{Y_2|\vec{X}}(1|\vec{x}) \\ \vdots \\ \hat{p}_{Y_K|\vec{X}}(1|\vec{x}) \end{bmatrix},$$

One-hot vectors and Cross-entropy loss

The cross-entropy loss, for a training database coded with one-hot vectors, is

$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ki} \ln \hat{y}_{ki}$$

This is useful because:

1. **Like MSE, Cross-Entropy has an asymptotic global optimum at:** $\hat{y}_k \rightarrow p_{Y_k|\vec{X}}(1|\vec{x})$.
2. **Unlike MSE, Cross-Entropy with a softmax nonlinearity suffers no vanishing gradient problem.**

Softmax Nonlinearity

The multinomial cross-entropy loss is only well-defined if $0 < \hat{y}_{ki} < 1$, and it is only well-interpretable if $\sum_k \hat{y}_{ki} = 1$. We can guarantee these two properties by setting

$$\hat{y}_k = \underset{k}{\text{softmax}}\left(W\vec{h}\right) = \frac{\exp(\bar{w}_k\vec{h})}{\sum_{\ell=1}^{K}\exp(\bar{w}_\ell\vec{h})},$$

where $\bar{w}_k$ is the k^{th} row of the W matrix.

Sigmoid is a special case of Softmax!

$$\text{softmax}_k\left(W\vec{h}\right) = \frac{\exp(\bar{w}_k\vec{h})}{\sum_{\ell=1}^{K}\exp(\bar{w}_\ell\vec{h})}.$$

Notice that, in the 2-class case, the softmax is just exactly a logistic sigmoid function:

$$\text{softmax}_1(W\vec{h}) = \frac{e^{\bar{w}_1\vec{h}}}{e^{\bar{w}_1\vec{h}} + e^{\bar{w}_2\vec{h}}} = \frac{1}{1 + e^{-(\bar{w}_1-\bar{w}_2)\vec{h}}} = \sigma\left((\bar{w}_1 - \bar{w}_2)\vec{h}\right)$$

so everything that you've already learned about the sigmoid applies equally well here.

Outline

1. Review: Neural Network
2. Learning the Parameters of a Neural Network
3. Definitions of Gradient, Partial Derivative, and Flow Graph
4. Back-Propagation
5. Computing the Weight Derivatives
6. Backprop Example: Semicircle $\rightarrow$ Parabola
7. Binary Cross Entropy Loss
8. Multinomial Classifier: Cross-Entropy Loss
9. Summary

Error Metrics Summarized

- Training is done using gradient descent.
- "Back-propagation" is the process of using the chain rule of differentiation in order to find the derivative of the loss with respect to each of the learnable weights and biases of the network.
- For a **regression** problem, use MSE to achieve $\hat{y} \rightarrow E\left[\vec{y}|\vec{x}\right]$.
- For a **binary classifier** with a sigmoid output, BCE loss gives you the MSE result without the vanishing gradient problem.
- For a **multi-class classifier** with a softmax output, CE loss gives you the MSE result without the vanishing gradient problem.