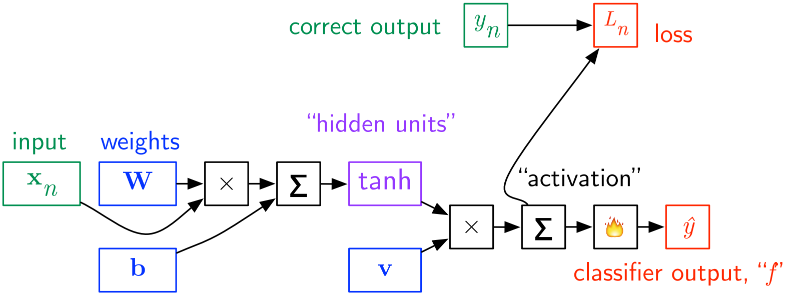
Machine Learning (CSE 446): Backpropagation

Sham M Kakade

University of Washington
cse446-staff@cs.washington.edu

Neuron-Inspired Classifiers



Two-Layer Neural Network

$$
\begin{aligned}
f(\mathbf{x}) &= \mathsf{sign}\left(\sum_{h=1}^{H} v_h \cdot \tanh\left(\mathbf{w}_h \cdot \mathbf{x} + b_h\right)\right) \\
&= \mathsf{sign}\left(\mathbf{v} \cdot \tanh\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)\right)
\end{aligned}
$$

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

- If we choose a differentiable loss, then the 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 *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.

$$\begin{aligned}
\frac{\partial L_n}{\partial a} &= \sum_{b:a\to b} \frac{\partial L_n}{\partial b} \cdot \frac{\partial b}{\partial a} \\
\bar{a} &= \sum_{b:a\to b} \bar{b} \cdot \frac{\partial b}{\partial a} \\
&= \sum_{b:a\to b} \bar{b} \cdot \begin{cases} 1 & \text{if } b = a + c \text{ for some } c \\ c & \text{if } b = a \cdot c \text{ for some } c \\ 1 - b^2 & \text{if } b = \tanh(a) \end{cases}
\end{aligned}$$

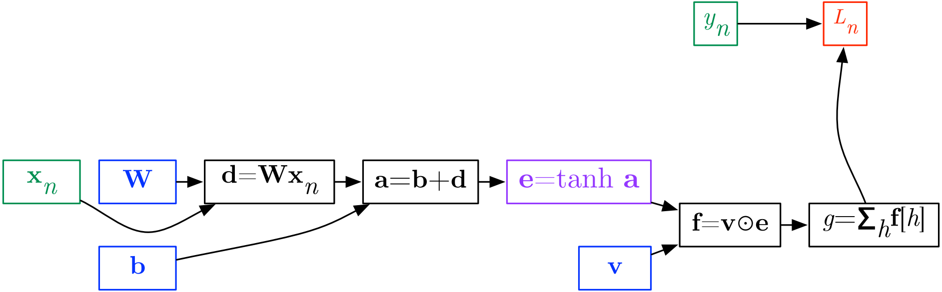
Backpropagation with Vectors

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

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

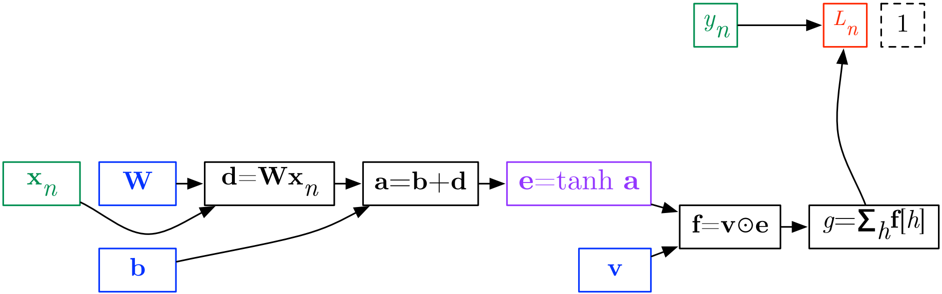
$$\begin{aligned} \bar{\mathbf{a}} &= \sum_{\mathbf{b}:\mathbf{a}\to\mathbf{b}} \sum_{i=1}^{|\mathbf{b}|} \bar{\mathbf{b}}[i] \cdot \frac{\partial \mathbf{b}[i]}{\partial \mathbf{a}} \\ &= \sum_{\mathbf{b}:\mathbf{a}\to\mathbf{b}} \begin{cases} \bar{\mathbf{b}} & \text{if } \mathbf{b} = \mathbf{a} + \mathbf{c} \text{ for some } \mathbf{c} \\ \bar{\mathbf{b}} \odot \mathbf{c} & \text{if } \mathbf{b} = \mathbf{a} \odot \mathbf{c} \text{ for some } \mathbf{c} \\ \bar{\mathbf{b}} \odot (\mathbf{1} - \mathbf{b} \odot \mathbf{b}) & \text{if } \mathbf{b} = \tanh(\mathbf{a}) \end{cases} \end{aligned}$$

Backpropagation, Illustrated



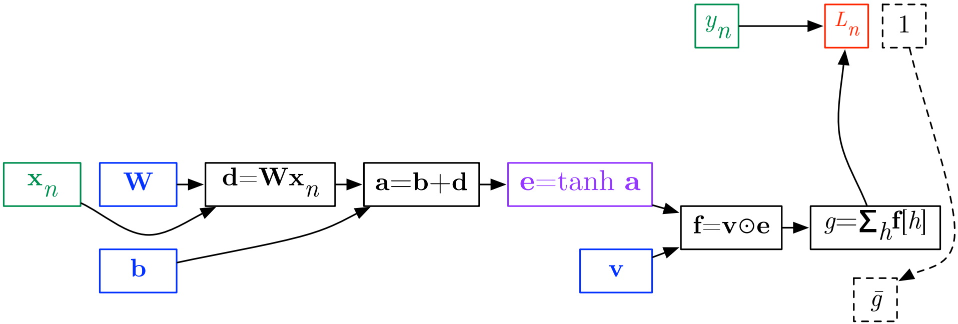
Intermediate nodes are de-anonymized, to make notation easier.

Backpropagation, Illustrated



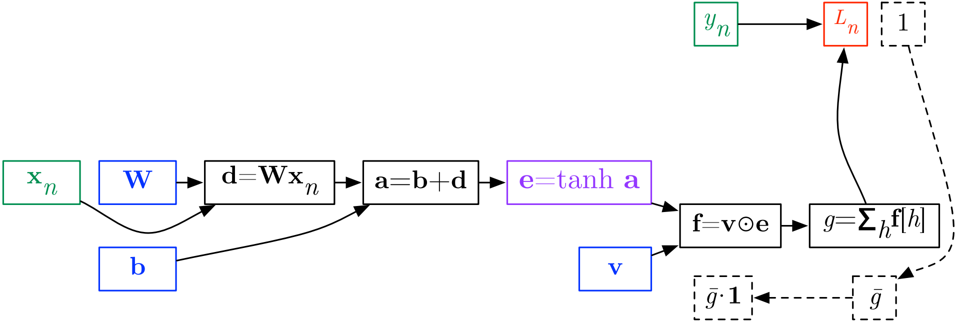
$$\frac{\partial L_n}{\partial L_n} = 1$$

Backpropagation, Illustrated



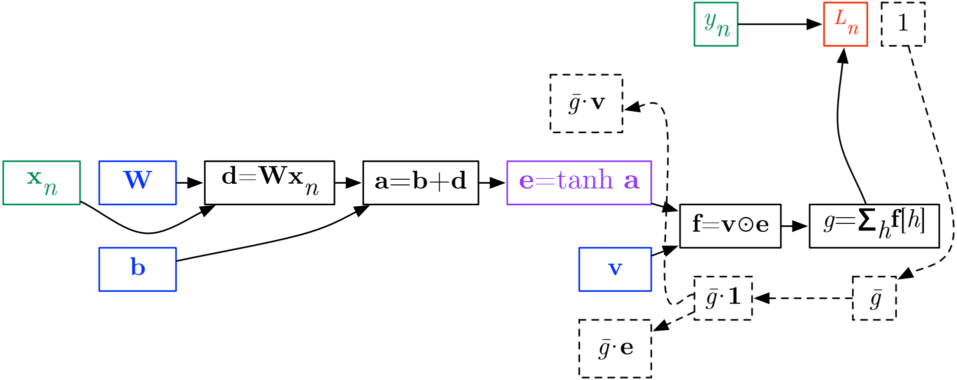
The form of $\bar{g}$ will be loss-function specific (e.g., $-2(y_n - g)$ for squared loss).

Backpropagation, Illustrated



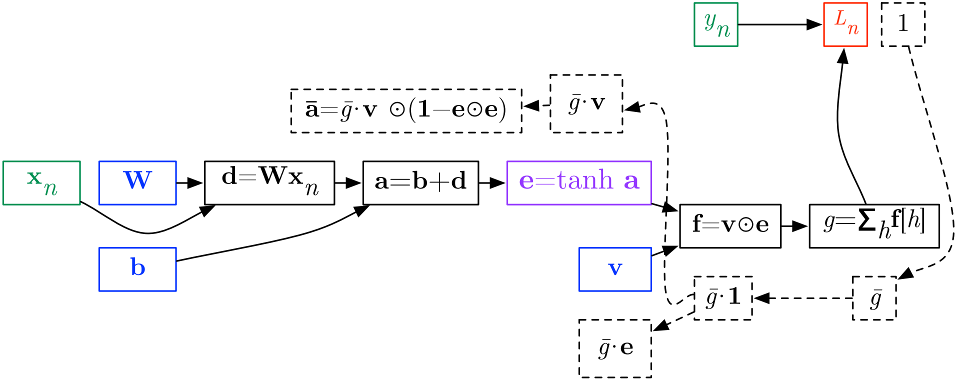
Sum.

Backpropagation, Illustrated



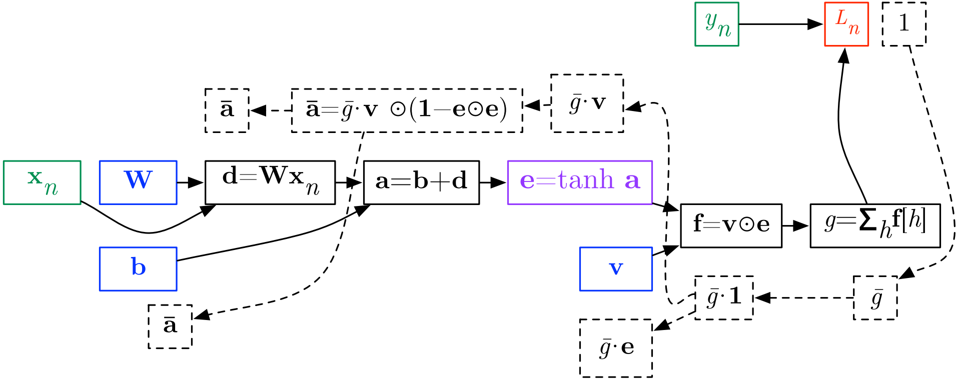
Product.

Backpropagation, Illustrated



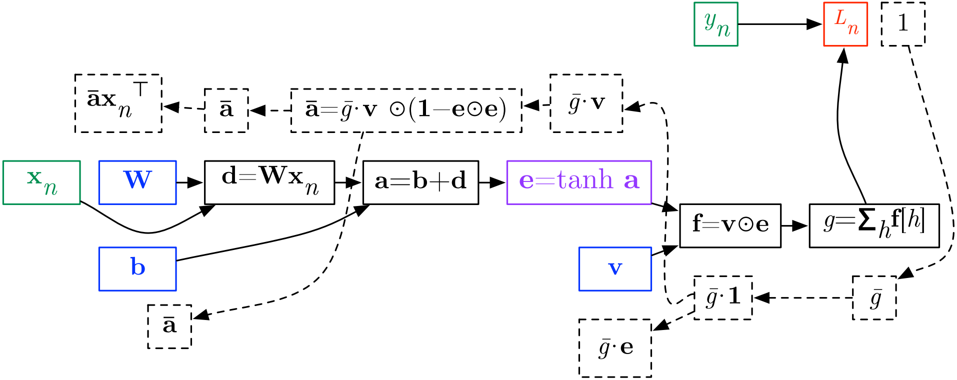
Hyperbolic tangent.

Backpropagation, Illustrated



Sum.

Backpropagation, Illustrated



Product.

Practical Notes

- Don't initalize all parameters to zero; add some random noise.

Practical Notes

- Don't initalize 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 initalize 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 initalize 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 initalize 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 initalize 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 initalize 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 initalize 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 initalize 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 initalize 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?

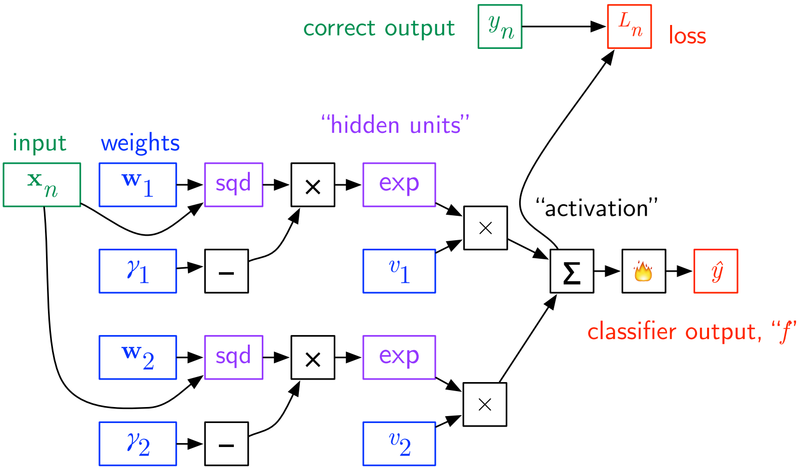
Practical Notes

- Don't initalize 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, $\text{sqd}(\mathbf{x}, \mathbf{w}) = \|\mathbf{x} - \mathbf{w}\|_2^2$.

Radial Basis Function Networks

Generalizing to H hidden units:

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

Each hidden unit is like a little "bump" in data space. $\mathbf{w}_h$ is the position of the bump, and γ_h is inversely proportional to its width.

A Gentle Reading on Backpropagation

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