CS 474/574 Machine Learning
6. Neural Networks

Prof. Dr. Forrest Sheng Bao
Dept. of Computer Science
Iowa State University
Ames, IA, USA

October 13, 2022

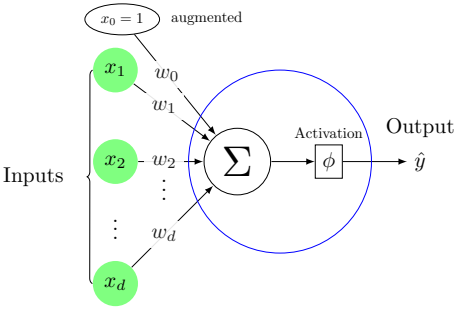
To compile this file, be sure you first compile all `.tex` files under `figs` folder into PDF.

`one_hidden_layer_step{1,2,3}.pdf` are produced from `one_hidden_layer.tex` by manually commenting different parts. So the step pdfs are in the repo.

Why artificial neural networks (ANNs) work

- Supervised or Reinforcement learning is about function fitting.
- To get the function, the analytical form sometimes doesn't matter.
- As long as we can mimic/fit it accurately enough, it's good.
- An ANN is a magical structure that can mimic any function [Fig. 5.3, Bishop book], if the ANN is "complex" enough. – Known as "Universal Approximation."

One neuron/perceptron



- A **neuron** (also called a **perceptron**) connects its inputs and output as follows:

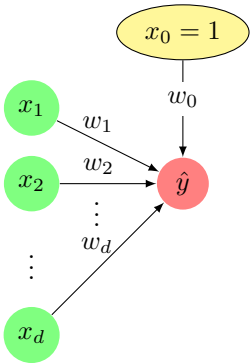
$$\hat{y} = \phi(\mathbf{w}^T\mathbf{x}) = \phi(w_0x_0 + w_1x_1 + w_2x_2 + w_dx_d) \quad (1)$$

where
  - $\mathbf{x} = [x_0, x_1, x_2, \ldots, x_d]^T$, $\mathbf{w} = [w_0, w_1, w_2, \ldots, w_d]^T$.
  - $x_0 = 1$ is augmented and w_0 is the bias,
  - the part of $\mathbf{x}$ without x_0 is denoted as $\mathbf{x}_{[1+]} = [x_1, x_2, \ldots, x_d]$, the feature vector of a sample or the raw input. Because in NNs, it is often the raw input, let's call it **input vector**.
  - $\phi(\cdot)$ is an **activation function**, which could be nonlinear, e.g., step or logistic (σ).
  - The output $\hat{y}$ of a neuron is also called the **activation**.

- This is a linear classifier: $\phi(\mathbf{w}^T\mathbf{x})$ where $\phi(\cdot)$ can be, e.g., a step or sign function. (If using a continuous function for $\phi(\cdot)$, we get a regressor. In the regressor case, ϕ is often nonlinear.)

Notations and terminology

We will use this style for a neuron. The inside of a neuron is hidden, and the label on the node is its output.

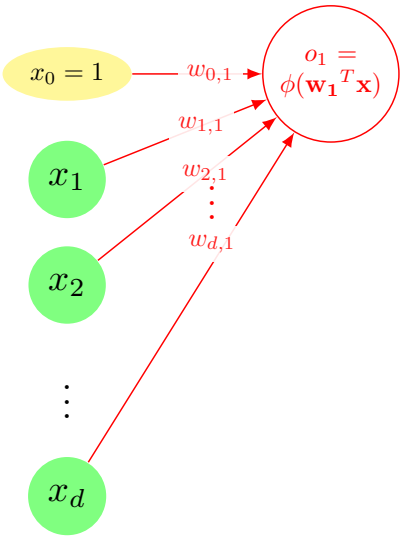


- A neuron or a neural network is often represented using a directed graph.
- On this graph, every **node** is a neuron and every edge is called a **connection**.
- How many connections do we have here?
- d + 1 (1 for bias)
- The green and yellow neurons **feed** their outputs $[x_1, x_2, \ldots, x_d]$ and x_0, respectively, to the red neuron.
- If bias is not wanted, just set w_0 to 0.

A perceptron vs. the perceptron algorithm

- Why is one algorithm seen earlier called the "perceptron" algorithm?
- Because $\phi(\mathbf{w}^T\mathbf{x})$ is exactly one neuron/perceptron in an ANN.
- Frank Rosenblatt published his perceptron algorithm in 1962 titled "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms."
- Linearly separable cases only! It cannot even do XOR.
- Therefore, Marvin Minsky jumped to the conclusion that ANNs were useless. [Perceptrons, Marvin Minsky and Seymour Papert, MIT Press, 1969]
- However, Minsky is an AAAI fellow but not a prophet.
- If we expand a perceptron into layers of perceptrons, we get an **artificial neural network (ANN)** or an **multi-layer perceptron (MLP)**, which is much more powerful, and for sure, can mimic XOR.

From a single neuron to a network of neurons I



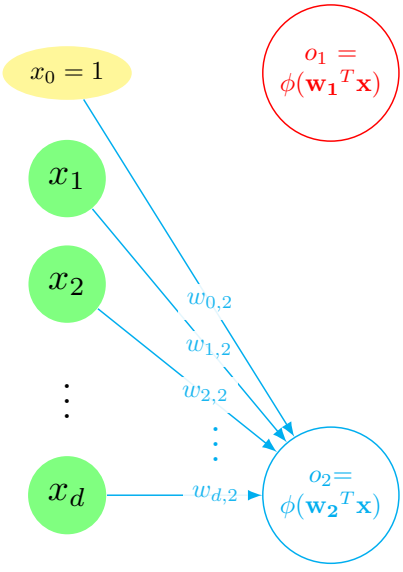
- I/O relation (recall Eq. 1) for the red-circled neuron:

$$\phi\left[\begin{pmatrix} w_{0,1} & w_{1,1} & w_{2,1} & \cdots & w_{d,1}\end{pmatrix}\begin{pmatrix}1\\x_1\\x_2\\\vdots\\x_d\end{pmatrix}\right] = \begin{pmatrix}o_1\end{pmatrix}$$

- o_1 is the output/activation of the neuron.
- For a matrix of only one element, we usually abbreviate it into a scalar. Hence (o_1) can be rewritten as o_1 when no ambiguity raises.
- Math note: Applying a scalar-domain function to a matrix means applying the function to each element of the matrix. For example, if

$$\phi(x) = \begin{cases}1, & if \quad x > 0\\ -1, & else.\end{cases} \text{ then } \phi\begin{pmatrix}1 & -2\\ -3 & 4\end{pmatrix} = \begin{pmatrix}1 & -1\\ -1 & 1\end{pmatrix}$$

From a single neuron to a network of neurons II

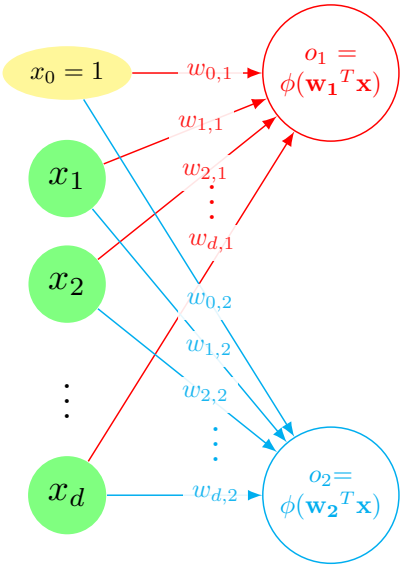


- I/O relation (recall Eq. 1) for blue-circled neuron:

$$\phi\left[\begin{pmatrix} w_{0,2} & w_{1,2} & w_{2,2} & \cdots & w_{d,2} \end{pmatrix} \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix}\right] = \begin{pmatrix} o_2 \end{pmatrix}$$

- (o_2) abbreviated into o_2.
- o_2 is the output/activation of the neuron.

From a single neuron to a network of neurons III



- Put results above together:

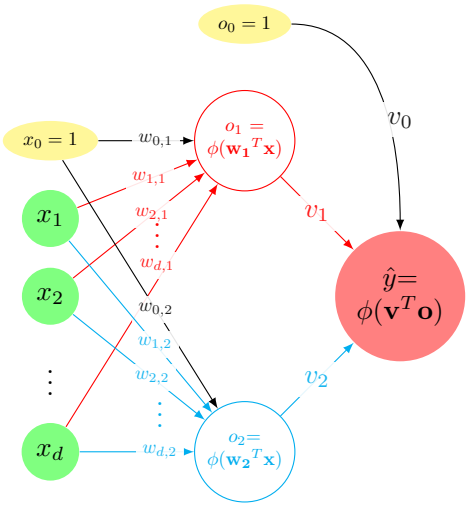
$$\phi\left[\begin{pmatrix} w_{0,1} & w_{1,1} & w_{2,1} & \cdots & w_{d,1} \\ w_{0,2} & w_{1,2} & w_{2,2} & \cdots & w_{d,2} \end{pmatrix}\begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_d \end{pmatrix}\right] = \begin{pmatrix} o_1 \\ o_2 \end{pmatrix} = \mathbf{o}_{[1+]}$$

- The subscript $+$ means non-bias neurons. We will see more about it in the next
- Rewrite into a shorter form:

$$\phi(\mathbb{W}^T\mathbf{x}) = \mathbf{o}_{[1+]}$$

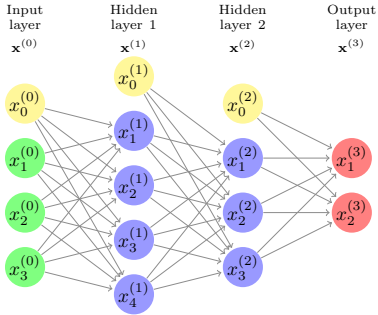
where $\mathbb{W} = \begin{pmatrix} w_{0,1} & w_{0,2} \\ w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \\ \vdots & \vdots \end{pmatrix} = \begin{pmatrix} | & | \\ \mathbf{W}_1 & \mathbf{W}_2 \\ | & | \end{pmatrix}$.

From a single neuron to a network of neurons IV



- Now we further expand the network by feeding o_1 and o_2 to another neuron $\hat{y}$.
- Three connections: $o_0 \to \hat{y}$ (the bias), $o_1 \to \hat{y}$, and $o_2 \to \hat{y}$.
- $\mathbf{o} = [1] \oplus \mathbf{o}_{[1+]}$ is the result of concatenating ($\oplus$) the bias 1 with the output $\mathbf{o}_{[1+]}$ from the previous slide.
- $\hat{y} = \phi\left[\begin{pmatrix} v_0 & v_1 & v_2 \end{pmatrix}\begin{pmatrix} o_0 \\ o_1 \\ o_2 \end{pmatrix}\right] = \phi(\mathbb{V}^T\mathbf{o})$
- $\mathbb{W}$ and $\mathbb{V}$ (just a font adjustment from $\mathbf{v}$) are called **weight matrix** or more clearly **transfer matrixes** as they connects two sets of neurons.
- In this example, $\mathbb{V}$ has only one column. Why?
- Every neuron ressembles a linear classifier. Its weights are one column in the corresponding transfer matrix.

Feedforward: the basic algorithm for ANNs to yield outputs I



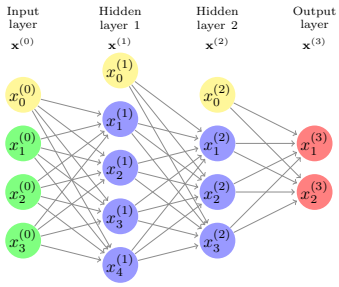
Yellow nodes are bias nodes. Layer index starts from 0.

- The transform $\mathbf{o} = \phi(\mathbb{W}^T\mathbf{x})$ or $\hat{y} = \phi(\mathbb{V}^T\mathbf{o})$ is called **feedforward** where the outputs from a **layer** (to be defined later) of neurons are **fed** into another laye
- To generalize, we use the notation $\mathbf{x}^{(l)} = [x_0^{(l)}, x_1^{(l)}, x_2^{(l)}, \dots]$ to represent neurons at layer l ($x_i^{(l)}$ is the i-th neuron in layer l), and $\mathbb{W}^{(l)}$ to denote the transfer matrix from layer l to layer $l+1$. We call l the **layer index** and i in this context the **node index**.
- Generalized feedforward between any two layers:

$$\mathbf{x}_{[1+]}^{(l+1)} = \phi(\mathbb{W}^{(l)}\mathbf{x}^{(l)}) = \phi(\mathbb{W}^{(l)}\begin{bmatrix}1 \\ \oplus \\ \mathbf{x}_{[1+]}^{(l)}\end{bmatrix})$$

  The subscript $_{[1+]}$ because bias terms are not produced by feedforward.
- The bias neuron $x_0^{(l)} = 1$ always except for the last/output layer which has n bias neuron (why?)
- Hence for the last/output layer, $\mathbf{x}^{(-1)} = [x_1^{(-1)}, x_2^{(-1)}, \dots]$.

Feedforward II



- Recursively feedforward, you can create a complex ANN:

$$\phi\left(\mathbb{W}^{(L)T}\cdots\left[1\oplus\phi\left(\overbrace{\mathbb{W}^{(1)T}\left[1\oplus\phi\left(\mathbb{W}^{(0)T}\mathbf{x}^{(0)}\right)\right]}^{\mathbf{x}^{(1)}_{[1+]}}\right)\right]\right)\overbrace{\phantom{\left[1\oplus\phi\left(\mathbb{W}^{(1)T}\left[1\oplus\phi\left(\mathbb{W}^{(0)T}\mathbf{x}^{(0)}\right)\right]\right)\right]}}^{\mathbf{x}^{(2)}_{[1+]}}\right)$$

- For the example NN in the figure left:

$$\underset{\mathbf{x^{(0)}}}{\begin{pmatrix}1\\x_1^{(0)}\\x_2^{(0)}\\x_3^{(0)}\end{pmatrix}}\xrightarrow{\phi,\mathbb{W}^{(1)}}\underset{\mathbf{x^{(1)}}}{\begin{pmatrix}1\\x_1^{(1)}\\x_2^{(1)}\\x_3^{(1)}\\x_4^{(1)}\end{pmatrix}}\xrightarrow{\phi,\mathbb{W}^{(2)}}\underset{\mathbf{x^{(2)}}}{\begin{pmatrix}1\\x_1^{(2)}\\x_2^{(2)}\\x_3^{(2)}\end{pmatrix}}\xrightarrow{\phi,\mathbb{W}^{(3)}}\underset{\mathbf{x}^3}{\begin{pmatrix}x_1^{(3)}\\x_2^{(3)}\end{pmatrix}}.$$

Once again, the last/output layer has no bias.

Layers

- Mathematically, all neurons right-multiplied with a transfer matrix form a **layer**, with the exception of output neurons where feedforward ends. Output neurons form the **output layer**.
- The starting point of feedforward is the **input layer**, whose values, except the bias if applicable, are given by the user.
- Layers between the input layer and the output layer are **hidden layers**.
- There is only one input layer and only one output layer per ANN.
- Each layer (including output layer) can have any number of neurons. Minimal is one.
- An ANN can have any arbitrary number of hidden layers. Even zero.

An ANN for XOR

Any logic operation: Fig. 2.9 of the Neural Network Ebook comes with the `neuralnetwork` package for LaTeX

Taking a break

Why did mathematicians invent matrixes?

Quiz

In order to approximate the relation $h = \frac{1}{2}gt^2$

- How many neurons are needed in the input layer?
- How many neurons are needed in the output layer?
- If the activation function is logistic function, are hidden layers necessary? Why?
- How many hidden neurons are needed at least?

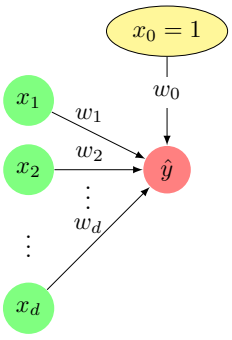
Training ANNs

- The power of an ANN is in its weights, e.g., $\mathbf{W}^{(1)}$, $\mathbf{W}^{(2)}, \ldots$.
- By "training" an ANN, we mean tuning its weights.
- But how?
- Due to the complexity of ANNs, it's difficult to get an analytical form of the solution of weights like what we did in simple linear classifiers.
- Gradient descent is commonly used, e.g., $\mathbf{W}^{(i)} \leftarrow \mathbf{W}^{(i)} + \rho \nabla \text{Loss}$ (such as error)
- But first, we need a loss function.

Loss function for training ANNs

- In early times of ANN research, mean squared error (MSE) was used as the loss function: $\frac{1}{N}\sum_i(\hat{y_i} - y_i)^2$ where y_i and $\hat{y}_i$ are ground truth target and prediction for the i-th sample, respectively, N is the number of samples. (Note it's difference to sum of squared error in the averaging part)
- Then it is noted that negative logistic loss is better (neg-log-loss) in that the logistic function panelizes a big error more than a small error. (See an explanation by Shuyu Luo in Towards Data Science where $h_\theta(x)$ is the prediction $\hat{y}$ used in our class.)
- We talked about log-loss in Unit 5 Regression. Google also has a good refreshing material
- Given a prediction $\hat{y}$ and a ground truth target y, the neg-log-loss is $-y\log\hat{y} - (1-y)\log(1-\hat{y})$
- **The discussion above about neg-log-loss is for classification only. For regression, MSE is still used de facto.**

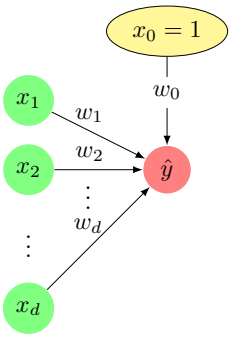
Gradient descent in ANNs: the simplest case, just one neuron



- If just one neuron, it's something similar to perceptron algorithm [Part I of Unit 4 SVMs slides], except that we will use neg-log-loss instead of mean/sum squared error.
- Warm-up: $\hat{y} = \phi(\mathbf{w}^T\mathbf{x})$
- Based on chain rule of derivative (J is a function [loss] of $\hat{y}$, which is a function [activation] of $\mathbf{w}^T\mathbf{x}$, which is a function [dot product] of w_i), gradient for each weight w_i is:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}}$$

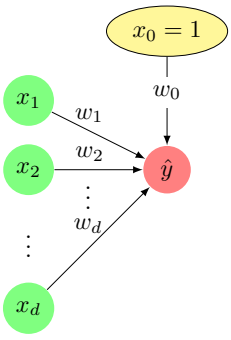
Gradient descent in ANNs: the simplest case, just one neuron



- If just one neuron, it's something similar to perceptron algorithm [Part I of Unit 4 SVMs slides], except that we will use neg-log-loss instead of mean/sum squared error.
- Warm-up: $\hat{y} = \phi(\mathbf{w}^T\mathbf{x})$
- Based on chain rule of derivative (J is a function [loss] of $\hat{y}$, which is a function [activation] of $\mathbf{w}^T\mathbf{x}$, which is a function [dot product] of w_i), gradient for each weight w_i is:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}^T\mathbf{x}}$$

Gradient descent in ANNs: the simplest case, just one neuron

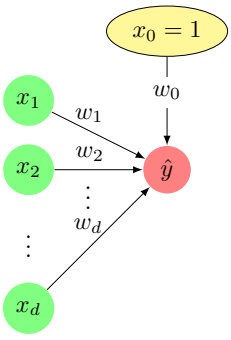


- If just one neuron, it's something similar to perceptron algorithm [Part I of Unit 4 SVMs slides], except that we will use neg-log-loss instead of mean/sum squared error.
- Warm-up: $\hat{y} = \phi(\mathbf{w}^T\mathbf{x})$
- Based on chain rule of derivative (J is a function [loss] of $\hat{y}$, which is a function [activation] of $\mathbf{w}^T\mathbf{x}$, which is a function [dot product] of w_i), gradient for each weight w_i is:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}^T\mathbf{x}} \frac{\partial \mathbf{w}^T\mathbf{x}}{\partial w_i} \tag{2}$$

- If the loss is neg-log-loss: $\frac{\partial J}{\partial \hat{y}} = \frac{\partial \ -y \log \hat{y} - (1-y)\log(1-\hat{y})}$

Gradient descent in ANNs: the simplest case, just one neuron

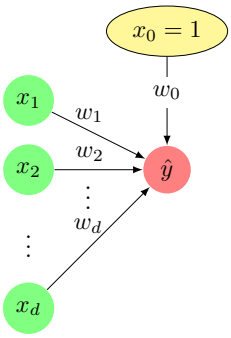


- If just one neuron, it's something similar to perceptron algorithm [Part I of Unit 4 SVMs slides], except that we will use neg-log-loss instead of mean/sum squared error.
- Warm-up: $\hat{y} = \phi(\mathbf{w}^T\mathbf{x})$
- Based on chain rule of derivative (J is a function [loss] of $\hat{y}$, which is a function [activation] of $\mathbf{w}^T\mathbf{x}$, which is a function [dot product] of w_i), gradient for each weight w_i is:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}^T\mathbf{x}} \frac{\partial \mathbf{w}^T\mathbf{x}}{\partial w_i} \tag{2}$$

- If the loss is neg-log-loss: $\frac{\partial J}{\partial \hat{y}} = \frac{\partial \ -y \log \hat{y} - (1-y)\log(1-\hat{y})}{\partial \hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$

Gradient descent in ANNs: the simplest case, just one neuron

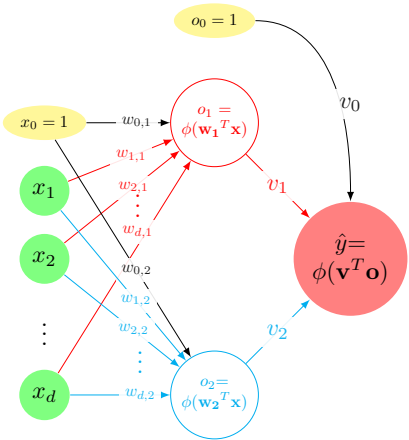


- If just one neuron, it's something similar to perceptron algorithm [Part I of Unit 4 SVMs slides], except that we will use neg-log-loss instead of mean/sum squared error.
- Warm-up: $\hat{y} = \phi(\mathbf{w}^T\mathbf{x})$
- Based on chain rule of derivative (J is a function [loss] of $\hat{y}$, which is a function [activation] of $\mathbf{w}^T\mathbf{x}$, which is a function [dot product] of w_i), gradient for each weight w_i is:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{w}^T\mathbf{x}} \frac{\partial \mathbf{w}^T\mathbf{x}}{\partial w_i} \tag{2}$$

- If the loss is neg-log-loss: $\frac{\partial J}{\partial \hat{y}} = \frac{\partial \ -y\log\hat{y} - (1-y)\log(1-\hat{y})}{\partial \hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$
- If using sigmoid activation function $\phi(x) = \sigma(x) = 1/(1+e^{-x})$, from calculus, its derivative is $\sigma(x)(1-\sigma(x))$, and thus
  $\frac{\partial \hat{y}}{\partial \mathbf{w}^T\mathbf{x}} = \frac{\partial \phi(\mathbf{w}^T\mathbf{x})}{\partial \mathbf{w}^T\mathbf{x}} = \frac{\partial \sigma(\mathbf{w}^T\mathbf{x})}{\partial \mathbf{w}^T\mathbf{x}} = \sigma\left(\mathbf{w}^T\mathbf{x}\right)\left(1-\sigma\left(\mathbf{w}^T\mathbf{x}\right)\right) = \hat{y}(1-\hat{y})$
- Lastly, $\frac{\partial \mathbf{w}^T\mathbf{x}}{\partial w_i} = x_i$.
- Put together: $\frac{\partial J}{\partial w_i} = (\hat{y}-y)x_i$
- Or in matrix form for all weights: $\frac{\partial J}{\partial \mathbf{w}} = (\hat{y}-y)\mathbf{x}$.

What about neurons more upstream?



- Apply the chain rule again:

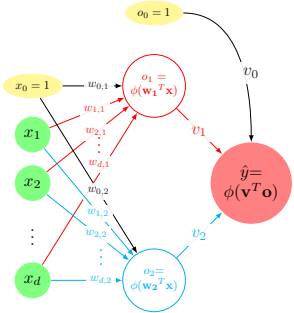
$$\frac{\partial J}{\partial w_{1,1}} = \frac{\partial J}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{1,1}} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{v}^T\mathbf{o}} \frac{\partial \mathbf{v}^T\mathbf{o}}{\partial o_1} \cdot \frac{\partial o_1}{\partial \mathbf{w_1}^T\mathbf{x}} \frac{\partial \mathbf{w_1}^T\mathbf{x}}{\partial w_{1,1}} \tag{3}$$

- This is the idea of **backpropagation**. In order to compute the partial derivative of the loss over a weight, first reversely propagate the loss to a neuron (here $\frac{\partial J}{\partial o_1}$) **destinated by the weight**, then compute the partial derivative of the neuron's output over the weight (here $\frac{\partial o_1}{\partial w_{1,1}}$), finally multiply the two.
- Because we will often use the partial derivative of an activation function ϕ over its input $\mathbf{w}^T\mathbf{x}$, we introduce a notation to express it in terms of its *output* $\phi(\mathbf{w}^T\mathbf{x})$:

$$\psi(\phi(\mathbf{w}^T\mathbf{x})) = \frac{\partial \phi(\mathbf{w}^T\mathbf{x})}{\partial \mathbf{w}^T\mathbf{x}} \tag{4}$$

- When the activation function is logistic, we have $\psi(\sigma(\mathbf{w}^T\mathbf{x})) = \sigma(\mathbf{w}^T\mathbf{x})(1 - \sigma(\mathbf{w}^T\mathbf{x}))$.

What about neurons more upstream? II



- Subsituting Eq. (4) into Eq. (3), we have:

$$\frac{\partial J}{\partial w_{1,1}} = \frac{\partial J}{\partial o_1} \cdot \frac{\partial o_1}{\partial w_{1,1}} = \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) v_1 \ \cdot \ \psi(o_1) x_1 \tag{5}$$

- Generalize for all weights in $\mathbf{w}_1 = [w_{0,1}, w_{1,1}, w_{2,1}, \ldots, w_{d,1}]^T$: $\partial J / \partial \mathbf{w}_1 = \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) v_1 \psi(o_1) \mathbf{x}$ (column vector)
- And for all weights in $\mathbf{w}_2 = [w_{0,2}, w_{1,2}, w_{2,2}, \ldots, w_{d,2}]^T$: $\partial J / \partial \mathbf{w}_2 = \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) v_2 \psi(o_2) \mathbf{x}$ (column vector)
- Note that there are two conventions on the orientation of partial derivatives of vectors. Here we use the denominator layout notation. In the denominator layout notation, $\nabla J_{\mathbf{w}} = \frac{\partial J}{\partial \mathbf{w}}$ for a column vector $\mathbf{w}$ and $\nabla J_{\mathbb{W}}$ has the same shape as a matrix $\mathbb{W}$. For more, read this.

- In matrix form ($(d+1) \times 2$):

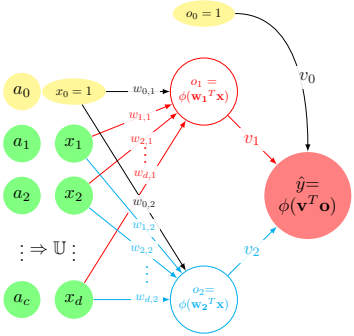
$$\nabla J_{\mathbf{w}} = \begin{pmatrix} | & | \\ \frac{\partial J}{\partial \mathbf{w_1}} & \frac{\partial J}{\partial \mathbf{w_2}} \\ | & | \end{pmatrix} = \mathbf{x} \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \begin{pmatrix} v_1 \psi(o_1) & v_2 \psi(o_2) \end{pmatrix} = \mathbf{x} \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \left[\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \circ \psi \begin{pmatrix} o_1 \\ o_2 \end{pmatrix} \right]^T \tag{6}$$

$$= \mathbf{x} \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \left[\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \circ \psi \begin{pmatrix} o_0 \\ o_1 \\ o_2 \end{pmatrix} \right]^T_{[1+]} = \mathbf{x} \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \left[\mathbb{V} \circ \psi(\mathbf{o}) \right]^T \underbrace{+}_{\text{drop the first row}} \tag{7}$$

where $\circ$ is Hadamard product or element-wise product of matrixes, and the subscript $+$ means dropping the first element/row corresponding to the bias term/terms – this means that bias terms have no impact to the the gradient of previous layers.

Backpropagation: further upstream

- Let's think one more step further: there is another layer $\mathbf{a}$ before $\mathbf{x}$, how do we backpropagate the derivative of loss from $\frac{\partial J}{\partial \mathbf{o}}$ to $\frac{\partial J}{\partial \mathbf{x}}$?



- Thus (don't be overwhelmed by the amount of math...)

$$\frac{\partial J}{\partial x_1} = \frac{\partial J}{\partial o_1} \cdot \frac{\partial o_1}{\partial x_1} + \frac{\partial J}{\partial o_2} \cdot \frac{\partial o_2}{\partial x_1}$$

$$= \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \frac{\partial \mathbf{v}^T\mathbf{o}}{\partial o_1} \cdot \frac{\partial o_1}{\partial \mathbf{w_1}^T\mathbf{x}} \frac{\partial \mathbf{w_1}^T\mathbf{x}}{\partial x_1} + \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \frac{\partial \mathbf{v}^T\mathbf{o}}{\partial o_2} \cdot \frac{\partial o_2}{\partial \mathbf{w_2}^T\mathbf{x}} \frac{\partial \mathbf{w_2}^T\mathbf{x}}{\partial x_1}$$

$$= \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) v_1 \psi(o_1) w_{1,1} + \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) v_2 \psi(o_2) w_{1,2}$$

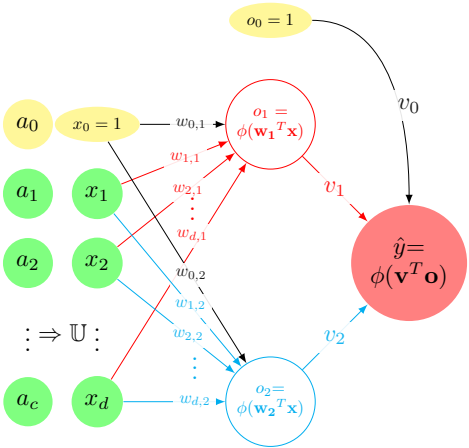
$$= \begin{pmatrix} w_{1,1} & w_{1,2} \end{pmatrix} \left\{ \psi \begin{pmatrix} o_1 \\ o_2 \end{pmatrix} \circ \left[\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \right] \right\}$$

$$= \begin{pmatrix} w_{1,1} & w_{1,2} \end{pmatrix} \left\{ \psi \begin{pmatrix} o_0 \\ o_1 \\ o_2 \end{pmatrix} \circ \left[\begin{pmatrix} v_0 \\ v_1 \\ v_2 \end{pmatrix} \frac{\partial J}{\partial \hat{y}} \psi(\hat{y}) \right] \right\}_{[1+]}$$

$$= \underbrace{\begin{pmatrix} w_{1,1} & w_{1,2} \end{pmatrix}}_{\text{row 1 of } \mathbb{W}} \left[\psi(\mathbf{o}) \circ \left(\mathbf{v} \underbrace{\frac{\partial J}{\partial \hat{y}} \psi(\hat{y})}_{\boldsymbol{\delta}^{(-1)}} \right) \right]_{[1+]} \tag{8}$$

- Plese note the $+$ subscript.

Backpropagation: a more generalized example I



- Generalize to all elements of $\mathbf{x}$:

$$\underbrace{\frac{\partial J}{\partial \mathbf{x}}}_{(d+1)\times 1} = \begin{pmatrix} w_{0,1} & w_{0,2} \\ w_{1,1} & w_{1,2} \\ \vdots & \vdots \end{pmatrix} \left[\psi(\mathbf{o}) \circ \left(\mathbf{v} \underbrace{\frac{\partial J}{\partial \hat{y}} \psi(\hat{y})}_{\boldsymbol{\delta}^{(-1)}} \right) \right]_{[1+]}$$

$$= \mathbb{W} \left[\psi(\mathbf{o}) \circ \left(\mathbb{V} \boldsymbol{\delta}^{(-1)} \right) \right]_{[1+]} = \mathbb{W} \boldsymbol{\delta}^{(-2)}_{[1+]} \tag{9}$$

where $\boldsymbol{\delta}^{(-2)}_{[1+]}$ are elements of $\boldsymbol{\delta}^{(-2)} = \left[\psi(\mathbf{o}) \circ \left(\mathbb{V} \boldsymbol{\delta}^{(-1)} \right) \right]$ without the first row corresponding to the bias term o_1, and we morph $\mathbf{v}$ to $\mathbb{V}$ just to keep the style for transfer/weighting matrixes consistent.

- Why not $\boldsymbol{\delta}^{(-1)}_{[1+]}$? Because no bias term in the output layer.

Backpropagation: a more genaralized example II

With $\underbrace{\partial J/\partial \mathbf{x}}_{(d+1)\times 1}$ in hand, we can compute $\underbrace{\partial J/\partial \mathbb{U}}_{c\times d} = \frac{\partial J}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial \mathbb{U}}$, where $\mathbf{x} = \mathbb{U}^T\mathbf{a}$ and $\mathbf{a} = [a_0, a_1, \ldots, a_c]^T$. Again, loss does not propagate from x_0 to previous layer ($\mathbf{a}$ layer) – because no connection.

$$\left(\frac{\partial J}{\partial \mathbb{U}}\right)^T = \underbrace{\begin{pmatrix} - & \left(\frac{\partial J}{\partial \mathbf{u}_{*,\mathbf{1}}}\right)^T & - \\ - & \left(\frac{\partial J}{\partial \mathbf{u}_{*,\mathbf{2}}}\right)^T & - \\ & \cdots & \\ - & \left(\frac{\partial J}{\partial \mathbf{u}_{*,\mathbf{d}}}\right)^T & - \end{pmatrix}}_{d\times c} = \underbrace{\left[\frac{\partial J}{\partial \mathbf{x}_{[1+]}} \circ \frac{\partial \mathbf{x}_{[1+]}}{\partial \mathbb{U}^T\mathbf{a}}\right]}_{d\times 1,\ \text{skip loss on } x_0} \frac{\partial \mathbb{U}^T\mathbf{a}}{\partial \mathbb{U}} = \underbrace{\left[\frac{\partial J}{\partial \mathbf{x}} \circ \frac{\partial \mathbf{x}}{\partial \mathbb{U}^T\mathbf{a}}\right]_{[1+]}}_{d\times 1,\ \text{skip loss on } x_0} \underbrace{\frac{\partial \mathbf{a}^T\mathbb{U}}{\partial \mathbb{U}}}_{1\times c}$$

$$= \underbrace{\left[\left(\mathbb{W}\boldsymbol{\delta}^{(-2)}_{[1+]}\right) \circ \psi(\mathbf{x})\right]_{[1+]}}_{\boldsymbol{\delta}^{(-3)}_{[1+]}} \mathbf{a}^T = \underbrace{\left[\psi(\mathbf{x}) \circ \left(\mathbb{W}\boldsymbol{\delta}^{(-2)}_{[1+]}\right)\right]_{[1+]}}_{\boldsymbol{\delta}^{(-3)}_{[1+]}} \mathbf{a}^T = \boldsymbol{\delta}^{(-3)}_{[1+]}\mathbf{a}^T \tag{10}$$

- Transpose both sides of Eq. (10): $\frac{\partial J}{\partial \mathbb{U}} = \begin{pmatrix} | & | & & | \\ \frac{\partial J}{\partial \mathbf{u}_{*,1}} & \frac{\partial J}{\partial \mathbf{u}_{*,2}} & \cdots & \frac{\partial J}{\partial \mathbf{u}_{*,d}} \\ | & | & & | \end{pmatrix} = \left(\boldsymbol{\delta}^{(-3)}_{[1+]}\mathbf{a}^T\right)^T = \mathbf{a}\left(\boldsymbol{\delta}^{(-3)}_{[1+]}\right)^T$

Backpropagation: Do you see a recursive pattern?

With the definitions of $\boldsymbol{\delta}^{(-2)}$ from Eq. (9) and $\boldsymbol{\delta}^{(-1)}$ from Eq. (8), we can expand Eq. (10) all the way to $\boldsymbol{\delta}^{(-1)}$:

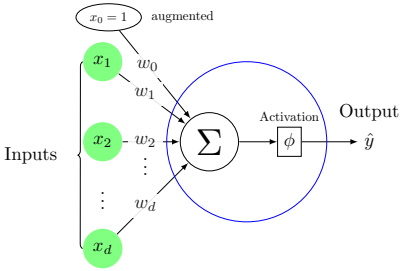
$$\boldsymbol{\delta}^{(-3)}_{[1+]}\mathbf{a}^T = \left[\psi(\mathbf{x}) \circ \left(\mathbb{W}\boldsymbol{\delta}^{(-2)}_{[1+]}\right)\right]_{[1+]} \mathbf{a}^T$$

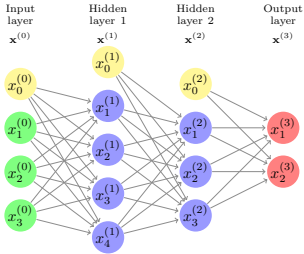
$$= \left[\psi(\mathbf{x}) \circ \left(\mathbb{W}\left[\psi\left(\mathbf{o}\right) \circ \left(\mathbb{V}\boldsymbol{\delta}^{(-1)}\right)\right]_{[1+]}\right)\right]_{[1+]} \mathbf{a}^T$$

$$= \left[\psi(\mathbf{x}) \circ \left(\mathbb{W}\overbrace{\left[\psi\left(\mathbf{o}\right) \circ \left(\mathbb{V}\underbrace{\left[\psi(\hat{y}) \circ \frac{\partial J}{\partial \hat{y}}\right]}_{\boldsymbol{\delta}^{(-1)}}\right)\right]_{[1+]}}^{\boldsymbol{\delta}^{(-2)}_{[1+]}}\right)\right]_{[1+]} \mathbf{a}^T$$

- We see a recursive pattern that a transform matrix $\mathbb{W}$ or $\mathbb{V}$ is left-multiplied with a term and then Hadamard product ($\circ$) with a $\psi(\cdot)$ term.
- It does two things: first, distribute the gradient on loss one layer backward/upstream, and second pass the gradient thru the activation function. Then the gradient is ready to be partial derivated with weights.
- If we keep applying this, we can compute the gradient of loss/error over weights of more upstream layers.

What exactly is $\boldsymbol{\delta}^{(l)}$?





There is NO $\delta_0^{(l)}$ if layer l is output.

- It's the partial derivative of loss/error over the input of the activation function at layer l or the weighted sums at layer l – between Σ and ϕ in the figure left. For convenience, let's call it "pre-activation" (or "post-weighted-sum") error.

$$\begin{aligned}\boldsymbol{\delta}^{(l)} &= \frac{\partial J}{\partial \mathbf{x}^{(l)}} \frac{\partial \mathbf{x}^{(l)}}{\partial \mathbb{W}^{(l-1)T}\mathbf{x}^{(l-1)}} \\ &= \frac{\partial J}{\partial \mathbf{x}^{(l)}} \frac{\partial \phi(\mathbb{W}^{(l-1)T}\mathbf{x}^{(l-1)})}{\partial \mathbb{W}^{(l-1)T}\mathbf{x}^{(l-1)}} = \frac{\partial J}{\partial \mathbf{x}^{(l)}} \psi(\mathbf{x}^{(l)})\end{aligned}$$

(For the definition of $\psi(\cdot)$, refer to Eq. 4)

- Yes, $\boldsymbol{\delta}^{(l)}$ is a vector. What's its shape?
- We will use the notation $\delta_i^{(l)}$ to denote the pre-activation loss on the i-th neuron at layer l. For example, to the network on the left, $\boldsymbol{\delta}^{(2)} = [\delta_0^{(2)}, \delta_1^{(2)}, \delta_2^{(2)}, \delta_3^{(2)}]^T$
- But note that $\delta_0^{(l)}$ for any **non-output** layer l is just a placeholder – because the bias neuron in any layer has no activation function, and it is not connected with any neurons in the layer in prior.

Generalized backpropagation I

- Let the error/loss **backpropagated** to layer l be $\boldsymbol{\delta}^{(l)}$ where the loss on the bias term is $\boldsymbol{\delta}_0^{(l)}$.
- Then the error/loss one layer back (pre-activation) is

$$\boldsymbol{\delta}^{(l-1)} = \begin{cases} \psi(\mathbf{x}^{(l-1)}) \circ \left(\mathbb{W}^{(l-1)}\boldsymbol{\delta}^{(l)}\right) & \text{if } l \text{ is the output layer} \\ \psi(\mathbf{x}^{(l-1)}) \circ \left(\mathbb{W}^{(l-1)}\boldsymbol{\delta}_{[1+]}^{(l)}\right) & \text{otherwise} \end{cases}$$

where $\boldsymbol{\delta}_{[1+]}^{(l)}$ means dropping the loss $\boldsymbol{\delta}_0^{(l)}$ on the bias term (Why?), $x^{(l-1)}$ is the output from layer $l-1$ and $\mathbb{W}^{(l-1)}$ is the transfer matrix from layers $l-1$ to l. This applies when there are multiple output neurons.

- Finally, for a weight $w_{i,j}^{(l-2)}$ connecting the i-th neuron in the $l-2$ layer to the j-th neuron in the $l-1$ layer, the partial derivative

$$\frac{\partial J}{\partial w_{i,j}^{(l-2)}} = \underbrace{\frac{\partial J}{\partial \mathbf{w}_j^{(l-2)}\mathbf{x}^{(l-2)}}}_{\text{scalar, pre-activation loss on neuron } x_j^{(l-1)}} \frac{\partial \mathbf{w}_j^{(l-2)}\mathbf{x}^{(l-2)}}{\partial w_{i,j}^{(l-2)}} = \delta_j^{(l-1)} x_i^{(l-2)}$$

Generalized backpropagation II

- To update $w_{i,j}^{(l-2)}$: $w_{i,j}^{(l-2)} \leftarrow w_{i,j}^{(l-2)} - \rho \frac{\partial J}{\partial w_{i,j}^{(l-2)}}$ where ρ is the learning rate.
- Vectorized version of the weight gradient above (using results in Eq. 10):

$$\nabla^{(l-2)} = \frac{\partial J}{\partial \mathbb{W}^{(l-2)}} = \underbrace{\frac{\partial J}{\partial \mathbb{W}^{(l-2)T}\mathbf{x}^{(l-2)}}}_{\substack{\text{pre-activation loss to} \\ \text{all neurons at layer } l-1}} \frac{\partial \mathbb{W}^{(l-2)T}\mathbf{x}^{(l-2)}}{\partial \mathbb{W}^{(l-2)}} = \begin{cases} \mathbf{x}_i^{(l-2)} \left(\boldsymbol{\delta}^{(l-1)}\right)^T & \text{if } l-1 \text{ is the output layer} \\ \mathbf{x}_i^{(l-2)} \left(\boldsymbol{\delta}_{[1+]}^{(l-1)}\right)^T & \text{otherwise} \end{cases}$$

- Vectorized weight update (subtract a matrix from a matrix):

$$\mathbb{W}^{(l-2)} \leftarrow \mathbb{W}^{(l-2)} - \rho \nabla^{(l-2)}$$

  where ρ, the learning rate, is a hyperparameter set by the user.
- What are their shapes?
- This is the training of an ANN! Repeat this for every sample. And you can code it up too!

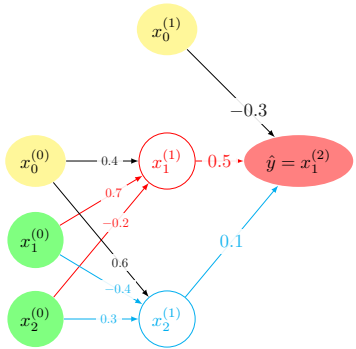
Recap: feedforward and backpropogation

- Two basic algorithms in ANNs.
- Feedforward: $\mathbf{x}_{[1+]}^{(l+1)} = \phi(\mathbb{W}^{(l)T}\mathbf{x}^{(l)})$. The subscript $1+$ because feedforward does not produce bias terms.
- Backpropagation (if layer l is non-output): $\boldsymbol{\delta}^{(l-1)} = \begin{cases} \psi(\mathbf{x}^{(l-1)}) \circ \left(\mathbb{W}^{(l-1)}\boldsymbol{\delta}^{(l)}\right) & \text{if } l \text{ is the output layer} \\ \psi(\mathbf{x}^{(l-1)}) \circ \left(\mathbb{W}^{(l-1)}\boldsymbol{\delta}_{[1+]}^{(l)}\right) & \text{otherwise} \end{cases}$
- Compute gradient: $\nabla^{(l-2)} = \begin{cases} \mathbf{x}_i^{(l-2)} \left(\boldsymbol{\delta}^{(l-1)}\right)^T & \text{if } l-1 \text{ is the output layer} \\ \mathbf{x}_i^{(l-2)} \left(\boldsymbol{\delta}_{[1+]}^{(l-1)}\right)^T & \text{otherwise} \end{cases}$
- To transfer forward (feedforward), use the tranpose of the transfer matrix: $\mathbb{W}^{(l)T}$
- To transfer backward (backpropagation) use the transfer matrix itself $\mathbb{W}^{(l)}$.
- If you know the activations at one layer, how do you get the activations at the previous layer?

Materials for further reading on backpropagation

- Andrew Ng's ML class exercise 4 If the link is dead, search "andrew ng ml class ex4" in Google and there are many copies on the Internet
  - In Andrew's notes, our $\psi(\mathbf{x}^{(l)})$ is called $g'(\mathbf{z}^{(l)})$ where $\mathbf{z}$ is our $\mathbb{W}^T\mathbf{x}^{(\mathbf{l-1})}$, the weighted sum of inputs before activation. In his notation, the output of a layer is denoted as $\mathbf{a}^{(l)} = g(\mathbf{z}^{(l)}) = g(\Theta^{(l-1)}\mathbf{a}^{(l-1)})$ where his g corresponds to our ϕ, the actiation function and his $\Theta^{(l)}$ corresponds to our $\mathbb{W}^{(l)T}$, the transfer matrix from layers l to $l-1$. Note that his $\Theta^{(l)}$ and our $\mathbb{W}^{(l)T}$ are transpose to each other.
- Peter Sadowski's Notes on backpropagation One typo in the notes: x_j should have been h_j.

A grounded example of feedforward and backpropagation

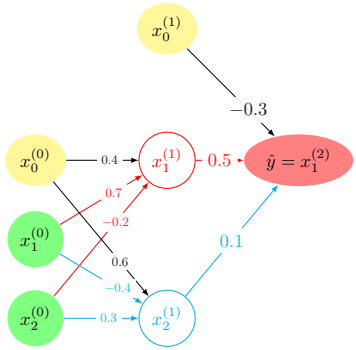


- Given a sample of feature vector $[0, 1]$ and target 1, show the feedforward process.
- From the figure left, the two transfer matrixes:

$$\mathbb{W}^{(0)} = \begin{pmatrix} w_{0,1}^{(0)} & w_{1,1}^{(0)} \\ w_{1,1}^{(0)} & w_{1,2}^{(0)} \\ w_{2,1}^{(0)} & w_{2,2}^{(0)} \end{pmatrix} = \begin{pmatrix} 0.4 & 0.6 \\ 0.7 & -0.4 \\ -0.2 & 0.3 \end{pmatrix} \text{ and } \mathbb{W}^{(1)} = \begin{pmatrix} -0.3 \\ 0.5 \\ 0.1 \end{pmatrix}$$

- First, construct layer 0 vector: $\mathbf{x}^{(0)} = [1, 0, 1]^T$ where the first 1 is augmented for the bias term.
- Then apply the transfer matrix to produce **non-bias** output for layer 1: $\mathbf{x}_{[1+]}^{(1)} = \phi(\mathbb{W}^{(0)T}\mathbf{x}^{(0)}) = [0.550, 0.711]^T$
- So the layer 1 output vector $\mathbf{x}^{(1)} = [1, 0.550, 0.711]^T$ (add 1 to the beginning for bias term)
- Second, feedforward from layer 1 to layer 2: $\mathbf{x}^{(2)} = \phi(\mathbb{W}^{(1)T}\mathbf{x}^{(1)}) = [0.512]$

A grounded example of feedforward and backpropagation II



- Now let's backpropagate.
- First, compute $\delta^{(2)} = \hat{y} - y = [0.512 - 1] = [-0.488]$.
- Second, backpropagate to layer 1:

$$\boldsymbol{\delta}^{(1)} = \psi(\mathbf{x}^{(1)}) \circ \left(\mathbb{W}^{(1)}\boldsymbol{\delta}^{(2)}\right) = \mathbf{x}^{(1)} \circ (1 - \mathbf{x}^{(1)}) \circ \left(\mathbb{W}^{(1)}\boldsymbol{\delta}^{(2)}\right) =$$

$$\begin{pmatrix} 1(1-1) \\ 0.550(1-0.550) \\ 0.711(1-0.711) \end{pmatrix} \circ \left(\begin{pmatrix} -0.3, \\ 0.5, \\ 0.1 \end{pmatrix} [-0.488] \right) = \begin{pmatrix} 0 \\ -0.060 \\ -0.010 \end{pmatrix}$$

- Do we need to propagate to layer 0, or in other words, compute $\boldsymbol{\delta}^{(0)}$?

- With δ's, we can compute the gradient of loss over all weights:
- $\nabla^{(1)} = \partial J / \partial \mathbb{W}^{(1)} = \mathbf{x}^{(1)} \left(\boldsymbol{\delta}^{(2)}\right)^T = \begin{pmatrix} 1.000 \\ 0.550 \\ 0.711 \end{pmatrix} [-0.488] = \begin{pmatrix} -0.488 \\ -0.269 \\ -0.347 \end{pmatrix}$

$$\nabla^{(0)} = \partial J / \partial \mathbb{W}^{(0)} = \mathbf{x}^{(0)} \left(\boldsymbol{\delta}^{(1)}_{[1+]}\right)^T = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} [-0.060, -0.010] = \begin{pmatrix} -0.060 & -0.010 \\ -0.000 & -0.000 \\ -0.060 & -0.010 \end{pmatrix}$$

A grounded example of feedforward and backpropagation III

- Update weights with gradient. Set $\rho = 1$.

$$\mathbb{W}^{(1)} \leftarrow \mathbb{W}^{(1)} - \rho\nabla^{(1)} = \begin{pmatrix} -0.3 \\ 0.5 \\ 0.1 \end{pmatrix} - \begin{pmatrix} -0.488 \\ -0.269 \\ -0.347 \end{pmatrix} = \begin{pmatrix} 0.188 \\ 0.769 \\ -0.447 \end{pmatrix}$$

$$\mathbb{W}^{(0)} \leftarrow \mathbb{W}^{(0)} - \rho\nabla^{(0)} = \begin{pmatrix} 0.4 & 0.6 \\ 0.7 & -0.4 \\ -0.2 & 0.3 \end{pmatrix} - \begin{pmatrix} -0.060 & -0.010 \\ -0.000 & -0.000 \\ -0.060 & -0.010 \end{pmatrix} = \begin{pmatrix} 0.460 & 0.610 \\ 0.700 & -0.400 \\ 0.460 & 0.610 \end{pmatrix}$$

- Let's see whether the new prediction is closer to the target 1 than previous 0.512: You should see $[0.722]$ with new $\mathbb{W}$'s. Great!

Coding hints

See the MiniNN demo.

Batch training in ANNs

- The backpropagation discussed above updates weights for every sample.
- It is also doable in batches, for saving time, and other benefits, e.g., avoiding overfitting or making the ANN more stable.
- For example:
  1. Collect losss on multiple samples.
  2. Use the mean loss (of course, losss should not cancel out) to compute the gradient
  3. Backpropagate using the gradient on mean loss.
  4. Repeat with another set (called **batch** or **minibatch**) of multiple samples.

Epoch in ANN training

- ANN training is done in gradient descent.
- Thus, we can do it again and again on the training data.
- A term frequently used in ANN training is **epoch** (`max_iter` in `scikit-learn`).
- One Epoch is when an ENTIRE dataset is used to train the neural network ONCE.
- After enough epoches, the graident is small enough and then we can stop.
- Too many epoches lead to overfitting. [More to discuss later on overfitting]

Activation functions I: linear, ReLU, and tanh

- A sweet sauce of ANNs is the activation functions.
- So far we have been using the logistic function. There are many out there.
- The simplest activation function is actually linear. Thus, $\phi(x) = x$. The weight sum of inputs is exactly the output/activation.
- A variant of linear activation function is rectified linear unit (ReLU): $\phi(x) = \max(0, x)$.
- Does it ring a bell? Hinge loss?
- As mentioned earlier, the term "sigmoid" could mean any S-shape functions, although in many cases people use it interchangeably with the logistic function.
- A large class of activation functions in ANNs are sigmoids. On top of logistics, hyperbolic tangent ($\tanh$) is another commonly used one.
- $\tanh$ can be considered as a recaled logistic function. The range of $\tanh$ is $[-1, 1]$ while that of logistic is $[0, 1]$.

Activation functions II: softmax

- Another common activation function is **softmax**. It's widely used for multi-class prediction problems formulated such that there are many neurons in the output layer, each of which corresponds to one discrete prediction outcome.
- E.g., in next-word prediction of an input method, each neuron could correspond to one word, and together we have tens or hundreds of thousands of neurons. The neuron will the highest activation corresponds to the word to be entered next.
- E.g., in object detection from photos, each output neuron could correspond to one object category. The one with the highest activation corresponds to the most likely category.
- Softmax is a normalization function. It scales the outputs from neurons into the range $[0, 1]$.

  $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ for $i \in [1..K]$

- Softmax is usually only used in the output layer.

Gradient vanishing problem

- Will the gradient get larger or smaller as backpropagation moves on?
- The derivative $\psi(\cdot)$ of many activation function yields a value in $[-1, 1]$.
- When you multiple a number with another number in $[-1, 1]$, it becomes smaller.
- Hence, the gradient becomes smaller and smaller (vanishes) as we backpropagate toward the input layer.
- It takes really long to update weights near the input layer.
- Solution: LTSM, residual networks, etc.

Saturation of a neuron and network initialization

- A neuron saturates if its output/activation is very close to the lower or upper limit of its activation function, e.g., -1 and 1 for $\tanh$.
- Saturation is bad: e.g., changes to weights or inputs bring little change on the output. The network looks like outputing the same regardless of the inputs or weight changes.
- Reason of saturation: $\mathbb{W}^{(l)}\mathbf{x}^{(l)}$, the input of the activation function, is too big.
- How to avoid:
  - Initialize $\mathbb{W}^{(l)}$ with random numbers in a small range, e.g., $[0, 1]$.
  - Scale your input data, or even scale after each layer (e.g., adding softmax layers), into a small range, e.g., $[0, 1]$.
- If your NN doesn't update fast enough (e.g., loss function doesn't drop much), monitor whether your run into saturation (e.g., how many neurons are close to limits of its range).
- Saturation could be a result of graident vanishing: not enough gradient to bring the output away from limits.

Overfitting in ANNs

- ANNs are very prone to overfitting.
- When training your ANN, monitor the loss on both training and test sets:
  - Alternate the use of training and test sets.
  - Train with training set for a while (usually measured in terms of **epoch**)
  - Check loss on test set.
  - Repeat
- If the loss on test set increases significantly above that on training set, overfitting might have happened. You need to do something.
- Traditionally, for shallow ANNs (a handleful of layers), L2-regularization is the common practice.
- After the raise of deep learning, many new techniques are developed:
  - dropout (nix the output from randomly picked neurons)
  - batch normalization (normalize/scale data within each training batch)
  - early stop (stop when the loss on test set drops too much)

Hyperparameters of an ANN

- Number and types of hidden layers, number of neurons in each hidden layer
- Activation function at each layer (usually it's the same activation function for the entire layer)
- Weights of regularization terms, e.g., α
- Threshold on loss to stop training and maximal number of iterations/epochs
- Other

Dark margic or alchemy?

Configuring the hyperparameters for an ANN is very empirical. Sometimes you don't know why it works or why it doesn't work. It's very much like an experimental science. Just imagine how Edison invented the light bulb.