Machine Learning: Part I

UMaine COS 470/570 – Introduction to AI

Spring 2019

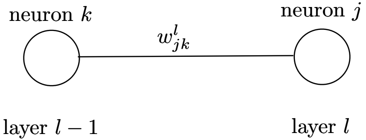
Part 2

From nets to tensors

- Want compact representation of network
- Want representation that can be mapped to parallel processors
- Insight:
  - Inputs can be considered a vector
  - Outputs can be considered a vector
  - Network is completely represented by its weights
  - A weight is between neuron *i* in layer *l* and neuron *j* in layer *l+1*
  - So all weights between two layers can be represented as $i \times j$ or $j \times i$ matrix
- Generalize: Scalars, vectors, matrices, and their *n*-dimensional counterparts: *tensors*
- Can map tensors and tensor operations onto parallel hardware (e.g., GPGPUs)

Notation (from Nielsen)

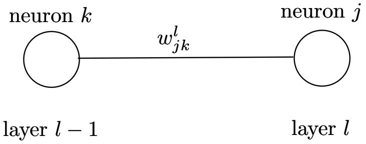
- Assume a multilayer FF network
- w^l_{jk}: wt from neuron k in layer $l-1$ to neuron j in layer l



- Subscript: *jk* for ease of calculation (later)
- b^l_j: bias of neuron j in layer l

Notation (from Nielsen)

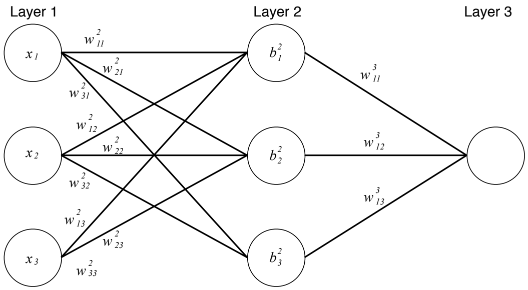
- Assume a multilayer FF network
- w^l_{jk}: wt from neuron k in layer $l-1$ to neuron j in layer l



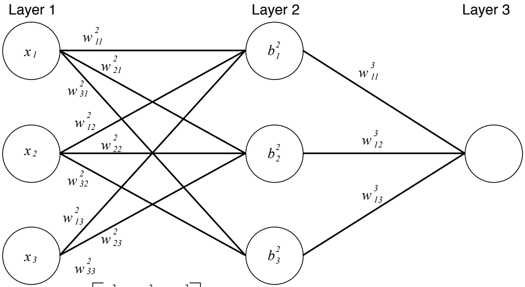
- Subscript: jk for ease of calculation (later)
- b^l_j: bias of neuron j in layer l
- a^l_j: activation (output) of neuron j in layer l

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

Matrix form of NN

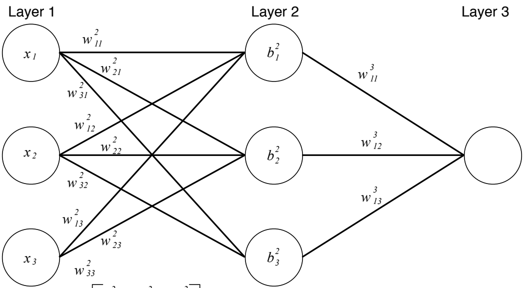


Matrix form of NN



$$\mathbf{w^2} = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \\ w^2_{21} & w^2_{22} & w^2_{23} \\ w^2_{31} & w^2_{32} & w^2_{33} \end{bmatrix} \qquad \mathbf{w^3} = \begin{bmatrix} w^3_{11} & w^3_{12} & w^3_{13} \end{bmatrix}$$

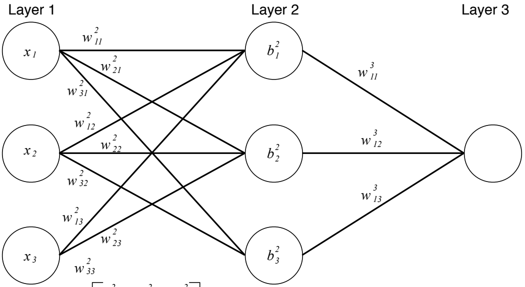
Matrix form of NN



$$\mathbf{w^2} = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \\ w^2_{21} & w^2_{22} & w^2_{23} \\ w^2_{31} & w^2_{32} & w^2_{33} \end{bmatrix} \qquad \mathbf{w^3} = \begin{bmatrix} w^3_{11} & w^3_{12} & w^3_{13} \end{bmatrix}$$

$$\mathbf{z^2} = \begin{bmatrix} (w^2_{11}x_1 + w^2_{12}x_2 + w^2_{13}x_3) + b^2_1 \\ (w^2_{21}x_1 + w^2_{22}x_2 + w^2_{23}x_3) + b^2_2 \\ (w^2_{31}x_1 + w^2_{32}x_2 + w^2_{33}x_3) + b^2_2 \end{bmatrix} = \begin{bmatrix} (w^2_{11}x_1 + w^2_{12}x_2 + w^2_{13}x_3) \\ (w^2_{21}x_1 + w^2_{22}x_2 + w^2_{23}x_3) \\ (w^2_{31}x_1 + w^2_{32}x_2 + w^2_{33}x_3) \end{bmatrix} + \begin{bmatrix} b^2_1 \\ b^2_2 \\ b^2_2 \end{bmatrix} = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \\ w^2_{21} & w^2_{22} & w^2_{23} \\ w^2_{31} & w^2_{32} & w^2_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b^2_1 \\ b^2_2 \\ b^2_3 \end{bmatrix} = \mathbf{w^2 x + b^2}$$

Matrix form of NN



$$\mathbf{w^2} = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \\ w^2_{21} & w^2_{22} & w^2_{23} \\ w^2_{31} & w^2_{32} & w^2_{33} \end{bmatrix} \qquad \mathbf{w^3} = \begin{bmatrix} w^3_{11} & w^3_{12} & w^3_{13} \end{bmatrix}$$

$$\mathbf{z^2} = \begin{bmatrix} (w^2_{11}x_1 + w^2_{12}x_2 + w^2_{13}x_3) + b^2_1 \\ (w^2_{21}x_1 + w^2_{22}x_2 + w^2_{23}x_3) + b^2_2 \\ (w^2_{31}x_1 + w^2_{32}x_2 + w^2_{33}x_3) + b^2_2 \end{bmatrix} = \begin{bmatrix} (w^2_{11}x_1 + w^2_{12}x_2 + w^2_{13}x_3) \\ (w^2_{21}x_1 + w^2_{22}x_2 + w^2_{23}x_3) \\ (w^2_{31}x_1 + w^2_{32}x_2 + w^2_{33}x_3) \end{bmatrix} + \begin{bmatrix} b^2_1 \\ b^2_2 \\ b^2_2 \end{bmatrix} = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \\ w^2_{21} & w^2_{22} & w^2_{23} \\ w^2_{31} & w^2_{32} & w^2_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b^2_1 \\ b^2_2 \\ b^2_3 \end{bmatrix} = \mathbf{w^2 x + b^2}$$

$$\boxed{\mathbf{a^2} = \sigma(\mathbf{w^2 x + b^2})}$$

Artificial Intelligence

Matrix form

- General equation:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

- σ is said to be "vectorized"
- Logit (weighted input) vector z^l is important, too

$$z^l = w^l a^{l-1} + b^l$$

- So $a^l = \sigma(z^l)$

Learning in feedforward ANN

- Want to adjust each weight so that NN has less error
- Have to define *error*
- Have to:
    - Determine how change in a each weight $\rightarrow$ change in error
    - Adjust the weight so as to minimize error

What are we learning?

- Network computes function of inputs
- Single output, n inputs $\mathbf{w}$: $h_{\mathbf{w}}(\mathbf{X})$
- What if $m > 1$ outputs?
  - Single layer net: separate into m nets, train separately
  - Multilayer: all outputs depend on hidden layer weights
  - $\Rightarrow$ vector function
- Output function $\mathbf{h}_{\mathbf{w}}(\mathbf{x})$:

What are we learning?

- Network computes function of inputs
- Single output, n inputs $\mathbf{w}$: $h_{\mathbf{w}}(\mathbf{X})$
- What if $m > 1$ outputs?
  - Single layer net: separate into m nets, train separately
  - Multilayer: all outputs depend on hidden layer weights
  - $\Rightarrow$ vector function
- Output function $\mathbf{h_w}(\mathbf{x})$:

$$\mathbf{h_w}(\mathbf{x}) \quad = \quad \mathbf{a}^L = \sigma(\mathbf{w}^L\mathbf{a}^{\mathbf{l-1}} + \mathbf{b}^L)$$

What are we learning?

- Network computes function of inputs
- Single output, n inputs $\mathbf{w}$: $h_{\mathbf{w}}(\mathbf{X})$
- What if $m > 1$ outputs?
  - Single layer net: separate into m nets, train separately
  - Multilayer: all outputs depend on hidden layer weights
  - $\Rightarrow$ vector function
- Output function $\mathbf{h}_{\mathbf{w}}(\mathbf{x})$:

$$\begin{aligned}
\mathbf{h}_{\mathbf{w}}(\mathbf{x}) &= \mathbf{a}^L = \sigma(\mathbf{w}^L\mathbf{a}^{\mathbf{l-1}} + \mathbf{b}^L) \\
&= \sigma(\mathbf{w}^L(\sigma(\mathbf{w}^{l-1}\mathbf{a}^{\mathbf{l-2}} + \mathbf{b}^{l-1}) + \mathbf{b}^L)
\end{aligned}$$

What are we learning?

- Network computes function of inputs
- Single output, n inputs $\mathbf{w}$: $h_{\mathbf{w}}(\mathbf{X})$
- What if $m > 1$ outputs?
  - Single layer net: separate into m nets, train separately
  - Multilayer: all outputs depend on hidden layer weights
  - $\Rightarrow$ vector function
- Output function $\mathbf{h}_{\mathbf{w}}(\mathbf{x})$:

$$
\begin{aligned}
\mathbf{h}_{\mathbf{w}}(\mathbf{x}) &= \mathbf{a}^L = \sigma(\mathbf{w}^L \mathbf{a}^{\mathbf{l-1}} + \mathbf{b}^L) \\
&= \sigma(\mathbf{w}^L(\sigma(\mathbf{w}^{l-1}\mathbf{a}^{\mathbf{l-2}} + \mathbf{b}^{l-1}) + \mathbf{b}^L) \\
\ldots \\
&= \sigma(\mathbf{w}^L(\sigma(\cdots\sigma(\mathbf{w}^2\mathbf{x} + b^2)\cdots)) + \mathbf{b}^L)
\end{aligned}
$$

Error function

- First, let's eliminate **b** $\Rightarrow$ into **x**
- Error of network:
  - Let **y** = desired output
  - Error on training example **x**:

$$\mathbf{E}_{\mathbf{w}}(\mathbf{x}) = \mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})$$

- But:
  - $\mathbf{E}_{\mathbf{w}}(\mathbf{x})$: positive/negative
  - We don't want any particular error element: want *average* error
  - Want to learn weights, so want a function of weights

Cost (loss) function

- Define a *cost* (loss, objective) function:

$$C_{\mathbf{x}}(\mathbf{w}) = \frac{1}{2}||(\mathbf{E}_{\mathbf{w}}(\mathbf{x}))||^2$$

Cost (loss) function

- Define a *cost* (loss, objective) function:

$$
\begin{aligned}
C_{\mathbf{x}}(\mathbf{w}) &= \frac{1}{2}||(\mathbf{E}_{\mathbf{w}}(\mathbf{x}))||^2 \\
&= \frac{1}{2}||\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})||^2
\end{aligned}
$$

Cost (loss) function

- Define a *cost* (loss, objective) function:

$$
\begin{aligned}
C_{\mathbf{x}}(\mathbf{w}) &= \frac{1}{2}||(\mathbf{E_w}(\mathbf{x}))||^2 \\
&= \frac{1}{2}||\mathbf{y} - \mathbf{h_w}(\mathbf{x})||^2 \\
&= \frac{1}{2}\sum_m (y_m - a_m^L)^2
\end{aligned}
$$

Cost (loss) function

- Define a *cost* (loss, objective) function:

$$
\begin{aligned}
C_{\mathbf{x}}(\mathbf{w}) &= \frac{1}{2}||(\mathbf{E}_{\mathbf{w}}(\mathbf{x}))||^2 \\
&= \frac{1}{2}||\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})||^2 \\
&= \frac{1}{2}\sum_m (y_m - a_m^L)^2
\end{aligned}
$$

- $C_{\mathbf{x}}(\mathbf{w})$: *quadratic cost (MSE) function*
- Entire cost function: average over all $\mathbf{x}_i$:

Cost (loss) function

- Define a *cost* (loss, objective) function:

$$
\begin{aligned}
C_{\mathbf{x}}(\mathbf{w}) &= \frac{1}{2}||(\mathbf{E}_{\mathbf{w}}(\mathbf{x}))||^2 \\
&= \frac{1}{2}||\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})||^2 \\
&= \frac{1}{2}\sum_m (y_m - a_m^L)^2
\end{aligned}
$$

- $C_{\mathbf{x}}(\mathbf{w})$: *quadratic cost (MSE) function*
- Entire cost function: average over all $\mathbf{x}_i$:

$$
C(\mathbf{w}) = \frac{1}{n}\sum_i C_{\mathbf{x}_i}(\mathbf{w})
$$

Cost (loss) function

- Define a *cost* (loss, objective) function:

$$C_{\mathbf{x}}(\mathbf{w}) = \frac{1}{2}||(\mathbf{E}_{\mathbf{w}}(\mathbf{x}))||^2$$
$$= \frac{1}{2}||\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})||^2$$
$$= \frac{1}{2}\sum_m (y_m - a_m^L)^2$$

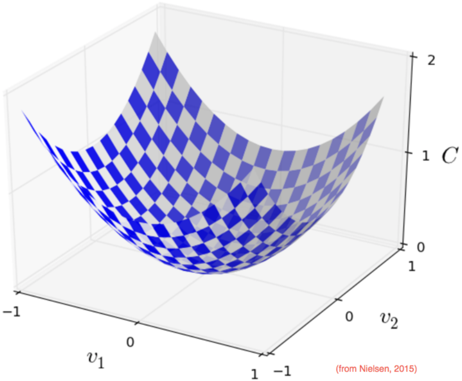
- $C_{\mathbf{x}}(\mathbf{w})$: *quadratic cost (MSE) function*
- Entire cost function: average over all $\mathbf{x}_i$:

$$C(\mathbf{w}) = \frac{1}{n}\sum_i C_{\mathbf{x}_i}(\mathbf{w})$$

- Always positive, $\to 0$ as output $\to \mathbf{y}$

Minimizing cost function

- If we minimize C, minimize $||\mathbf{E}||$
- Using calculus, can find analytical solution
- But with n weights, $n+1$-dimensional curve
- E.g., two dimension:



- Largest nets: *billions* of weights

Review: Gradient descent search

- *Gradient descent search* instead of analytical solution
- Find *local gradients* wrt weights
- $\Rightarrow$ n *partial derivatives* of C
- Take a small step in direction of decrease in *all* the derivatives
- Repeat until close enough to minimum

What is the local gradient?

- For simplicity: two variables, v_1, v_2

What is the local gradient?

- For simplicity: two variables, v_1, v_2
- Then:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

What is the local gradient?

- For simplicity: two variables, v_1, v_2
- Then:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- Let $\Delta \mathbf{v} = [\Delta v_1 \ \Delta v_2]$

What is the local gradient?

- For simplicity: two variables, v_1, v_2
- Then:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- Let $\Delta \mathbf{v} = [\Delta v_1 \ \Delta v_2]$
- Then *gradient* of C is:

$$\nabla C = \left[\frac{\partial C}{\partial v_1} \ \frac{\partial C}{\partial v_2}\right]$$

What is the local gradient?

- For simplicity: two variables, v_1, v_2
- Then:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- Let $\Delta \mathbf{v} = [\Delta v_1 \ \Delta v_2]$
- Then *gradient* of C is:

$$\nabla C = \left[\frac{\partial C}{\partial v_1} \ \frac{\partial C}{\partial v_2}\right]$$

- Thus $\Delta C \approx \nabla C \cdot \Delta \mathbf{v}$

Updating the variables

- Given that:
  - $\Delta C \approx \nabla C \cdot \Delta \mathbf{v}$
  - We want to minimize ΔC
- How shall we choose $\Delta \mathbf{v}$?
- Want any change in $\Delta \mathbf{v}$ to cause ΔC to be negative

Choosing $\Delta\mathbf{v}$

$$\Delta C \approx \nabla C \cdot \Delta\mathbf{v}$$

- Suppose we choose $\Delta\mathbf{v}$ like this:

$$\Delta\mathbf{v} = -\eta\nabla C$$

- The

$$\begin{aligned} \Delta C &\approx \nabla C \cdot -\eta\nabla C \\ &= -\eta(\nabla C \cdot \nabla C) \\ &= -\eta\Sigma_i c_i c_i = -\eta\Sigma_i c_i^2 \end{aligned}$$

- Since $||\mathbf{A}|| = \sqrt{\Sigma_i a_i^2}$,

$$\Delta C \approx -\eta||\nabla C||^2$$

Choosing $\Delta\mathbf{v}$

$$\Delta C \approx -\eta ||\nabla C||^2$$

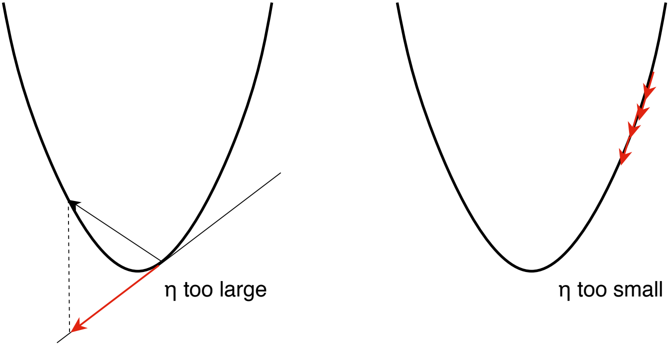
- With $\Delta v = -\eta \nabla C$:
  - Cost function always negative
  - For $||\Delta v|| \leq \epsilon$, minimizes $\nabla C \cdot \Delta v$
- η is learning rate (sometimes α)
- Next value of **v**:

$$\mathbf{v}_{t+1} = \mathbf{v}_t - \eta \nabla C$$

- Now generalize $\mathbf{v} \rightarrow \mathbf{w}$ (including **b**)
- Other gradient descent functions have been tried

Choosing learning rate

- How to choose η?



- If too large $\Rightarrow$ may overshoot minimum
- If too small $\Rightarrow$ will take a very long time to find minimum

Computing gradient

- Difficult
- Cost function: Must compute all C_x then average

$$C = \frac{1}{n}\sum_x C_x = \frac{1}{n}\sum_x \frac{||y(x) - a||^2}{2}$$

- To find overall gradient ∇C:

$$\nabla C = \frac{1}{n}\sum_x \nabla C_x$$

- With many training examples, costly $\Rightarrow$ slow learning

Stochastic gradient descent

- *Stochastic gradient descent*
- Speeds up learning
- Estimate ∇C:
  - Choose small sample of inputs randomly: a *mini-batch*
  - Compute ∇C_x for these to estimate ∇C
- If batch size is large enough, average $\approx \nabla C$
- Idea:
  - Randomly partition training examples into mini-batches
  - Train with each mini-batch
- Doing this: *epoch*
- Repeat until error is satisfactory

Stochastic gradient descent

- *Stochastic gradient descent*
- Speeds up learning
- Estimate ∇C:
  - Choose small sample of inputs randomly: a *mini-batch*
  - Compute ∇C_x for these to estimate ∇C
- If batch size is large enough, average $\approx \nabla C$
- Idea:
  - Randomly partition training examples into mini-batches
  - Train with each mini-batch
- Doing this: *epoch*
- Repeat until error is satisfactory
- Problem: Don't know how to calculate ∇C with hidden layers!

Gradient descent

- Computing the gradient ∇C of the cost function:
  - Composed of $\frac{\partial C}{\partial \mathbf{w}}, \frac{\partial C}{\partial \mathbf{b}}$ – where w, b are vectors
  - May be very difficult to compute

Backpropagation

- *Backpropagation* algorithm (Rumelhart, Hinton, & Williams, 1986)
- Rather than trying to adjust all weights at once, do it by layers
- Compare output layer with target

Backpropagation

- *Backpropagation* algorithm (Rumelhart, Hinton, & Williams, 1986)
- Rather than trying to adjust all weights at once, do it by layers
- Compare output layer with target
- Compute error, use it to update weights from previous hidden layer to output layer

Backpropagation

- *Backpropagation* algorithm (Rumelhart, Hinton, & Williams, 1986)
- Rather than trying to adjust all weights at once, do it by layers
- Compare output layer with target
- Compute error, use it to update weights from previous hidden layer to output layer
- Now propagate error in expected outputs of hidden layer backward, etc.

Backpropagation

- *Backpropagation* algorithm (Rumelhart, Hinton, & Williams, 1986)
- Rather than trying to adjust all weights at once, do it by layers
- Compare output layer with target
- Compute error, use it to update weights from previous hidden layer to output layer
- Now propagate error in expected outputs of hidden layer backward, etc.
- Propagate by dividing responsibility for error at neuron in l according to contribution from each neuron in $l-1$

Error in output layer

- First define vector δ^{L}, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

where:

Error in output layer

- First define vector δ^{L}, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

where:

  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output

Error in output layer

- First define vector δ^{L}, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

where:

  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
  - $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$

Error in output layer

- First define vector δ^{L}, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

where:

  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
  - $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$
  - z_j^L: weighted input to j

Error in output layer

- First define vector δ^L, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

where:

- $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
- $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$
- z_j^L: weighted input to j
- Thus $\sigma'(z_j^L)$ is how fast σ is changing at z_j^L

Error in output layer

- First define vector δ^L, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

  where:
  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
  - $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$
  - z_j^L: weighted input to j
  - Thus $\sigma'(z_j^L)$ is how fast σ is changing at z_j^L
- δ^L is a measure of error at L

Error in output layer

- First define vector δ^{L}, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

  where:
  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
  - $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$
  - z_j^L: weighted input to j
  - Thus $\sigma'(z_j^L)$ is how fast σ is changing at z_j^L
- δ^L is a measure of error at L
- z_j^L already computed, $\sigma'(z_j^L)$ easy to compute

Error in output layer

- First define vector δ^{L}, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

  where:
  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
  - $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$
  - z_j^L: weighted input to j
  - Thus $\sigma'(z_j^L)$ is how fast σ is changing at z_j^L
- δ^L is a measure of error at L
- z_j^L already computed, $\sigma'(z_j^L)$ easy to compute
- $\frac{\partial C}{\partial a_j^L}$ for quadratic cost function: $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$

Artificial Intelligence

Error in output layer

- First define vector δ^L, where for element j:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$$

  where:
  - $\frac{\partial C}{\partial a_j^L}$: how fast the cost function is changing due to j's output
  - $\sigma'(\cdot)$: 1st deriv. of $\sigma(\cdot)$
  - z_j^L: weighted input to j
  - Thus $\sigma'(z_j^L)$ is how fast σ is changing at z_j^L
- δ^L is a measure of error at L
- z_j^L already computed, $\sigma'(z_j^L)$ easy to compute
- $\frac{\partial C}{\partial a_j^L}$ for quadratic cost function: $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$
- So for quadratic: $\delta_j^L = (a_j^L - y_j)\sigma'(z_j^L)$

Hadamard product

- Need a new operator to simplify expressions
- Define *Hadamard product* as: $\mathbf{s} \odot \mathbf{t} = \mathbf{h}$ s.t. $h_j = s_j \times t_j$
- I.e., elementwise product – e.g.:
  $$\begin{bmatrix} -2 \\ 20 \\ 3 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -6 \\ 40 \\ 3 \end{bmatrix}$$

Error in output layer

- $\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L)$
- Can be rewritten as:

$$\delta^L = \nabla_a C \odot \sigma'(\mathbf{z}^L)$$

- Or

$$\delta^L = (\mathbf{a}^L - \mathbf{y}) \odot \sigma'(\mathbf{z}^L)$$

Finding previous layer's error

- If we know δ^{l+1}, can we find δ^l?
- $(\mathbf{w^{l+1}})^T$ = transpose of weight matrix into $l+1$
- $(\mathbf{w^{l+1}})^T\delta^{l+1}$:
  - Moves error backward
  - Gives measure of error at layer l
- Then
$$\delta^l = ((\mathbf{w}^{l+1})^T\delta^{l+1}) \odot \sigma'(\mathbf{z}^l)$$
- Can now compute the error at any layer

Rate of change for biases, weights

- For any weight in the network:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

- For any bias in the network:

$$\frac{\partial C}{\partial b^l_j} = \delta^L_j$$

since "activation" for any bias is just 1

Backpropagation & gradient descent

- For each $x \in m$ training examples:

Backpropagation & gradient descent

- For each $x \in m$ training examples:
  - Feedforward: for each layer l, compute:

Backpropagation & gradient descent

- For each x $\in$ *m* training examples:
  - Feedforward: for each layer *l*, compute:
    - $\mathbf{z^{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$

Backpropagation & gradient descent

- For each x $\in$ *m* training examples:
  - Feedforward: for each layer *l*, compute:
    - $\mathbf{z}^{\mathbf{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$

Backpropagation & gradient descent

- For each x $\in$ *m* training examples:
  - Feedforward: for each layer *l*, compute:
    - $\mathbf{z^{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:

Backpropagation & gradient descent

- For each x $\in$ m training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x},\mathbf{l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$

Backpropagation & gradient descent

- For each x $\in m$ training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:

Artificial Intelligence

Backpropagation & gradient descent

- For each x $\in$ m training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x},\mathbf{l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:
    - $\delta^{x,l} = ((\mathbf{w}^{l+1})^T \delta^{x,l+1}) \odot \sigma'(\mathbf{z}^{x,l})$

Backpropagation & gradient descent

- For each $x \in m$ training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:
    - $\delta^{x,l} = ((\mathbf{w}^{l+1})^T \delta^{x,l+1}) \odot \sigma'(\mathbf{z}^{x,l})$
- Gradient descent: For each layer from L $\rightarrow$ 2:

Backpropagation & gradient descent

- For each x $\in$ m training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:
    - $\delta^{x,l} = ((\mathbf{w}^{l+1})^T \delta^{x,l+1}) \odot \sigma'(\mathbf{z}^{x,l})$
- Gradient descent: For each layer from L $\rightarrow$ 2:
  - Next $\mathbf{w}^l = \mathbf{w}^l - \frac{\eta}{m} \sum_x \delta^{x,l} (\mathbf{a}^{x,l-1})^T$

Backpropagation & gradient descent

- For each $x \in m$ training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:
    - $\delta^{x,l} = ((\mathbf{w}^{l+1})^T \delta^{x,l+1}) \odot \sigma'(\mathbf{z}^{x,l})$
- Gradient descent: For each layer from $L \rightarrow 2$:
  - Next $\mathbf{w}^l = \mathbf{w}^l - \dfrac{\eta}{m} \sum_x \delta^{x,l} (\mathbf{a}^{x,l-1})^T$
  - Next $\mathbf{b}^l = \mathbf{b}^l - \dfrac{\eta}{m} \sum_x \delta^{x,l}$

Backpropagation & gradient descent

- For each x $\in m$ training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x,l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:
    - $\delta^{x,l} = ((\mathbf{w}^{l+1})^T \delta^{x,l+1}) \odot \sigma'(\mathbf{z}^{x,l})$
- Gradient descent: For each layer from L $\rightarrow$ 2:
  - Next $\mathbf{w}^l = \mathbf{w}^l - \frac{\eta}{m} \sum_x \delta^{x,l} (\mathbf{a}^{x,l-1})^T$
  - Next $\mathbf{b}^l = \mathbf{b}^l - \frac{\eta}{m} \sum_x \delta^{x,l}$

Backpropagation & gradient descent

- For each x $\in$ m training examples:
  - Feedforward: for each layer l, compute:
    - $\mathbf{z}^{\mathbf{x},\mathbf{l}} = \mathbf{w}^l \mathbf{a}^{x,l-1} + \mathbf{b}^l$
    - $\mathbf{a}^{x,l} = \sigma(\mathbf{z}^{x,l})$
  - Compute the output error:
    - $\delta^{x,L} = \nabla_a C_x \odot \sigma'(\mathbf{z}^{x,L})$
  - Backpropagate error for each layer l:
    - $\delta^{x,l} = ((\mathbf{w}^{l+1})^T \delta^{x,l+1}) \odot \sigma'(\mathbf{z}^{x,l})$
- Gradient descent: For each layer from L $\to$ 2:
  - Next $\mathbf{w}^l = \mathbf{w}^l - \dfrac{\eta}{m} \sum_x \delta^{x,l} (\mathbf{a}^{x,l-1})^T$
  - Next $\mathbf{b}^l = \mathbf{b}^l - \dfrac{\eta}{m} \sum_x \delta^{x,l}$

Do for some # of epochs, some # mini-batches each.

Backprop algorithm

function BACK-PROP-LEARNING(*examples*, *network*) **returns** a neural network
  inputs: *examples*, a set of examples, each with input vector $\mathbf{x}$ and output vector $\mathbf{y}$
    network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
  local variables: Δ, a vector of errors, indexed by network node

  repeat
    for each weight $w_{i,j}$ in *network* **do**
      $w_{i,j} \leftarrow$ a small random number
    for each example $(\mathbf{x}, \mathbf{y})$ in *examples* **do**
      / Propagate the inputs forward to compute the outputs */*
      for each node i in the input layer **do**
        $a_i \leftarrow x_i$
      for $\ell = 2$ **to** L **do**
        for each node j in layer ℓ **do**
          $in_j \leftarrow \sum_i w_{i,j} a_i$
          $a_j \leftarrow g(in_j)$
      / Propagate deltas backward from output layer to input layer */*
      for each node j in the output layer **do**
        $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
      for $\ell = L - 1$ **to** 1 **do**
        for each node i in layer ℓ **do**
          $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$
      / Update every weight in network using deltas */*
      for each weight $w_{i,j}$ in *network* **do**
        $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
  until some stopping criterion is satisfied
  return *network*

Speed of backprop

- What if we had instead done what we did in HC?
  - For each timestep, look at small changes in the weights
  - Pick set that decreases error
- Could do this for each weight separately, too
- If we have millions of weights, requires *millions* of passes through network
- With backprop: one forward, one backward pass, no matter how many weights

Deep learning

- Networks with $\geq$ 2 hidden layers are *deep networks*

Deep learning

- Networks with $\geq$ 2 hidden layers are *deep networks*
- Backprop will still work for them

Deep learning

- Networks with ≥ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:

Deep learning

- Networks with ≥ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:
  - Tend to lose the error “signal” as propagate back through network

Deep learning

- Networks with $\geq$ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:
  - Tend to lose the error “signal” as propagate back through network
  - Each neuron in earlier layers have less and less impact on output error

Deep learning

- Networks with $\geq$ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:
  - Tend to lose the error "signal" as propagate back through network
  - Each neuron in earlier layers have less and less impact on output error
  - *Vanishing gradient problem*

Deep learning

- Networks with ≥ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:
  - Tend to lose the error "signal" as propagate back through network
  - Each neuron in earlier layers have less and less impact on output error
  - *Vanishing gradient problem*
- $\Rightarrow$ *extremely* slow learning rate

Deep learning

- Networks with ≥ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:
  - Tend to lose the error “signal” as propagate back through network
  - Each neuron in earlier layers have less and less impact on output error
  - *Vanishing gradient problem*
- $\Rightarrow$ *extremely* slow learning rate
- Can have opposite problem, depending on net: *exploding gradient problem*

Deep learning

- Networks with ≥ 2 hidden layers are *deep networks*
- Backprop will still work for them
- But:
  - Tend to lose the error "signal" as propagate back through network
  - Each neuron in earlier layers have less and less impact on output error
  - *Vanishing gradient problem*
- $\Rightarrow$ *extremely* slow learning rate
- Can have opposite problem, depending on net: *exploding gradient problem*
- Stymied researchers for many years

Deep learning

- So what changed?
  - Faster machines
  - Other activation functions
  - Better versions of backprop-ish algorithms invented
  - Other kinds of networks
- Examples of other networks:
  - Convolutional neural networks
  - LSTM
- Led to tremendous increase in deep learning research, applications

- Front-end to TensorFlow, Theano, MS learning systems
- Static networks (though some dynamic work coming along)
- Use of GPUs automatically
- Choice of back-end
- Easy to use – automates a lot, more declarative

- Dynamic networks
- Easier to play around with, debug
- Very "Python-like" – OO, more imperative

Autoencoders

- Particular kind of neural network for unsupervised learning
- Input layer, ≥ 1 hidden layer, output layer
- Error is computed by comparing output to input
- Goal is to reconstruct the input: i.e., learn the identity function
- Why?

Artificial Intelligence

Autoencoders

- Key is that hidden layer(s) have fewer neurons than input/output
- Network learns a compressed version of input: dimensionality reduction
- Hidden layers: features discovered by network during training
- Useful for:
  - Learning features of input examples
  - Denoising input
  - Information compression
  - Information retrieval:
    - Train to produce reduced low-dimension *binary code* in internal layer(s)
    - Use that code as hash key for information
    - Can $\Rightarrow$ very efficient retrieval

Restricted Boltzmann machines

- Boltzmann machines learn probability distributions over inputs
- Layers of neurons, but undirected links
- Restricted Boltzmann machines [Smolensky, 1986]: no intra-layer links
- First fast deep-learning algorithms [Hinton – mid-2000s]:
  - Treat first hidden layer as RBM – train it.
  - Treat second hidden layer as RBM with inputs from first hidden layer, train it.
  - Etc.
  - Fine-tune with backprop/gradient descent

Neural network training:

"You process a lot of data in a quiet way, don't you, Larry!"

. . . and. . .

You'd like to ask Roy if he's really thought this through.

Preview

- Convolutional NNs: when covering image recognition
- LSTM: when looking at NLP
- GANs: when looking at creativity