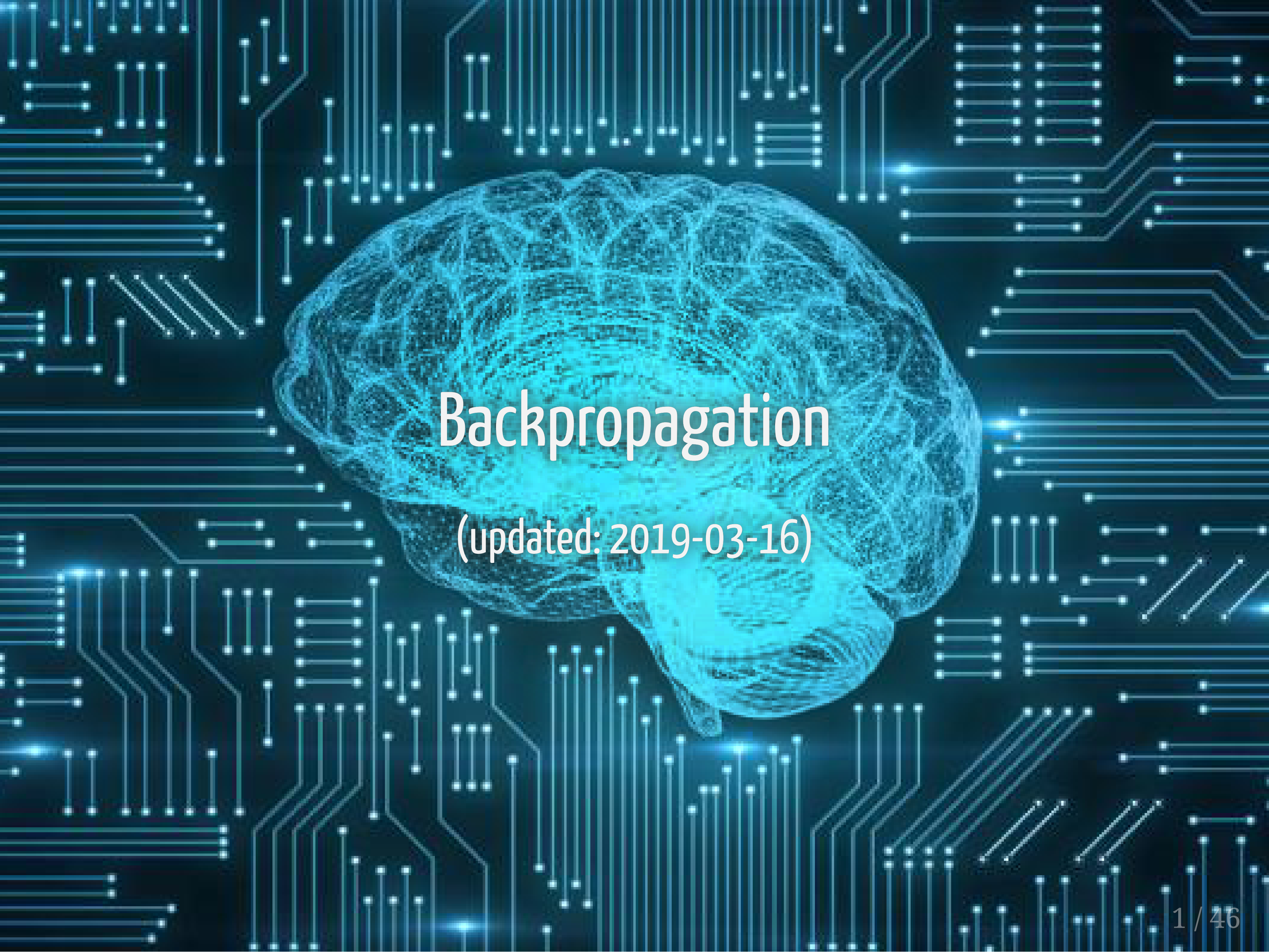


Backpropagation

(updated: 2019-03-16)

Class 7

One Hidden Layer Neural Networks

- Derive full backpropagation equations for deep network
- Introduce additional forms of gradient descent
  - Gradient descent with momentum
  - Nesterov momentum
  - Adaptive gradient methods
- Activation functions
  - sigmoid, tanh, ReLU, Leaky ReLU, softplus, swish
- Learning rate schedules

Neural Networks

We've covered 3 networks with inputs directly connected to the output(s):

(1) Linear regression

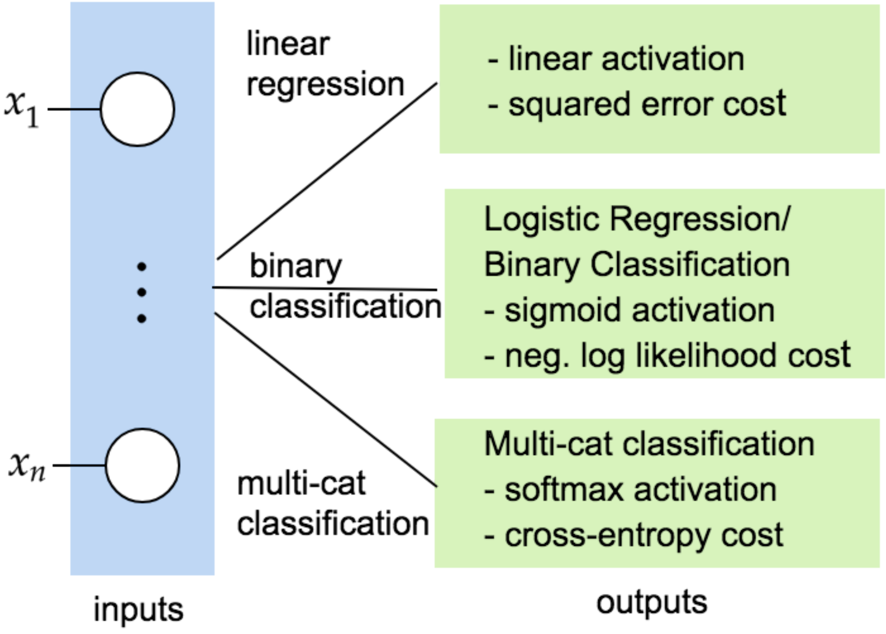
$C = \frac{1}{2}\|\hat{y} - t\|^2$

(2) Logistic regression/binary classification

$C = -t \ln \hat{y} - (1-t) \ln(1-\hat{y})$

(3) Multicategory classification

$C = -\sum_k t_k \ln \hat{y}_k$



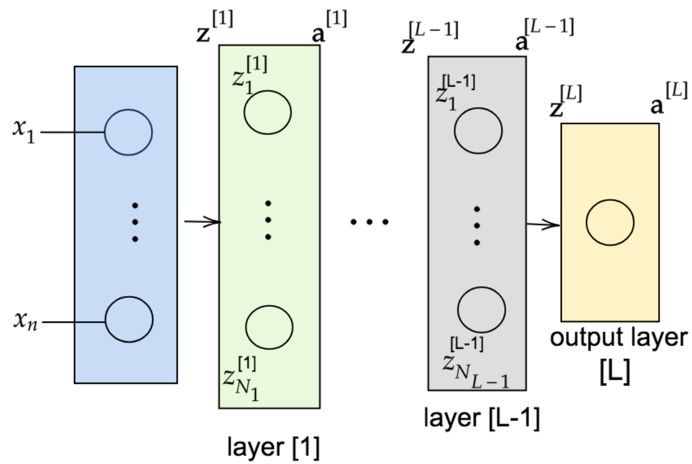
Neural Networks

No Hidden Layers When inputs are directly connected to the output there is only one set of weights $\mathbf{w}$ and biases $\mathbf{b}$.

- Gradients can easily be derived from the cost functions

Hidden Layers Adding a layer of nodes between the inputs and outputs gives the network **universal** properties.

- It also greatly complicates the calculation of the cost function gradients in terms of the now multiple sets (per layer) of weights and biases.



Biases, Weights

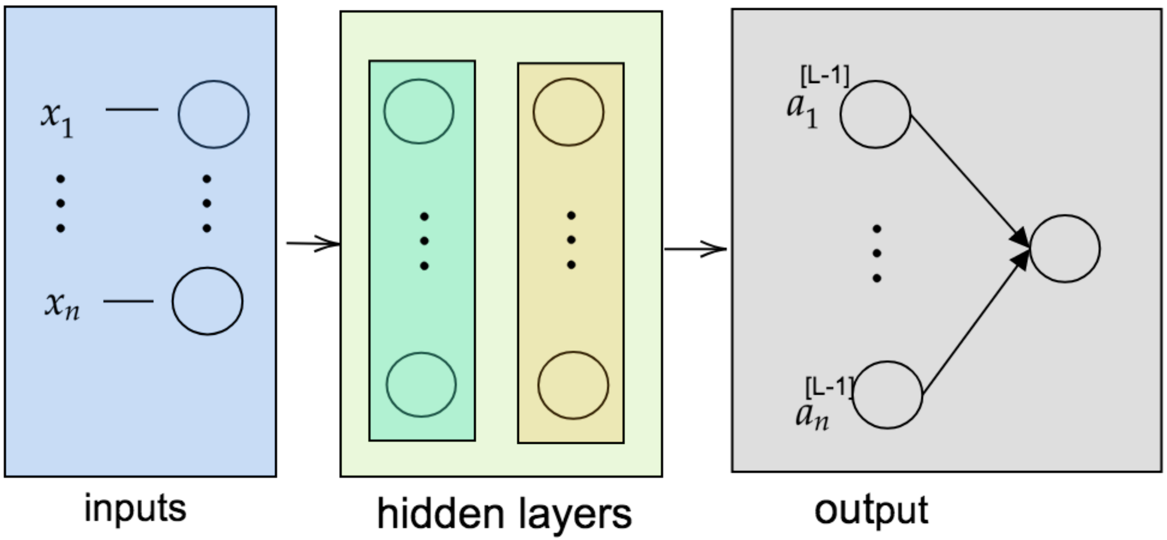
$$N_b = N_1 + \ldots + N_L$$
$$N_w = nN_1 + N_1N_2 \ldots + N_{L-1}N_L$$

Example MNIST input, 100 hidden nodes

$$100 + 10 + 784 * 100 + 100 * 10 = 79,510$$

Neural Networks

Another (powerful) view.



The inputs and hidden layers act to pre-process the inputs so output layer is more effective.

- The preprocessing could be done by a previously built model
- Preprocessing using unsupervised training is very effective.

Neural Networks

Counting layers

- Will use the # of adjustable parameter sets as the # of layers.
- Perceptron is a single layer network.
- A network with 1-hidden layer is then a 2-layer network.
- Specifying the number of hidden layers is probably clearest way to describe a network.
- $N_0/N_1/\ldots/N_L$

  N_0 = # inputs

  N_1 = # nodes in 1st hidden layer
  $\vdots$
  N_L = nodes in output layer

Backpropagation

Neural networks are trained using a form of gradient descent called mini-batch gradient descent. The (mini-batch) gradients are computed using "Backpropagation."

Backpropagation

- Efficient (using matrix algebra) calculation of gradients is one of the main reasons why neural networks are effective.

- The layer-by-layer structure of the feed-forward neural networks, along with the backpropagation algorithm make training efficient.

- The availability of low cost array processors (GPU's) has been a huge boost to neural networks.

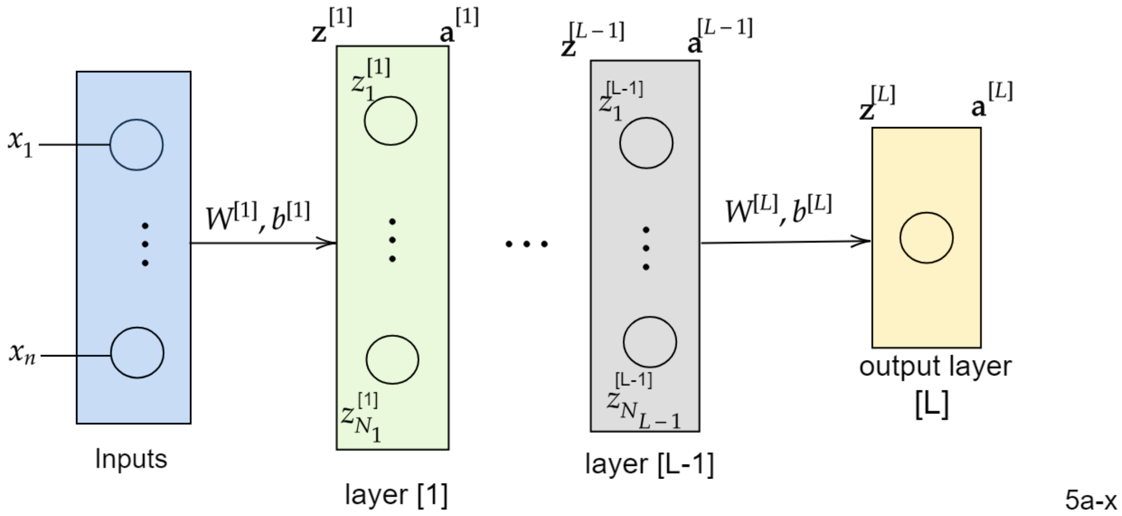
Algorithm Components

- Forward propagation
- Backward propagation (backprop)
- Weight updates (all adjustable parameters)

Backpropagation

For layer activation function f and output activation g

$$
\begin{aligned}
C(\hat{y}, t) &= C(g(\mathbf{z}^{[L]}), t) \\
g(\mathbf{z}^{[L]}) &= g(f(\mathbf{z}^{[L-1]};\ \mathbf{b}^{[L]}, W^{[L]})) \\
&= g(f(f(\mathbf{z}^{[L-2]};\ \mathbf{b}^{[L-1]}, W^{[L-1]});\ \mathbf{b}^{[L]}, W^{[L]})) \\
&= g(f(f(\ldots f(\mathbf{x}; \mathbf{b}^{[1]}, W^{[1]}))))
\end{aligned}
$$



Backpropagation

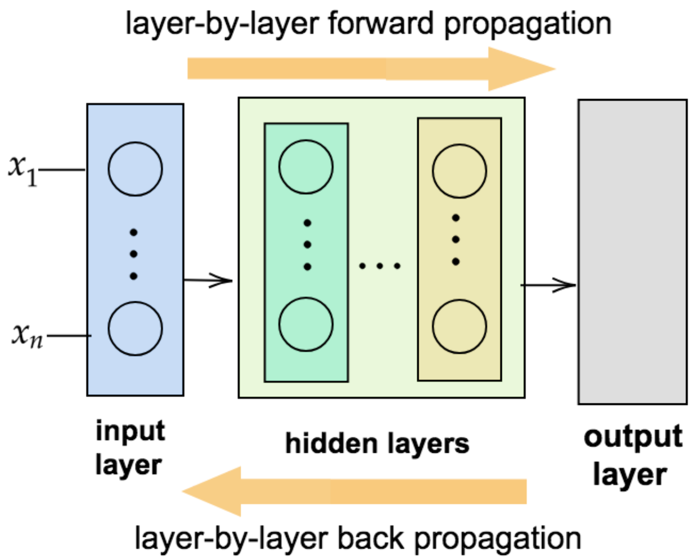
Backpropagation is a natural outgrowth of gradient descent applied to the Perceptron network

- Apparently 'discovered' several times
- Applied to neural networks in 1974 Ph.D. thesis of Paul Werbos.
- Backpropagation popularized through the 1986 work of Rumelhart, Hinton, and Williams[1]
- In 1993, Wan was the first to win an international pattern recognition contest using Backpropagation.[2]
- In 2010, the availability of GPU's led to dramatically improved neural network performance.

[1] Rumelhard et.al

[2] Deep Learning in Neural Networks

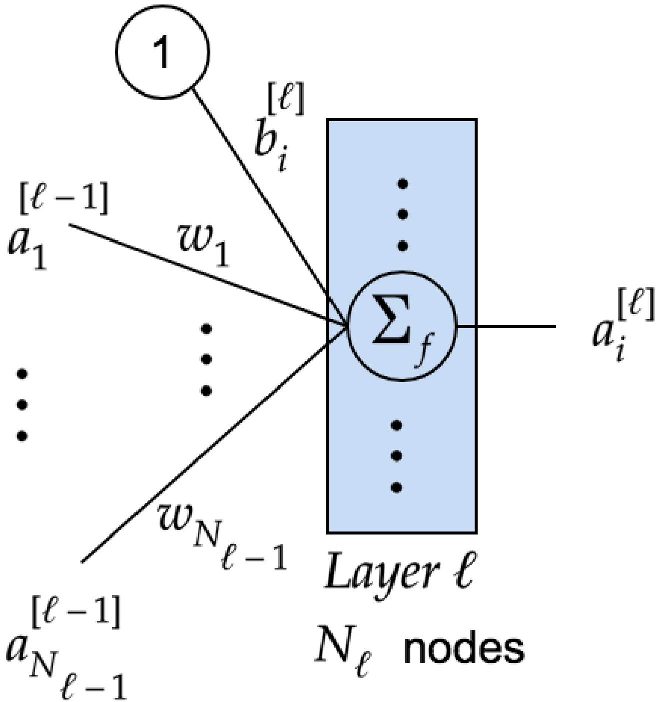
Backpropagation



- Each step (forward/back) uses efficient matrix algebra.
- The backward steps will use values precomputed during forward prop.
- After each forward-back cycle, the adjustable parameters are updated (Weight update).

Notation

Generic i^{th} node in layer 'l'

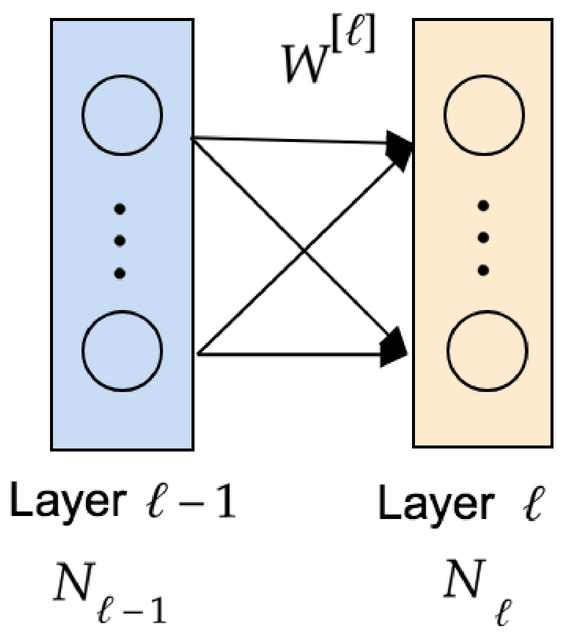


For a single node in layer $'\ell'$

$$z_i^{[\ell]} = \mathbf{b}_i^{[\ell]} + w^{[\ell]} \cdot a^{[l-1]}$$
$$a_i^{[\ell]} = f(z_i^{[\ell]})$$

- Layer ℓ has N_ℓ nodes
- Node input $z_i^{[\ell]}$ is often called the 'net' value
- The activation function acts only on the inputs to that node
- The outputs of the nodes are labelled a because they are activation values

Notation



- Arrange biases into vectors $\mathbf{b}^{[\ell]}$ of length N_ℓ
- Arrange weights for a node as rows in a matrix $W^{[\ell]}$
- $W^{[\ell]}$ maps $N_{(\ell-1)}$ vectors to length N_ℓ vectors. $W^{[\ell]}$ is $N_\ell \times N_{(l-1)}$
- Now all nodes can be computed using:

$$\mathbf{z}^{[\ell]} = \mathbf{b}^{[\ell]} + W^{[\ell]}\mathbf{a}^{[l-1]}$$
$$\mathbf{a}^{[\ell]} = f(\mathbf{z}^{[\ell]})$$

This looks just like the equation for a single node, but now it is a vector/matrix equation

Notation

The activation function for all layers except the last layer acts element-wise:

$$f(\mathbf{z}^{[\ell]}) \triangleq \begin{bmatrix} f(z_1^{[\ell]}) \\ \vdots \\ f(z_{N^{[\ell]}}^{[\ell]}) \end{bmatrix}$$

Note for later. The Hadamard product of two vectors $\mathbf{x} \odot \mathbf{y}$ is element-wise:

$$\mathbf{x} \odot \mathbf{y} = \begin{bmatrix} x_1 y_1 \\ \vdots \\ x_n y_n \end{bmatrix}$$

Unfortunately there is no notation to say a function acts on a vector element by element.

At this point the activation function is not specified, keeping the equations generic

Network Design

A network is defined by:

- The length of the input vector. For MNIST it would be $28 \cdot 28 = 784$.
- The number of hidden layers.
- The number of nodes in each hidden layer.
- The activation function for the hidden layers (usually the same for all).
- The cost function determined by the task.
- The output layer activation typically chosen based on the task/cost function.
- The hidden layer activation functions act element-wise. This is not true for the softmax output activation used for multi-cat classification.

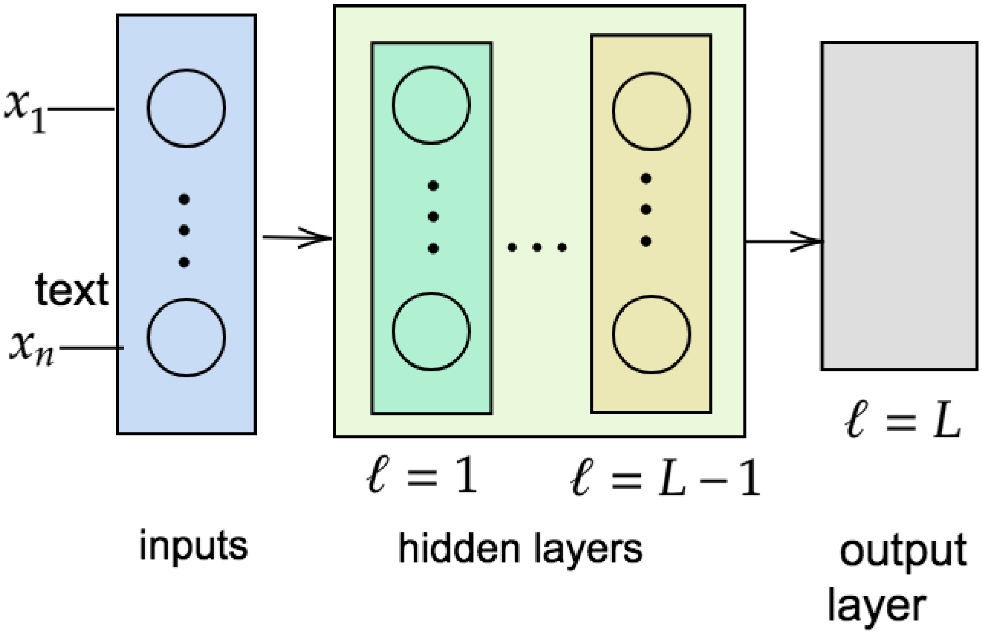
Network Design

pseudo-code to 'create' network?

TBD

Forward Propagation

Forward propagation computes network outputs $\hat{y}$ from input data given values for adjustable parameters.

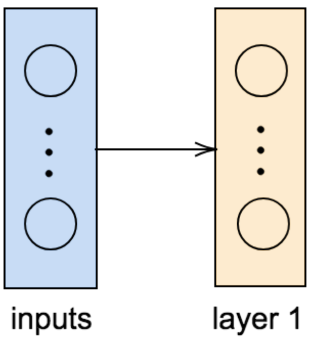


Forward Propagation

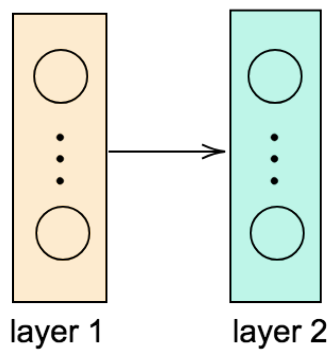
Let layer zero be the input layer. N_0 is the number of inputs and $\mathbf{a}^{[0]}$ is one vector of input data. Technically, the i^{th} input vector is $\mathbf{a}^{[0](i)}$.

Forward propagation computes net-values and activation values layer by layer

Step 1: compute $\mathbf{z}^{[1]}, \mathbf{a}^{[1]}$



Step 2: compute $\mathbf{z}^{[2]}, \mathbf{a}^{[2]}$



Values of $\mathbf{z}$ and $\mathbf{a}$ should be retained during forward pass for later use in back propagation

Forward propagation algorithm

Let data be $\mathbf{x}^{(i)} \in \mathbb{R}^n, i = 1, \cdots, m$. With labels/tags $t^{(i)}$. For now consider just a single input vector at a time, so drop the i superscript

Initialize $\mathbf{b}, W$ for all layers

Select hidden node activation f and output activation g

{Scale Data}

For layer ℓ in $\{1, \cdots, L-1\}$

$$\mathbf{z}^{[\ell]} = \mathbf{b}^{[\ell]} + W^{[\ell]}\mathbf{a}^{[l-1]}$$
$$\mathbf{a}^{[\ell]} = f(\mathbf{z}^{[\ell]})$$

end

$$\mathbf{z}^{[L]} = \mathbf{b}^{[L]} + W^{[L]}\mathbf{a}^{[L-1]}$$
$$\mathbf{a}^{[L]} = g(\mathbf{z}^{[L]})$$

note: $\mathbf{a}^{[L]}$ is the network output, usually written as $\hat{y}$

Backpropagation

Forward propagation computes the network prediction (scalar or vector). Will use $\mathbf{a}^{[L]}$ and $\hat{\mathbf{y}}$ interchangeably for the network output

Gradient Descent

- Gradient descent requires the gradient of the cost function C w.r.t all adjustable parameters
- At times will use θ as a symbol to represent all of the adjustable parameters

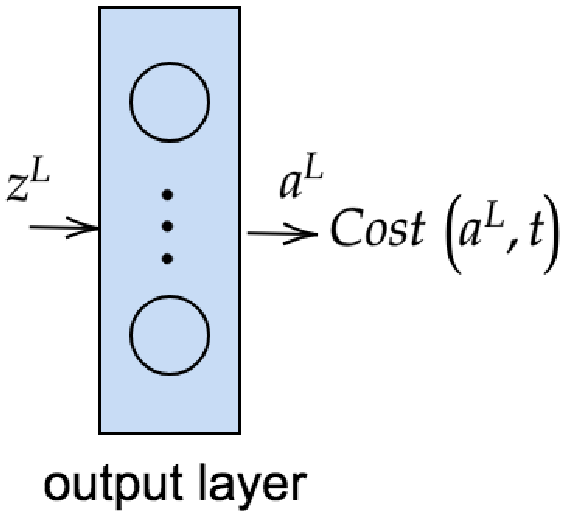
$$\theta = \left\{ \mathbf{b}^{[1]}, W^{[1]}, \cdots, \mathbf{b}^{[L]}, W^{[L]}, \right\}$$

- Cost is an explicit function of the network output and desired response $C = C(\hat{y}^{(i)}, t^{(i)})$
- It is an implicit function of the parameters θ and the $\mathbf{z}^{[\ell]}$ and $\mathbf{a}^{[\ell]}$ values computed during forward prop

Backpropagation

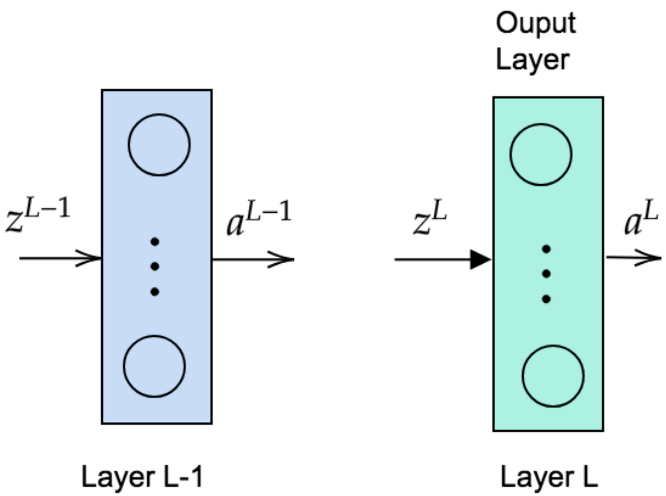
Step 1. Compute

$$\frac{\partial C}{\partial z^{[L]}}$$



Step 2. Compute

$$\frac{\partial C}{\partial z^{[L-1]}} \quad \text{from} \quad \frac{\partial C}{\partial z^{[\ell]}}$$



Gradient descent needs $\partial C/\partial \mathbf{b}$ and $\partial C/\partial W$. These will be computed from $\partial C/\partial \mathbf{z}$

Backpropagation

Equating output $\hat{y}$ with $\mathbf{a}^{[L]}$, write generic cost function partial (ignoring for now that $\hat{y}$ could be a scalar or a vector)

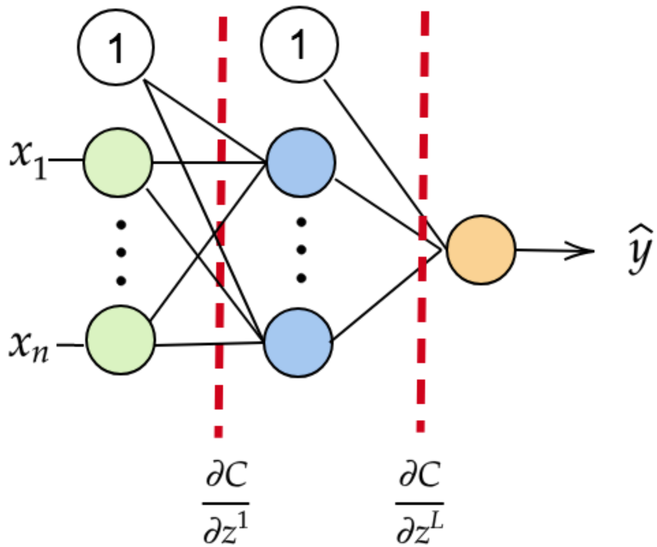
$$\partial C = \frac{\partial C}{\partial \hat{y}} \partial \hat{y}$$

The network output $\hat{y}$ is a function of $\mathbf{z}^{[L]}$ (via output activation) so the chain rule gives

$$\partial C = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \mathbf{z}^{[L]}} \partial \mathbf{z}^{[L]}$$

but $\mathbf{z}^{[L]}$ is a function of $\mathbf{z}^{[L-1]}$ and so the partials work their way up through the layers

Backpropagation



Some authors (Nielsen) use the notation:

$$\delta^\ell = \frac{\partial C}{\partial z^{[\ell]}}$$

Dimensions:

$$\frac{\partial C}{\partial \mathbf{z}^{[\ell]}} \quad \text{has length} \quad N_\ell$$

Will find:

$$\frac{\partial C}{\partial \mathbf{b}^{[\ell]}} = \frac{\partial C}{\partial \mathbf{z}^{[\ell]}}$$

and

$$\frac{\partial C}{\partial W^{[\ell]}} = \frac{\partial C}{\partial \mathbf{z}^{[\ell]}} (\mathbf{a}^{[l-1]})^T$$

Note that if $\ell = 1$ then

$$\frac{\partial C}{\partial W^{[1]}} = \frac{\partial C}{\partial \mathbf{z}^{[1]}} \mathbf{x}^T$$

Backpropagation

Quick recap of results from earlier notes:

- **Linear regression**

$$C = \frac{1}{2}(\hat{y} - t)^2$$

$$\hat{y} = \mathbf{z}^{[L]}$$

$$\partial C = (\hat{y} - t)\ \partial \mathbf{z}^{[L]}$$

- **Softmax Regression**

$$C = -\sum_k t_k \ln \hat{y}_k$$

$$\hat{y}_k = \frac{e^{\mathbf{z}_k^{[L]}}}{\sum_j e^{\mathbf{z}_j^{[L]}}}$$

$$\partial C = (\hat{y} - \mathbf{t})\partial \mathbf{z}^{[L]}$$

- **Logistic Regression**

$$C = -t \ln \hat{y} - (1 - t) \ln(1 - \hat{y})$$

$$\hat{y} = \sigma(\mathbf{z}^{[L]})$$

$$\partial C = (\hat{y} - t)\partial \mathbf{z}^{[L]}$$

This pattern does not hold when using squared error cost with σ output.

Backpropagation

Gradient descent depends on the partials of the cost w.r.t the adjustable parameters

$$\frac{\partial C}{\partial \mathbf{b}^{[L]}}, \quad \frac{\partial C}{\partial W^{[L]}}$$

Using

$$\mathbf{z}^{[L]} = \mathbf{b}^{[L]} + W^{[L]}\, a^{[L-1]}$$

Gives (note: vector by vector derivative)

$$\frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{b}^{[L]}} = I, \quad (N_L \times N_L)$$

This gives:

$$\frac{\partial C}{\partial \mathbf{b}^{[L]}} = \frac{\partial C}{\partial \mathbf{z}^{[L]}} \frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{b}^{[L]}} = \frac{\partial C}{\partial \mathbf{z}^{[L]}}$$

Backpropagation

Now need:

$$\frac{\partial \mathbf{z}^{[L]}}{\partial W^{[L]}}$$

This is a derivative of a vector by a matrix, so is a 3-dimensional tensor. Fortunately, it is sparse and collapses when multiplied so the full 3-d structure does not need to be computed.

$$\begin{aligned}
\frac{\partial z_i^{[L]}}{\partial W_{j,k}^{[L]}} &= \frac{\partial}{\partial W_{jk}^{[L]}}\Big(\sum_q W_{i,q}^{[L]} a_q^{[L-1]}\Big) \\
&= \sum_q \delta_{i,j}\delta_{q,k} a_q^{[L-1]} \\
&= \delta_{i,j} \sum_q \delta_{q,k} a_q^{[L-1]} \\
&= \delta_{i,j} a_k^{[L-1]}
\end{aligned}$$

$\mathbf{a}^{[L-1]}$ is computed during forward prop. The result has 3 indices but is sparse and will collapse when multiplied

Backpropagation

$\frac{\partial C}{\partial W}$ should have the dimensions of W

$$
\begin{aligned}
\Big(\frac{\partial C}{\partial W^{[L]}}\Big)_{i,j} &= \Big(\frac{\partial C}{\partial \mathbf{z}^{[L]}}\frac{\partial \mathbf{z}^{[L]}}{\partial W^{[L]}}\Big)_{i,j} \\
&= \sum_k \frac{\partial C}{\partial z_k^{[L]}}\frac{\partial z_k^{[L]}}{\partial W_{ij}^{[L]}}) \\
&= \sum_k \frac{\partial C}{\partial z_k^{[L]}}\delta_{ki} a_j^{[L-1]} \\
&= \frac{\partial C}{\partial z_i^{[L]}} a_j^{[L-1]}
\end{aligned}
$$

In vector form

$$
= \begin{bmatrix} \frac{\partial C}{\partial z_1^{[L]}} \\ \vdots \\ \frac{\partial C}{\partial z_{N_L}^{[L]}} \end{bmatrix} \left[a_1^{[L-1]}, \cdots, a_{N_{L-1}}^{[L-1]}\right]
$$

$$
= \frac{\partial C}{\partial \mathbf{z}^{[L]}}(\mathbf{a}^{[L-1]})^T
$$

Calculated partial has dimensions $N_L \times N_{L-1}$

Check: what is the dimension of $W^{[L]}$? $W^{[L]}$ maps $\mathbf{a}^{[L-1]}$ to $\mathbf{z}^{[L]}$ so is $N_L \times N_{L-1}$

Backpropagation

Completed following steps:

1. $\partial C/\partial \mathbf{z}^{[L]}$
2. $\partial C/\partial \mathbf{b}^{[L]}$ from $\partial C/\partial \mathbf{z}^{[L]}$
3. $\partial C/\partial W^{[L]}$ from $\partial C/\partial \mathbf{z}^{[L]}$

Next compute $\partial C/\partial \mathbf{z}^{[L-1]}$ from $\partial C/\partial \mathbf{z}^{[L]}$, then have sequential process to find all needed derivatives:

$$\mathbf{z}^{[L]} = \mathbf{b}^{[L]} + W^{[L]}\mathbf{a}^{[L-1]}$$

$$\mathbf{a}^{[L-1]} = f(\mathbf{z}^{[L-1]})$$

Backpropagation

Given $\partial C/\partial \mathbf{z}^{[L]}$ use chain rule to compute $\partial C/\partial \mathbf{z}^{[L-]}$

First, using index notation

$$\frac{\partial C}{\partial \mathbf{z}_i^{[L-1]}} = \sum_k \sum_j \frac{\partial C}{\partial \mathbf{z}_j^{[L]}} \frac{\partial \mathbf{z}_j^{[L]}}{\partial \mathbf{a}_k^{[L-1]}} \frac{\partial a_k^{[L-1]}}{\partial \mathbf{z}_i^{[L-1]}}$$

Translating to vector notation

$$\frac{\partial C}{\partial \mathbf{z}^{[L-1]}} = \Big(\underbrace{\Big(\frac{\partial C}{\partial \mathbf{z}^{[L]}}\Big)^T}_{vector} \underbrace{\frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{a}^{[L-1]}}}_{matrix}\Big) \underbrace{\frac{\partial \mathbf{a}^{[L-1]}}{\partial \mathbf{z}^{[L-1]}}}_{matrix}$$

Note: Performing vector-matrix product first is more efficient.

Backpropagation

Recall that the activation for all layers except the last is 'component-wise'

$$a_i^{[L-1]} = f(z_i^{[L-1])})$$

$$\frac{\partial \mathbf{a}^{[L-1]}}{\partial \mathbf{z}^{[L-1]}} = \begin{bmatrix} f'(z_1^{[L-1]}) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & f'(z_{N_{L-1}}^{[L-1]}) \end{bmatrix}$$

For diagonal matrix D, AD multiples column i of A by D_{ii} Now just need $\partial z^{[L]} / \partial a^{[L-1]}$

$$\begin{aligned} \frac{\partial z_i^{[L]}}{\partial a_j^{[L-1]}} &= \frac{\partial}{\partial a_j^{[L-1]}} (\sum_k W_{i,k}^{[L]} a_k^{[L-1]}) \\ &= \sum_k W_{i,k}^{[L]} \delta_{j,k} \\ &= W_{i,j}^{[L]} \end{aligned}$$

Putting Pieces Together

$$\frac{\partial C}{\partial \mathbf{z}^{[L-1]}} = \left(\left(\frac{\partial C}{\partial \mathbf{z}^{[L]}}\right)^T \frac{\partial \mathbf{z}^{[L]}}{\partial \mathbf{a}^{[L-1]}}\right) \frac{\partial a^{[L-1]}}{\partial \mathbf{z}^{[L-1]}}$$

First term

$$\sum_i \frac{\partial C}{\partial z_i^{[L]}} W_{ij}^{[L]} = (W^{[L]})^T \frac{\partial C}{\partial z^{[L]}}$$

Again for performance reasons let the vector

$$\mathbf{d} = \mathrm{diag}(\frac{\partial \mathbf{a}^{[L-1]}}{\partial \mathbf{z}^{[L-1]}}) = \begin{bmatrix} f'(z_1^{[L-1]}) \\ \vdots \\ f'(z_{N_{L-1}}^{[L-1]}) \end{bmatrix}$$

Then using the Hadamard product $\odot$ gives:

$$\frac{\partial C}{\partial z^{[L-1]}} = \mathbf{d} \odot \left((W^{[L]})^T \frac{\partial C}{\partial z^{[L]}}\right)$$

Back Propagation Algorithm

For this algorithm, interpret $a^{[0]}$ as the input vector $\mathbf{x}$

First, run forward prop to calculate $\mathbf{z}^{[\ell]}, \mathbf{a}^{[\ell]}$ for $\ell = 1, \ldots, L$

Initialize

$\ell = L$
$\boldsymbol{\delta}^{[\ell]} = \partial C / \partial \mathbf{z}^{[L]}$

while ℓ is ≥ 1

$\partial C / \partial \mathbf{b}^{[\ell]} = \boldsymbol{\delta}^{[\ell]}$
$\partial C / \partial W^{[\ell]} = \boldsymbol{\delta}^{[\ell]} (\mathbf{a}^{[\ell-1]})^T$

if $\ell > 1$

$\mathbf{d}_i = f'(z_i^{[l-1]})$
$\boldsymbol{\delta}^{[\ell]} = \mathbf{d} \odot ((W^{[\ell]})^T \boldsymbol{\delta}^{[\ell]})$

$\ell = \ell - 1$

end

Back Propagation Observations

Observations from the back-prop equations?

- If $W = 0$, then $\delta = 0$ and so all gradients $(\mathbf{b}, W)$ are 0
  - Should not initialize W to zero
- The hidden layer activation derivative f' is a multiplicative factor of δ at each step
- Activation functions like σ have near zero derivative when the argument is not near zero. This is the cause of vanishing gradients which halts the training of deep nets.
  - Use ReLU (or other alternatives) instead

Mini-Batch Processing

Forward prop and Back prop equations were derived for a single data vector $\mathbf{x}^{(i)}$. Now re-derive gradients for a batch of inputs.

Arrange input data and tagging data as matrices

$$X = \left[\, \mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)} \,\right] = \begin{bmatrix} x_1^{(1)} & \cdots & x_1^{(m)} \\ \vdots & & \vdots \\ x_n^{(1)} & \cdots & x_n^{(m)} \end{bmatrix}$$

For a batch of m sample vectors with associated tags $T = \left[t^{(1)}, \cdots, t^{(m)}\right]^T$ for scalar model and

$$T = \begin{bmatrix} t_1^{(1)} & \cdots & t_1^{(m)} \\ \vdots & & \vdots \\ t_k^{(1)} & \cdots & t_k^{(m)} \end{bmatrix}$$

For a model with K class outputs

Mini-Batch Processing

Because $WX = \left[W\mathbf{x}^{(1)}, \cdots, W\mathbf{x}^{(m)}\right]$

The forward propagate equations are essentially unchanged. Basically just replacing lower-case with upper-case

$$Z^{[1]} = \mathbf{b}^{[1]} + W^{[1]}X$$

(no longer $\mathbf{x}^{(i)}$)

Dimensions:

$$\dim(X) = n \times m$$

$$\dim(Z^{[1]}) = N_1 \times m$$

Where N_1 is the number of nodes in the first hidden layer

Mini-Batch Processing - Forward Propagation

Let f be the hidden layer activation function. The outputs of the first hidden layer are computed by applying the activation function to the input net-values $Z^{[1]}$

$$A^{[1]} = f(Z^{[1]}) \qquad \text{applied component-wise}$$

$$A^{[1]} = \begin{bmatrix} f(Z_1^{[1](1)}) & \cdots & f(Z_1^{[1](m)}) \\ \vdots & & \vdots \\ f(Z_{N_1}^{[1](1)}) & \cdots & f(Z_{N_1}^{[1](m)}) \end{bmatrix}$$

Let g be the output layer activation function. Using capital Y for the matrix version of $\hat{y}$

$$Z^{[L]} = b^{[L]} + W^{[L]} A^{[L-1]}$$
$$\hat{Y} = g(Z^{[L]})$$

Mini-Batch Processing

Dimensions

- Feed in m vectors $\mathbf{x}^{(i)} \in \mathbb{R}^n, i = 1, 2, \ldots m$
- Output layer produces $\hat{Y}$
  - $\hat{Y}$ is a $1 \times m$ matrix if output for a single $\mathbf{x}^{(i)}$ is scalar (binary classification or regression)
  - $\hat{Y}$ is a $K \times m$ matrix if output for a single $\mathbf{x}^{(i)}$ is length K (K-way classification)

Batch Backpropogation Algorithm

Cost for a single sample $\mathbf{x}^{(i)}$ is $C^{(i)} = C(\mathbf{x}^{(i)}, \theta)$

For a batch of m samples $C = \frac{1}{m}\sum_{i=1}^{m} C^{(i)}$

It follows that a generic partial can be written:

$\partial C = \frac{1}{m}\sum_{i=1}^{m} \partial C^{(i)}$

Recall single point back-prop:

- Compute $\partial C^{(i)}/\partial \mathbf{z}^{[L]}$ given cost function
- Iterate to get $\partial C^{(i)}/\partial \mathbf{z}^{[\ell]}$ for each $\ell = L-1, \cdots, 1$
- Use $\partial C^{(i)}/\partial \mathbf{z}^{[\ell]}$ to find $\partial C^{(i)}/\partial \mathbf{b}^{[\ell]}, \partial C^{(i)}/\partial W^{[\ell]}$

end

In next few slides will vectorize gradient calculation for mini-batches

Mini-Batch Backpropogation

$C = \frac{1}{m} \sum_{i=1}^{m} C^{(i)}$

The single data sample backpropogation algorithm starts by computing

$$\frac{\partial C^{(i)}}{\partial \mathbf{z}^{[\ell]}}$$

In the following, C is a sum of sample costs, and $Z^{[\ell]}$ is an array of network values computed from input X using forward propagation.

If the batch size is m, will compute B gradients and then average them.

For a single vector $\mathbf{z}^{[L]}$, $\frac{\partial C^{(i)}}{\partial \mathbf{z}^{[L]}}$ is a vector with length N_L . Given array $Z^{[L]}$:

$$Z_{batch}^{[L]} = \begin{bmatrix} z_1^{(1)} & \dots & z_1^{(m)} \\ \vdots & & \vdots \\ z_{N_L}^{(1)} & \dots & z_{N_L}^{(m)} \end{bmatrix}$$

Will compute

$$\frac{\partial C}{\partial Z_{batch}^{[L]}}$$

Mini-Batch Backpropogation

Single sample:

$$\underbrace{\frac{\partial C}{\partial \mathbf{z}^{[l-1]}}}_{vector} = \underbrace{f^{'}(\mathbf{z}^{[l-1]})}_{vector} \odot \underbrace{\left[(W^{[\ell]})^T \frac{\partial C}{\partial \mathbf{z}^{[\ell]}}\right]}_{vector}$$

Batch of samples:

$$\underbrace{\frac{\partial C}{\partial Z^{[l-1]}}}_{Matrix} = f^{'}(Z^{[l-1]}) \odot \left[(W^{[\ell]})^T \frac{\partial C}{\partial Z^{[\ell]}}\right]$$

Note that these are NOT averaged

Mini-batch Backpropagation

Showed earlier that for a single input vector

$$\frac{\partial C}{\partial \mathbf{b}^{[\ell]}} = \frac{\partial C}{\partial \mathbf{z}^{[\ell]}}$$

For a batch of data, will have m such gradients that are averaged

$$\frac{\partial C}{\partial \mathbf{b}^{[\ell]}} = \frac{1}{m} \frac{\partial C}{\partial \mathbf{Z}^{[\ell]}} \underbrace{\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}}_{\text{length } m}$$

Mini-Batch Backpropogation

For single sample

$$\frac{\partial C}{\partial W^{[\ell]}} = \frac{\partial C}{\partial \mathbf{z}^{[\ell]}}\left(\mathbf{a}^{[\ell-1]}\right)^T$$

For a batch of samples

$$\frac{\partial C}{\partial W^{[\ell]}} = \frac{1}{m}\sum_i \frac{\partial C}{\partial \mathbf{z}^{[\ell](i)}}\left(\mathbf{a}^{[\ell-1](i)}\right)^T$$

For batch, have m vectors in $\frac{\partial C}{\partial Z^{[\ell]}}$ and m vectors in $A^{[\ell-1]}$. Justify that the sum above is given by the following matrix product

$$\frac{\partial C}{\partial W^{[\ell]}} = \frac{1}{m}\frac{\partial C}{\partial Z^{[\ell]}}(A^{[\ell-1]})^T$$

Notes:

$Z^{[\ell]}$ is a matrix containing $\mathbf{z}^{[\ell](i)}$ in column i
$A^{[\ell-1]}$ is a matrix containing $\mathbf{a}^{[\ell-1](i)}$ in column i

Mini-batch Backpropagation

For a neural network with depth L, the adjustable parameters are

$$\left\{\mathbf{b}^{[1]}, W^{[1]}, \dots \mathbf{b}^{[L]}, W^{[L]}\right\}$$

backpropagation steps

(1) Compute derivative of cost function w.r.t last net values

$$\frac{\partial C}{\partial Z^{[L]}}$$

(2) Compute derivative of cost function w.r.t. net values a layer $\ell - 1$ from derivative a net values ℓ

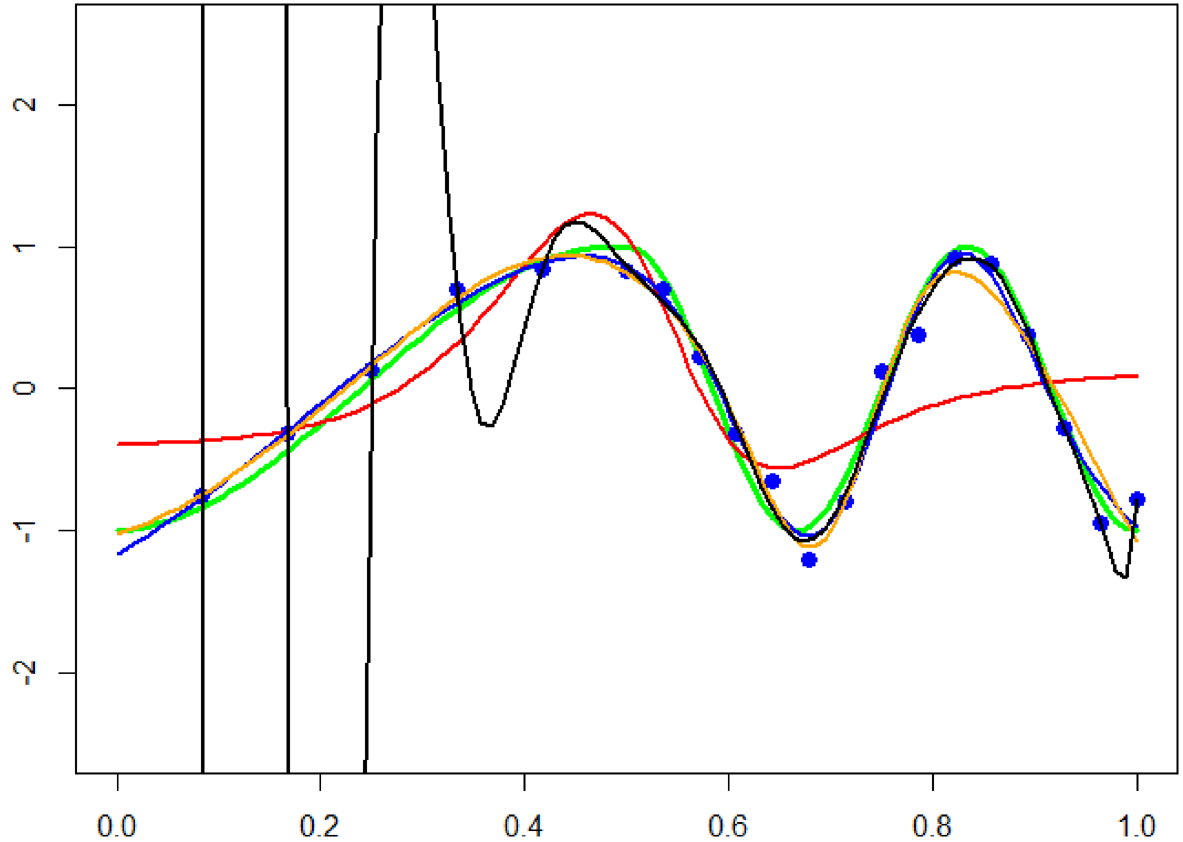
$$\frac{\partial C}{\partial Z^{[\ell-1]}} \quad \text{from} \quad \frac{\partial C}{\partial Z^{[\ell]}}$$

(3) Compute derivative of cost function w.r.t. adjustable parameters at layer ℓ from derivatives w.r.t. net values at layer ℓ

$$\frac{\partial C}{\partial \mathbf{b}^{[\ell]}} \quad \text{from} \quad \frac{\partial C}{\partial Z^{[\ell]}}$$

$$\frac{\partial C}{\partial W^{[\ell]}} \quad \text{from} \quad \frac{\partial C}{\partial Z^{[\ell]}}$$

Homework plot



polynomial interpolation fit

```
c(X,Y,xgrid,ygrid) %<-% data.lawrence.giles(12345)
degree=15     # larger values have numerical problems
x.set=c(X); y.set=c(Y) # convert to vectors
lm.fit = lm(y ~ poly(x,degree,raw=FALSE),
            data=data.frame(y=y.set,x=x.set))
lm.fit$residuals
```

```
##             1             2             3             4             5
##  1.539552e-07 -3.378065e-06  3.810329e-05 -3.104312e-04  2.340646e-03
##             6             7             8             9            10
## -3.099246e-02  8.253468e-02 -6.804435e-02 -6.491946e-02  1.925414e-01
##            11            12            13            14            15
## -1.383460e-01 -6.243021e-02  1.952435e-01 -1.561774e-01  4.403841e-02
##            16            17            18            19            20
##  2.095602e-02 -2.499755e-02  1.057952e-02 -2.252298e-03  2.011128e-04
```

```
# compute fit values at grid points
y = predict.lm(lm.fit,data.frame(x=xgrid))
```

Batch generator

```r
chunker.cl=function(M,chunk.size){
  chunk=0
  defect= M %% chunk.size
  num.chunks=floor((M+chunk.size-1)/chunk.size)
  # this is very wasteful if chunk.size evenly divides M
  chunks=rep(chunk.size,num.chunks)
  if(sum(chunks)>M) chunks[num.chunks]=M-sum(chunks[-num.chunks])
  sum.chunks=0 # placeholder
  sequence=0   # placeholder

  function(){
    if(chunk==0){
      # re-order chunks
      chunks<<-chunks[sample(1:num.chunks,num.chunks,replace=FALSE)]
      sum.chunks<<-cumsum(chunks)
      sequence<<-sample(1:M,M,replace=FALSE)
    }
    chunk<<-chunk+1
    if(chunk > num.chunks) {
      chunk<<-0; return(NULL)
    }else{
      start=ifelse(chunk==1,1,sum.chunks[chunk-1]+1)
      sequence[start:sum.chunks[chunk]]
    }
  }
```

Chunker code - batch generation

```
# example, how to use chunker.cl
  m=32    # number of samples
  chunker=chunker.cl(m,12)  # 12=mini-batch size
  test.chunker=c()
  while(TRUE){
    samples=chunker()
    if(is.null(samples)) break;
    test.chunker=c(test.chunker,samples)
    print(samples)
  }
```

```
##  [1] 23 20 12 21 16  7 13 31  1  5 28  8
##  [1] 29 17 19 11  3 30 27 26 10 22  4 24
## [1]  9 25 15 32  6 14 18  2
```

```
  all(sort(test.chunker)==(1:m))
```

```
## [1] TRUE
```