CAP 4453
Robot Vision
Dr. Gonzalo Vaca-Castaño
gonzalo.vacacastano@ucf.edu
Administrative details

• Correction of the midterm exam
Credits

• Some slides comes directly from:
  • Yosesh Rawat
  • Andrew Ng
Robot Vision

17. Introduction to Deep Learning II
Outline

- Fully connected Neural network
  - Activation functions:
    - Forward and backward
  - Back propagation
  - Network definitions
  - Initialization
  - Training
    - Hyper parameters
    - Gradient updates: RMS prop,
    - Amount of training data
    - Batch normalization
- Dataset
  - Train set, test set, validation set
  - Bias and variance

- Implementation network to solve digit identification
Fully connected networks: The math
A REVIEW

Fully connected Neural network

• A deep network is a neural network with many layers
• A neuron in a linear function followed for an activation function
• Activation function must be non-linear
• A loss function measures how close is the created function (network) from a desired output
• The “training” is the process of find parameters (‘weights’) that reduces the loss functions
• Updating the weights as $w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$ reduces the loss
• An algorithm named back-propagation allows to compute $\frac{dJ}{dW}$ for all the weights of the network in 2 steps: 1 forward, 1 backward
A Neuron
A REVIEW

\[ z = w^T x \]
\[ y = f(z) = f(w^T x) \]

\[ x = [x_1, x_2, x_3, 1] \]

Activations and their derivatives

<table>
<thead>
<tr>
<th>( f(z) = \frac{1}{1 + \exp(-z)} )</th>
<th>( f'(z) = f(z)(1 - f(z)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(z) = \tanh(z) )</td>
<td>( f'(z) = (1 - f^2(z)) )</td>
</tr>
<tr>
<td>( f(z) = \begin{cases} 0, &amp; z &lt; 0 \ z, &amp; z \geq 0 \end{cases} )</td>
<td>This space left intentionally (kind of) blank</td>
</tr>
<tr>
<td>( f(z) = \log(1 + \exp(z)) )</td>
<td>( f'(z) = \frac{1}{1 + \exp(-z)} )</td>
</tr>
</tbody>
</table>
How to minimize a function?

In our case the loss function

\[ w_{new} = w_{prev} - \alpha \frac{dJ}{dW} \]

Repeat until there is almost not change

How to compute this gradient?
Gradient descent

\[ f(x) \]

\[ x \]
General approach

Pick random starting point.
General approach

Compute gradient at point (analytically or by finite differences)
General approach

Move along parameter space in direction of negative gradient

\[ \gamma = \text{amount to move} \]
\[ = \text{learning rate} \]

\[ a_2 = a_1 - \gamma \nabla f(a_1) \]
General approach

Move along parameter space in direction of negative gradient.

\[ a_3 = a_2 - \gamma \nabla f(a_2) \]

\[ \gamma = \text{amount to move} \]
\[ = \text{learning rate} \]
General approach

Stop when we don’t move any more.

\[ a_{\text{stop}}: \quad a_{n-1} - \gamma \nabla f(a_{n-1}) = 0 \]
Gradient Descent

• The gradient is the direction of fastest increase in $J(X)$
• Updating the weights as $w_{new} = w_{prev} - \alpha \frac{dJ}{dW}$ reduces the loss

Learning rate \quad \text{gradient}

The Approach of Gradient Descent

• Iterative solution:
  - Start at some point
  - Find direction in which to shift this point to decrease error
    • This can be found from the derivative of the function
      – A positive derivative $\rightarrow$ moving left decreases error
      – A negative derivative $\rightarrow$ moving right decreases error
    – Shift point in this direction

Overall Gradient Descent Algorithm

• Initialize:
  – $x^0$
  – $k = 0$

• While $|f(x^{k+1}) - f(x^k)| > \varepsilon$
  – $x^{k+1} = x^k - \eta^k \nabla f(x^k)$
  – $k = k + 1$
Train with Gradient Descent

- $x^i, y^i = n$ training examples
- $f(x) =$ feed forward network
- $L(x, y; \theta) =$ some loss function

*Loss function* measures how ‘good’ our network is at classifying the training examples wrt. the parameters of the model (the perceptron weights).
Loss Function

• Way to define how good the network is performing
  • In terms of prediction

• Network training (Optimization)
  • Find the best network parameters to minimize the loss

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) \]

- Input
- Ground truth
- Loss function
- Network parameters
Loss Functions

• Cross entropy
  \[-\frac{1}{N} \sum_{i=1}^{N} (y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))\]

  [Ground-truth]  [Predicted value]

• Mean squared error (MSE)
  \[\frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2\]
Learning rate

![Graph showing the change of learning rate with epochs.](image)

- $a_1$, $a_2$, $a_3$, $a_{stop}$

![Graph showing the change of loss with epochs.](image)

- Very high learning rate
- Low learning rate
- High learning rate
- Good learning rate
Notation

- The input layer is the $0^{th}$ layer
- We will represent the output of the i-th perceptron of the $k^{th}$ layer as $y_i^{(k)}$
  - Input to network: $y_i^{(0)} = x_i$
  - Output of network: $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i-th unit of the $k-1$th layer and the jth unit of the $k$-th layer as $w_{ij}^{(k)}$
  - The bias to the jth unit of the k-th layer is $b_j^{(k)}$
Training steps

• Define network
• Loss function
• Initialize network parameters
• Get training data
  • Prepare batches
• Feedforward one batch
  • Compute loss
  • Update network parameters
  • Repeat
How to minimize a function?

\[ w_{\text{new}} = w_{\text{prev}} - \alpha \frac{dJ}{dW} \]

Repeat until there is almost not change
Training Neural Nets through Gradient Descent

Total training error:

\[ \text{Err} = \frac{1}{T} \sum_{t} \text{Div}(Y_t, d_t) \]

- Gradient descent algorithm:
- Initialize all weights and biases \( \{w_{ij}^{(k)}\} \)
  - Using the extended notation: the bias is also a weight
- Do:
  - For every layer \( k \) for all \( i, j \), update:
    - \( w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{d\text{Err}}{dw_{i,j}^{(k)}} \)
- Until \( \text{Err} \) has converged

Example: L2

\[ \text{Div} = \frac{1}{2} (y_t - d_t)^2 \]

\[ \frac{d\text{Div}}{dy_t} = (y_t - d_t) \]
The derivative

Total training error:

\[ Err = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t) \]

- Computing the derivative

Total derivative:

\[ \frac{dErr}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_t \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \]
The derivative

Total training error:

\[ Err = \frac{1}{T} \sum_{t} \text{Div}(Y_t, d_t) \]

Total derivative:

\[ \frac{dErr}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_{t} \frac{d\text{Div}(Y_t, d_t)}{dw_{i,j}^{(k)}} \]

• So we must first figure out how to compute the derivative of divergences of individual training inputs
Calculus Refresher: Basic rules of calculus

For any differentiable function
\[ y = f(x) \]
with derivative
\[ \frac{dy}{dx} \]
the following must hold for sufficiently small \( \Delta x \)
\[ \Delta y \approx \frac{dy}{dx} \Delta x \]

For any differentiable function
\[ y = f(x_1, x_2, ..., x_M) \]
with partial derivatives
\[ \frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, ..., \frac{\partial y}{\partial x_M} \]
the following must hold for sufficiently small \( \Delta x_1, \Delta x_2, ..., \Delta x_M \)
\[ \Delta y \approx \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \cdots + \frac{\partial y}{\partial x_M} \Delta x_M \]
Calculus Refresher: Chain rule

For any nested function \( y = f(g(x)) \)

\[
\frac{dy}{dx} = \frac{\partial y}{\partial g(x)} \frac{dg(x)}{dx}
\]

Check - we can confirm that:

\[
\Delta y = \frac{dy}{dx} \Delta x
\]

\( z = g(x) \)  \( \Delta z = \frac{dg(x)}{dx} \Delta x \)

\( y = f(z) \)  \( \Delta y = \frac{dy}{dz} \Delta z = \frac{dy}{dz} \frac{dg(x)}{dx} \Delta x \)
Calculus Refresher: Distributed Chain rule

\[ y = f(g_1(x), g_1(x), \ldots, g_M(x)) \]

\[
\frac{dy}{dx} = \frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \ldots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx}
\]

Check: \[ \Delta y = \frac{dy}{dx} \Delta x \]

\[
\Delta y = \frac{\partial y}{\partial g_1(x)} \Delta g_1(x) + \frac{\partial y}{\partial g_2(x)} \Delta g_2(x) + \ldots + \frac{\partial y}{\partial g_M(x)} \Delta g_M(x)
\]

\[
\Delta y = \frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} \Delta x + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} \Delta x + \ldots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx} \Delta x
\]

\[
\Delta y = \left( \frac{\partial y}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial y}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \ldots + \frac{\partial y}{\partial g_M(x)} \frac{dg_M(x)}{dx} \right) \Delta x
\]
Distributed Chain Rule: Influence Diagram

- Small perturbations in $x$ cause small perturbations in each of $g_1 \ldots g_M$, each of which individually additively perturbs $y$
A first closer look at the network

- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs
- Expanded **with all weights and activations shown**
- The overall function is differentiable w.r.t every weight, bias and input
Forward Computation

\[ y_i^{(0)} = x_i \]

ITERATE FOR \( k = 1:N \)

for \( j = 1:\text{layer-width} \)

\[ z_j^{(k)} = \sum_i w_{ij}^{(k)} y_i^{(k-1)} \]

\[ y_j^{(k)} = f_k \left( z_j^{(k)} \right) \]
Gradients: Backward Computation

\[ \frac{\partial \text{Div}(Y, d)}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}} \]

\[ \text{Div} = \frac{1}{2} (y_t - d_t)^2 \]

\[ \frac{d \text{Div}}{dy_i} = (y_t - d_t) \]
Gradients: Backward Computation

\[
\frac{\partial \text{Div}(Y, d)}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}
\]

\[
\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial \text{Div}}{\partial y_i} = f_N'(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}
\]
\[ y_i^{[N]} = f(z_i^{[N]}) \]
\[ \frac{\partial y_i^{[N]}}{\partial z_i^{[N]}} = f^{[N]'}(z_i^{[N]}) \]

\[ \frac{\partial \text{Div}(Y, d)}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}} \]

\[ \frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial \text{Div}}{\partial y_i} = f_N'(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \]
Gradients: Backward Computation

$z_{i}^{(N)}$ computed during the forward pass

\[
\frac{\partial \text{Div}(Y, d)}{\partial y_{i}^{(N)}} = \frac{\partial \text{Div}(Y, d)}{\partial y_{i}}
\]

\[
\frac{\partial \text{Div}}{\partial z_{i}^{(N)}} = \frac{\partial y_{i}^{(N)}}{\partial z_{i}^{(N)}} \frac{\partial \text{Div}}{\partial y_{i}} = f_{N}'(z_{i}^{(N)}) \frac{\partial \text{Div}}{\partial y_{i}^{(N)}}
\]
Gradients: Backward Computation

\[
\frac{\partial \text{Div}(Y, d)}{\partial Y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}
\]

Derivative of the activation function of Nth layer

\[
\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial \text{Div}}{\partial y_i} = f_N'(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}
\]
Gradients: Backward Computation

\[
\frac{\partial \text{Div}}{\partial y_i^{(N-1)}} = \sum_j \frac{\partial y_i^{(N-1)}}{\partial z_j^{(N)}} \frac{\partial \text{Div}}{\partial z_j^{(N)}} = \sum_j w_{ij}^{(N)} \frac{\partial \text{Div}}{\partial z_j^{(N)}}
\]

Because:

\[
\frac{\partial z_j^{(N)}}{\partial y_i^{(N-1)}} = w_{ij}^{(N)}
\]

\[
\frac{\partial \text{Div}}{\partial y_i^{(N)}} = f'_N \left(z_i^{(N)}\right) \frac{\partial \text{Div}}{\partial y_i^{(N)}}
\]

\[
\frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y,d)}{\partial y_i^{(N)}}
\]
But in this case the input is the output from previous layer

\[ z^{(N)} = w^{T} y^{[N-1]} \]
Gradients: Backward Computation

computed during the forward pass

\[ \frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}} \]

\[ \frac{\partial \text{Div}}{\partial y_i^{(N)}} = f_N'(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \]

\[ \frac{\partial \text{Div}}{\partial y_i^{(N-1)}} = \sum_j w_{ij}^{(N)} \frac{\partial \text{Div}}{\partial z_j^{(N)}} \]
Gradients: Backward Computation

\[ \frac{\partial \text{Div}}{\partial y_{i}^{(k-1)}} = \sum_{j} \frac{\partial z_{j}^{(k)}}{\partial y_{i}^{(k-1)}} \frac{\partial \text{Div}}{\partial z_{j}^{(k)}} = \sum_{j} w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_{j}^{(k)}} \]

\[ \frac{\partial \text{Div}}{\partial y_{i}^{(N)}} = f_{N}^{T} \left( z_{i}^{(N)} \right) \frac{\partial \text{Div}}{\partial y_{i}^{(N)}} \]
Gradients: Backward Computation

\[ \frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = \frac{\partial z_j^{(k)}}{\partial w_{ij}^{(k)}} \frac{\partial \text{Div}}{\partial z_j^{(k)}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial z_j^{(k)}} \]

\[ \frac{\partial \text{Div}}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k)}} \]

\[ \frac{\partial \text{Div}}{\partial y_i^{(N)}} = f_N(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \]

\[ \frac{\partial \text{Div}}{\partial Y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}} \]
Initialize: Gradient w.r.t. network output

\[
\frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}
\]

For \( k = N \) \( \cdot \) 1
For \( i = 1 \) \( \cdot \) layer \( \cdot \) width

\[
\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f_k'(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}}
\]

\[
\frac{\partial \text{Div}}{\partial y_i^{(k-1)}} = \sum_j w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}
\]

\[
\frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}
\]
Training by BackProp

- Initialize all weights \( (W^{(1)}, W^{(2)}, \ldots, W^{(K)}) \)
- Do:
  - Initialize \( Err = 0 \); For all \( i, j, k \), initialize \( \frac{dErr}{dw_{i,j}^{(k)}} = 0 \)
  - For all \( t = 1: T \) (Loop over training instances)
    - **Forward pass:** Compute
      - Output \( Y_t \)
      - \( Err += Div(Y_t, d_t) \)
    - **Backward pass:** For all \( i, j, k \):
      - Compute \( \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \)
      - Compute \( \frac{dErr}{dw_{i,j}^{(k)}} + \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \)
  - For all \( i, j, k \), update:
    \[
    w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dErr}{T dw_{i,j}^{(k)}}
    \]
- Until \( Err \) has converged
Exercise

1. Expected output: 1

\[ \text{Div} = \frac{(\hat{y}_i - y_i)^2}{2} \]

\[ (\hat{y}_i - y_i) \]

Activations and their derivatives

\[ f(z) = \frac{1}{1 + \exp(-z)} \quad f'(z) = f(z)(1 - f(z)) \]

\[ f(z) = \tanh(z) \quad f'(z) = (1 - f^2(z)) \]

\[ f(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases} \quad f'(z) = \text{This space left intentionally (kind of) blank} \]

\[ f(z) = \log(1 + \exp(z)) \quad f'(z) = \frac{1}{1 + \exp(-z)} \]

Gradients: Backward Computation

\[ \text{Initialize: Gradient w.r.t. network output} \]

\[ \frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y(i)} \]

\[ \frac{\partial \text{Div}}{\partial w_{ij}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial y(j)} \]

Figure assumes, but does not show the "1" bias nodes.
A real example
Digit classification

- MNIST dataset:
  - 70000 grayscale images of digits scanned.
  - 60000 for training
  - 10000 for testing

- Loss function

\[ J_2(w) = \frac{1}{m} \sum_{\text{train}} (\hat{y}_i - y_i)^2 \]
Digit classification

Typical Problem statement:
multiclass classification

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job
A look in the code

- To run this code do:
  - import network
  - net = network.Network([784, 30, 10])
  - net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
A look in code

```
def backprop(self, x, y):
    """Return a tuple ``(nabla_b, nabla_w)` representing the gradient for the cost function C_x.```
    """nabla_b'' and 
    """nabla_w'' are layer-by-layer lists of numpy arrays, similar 
    to ``self.biases'' and ``self.weights''.``"
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * 
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-1].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book. Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on. It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
        z = zs[-l]
        delta = np.dot(self.weights[-l-1].transpose(), delta) * 
        sigmoid_prime(zs[-l])
        nabla_b[-l-1] = delta
        nabla_w[-l-1] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)

def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial C_x / \partial a for the output activations.``"
    return (output_activations-y)

## Miscellaneous functions

def sigmoid(z):
    """The sigmoid function.``"
    return 1.0 / (1.0 + np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function.``"
    return sigmoid(z)*(1-sigmoid(z))
```
A look in code

Initialize: Gradient w.r.t network output

\[
\frac{\partial D}{\partial y_j} = \frac{\partial D(Y, d)}{\partial y_j^{(N)}}
\]

\[
\frac{\partial D}{\partial y_l^{(k-1)}} = \sum w_{lj} \frac{\partial D}{\partial y_j^{(k)}}
\]

\[
\frac{\partial D}{\partial w_{lj}^{(k)}} = y_l^{(k-1)} \frac{\partial D}{\partial y_l^{(k)}}
\]

For \( k = N \cdot 1 \)

For \( l = 1 \) to \( \text{layer width} \)

```python
def backprop(self, x, y):
    """Return a tuple (\'nabla_b\', \'nabla_w\') representing the
    gradient for the cost function C(x, \cdot) = \|y - y_h\|^2
    \"""  
    # Forwrad
    activation = x
    activations = [x]  # list to store all the activations, layer by layer
    zs = []  # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # Backward pass
    delta = self.cost_derivative(activations[-1], y) * 
            sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-1].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book. Here,
    # l = 1 means the last layer of neurons, 1 = the
    # second-to-last layer, and so on. It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    for l in range(2, self.num_layers):
        zs[l-2] = zs[l-2]*sigmoid_prime(zs[l-1])
    delta = np.dot(self.weights[-1].transpose(), delta) * 
            sigmoid_prime(zs[-1])
    nabla_b[-2] = delta
    nabla_w[-2] = np.dot(delta, activations[-2].transpose())
    return (nabla_b, nabla_w)
```

```python
def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial C / \partial x
    for the output activations."
    return (output_activations-y)
```

`sigmoid(z)`

`sigmoid_prime(z)`

### Miscellaneous functions

```python
def sigmoid(z):
    """The sigmoid function."
    return 1.0 / (1.0 + np.exp(-z))
```

```python
def sigmoid_prime(z):
    """Derivative of the sigmoid function."
    return sigmoid(z)*(1-sigmoid(z))
```
A look in the code

random Initialization
Feed forward ‘a’ thru all the layers

A Epoch is when all the training data has been used to update weights
A minibatch is a subset of all the data used to obtain a ‘quick’ weight updates
If there is test data perform evaluation
A look in the code

Add errors from all the training data from the mini-batch

Update the weights
references

• http://neuralnetworksanddeeplearning.com/chap1.html
• https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2015/
Questions?