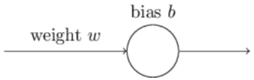
Chapter 3 of Neural Networks and Deep Learning

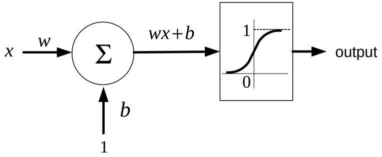
John Chiasson
Boise State University

These lecture slides follow the book **Neural Networks and Deep Learning** by Michael A. Nielsen, Determination Press, 2015

Problem with Sigmoids

Consider the simple network:

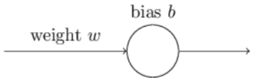


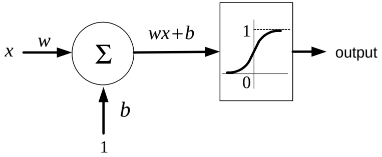


$$z = wx + b, \quad a = \sigma(z) = \frac{1}{1+e^{-z}}, \quad \sigma'(z) = \sigma(z)(1-\sigma(z))$$

- We want $x = 1$ to correspond to $y = 0$, $i.e.$, $a \approx 0$.
- $C = \frac{1}{2}(a-y)^2$
- $\dfrac{\partial C}{\partial w} = (a-y)\dfrac{\partial a}{\partial w} = (a-y)\sigma'(z)x$
- $w^{(new)} = w^{(old)} - \eta\dfrac{\partial C}{\partial w} = w^{(old)} - \eta(a-y)\sigma'(z)x$
- $b^{(new)} = b^{(old)} - \eta\dfrac{\partial C}{\partial b} = b^{(old)} - \eta(a-y)\sigma'(z)$

Problem with Sigmoids

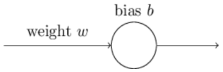


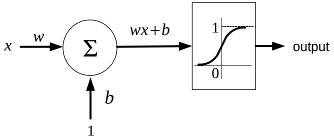


- $w^{(new)} = w^{(old)} - \eta \dfrac{\partial C}{\partial w} = w^{(old)} - \eta (a-y)\sigma'(wx+b)x$
- $b^{(new)} = b^{(old)} - \eta \dfrac{\partial C}{\partial b} = b^{(old)} - \eta (a-y)\sigma'(wx+b)$

$$
\begin{aligned}
w^{(0)} &= 0.6, b^{(0)} = 0.9, \eta = 0.15 \\
z^{(0)} &= w^{(0)}x + b^{(0)} = (0.6)(1) + 0.9 = 1.5 \\
a^{(0)} &= \sigma(1.5) = \frac{1}{1+e^{-1.5}} = 0.817 \quad (\text{we want } a \approx 0) \\
\sigma'(1.5) &= \frac{1}{1+e^{-1.5}} \frac{e^{-1.5}}{1+e^{-1.5}} = 0.149
\end{aligned}
$$

Problem with Sigmoids and Quadratic Cost





- $w^{(new)} = w^{(old)} - \eta \frac{\partial C}{\partial w} = w^{(old)} - \eta(a-y)\sigma'(wx+b)x$
- $b^{(new)} = b^{(old)} - \eta \frac{\partial C}{\partial b} = b^{(old)} - \eta(a-y)\sigma'(wx+b)$

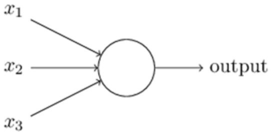
$$w^{(0)} = 2.0, b^{(0)} = 2.0, \eta = 0.15$$
$$z^{(0)} = w^{(0)}x + b^{(0)} = (2)(1) + 2 = 4$$
$$a^{(0)} = \sigma(4) = \frac{1}{1+e^{-4}} = 0.982 \quad (\text{we want } a \approx 0)$$
$$\sigma'(4) = \frac{1}{1+e^{-4}} \frac{e^{-4}}{1+e^{-4}} = 1.766\,3 \times 10^{-2}$$

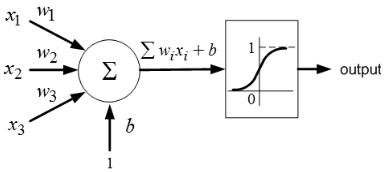
- When the initial weights are way off the sigmoid can be close to saturation.

$\Longrightarrow \sigma'(z) \approx 0 \Longrightarrow \frac{\partial C}{\partial w} \approx 0$ and the weight updates **slowly**.

Cross Entropy with Sigmoid

Simple network: 3 inputs and 1 output.





- Let the training data be $\{x^{(k)} \in \mathbb{R}^3, y^{(k)} \in \{0,1\} \text{ for } k = 1, 2, ..., n\}$.
- $z^{(k)} = \sum_{j=1}^{3} w_j x_j^{(k)} + b, \;\; a^{(k)} = \sigma(z^{(k)})$

Define the **cross-entropy** cost function to be

$$C \triangleq -\frac{1}{n} \sum_{k=1}^{n} \left(y^{(k)} \ln(a^{(k)}) \; + (1 - y^{(k)}) \ln(1 - a^{(k)}) \right)$$

Properties of the Cross Entropy Cost Function

cross-entropy cost function:

$$C \triangleq -\frac{1}{n}\sum_{k=1}^{n}\left(y^{(k)}\ln(a^{(k)}) + (1-y^{(k)})\ln(1-a^{(k)})\right)$$

- The cost $C \geq 0$
  - Each $y^{(k)}$ is 0 or 1. So $1-y^{(k)}$ is 0 or 1.
  - Sigmoid activation so $0 \leq a^{(k)} \leq 1$ and $0 \leq 1-a^{(k)} \leq 1$.

$$\begin{aligned} \Longrightarrow &\quad \ln(a^{(k)}) \leq 0 \text{ and } \ln(1-a^{(k)}) \leq 0. \\ \Longrightarrow &\quad -\left(y^{(k)}\ln(a^{(k)}) + (1-y^{(k)})\ln(1-a^{(k)})\right) \geq 0 \text{ for all } k. \end{aligned}$$

Properties of the Cross Entropy Cost Function

$$C \triangleq -\frac{1}{n}\sum_{k=1}^{n}\left(y^{(k)}\ln(a^{(k)}) + (1-y^{(k)})\ln(1-a^{(k)})\right)$$

- The cost $C = 0$ if and only if for all k:
  - If $y^{(k)} = 0$ then $a^{(k)}$ must be 0.
  - If $y^{(k)} = 1$ then $a^{(k)}$ must be 1.

Proof: First note that

$$0\ln(0) \triangleq \lim_{a\to 0^+} a\ln(a) = \lim_{a\to 0^+}\frac{\ln(a)}{1/a} = \lim_{a\to 0^+}\frac{1/a}{-(1/a^2)} = \lim_{a\to 0^+}\frac{-a}{1} = 0$$

If $y^{(k)} = 0$ then

$$-\left(y^{(k)}\ln(a^{(k)}) + (1-y^{(k)})\ln(1-a^{(k)})\right)\Big|_{y^{(k)}=0} = -\ln(1-a^{(k)})$$

and $\ln(1-a^{(k)}) = 0$ if and only if $a^{(k)} = 0$.

Similarly if $y^{(k)} = 1$.

Properties of the Cross Entropy Cost Function

$$C \triangleq -\frac{1}{n}\sum_{k=1}^{n}\left(y^{(k)}\ln(a^{(k)}) \; + (1-y^{(k)})\ln(1-a^{(k)})\right), \quad z^{(k)} = \sum_{j=1}^{3} w_j x_j^{(k)} + b$$

$$\begin{aligned}
\frac{\partial C}{\partial w_j} &= -\frac{1}{n}\sum_{k=1}^{n}\left(\frac{y^{(k)}}{\sigma(z^{(k)})} - \frac{1-y^{(k)}}{1-\sigma(z^{(k)})}\right)\frac{\partial \sigma(z^{(k)})}{\partial w_j} = -\frac{1}{n}\sum_{k=1}^{n}\left(\frac{y^{(k)}}{\sigma(z^{(k)})} - \frac{1-y^{(k)}}{1-\sigma(z^{(k)})}\right)\sigma'(z^{(k)})x_j^{(k)} \\
&= -\frac{1}{n}\sum_{k=1}^{n}\left(\frac{y^{(k)}}{\sigma(z^{(k)})}\frac{1-\sigma(z^{(k)})}{1-\sigma(z^{(k)})} - \frac{1-y^{(k)}}{1-\sigma(z^{(k)})}\frac{\sigma(z^{(k)})}{\sigma(z^{(k)})}\right)\sigma'(z^{(k)})x_j^{(k)} \\
&= \frac{1}{n}\sum_{k=1}^{n}\left(\frac{\sigma'(z^{(k)})x_j^{(k)}}{\sigma(z^{(k)})(1-\sigma(z^{(k)}))}\right)\left(\sigma(z^{(k)}) \; - y^{(k)}\right) \\
&= \frac{1}{n}\sum_{k=1}^{n} x_j^{(k)}\left(\sigma(z^{(k)}) \; - y^{(k)}\right)
\end{aligned}$$

- The gradient $\partial C/\partial w_j$ is proportional to the error $\sigma(z^{(k)}) - y^{(k)}$.
- The larger the error, the larger $\frac{\partial C}{\partial w_j}$, and thus the faster it learns.
- Exercise: Show that $\frac{\partial C}{\partial z^{(k)}} = \frac{\partial C}{\partial b} = \frac{1}{n}\sum_{k=1}^{n}\left(\sigma(z^{(k)}) \; - y^{(k)}\right)$

Cross Entropy with Multiple Outputs

- L denotes the **output layer**.

$$C \triangleq -\frac{1}{n}\sum_{k=1}^{n}\left(y^{(k)}\ln(a^{(k)}) + (1-y^{(k)})\ln(1-a^{(k)})\right)$$

$$= -\frac{1}{n}\sum_{x}\left(y(x)\ln a^{L}(x) + \Big(1-y(x)\Big)\ln(1-a^{L}(x))\right)$$

$$= -\frac{1}{n}\sum_{x}\left(y(x)\ln a^{L}(x,W,b) + \Big(1-y(x)\Big)\ln\Big(1-a^{L}(x,W,b)\Big)\right)$$

- Let there now be n_o outputs.
- Let $y_1(x), y_2(x), ..., y_{n_o}(x)$ be the output labels for the input x.
- $0 \le y_j(x) \le 1$ for $j = 1, 2, ..., n_o$

$$C \triangleq -\frac{1}{n}\sum_{x}\sum_{j=1}^{n_o}\left(y_j(x)\ln a_j^{L}(x,W,b) + \Big(1-y_j(x)\Big)\ln\Big(1-a_j^{L}(x,W,b)\Big)\right)$$

$$= -\frac{1}{n}\sum_{x}\sum_{j=1}^{n_o}\left(y_j\ln a_j^{L} + (1-y_j)\ln(1-a_j^{L})\right) \qquad \text{(Nielsen's notation)}$$

Cross Entropy with Multiple Outputs

- $y_1(x), y_2(x), \ldots, y_{n_o}(x)$ are the output labels for the input x.
- C is the average cost over the training set.

$$
\begin{aligned}
C &\triangleq -\frac{1}{n}\sum_x \sum_{j=1}^{n_o}\Big(y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\Big)\\
&= -\frac{1}{n}\sum_x \sum_{j=1}^{n_o}\Big(y_j(x)\ln a_j^L(x,W,b) + \Big(1-y_j(x)\Big)\ln\Big(1-a_j^L(x,W,b)\Big)\Big)
\end{aligned}
$$

- $W = \{W^{(2)}, \ldots, W^{(L-1)}, W^{(L)}\}$ are the weights for all layers.
- $b = \{b^{(2)}, \ldots, b^{(L-1)}, b^{(L)}\}$ are the biases for all layers.
- With n_{L-1} the number of neurons in layer $L-1$ we have

$$
z_i^L = \sum_{k=1}^{n_{L-1}} w_{ik}^{(L)} a_k^{L-1} + b_i^L, \quad a_i^L = \sigma(z_i^L)
$$

and

$$
\begin{aligned}
\frac{\partial C}{\partial w_{ij}^{(L)}} &= \frac{1}{n}\sum_x a_j^{L-1}(a_i^L - y_i)\\
\frac{\partial C}{\partial b_i^{(L)}} &= \frac{1}{n}\sum_x (a_i^L - y_i)
\end{aligned}
$$

Cross Entropy with Multiple Outputs and the Mini-Batch

- $y_1(x), y_2(x), \ldots, y_{n_o}(x)$ are the output labels for the input x.
- C is the average cost over a minibatch.

$$C \triangleq -\frac{1}{m}\sum_{x\in\ \text{minibatch}}\sum_{j=1}^{n_o}\Big(y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\Big)$$
$$= -\frac{1}{m}\sum_{x\in\ \text{minibatch}}\sum_{j=1}^{n_o}\Big(y_j(x)\ln a_j^L(x,W,b) + \Big(1-y_j(x)\Big)\ln\Big(1-a_j^L(x,W,b)\Big)\Big)$$

- $W = \{W^{(2)}, \ldots, W^{(L-1)}, W^{(L)}\}$ are the weights for all layers.
- $b = \{b^{(2)}, \ldots, b^{(L-1)}, b^{(L)}\}$ are the biases for all layers.
- With n_{L-1} the number of neurons in layer $L-1$ we have

$$z_i^L = \sum_{k=1}^{n_{L-1}} w_{ik}^{(L)} a_k^{L-1} + b_i^L, \quad a_i^L = \sigma(z_i^L)$$

and

$$\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{1}{m}\sum_{x\in\ \text{minibatch}} a_j^{L-1}(a_i^L - y_i)$$
$$\frac{\partial C}{\partial b_i^{(L)}} = \frac{1}{m}\sum_{x\in\ \text{minibatch}} (a_i^L - y_i)$$

Backpropagation Equations - Cross Entropy Cost

Cross Entropy Cost on Output Layer and Sigmoid on Hidden Layer

$$\delta^{(3)} = \frac{\partial C}{\partial z^{(3)}} = \frac{1}{m} \sum_{x \in \text{ minibatch}} \begin{bmatrix} (a_1^{(3)} - y_1) \\ \vdots \\ (a_{n_o}^{(3)} - y_{n_o}) \end{bmatrix} \in \mathbb{R}^{n_o} \tag{BP1}$$

$$\frac{\partial C}{\partial b^{(3)}} = \delta^{(3)} \in \mathbb{R}^{n_o} \tag{BP3}$$

$$\frac{\partial C}{\partial W^{(3)}} = \delta^{(3)} a^{(2)\,T} \in \mathbb{R}^{n_o \times n_h}, \quad a^{(2)} \in \mathbb{R}^{n_h} \tag{BP4}$$

$$\delta^{(2)} = W^{(3)\,T} \delta^{(3)} \odot \sigma'(z^{(2)}) = \frac{\partial C}{\partial z^{(2)}} \in \mathbb{R}^{n_h} \tag{BP2}$$

$$\frac{\partial C}{\partial b^{(2)}} = \delta^{(2)} \in \mathbb{R}^{n_h} \tag{BP3}$$

$$\frac{\partial C}{\partial W^{(2)}} = \delta^{(2)} a^{(1)\,T} \in \mathbb{R}^{n_h \times n_i}, \quad a^{(1)} = x \in \mathbb{R}^{n_i} \tag{BP4}$$

Digit Recognition with a Cross Entropy Cost Function

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
    monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

- Mini-batch size = 10, $\eta = 0.5$, 30 epochs, **30 hidden neurons**
- Cost is monotonically decreasing.

```
Epoch 1 training complete
Cost on training data:  0.460579340682
Accuracy on evaluation data:  9280 / 10000
Epoch 2 training complete
Cost on training data:  0.389641072453
Accuracy on evaluation data:  9370 / 10000
                   ⋮
Epoch 28 training complete
Cost on training data:  0.173435561264
Accuracy on evaluation data:  9507 / 10000
Epoch 29 training complete
Cost on training data:  0.187956075344
Accuracy on evaluation data:  9502 / 10000
```

Digit Recognition with a Cross Entropy Cost Function

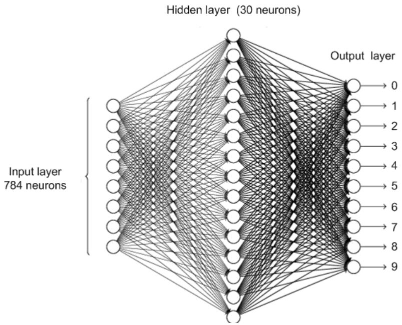
```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
net = network2.Network([784, 100, 10], cost=network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(training_data, 30, 10, 0.5, evaluation_data=test_data,
    monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

- Mini-batch size = 10, $\eta = 0.5$, 30 epochs, **100 hidden neurons**
- Cost is monotonically decreasing.

```
Epoch 0 training complete
Cost on training data:  0.446182191452
Accuracy on evaluation data:  9323 / 10000
Epoch 1 training complete
Cost on training data:  0.329450977928
Accuracy on evaluation data:  9463 / 10000
                 ⋮
Epoch 28 training complete
Cost on training data:  0.0196113354189
Accuracy on evaluation data:  9667 / 10000
Epoch 29 training complete
Cost on training data:  0.016995307966
Accuracy on evaluation data:  9673 / 10000
```

Overfitting and Regularization

- $x \in \mathbb{R}^{784}$ are the pixel values of a training image.
- $a^{(3)} = f(x, W^{(2)}, b^{(2)}, W^{(3)}, b^{(3)}) \in \mathbb{R}^{10}$ are the output values of the network.
- $W^{(2)} \in \mathbb{R}^{30\times784}$, $b^{(2)} \in \mathbb{R}^{30}$, $W^{(3)} \in \mathbb{R}^{10\times30}$, $b^{(3)} \in \mathbb{R}^{10}$.
- The total number of parameters is $30 \times 784 + 30 + 10 \times 30 + 10 = 23,860$



Overfitting: Are we using too many parameters?

- Are the parameters being adjusted to learn the noisy training data?
- The parameters should be adjusted for classification of new (unseen) data

Overfitting

Overfitting:

- The network learns with the (noisy) training data.
- However, the network does not perform well on new (noisy) test data.

Demonstration: Train the network so that is does **not** generalize to new data.

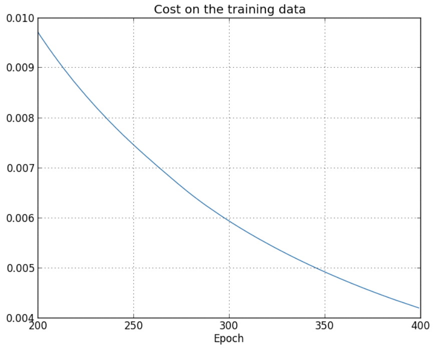
```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(training_data[:1000], 400, 10, 0.5, evaluation_data=test_data,
    monitor_evaluation_accuracy=True, monitor_training_cost=True)
```

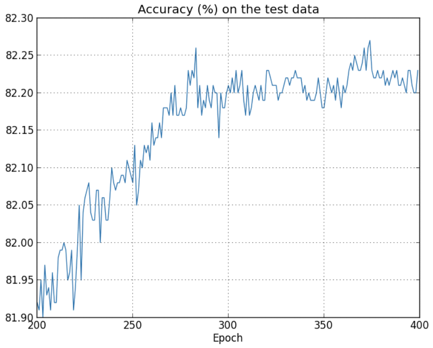
- Only use 1000 training images (instead of 50,000)
- mini-batch size $= 10$ so there are only 100 weight updates per epoch.
- 400 epochs for a total of $400 \times 100 = 40,000$ weight updates.

Overfitting

$$C \triangleq -\frac{1}{1000}\sum_x \sum_{j=1}^{10}\left(y_j \ln a_j^{(3)} + (1-y_j)\ln(1-a_j^{(3)})\right)$$

- Left side is cost on **training data** from epoch 200 to epoch 400.
- Right side is the percent accuracy on the 10,000 **test digits**.
- After about epoch 280 the accuracy on the test data only fluctuates about 82%.
- We say the network is **overfitting** or **overtraining** beyond epoch 280.

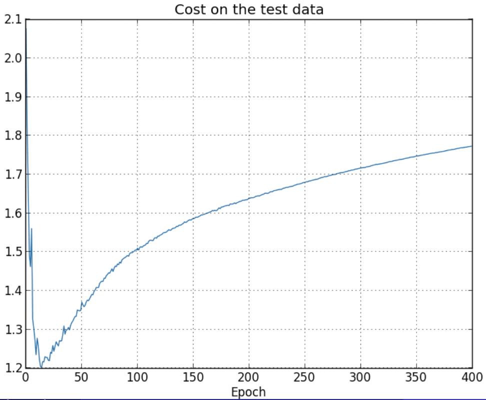




Overfitting

$$C \triangleq -\frac{1}{1000}\sum_x \sum_{j=1}^{10}\left(y_j \ln a_j^{(3)} + (1-y_j)\ln(1-a_j^{(3)})\right)$$

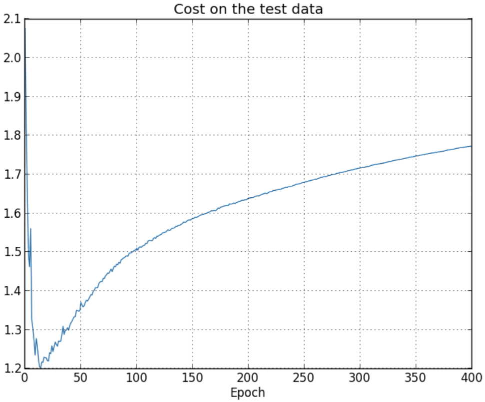
- Plot of the cost on the **test data** versus epoch.
- After epoch 15 the cost on the **test data** gets worse!
- Does overfitting start at epoch 15 rather than 280?



Overfitting

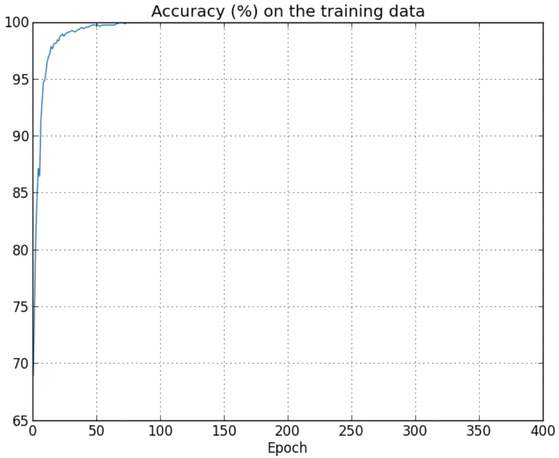
$$C \triangleq -\frac{1}{1000}\sum_x \sum_{j=1}^{10}\left(y_j \ln a_j^{(3)} + (1-y_j)\ln(1-a_j^{(3)})\right)$$

- Practical point of view
  - Improve the **classification accuracy** on the **test data**!
  - Take overfitting to start at epoch 280.



Overfitting

- After about 75 epochs the network correctly classifies all 1000 **training images**.
- The classification accuracy on the **test images** is only about 82% (8200/10000).
  - The network learns peculiarities (due to noise) of the training set.
  - Not recognizing digits in general.



Overfitting

- Overfitting is a major problem in neural networks
- Don't want the weights & biases adjusted to fit the noise in the training data.
- Use the `validation_data` (instead of `test_data`) to prevent overfitting.

Detect Overfitting:

(1) Compute classification accuracy on the `validation_data` after each epoch.
(2) If the accuracy on the `validation_data` no longer improves, stop training.

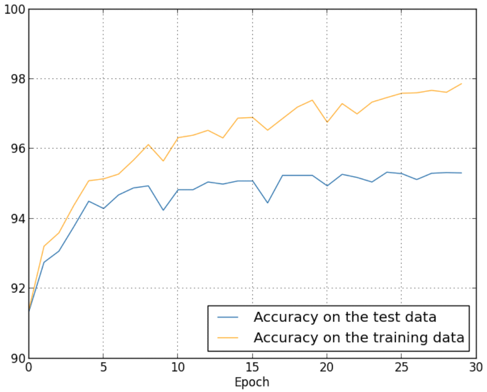
Overfitting

Why use `validation_data` rather than `test_data` to prevent overfitting?

- The MNIST set has 60,000 training images and 10,000 testing images.
- The `test_data` represents data **after** the network is trained, e.g., the network used by the Post Office to read zip codes.
- The 60,000 training images are *subdivided* into 50,000 images for the `training_data` and 10,000 images for the `validation_data`.
- Recall the hyper-parameters: n_h number of hidden neurons, η learning rate, number of epochs, mini-batch size, etc.
- Use the `validation_data` to help set the hyper-parameters.
- The **final evaluation** is done using the `test_data`.

Redo with 50000 Training Images

- Mini-batch size $= 10$, $\eta = 0.5$, 30 hidden neurons, cross-entropy cost
- 50,000 training images for 30 epochs
- 97.86% classification accuracy on the 50,000 training images.
- 95.49% classification accuracy on the 10,000 test images.



Overfitting example: 100% on 1000 training images and 82% on test images.

Regularization - Keep the weights from getting large

- Add a *regularization* term to the cost. I.e.,

$$C \triangleq -\frac{1}{n}\sum_x \sum_{j=1}^{n_o}\left(y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right) + \frac{\lambda}{2}\frac{1}{n}\sum_w w^2$$

  or with the mini-batch

$$C \triangleq -\frac{1}{m}\sum_{x\in \text{ minibatch}} \sum_{j=1}^{n_o}\left(y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right) + \frac{\lambda}{2}\frac{1}{n}\sum_w w^2.$$

  Note that the regularization terms stayed the same.
- $\lambda > 0$ is the *regularization parameter*.
- This can be done on the quadratic cost as well:

$$C \triangleq \frac{1}{m}\sum_{x\in \text{ minibatch}} \frac{1}{2}\left\|y - a^L\right\|^2 + \frac{\lambda}{2}\frac{1}{n}\sum_w w^2$$

- More generally

$$C \triangleq C_0 + \frac{\lambda}{2}\frac{1}{n}\sum_w w^2$$

  $C_0 \triangleq -\frac{1}{m}\sum_{x\in \text{ minibatch}}\sum_{j=1}^{n_o}\left(y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right)$ or $\frac{1}{m}\sum_{x\in \text{ minibatch}} \frac{1}{2}\left\|y - a^L\right\|^2$.
- Minimizing C is a compromise between small weights and small C_0.
- Regularization is done to reduce overfitting. We show why below.

Regularization

$$C \triangleq C_0 + \frac{\lambda}{2}\frac{1}{n}\sum_w w^2$$

The gradients of the regularized cost are

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$

$$\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}.$$

The learning rule (parameter update) becomes

$$w \rightarrow w - \eta\frac{\partial C_0}{\partial w} - \eta\frac{\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta\frac{\partial C_0}{\partial w}$$
$$b \rightarrow b - \eta\frac{\partial C_0}{\partial b}.$$

- At each update, each weight is scaled by $1 - \frac{\eta\lambda}{n}$ with η, λ chosen so that $0 < 1 - \frac{\eta\lambda}{n} < 1$.
- This rescaling is often called *weight decay*.

Regularization

$$C \triangleq C_0 + \frac{\lambda}{2} \sum_w w^2$$

When using stochastic gradient descent the update

$$w \longrightarrow \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}$$

$$b \longrightarrow b - \eta \frac{\partial C_0}{\partial b}.$$

With

$$C_{0x} \triangleq -\sum_{j=1}^{n_o} \left(y_j(x) \ln a_j^L(x) + (1 - y_j(x)) \ln(1 - a_j^L(x) \right)$$

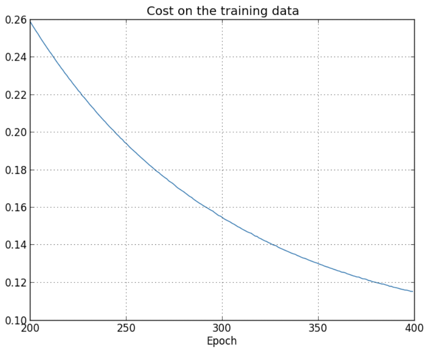
and using mini-batches we have

$$w \longrightarrow \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_{x \in \text{ minibatch}} \frac{\partial C_{0x}}{\partial w}$$

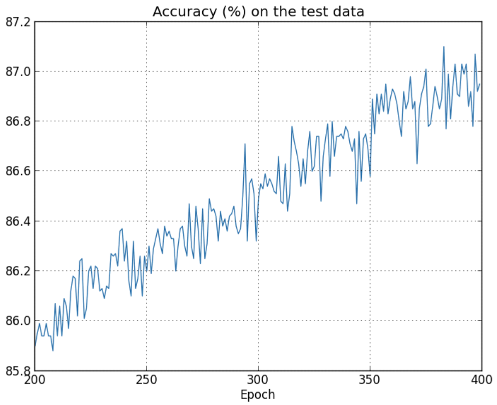
$$b \longrightarrow b - \frac{\eta}{m} \sum_{x \in \text{ minibatch}} \frac{\partial C_{0x}}{\partial b}.$$

Rerun Code with Regularization

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(training_data[:1000],400,10,0.5,evaluation_data=test_data,lmbda = 0.1,
    monitor_evaluation_cost=True, monitor_evaluation_accuracy=True,
    monitor_training_cost=True, monitor_training_accuracy=True)
```



Rerun Code with Regularization



- Recall that previously classification accuracy stopped at about epoch 280.
- With regularization the classification accuracy is still increasing at epoch 400!
- Classification accuracy is now 87% (82% without regularization) at epoch 400.

Run with Full MNIST Training Data with Regularization

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(training_data,30,10,0.5,evaluation_data=test_data,lmbda = 5,
    monitor_evaluation_cost=True, monitor_evaluation_accuracy=True,
    monitor_training_cost=True, monitor_training_accuracy=True)
```

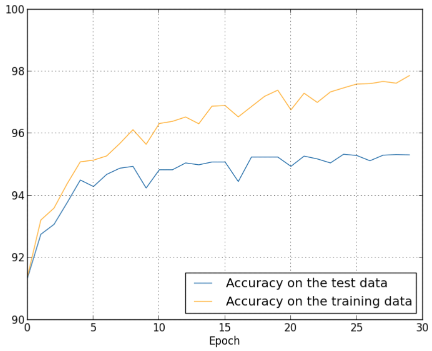
- Weight decay $1 - \frac{\eta\lambda}{n} = 1 - 0.5\frac{5}{50000} = 0.99995$
- With only 1000 training images:

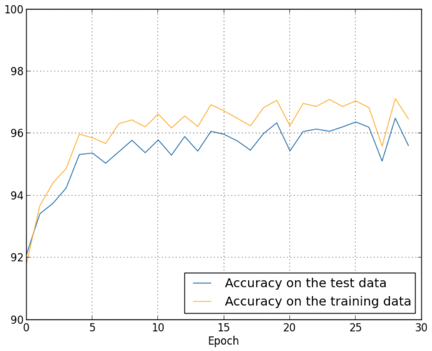
  Weight decay $1 - \frac{\eta\lambda}{n} = 1 - 0.5\frac{0.1}{1000} = 0.99995$

Run Regularization with Full MNIST Training Data & 30 Hidden Neurons

- 50,000 training images, 30 hidden neurons
- 30 epochs, $\eta = 0.5$, $\lambda = 5$

Left Figure: Without regularization. **Right Figure:** With regularization.





- Classification accuracy is now 96.49% with regularization (Right Figure).
- Classification accuracy was 95.49% without regularization (Left Figure).

Run with Full MNIST Training Data & 100 Hidden Neurons

- 50,000 training images
- 100 hidden neurons
- 30 epochs, $\eta = 0.5$, $\lambda = 5$

```
net = network2.Network([784, 100, 10], cost=network2.CrossEntropyCost)
net.large_weight_initializer()
net.SGD(training_data, 30, 10, 0.5,
lmbda=5.0,evaluation_data=validation_data,
    monitor_evaluation_accuracy=True)
```

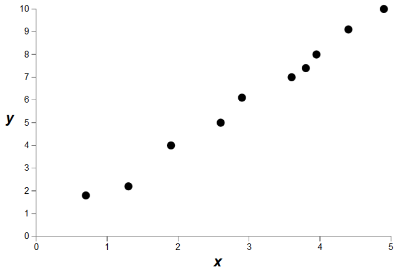
- Classification accuracy is now 97.92% on the **validation data**.

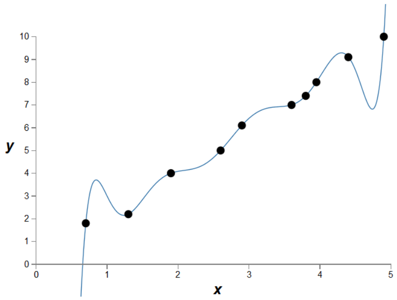
Comment

- Change to 60 epochs and $\eta = 0.1$ to get 98.04% accuracy on the **validation data**.

More on Overfitting

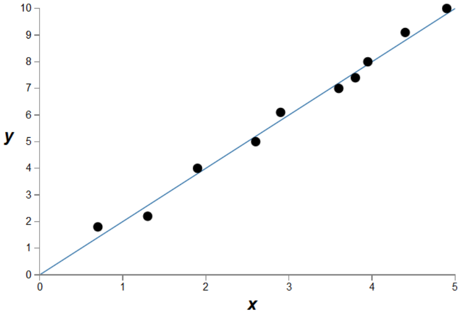
Left: Ten raw data points.
Right: Fit a polynomial $y = a_9x^9 + a_8x^8 + \cdots + a_1x_1 + a_0$ to the data.





More on Overfitting

- Least squares fit of $y = mx + b$ to the data.
- We think the data is $y = mx + b +$ noise.



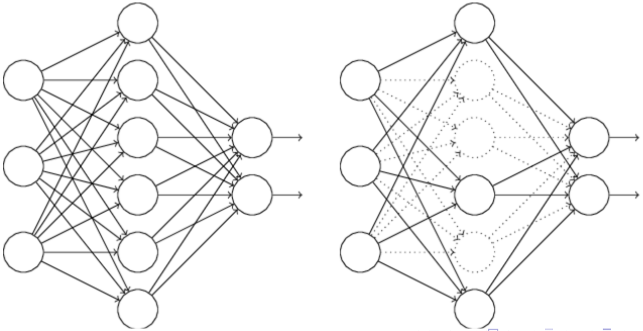
Why don't we include the biases in regularization?

- Empirically it doesn't seem to help.
- Biases are not sensitive to input noise.
- Large weights multiplying the (noisy) inputs makes the neuron sensitive to the noise.

Dropout

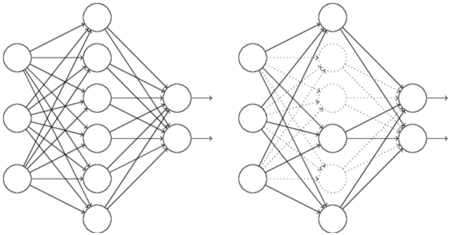
(1) Randomly choose 1/2 of the hidden neurons to "dropout".
(2) Forward-propagate the input x through the modified network.
(3) Backpropagate the result, also through the modified network.
(4) Do this over a mini-batch.
(5) Update the **appropriate** weights and biases.

- Repeat the above on the next minibatch.
  - I.e., randomly choose another half of the hidden neurons to dropout, etc.



Dropout

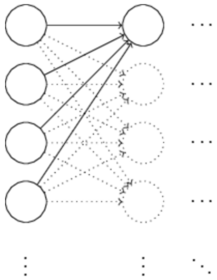
- At each update only half the weights are updated.
- After training run the **full** network.
  - **Twice** as many hidden neurons will be active compared to training.
  - Compensate this by **halving** the weights **outgoing** from the hidden neurons.



- We can look at dropout as **averaging** over different networks.
- Dropout means that the network learns without relying on being connected to specific other neurons.
- Dropout makes the model robust to the loss of any individual piece of evidence.

Weight Initialization

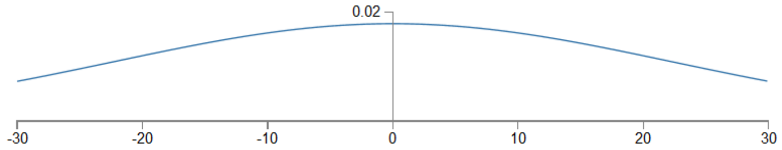
- Weights and biases have been initialized as independent $\mathcal{N}(0,1)$ Gaussian RVs.
- Not such a good thing to do!



- Consider a binary picture of 1000 pixels so $x_j = 0$ or 1 for $j = 1, \ldots, 1000$.
- Let $z = \sum_{j=1}^{1000} w_j x_j + b$ be the input to one of the hidden neurons.
- Suppose half the input neurons are on and the other half off.
- $E[z] = E\left[\sum_{j=1}^{1000} w_j x_j + b\right] = 0$ as $E[w_j] = 0, E[b] = 0$.
- $E[z^2] = E\left[\left(\sum_{j=1}^{1000} w_j x_j + b\right)^2\right] = \sum_{j=1}^{1000} E[w_j^2]x_j^2 + E[b^2] = 501$ as $x_j = 0$ for 500 of the pixels

Weight Initialization

- So $z = \sum_{j=1}^{1000} w_j x_j + b$ is a zero mean Gaussian RV with variance $\sigma^2 = 501$.
- z has standard deviation $\sigma = \sqrt{501} \approx 22.4$
- The probability density function (pdf) of z is broad as shown below.



- So, at initialization, often we will have $z >> 1$ or $z << -1$.
  $\implies \sigma(z)$ will be close to 1 or 0.
- Thus $\sigma'(z) \approx 0$ making learning slow.
  - Slow learning just means weights don't change much at each update.
  - Cross-entropy only eliminates the problem of $\sigma'(z) \approx 0$ at the **output** layer.
  - This problem will be at all the hidden layers.

Weight Initialization

- Let $n_{in} = 1000$ denote the number of input pixels.
- Now choose the weights as independent $\mathcal{N}(0, 1/n_{in})$ Gaussian RVs.
  - So $\sigma^2 = \dfrac{1}{n_{in}}$ or $\sigma = \dfrac{1}{\sqrt{n_{in}}}$.
- Then

$$z = \sum_{j=1}^{1000} w_j x_j + b$$

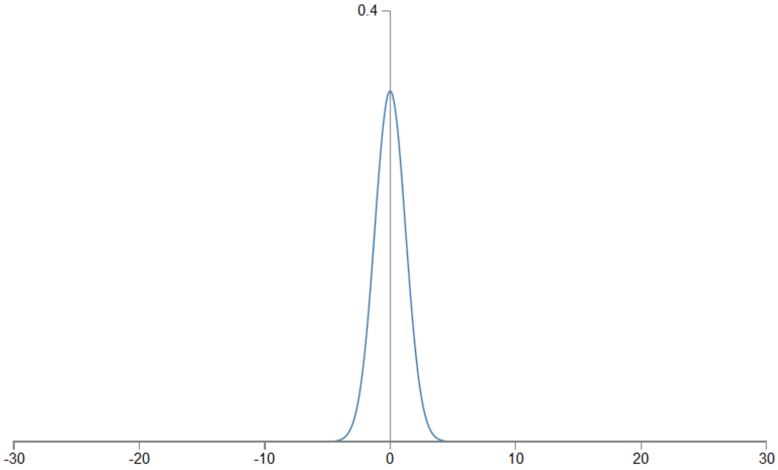
has mean zero and variance

$$\sigma^2 = E[z^2] = \sum_{j=1}^{1000} E[w_j^2]x_j^2 + E[b^2] = 500 \times \frac{1}{1000} + 1 = \frac{3}{2}.$$

- The probability density function (pdf) of z is now peaked (see next slide).

Weight Initialization

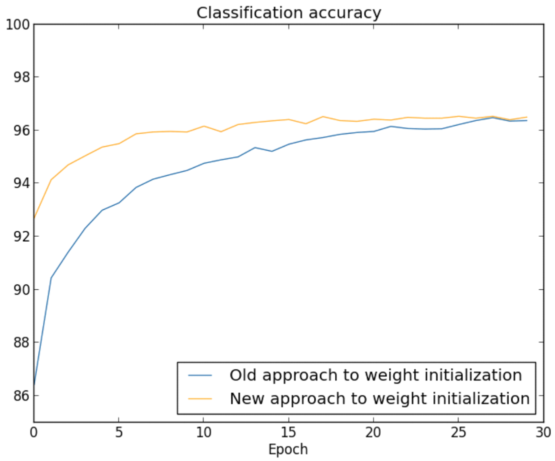
- Probability density function of z with weights independent $\mathcal{N}(0, 1/n_{in})$ RVs.



- It doesn't matter much how the biases are initialized.

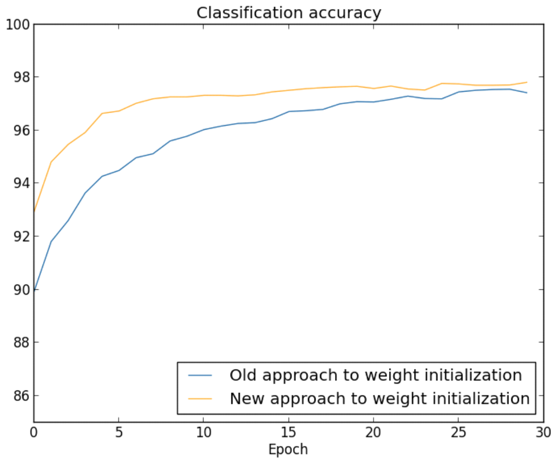
Comparison of Old and New Weight Initialization

- Use 30 hidden neurons, and the cross-entropy cost function.
- Mini-batch size of 10, $\lambda = 5.0$, $\eta = 0.1$.



Comparison of Old and New Weight Initialization

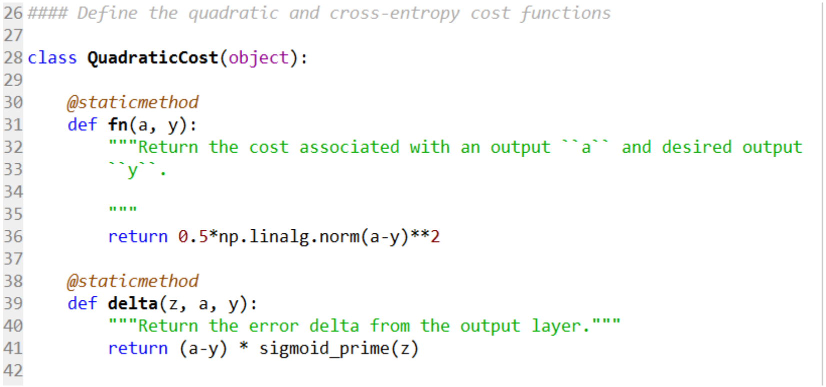
- Use 100 hidden neurons, and the cross-entropy cost function
- Mini-batch size of 10, $\lambda = 5.0$, $\eta = 0.1$.



Code for Network2.py

```
"""network2.py
~~~~~~~~~~~~~~

An improved version of network.py, implementing the stochastic
gradient descent learning algorithm for a feedforward neural network.
Improvements include the addition of the cross-entropy cost function,
regularization, and better initialization of network weights.  Note
that I have focused on making the code simple, easily readable, and
easily modifiable.  It is not optimized, and omits many desirable
features.

"""

#### Libraries
# Standard library
import json
"Java Script Object Notation - similar to pickle."
import random
import sys
"System-specific parameters and functions"

# Third-party libraries
import numpy as np
```

Code for Network2.py



```
#### Define the quadratic and cross-entropy cost functions

class QuadraticCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output ``a`` and desired output
        ``y``.

        """
        return 0.5*np.linalg.norm(a-y)**2

    @staticmethod
    def delta(z, a, y):
        """Return the error delta from the output layer."""
        return (a-y) * sigmoid_prime(z)

```

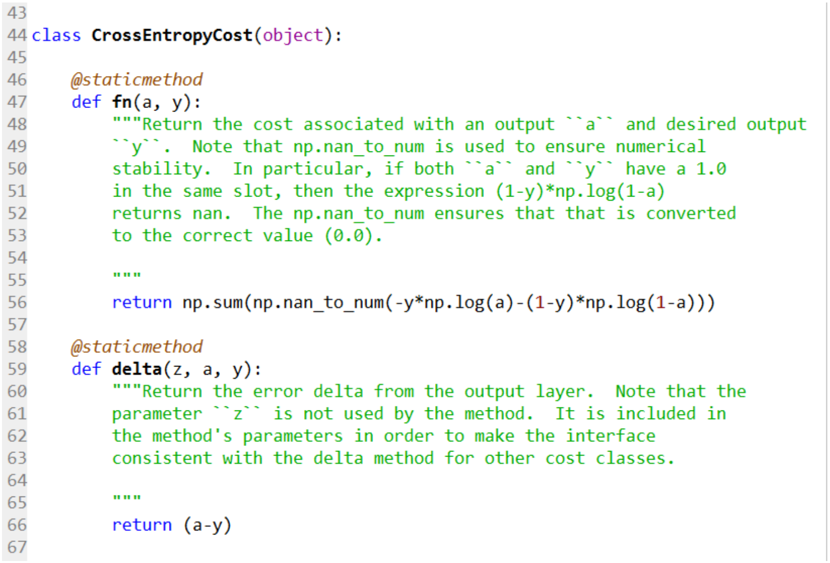
@staticmethod

- Tells Python the method doesn't depend on the `object` in any way.
- Note that "*`self`*" isn't passed as a parameter to the **fn** and **delta** methods.

  I.e., we don't write `def` **fn**(*`self`*, `a, y):`

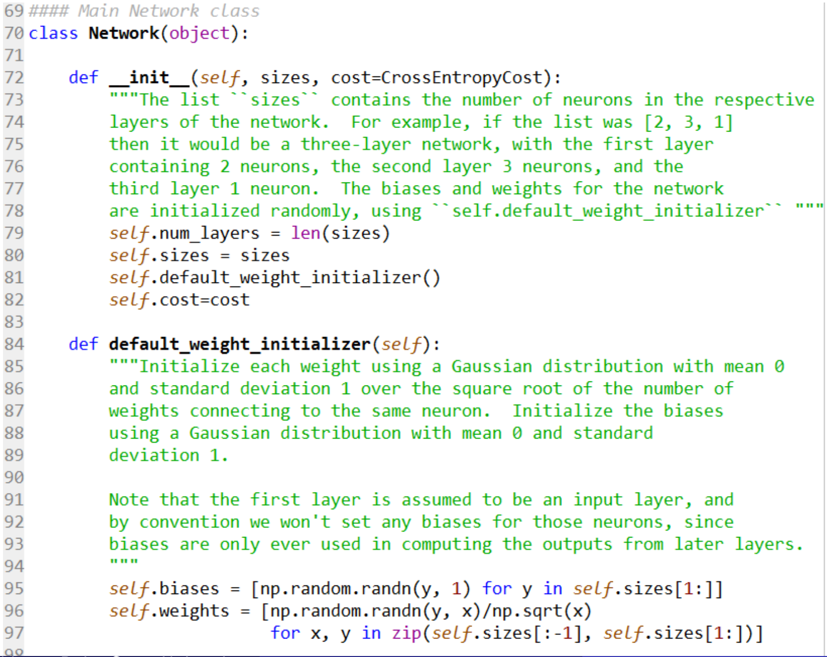
  (See slide 46)

Code for Network2.py



```python

class CrossEntropyCost(object):

    @staticmethod
    def fn(a, y):
        """Return the cost associated with an output ``a`` and desired output
        ``y``.  Note that np.nan_to_num is used to ensure numerical
        stability.  In particular, if both ``a`` and ``y`` have a 1.0
        in the same slot, then the expression (1-y)*np.log(1-a)
        returns nan.  The np.nan_to_num ensures that that is converted
        to the correct value (0.0).

        """
        return np.sum(np.nan_to_num(-y*np.log(a)-(1-y)*np.log(1-a)))

    @staticmethod
    def delta(z, a, y):
        """Return the error delta from the output layer.  Note that the
        parameter ``z`` is not used by the method.  It is included in
        the method's parameters in order to make the interface
        consistent with the delta method for other cost classes.

        """
        return (a-y)

```

Code for network2.py



```python
#### Main Network class
class Network(object):

    def __init__(self, sizes, cost=CrossEntropyCost):
        """The list ``sizes`` contains the number of neurons in the respective
        layers of the network.  For example, if the list was [2, 3, 1]
        then it would be a three-layer network, with the first layer
        containing 2 neurons, the second layer 3 neurons, and the
        third layer 1 neuron.  The biases and weights for the network
        are initialized randomly, using ``self.default_weight_initializer`` """
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.default_weight_initializer()
        self.cost=cost

    def default_weight_initializer(self):
        """Initialize each weight using a Gaussian distribution with mean 0
        and standard deviation 1 over the square root of the number of
        weights connecting to the same neuron.  Initialize the biases
        using a Gaussian distribution with mean 0 and standard
        deviation 1.

        Note that the first layer is assumed to be an input layer, and
        by convention we won't set any biases for those neurons, since
        biases are only ever used in computing the outputs from later layers.
        """
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)/np.sqrt(x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]
```

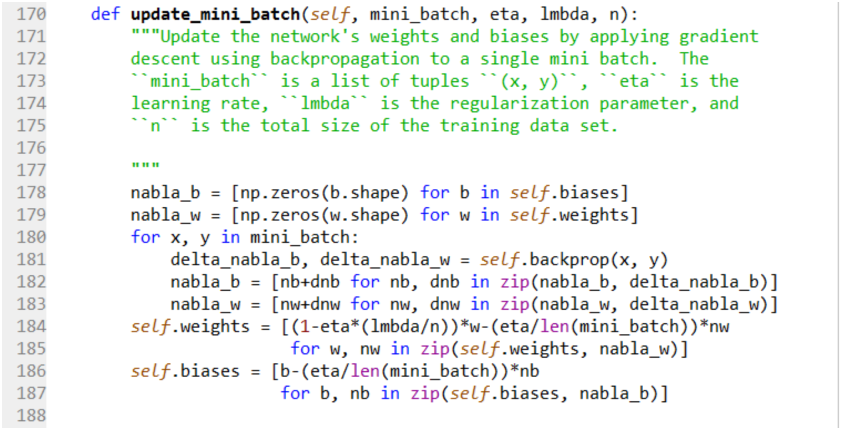
Code for Network2.py

```python

    def large_weight_initializer(self):
        """Initialize the weights using a Gaussian distribution with mean 0
        and standard deviation 1.  Initialize the biases using a
        Gaussian distribution with mean 0 and standard deviation 1.

        Note that the first layer is assumed to be an input layer, and
        by convention we won't set any biases for those neurons, since
        biases are only ever used in computing the outputs from later
        layers.

        This weight and bias initializer uses the same approach as in
        Chapter 1, and is included for purposes of comparison.  It
        will usually be better to use the default weight initializer
        instead.

        """
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(self.sizes[:-1], self.sizes[1:])]

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a
```

Code for Network2.py

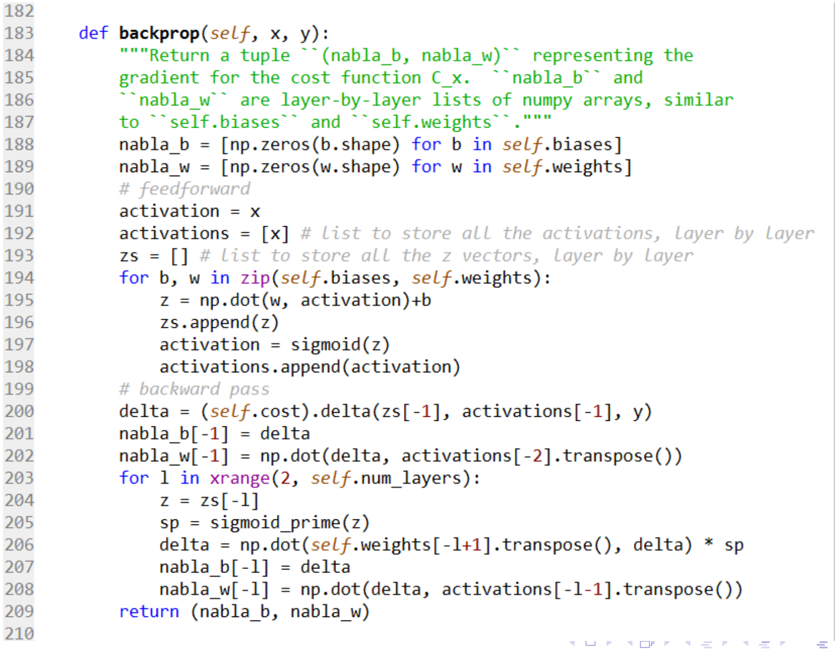
```

    def feedforward(self, a):
        """Return the output of the network if ``a`` is input."""
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            lmbda = 0.0,
            evaluation_data=None,
            monitor_evaluation_cost=False,
            monitor_evaluation_accuracy=False,
            monitor_training_cost=False,
            monitor_training_accuracy=False):
        """Train the neural network using mini-batch stochastic gradient
        descent.  The ``training_data`` is a list of tuples ``(x, y)``
        representing the training inputs and the desired outputs.  The
        other non-optional parameters are self-explanatory, as is the
        regularization parameter ``lmbda``.  The method also accepts
        ``evaluation_data``, usually either the validation or test
        data.  We can monitor the cost and accuracy on either the
        evaluation data or the training data, by setting the
        appropriate flags.  The method returns a tuple containing four
        lists: the (per-epoch) costs on the evaluation data, the
        accuracies on the evaluation data, the costs on the training
        data, and the accuracies on the training data.  All values are
        evaluated at the end of each training epoch.  So, for example,
        if we train for 30 epochs, then the first element of the tuple
        will be a 30-element list containing the cost on the
        evaluation data at the end of each epoch. Note that the lists
        are empty if the corresponding flag is not set.

        """
    def SGD(self, training_data, epochs, mini_batch_size, eta,lmbda = 0.0,
            evaluation_data=None,
            monitor_evaluation_cost=False,
            monitor_evaluation_accuracy=False,
            monitor_training_cost=False,
            monitor_training_accuracy=False):

        if evaluation_data: n_data = len(evaluation_data)
        n = len(training_data)
        evaluation_cost, evaluation_accuracy = [], []
        training_cost, training_accuracy = [], []
        for j in xrange(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in xrange(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta, lmbda, len(training_data))
            print "Epoch %s training complete" % j
            if monitor_training_cost:
                cost = self.total_cost(training_data, lmbda)
                training_cost.append(cost)
                print "Cost on training data: {}".format(cost)
            if monitor_training_accuracy:
                accuracy = self.accuracy(training_data, convert=True)
                training_accuracy.append(accuracy)
                print "Accuracy on training data: {} / {}".format(accuracy, n)
            if monitor_evaluation_cost:
                cost = self.total_cost(evaluation_data, lmbda, convert=True)
                evaluation_cost.append(cost)
                print "Cost on evaluation data: {}".format(cost)
            if monitor_evaluation_accuracy:
                accuracy = self.accuracy(evaluation_data)
                evaluation_accuracy.append(accuracy)
                print "Accuracy on evaluation data: {} / {}".format(
                    self.accuracy(evaluation_data), n_data)
            print
        return evaluation_cost, evaluation_accuracy, training_cost, training_accuracy
```

Code for Network2.py



```python
    def update_mini_batch(self, mini_batch, eta, lmbda, n):
        """Update the network's weights and biases by applying gradient
        descent using backpropagation to a single mini batch.  The
        ``mini_batch`` is a list of tuples ``(x, y)``, ``eta`` is the
        learning rate, ``lmbda`` is the regularization parameter, and
        ``n`` is the total size of the training data set.

        """
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        self.weights = [(1-eta*(lmbda/n))*w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

```

Code for Network2.py



```

    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).delta(zs[-1], activations[-1], y)
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

```

Backprop Code

```
class Network(object):
    def backprop(self, x, y):
        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] # store all the activations, layer by layer
        zs = [] # store 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  zs = [$z^{(2)}$  $z^{(3)}$] and activations = [$x$  $a^{(2)}$  $a^{(3)}$]

        delta = (self.cost).delta(zs[-1], activations[-1], y)
```

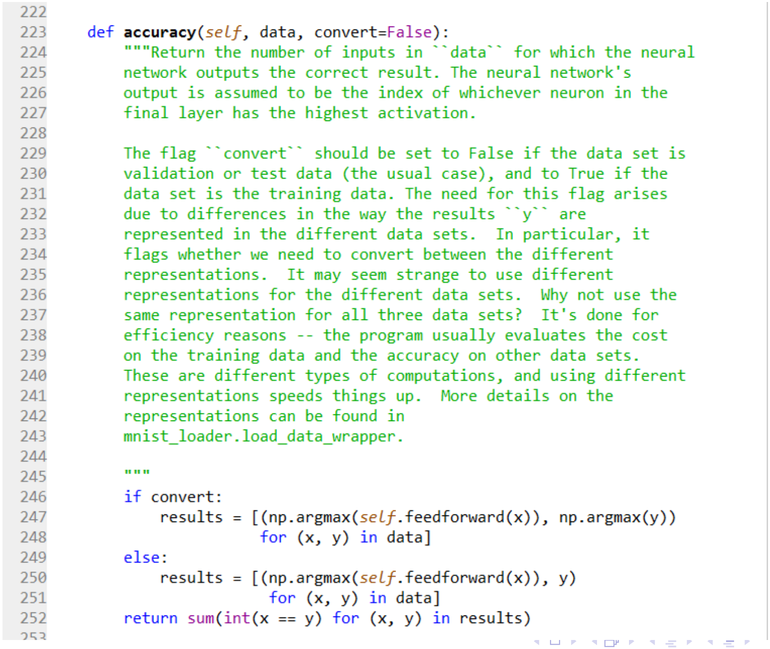
$$\texttt{nabla\_b[-1] = } \underbrace{\texttt{delta}}_{\delta^{(3)} \triangleq \frac{\partial C}{\partial z^{(3)}} = \frac{\partial C}{\partial b^{(3)}}}$$

$$\texttt{nabla\_w[-1] = np.dot(} \underbrace{\texttt{delta}}_{\delta^{(3)} \triangleq \frac{\partial C}{\partial z^{(3)}}} \texttt{,} \underbrace{\texttt{activations}[-2]\texttt{.transpose}()}_{a^{(2)T}}\texttt{)} \quad \texttt{\#}\ \delta^{(3)} a^{(2)T} \in \mathbb{R}^{n_o \times n_h}$$

Backprop Code

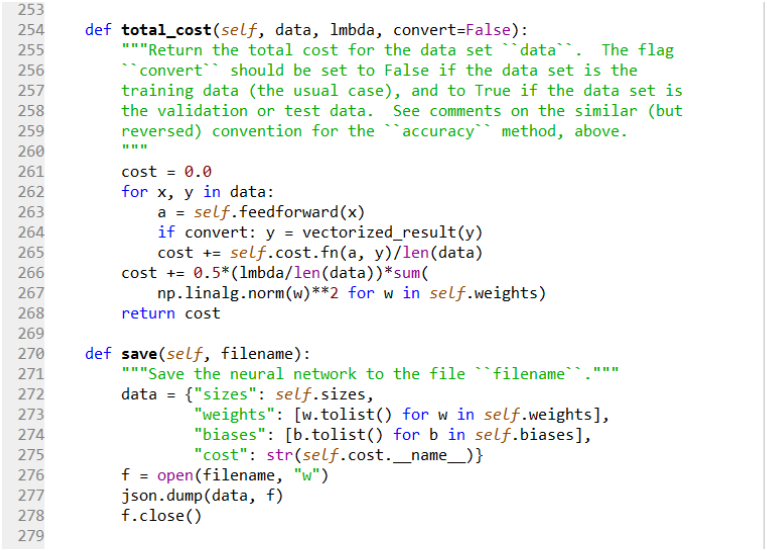
```
class Network(object):

        # backward pass  # zs = [$z^{(2)}$  $z^{(3)}$]and activations = [$x$  $a^{(2)}$  $a^{(3)}$]

        delta = (self.cost).delta(zs[-1], activations[-1], y)

        nabla_b[-1] =  $\underbrace{\texttt{delta}}_{\frac{\partial C}{\partial b^{(3)}}=\delta^{(3)}\in\mathbb{R}^{n_o}}$

        nabla_w[-1] = $\underbrace{\texttt{np.dot(delta, activations[-2].transpose())}}_{\frac{\partial C}{\partial W^{(3)}}=\delta^{(3)}a^{(2)T}\in\mathbb{R}^{n_o\times n_h}}$

        for l in xrange(2, self.num_layers):

            z = zs[-l]  # zs[-2] = $z^{(2)}$

            sp = sigmoid_prime(z)

            delta = $\underbrace{\texttt{np.dot(self.weights[-l+1].transpose(), delta) * sp}}_{\delta^{(2)}=W^{(3)T}\delta^{(3)}\odot\sigma'(z^{(2)})}$

            nabla_b[-l] = delta # $\frac{\partial C}{\partial b^{(2)}} = \delta^{(2)}$

            nabla_w[-l] = $\underbrace{\texttt{np.dot(delta, activations[-l-1].transpose())}}_{\frac{\partial C}{\partial W^{(2)}}=\delta^{(2)}x^T}$

            return (nabla_b, nabla_w)
```

Code for Network2.py



```python
    def accuracy(self, data, convert=False):
        """Return the number of inputs in ``data`` for which the neural
        network outputs the correct result. The neural network's
        output is assumed to be the index of whichever neuron in the
        final layer has the highest activation.

        The flag ``convert`` should be set to False if the data set is
        validation or test data (the usual case), and to True if the
        data set is the training data. The need for this flag arises
        due to differences in the way the results ``y`` are
        represented in the different data sets.  In particular, it
        flags whether we need to convert between the different
        representations.  It may seem strange to use different
        representations for the different data sets.  Why not use the
        same representation for all three data sets?  It's done for
        efficiency reasons -- the program usually evaluates the cost
        on the training data and the accuracy on other data sets.
        These are different types of computations, and using different
        representations speeds things up.  More details on the
        representations can be found in
        mnist_loader.load_data_wrapper.

        """
        if convert:
            results = [(np.argmax(self.feedforward(x)), np.argmax(y))
                       for (x, y) in data]
        else:
            results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in data]
        return sum(int(x == y) for (x, y) in results)
```

Code for Network2.py



```python

    def total_cost(self, data, lmbda, convert=False):
        """Return the total cost for the data set ``data``.  The flag
        ``convert`` should be set to False if the data set is the
        training data (the usual case), and to True if the data set is
        the validation or test data.  See comments on the similar (but
        reversed) convention for the ``accuracy`` method, above.
        """
        cost = 0.0
        for x, y in data:
            a = self.feedforward(x)
            if convert: y = vectorized_result(y)
            cost += self.cost.fn(a, y)/len(data)
        cost += 0.5*(lmbda/len(data))*sum(
            np.linalg.norm(w)**2 for w in self.weights)
        return cost

    def save(self, filename):
        """Save the neural network to the file ``filename``."""
        data = {"sizes": self.sizes,
                "weights": [w.tolist() for w in self.weights],
                "biases": [b.tolist() for b in self.biases],
                "cost": str(self.cost.__name__)}
        f = open(filename, "w")
        json.dump(data, f)
        f.close()

```

- `w.tolist()` converts the numpy array `w` to a python list

Code for Network2.py

```python

#### Loading a Network
def load(filename):
    """Load a neural network from the file ``filename``.  Returns an
    instance of Network.

    """
    f = open(filename, "r")
    data = json.load(f)
    f.close()
    cost = getattr(sys.modules[__name__], data["cost"])
    net = Network(data["sizes"], cost=cost)
    net.weights = [np.array(w) for w in data["weights"]]
    net.biases = [np.array(b) for b in data["biases"]]
    return net

#### Miscellaneous functions
def vectorized_result(j):
    """Return a 47-dimensional unit vector with a 1.0 in the j'th position
    and zeroes elsewhere.  This is used to convert a digit (0...9)
    into a corresponding desired output from the neural network.

    """
    e = np.zeros((47, 1))
    e[j] = 1.0
    return e

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))
```

Choosing Hyper-Parameters

- 30 hidden neurons, 30 epochs, mini-batch of 10,and the cross-entropy cost.
- $\lambda = 1000.0$, $\eta = 10$ (**bad guess for the MNIST data).**

```
import mnist_loader
training_data, validation_data, test_data = mnist_loader.load_data_wrapper()
import network2
net = network2.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 10.0, lmbda = 1000.0,
...  evaluation_data=validation_data, monitor_evaluation_accuracy=True)

Epoch 0 training complete
Accuracy on evaluation data:  1030 / 10000
Epoch 1 training complete
Accuracy on evaluation data:  990 / 10000
Epoch 2 training complete
Accuracy on evaluation data:  1009 / 10000
...
Epoch 27 training complete
Accuracy on evaluation data:  1009 / 10000
Epoch 28 training complete
Accuracy on evaluation data:  983 / 10000
Epoch 29 training complete
Accuracy on evaluation data:  967 / 10000
```

Our classification accuracies are no better than chance!

Choosing Hyper-Parameters

General approach to choosing hyper-parameters is to simplify the problem!

- E.g., eliminate all the data except the 0 digits and 1 digits.
  This reduces the data by 80% thus speeding up the training.
- Try a simple network. Perhaps [784 10] instead of [784 30 10].
- Increase the frequency of monitoring.
  E.g., we have 50000 training images on a [784 30 10] network.
  Try monitoring the validation accuracy after (say) training on
  1000 images instead of waiting to do so after 50,000 images
- Use fewer validation images.
  Instead of computing accuracy on 10,000 validation images use only 1000 images.

Choosing Hyper-Parameters

Example 1

```
net = network2.Network([784, 10])
net.SGD(training_data[:1000], 30, 10, 10.0, lmbda = 1000.0, \
....evaluation_data=validation_data[:100], \
....monitor_evaluation_accuracy=True)

Epoch 0 training complete
Accuracy on evaluation data:  10 / 100
Epoch 1 training complete
Accuracy on evaluation data:  10 / 100
Epoch 2 training complete
Accuracy on evaluation data:  10 / 100
..
```

- Pure noise as we only get 10% accuracy.
- However, we found this quickly as each epoch only takes a fraction of a second.

Choosing Hyper-Parameters

Example 2
We originally took the (bad) guess of $\lambda = 1000$.
We then reduced the training data from 50,000 to 1000.
We should reduce λ by 50 as well, i.e., take $\lambda = 20$.
(Recall the factor $1 - \frac{\eta\lambda}{n}$ to implement regularization. See slide 29)

```
net = network2.Network([784, 10])
net.SGD(training_data[:1000], 30, 10, 10.0, lmbda = 20.0, \
...  evaluation_data=validation_data[:100], \
...  monitor_evaluation_accuracy=True)
```

```
Epoch 0 training complete
Accuracy on evaluation data:  12 / 100
Epoch 1 training complete
Accuracy on evaluation data:  14 / 100
Epoch 2 training complete
Accuracy on evaluation data:  25 / 100
Epoch 3 training complete
Accuracy on evaluation data:  18 / 100
...
```

- We have something though not very good!

Choosing Hyper-Parameters

Example 3

We originally took the (bad) guess of $\eta = 10$.

Try $\eta = 100$

```
net = network2.Network([784, 10])
net.SGD(training_data[:1000], 30, 10, 100.0, lmbda = 20.0, \
...   evaluation_data=validation_data[:100], \
...   monitor_evaluation_accuracy=True)
```

```
Epoch 1 training complete
Accuracy on evaluation data:  10 / 100
Epoch 2 training complete
Accuracy on evaluation data:  10 / 100
Epoch 3 training complete
Accuracy on evaluation data:  10 / 100
...
```

- Back to noise! Need to reduce η.

Choosing Hyper-Parameters

Example 4
We originally took the (bad) guess of $\eta = 10$.
$\eta = 100$ was worse so try $\eta = 1$.

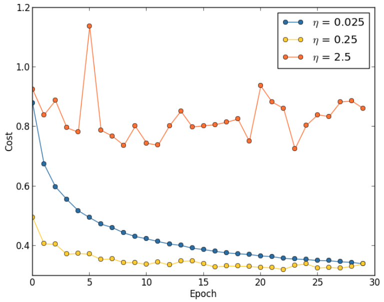
```
net = network2.Network([784, 10])
net.SGD(training_data[:1000], 30, 10, 1.0, lmbda = 20.0, \
...  evaluation_data=validation_data[:100], \
...  monitor_evaluation_accuracy=True)
```

```
Epoch 0 training complete
Accuracy on evaluation data:  62 / 100
Epoch 1 training complete
Accuracy on evaluation data:  42 / 100
Epoch 2 training complete
Accuracy on evaluation data:  43 / 100
Epoch 3 training complete
Accuracy on evaluation data:  61 / 100
```

- Better!
- Keep going: Add hidden neurons, adjust other hyper-parameters based on validation accuracy, etc.

Choosing Hyper-Parameters

Example 5 Learning Rate
$\eta = 0.025, \eta = 0.25, \eta = 2.5$.
30 epochs, mini-batch size of 10, $\lambda = 5.0$.
50000 training images.

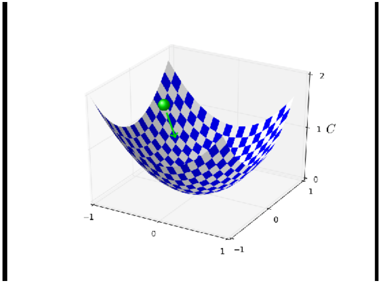


$$w^{(new)} = \left(1 - \frac{\eta\lambda}{n}\right) w^{(old)} - \eta \frac{\partial C_0}{\partial w}$$

Choosing Hyper-Parameters

Example 4 Learning Rate (continued)
Oscillations occur if η is too big.

$$w^{(new)} = \left(1 - \frac{\eta\lambda}{n}\right) w^{(old)} - \eta \frac{\partial C_0}{\partial w}$$



- We (theoretically) want to update the weights until $\dfrac{\partial C}{\partial w} = 0$.
- However, with η too big we can keep overshooting the minimum, i.e., oscillate.
- If η is too small then it will take "forever" to descend to the minimum value.

Learning Rate

- Choose η based on the training cost.
  $\eta's$ purpose is to control the step size in gradient descent.
  Monitoring the training cost is best way to detect if the step size is too big.
- First estimate the **threshold** (maximum value) of η as the value
  at which the cost on the training data decreases during the first few epochs.
  I.e., the cost should not be oscillating or increasing.
- E.g., try $\eta = 0.01$. If the cost decreases during the first few epochs,
  then sucessively try $\eta = 0.1, \eta = 1.0, \ldots$
- Keep increasing η until the cost oscillates or increases during the first few epochs.
- If the cost oscillates or increases with $\eta = 0.01$ then sucessively try
- $\eta = 0.001, 0.0001, \ldots$until the cost decreases during the first few epochs.
- The **threshold** is then the largest η at which the cost decreases during the first few epochs.
- To use η over many epochs one perhaps divide it by 2 after (say) five epochs
  so the steps are smaller as you get closer to the minimum.

Number of Training Epochs and Learning Rate Schedule

Number of Epochs

- At the end of each epoch compute the classification accuracy on the validation data.
- When the classification accuracy stops improving, terminate the weight updating.
- **Early Stopping rule:**

  If the classification accuracy hasn't improved during the last 10 epochs, then terminate.

  Or in the last 20 epochs, etc. Stopping rule gives a new hyper-parameter.

Learning Rate Schedule

We have been holding η constant.

It would be better to decrease η as the epochs go on.

How do we schedule the decreases for η? Use early stopping!

- Hold the learning rate constant until the validation accuracy starts to get worse.
- Then decrease the learning rate by (say) a factor of two.
- Continue until (say) η is $2^{10} = 1024.0$ times lower than its initial value.
- Then terminate.

Regularization Parameter

How do we choose λ?

- Set $\lambda = 0$ and determine the value for η.
- Then select a good value of λ using the validation data.

  Set λ to 1.

  If the accuracy goes up increase λ by a factor of 10.

  Else, decrease λ by a factor of 10.

Gradient Descent

Let $C(w_1, w_2)$ be a function of the two variables w_1, w_2.

A Taylor series expansion gives the approximation

$$C(w_1, w_2) \approx C(w_{01}, w_{02}) + \underbrace{\begin{bmatrix} \dfrac{\partial C(w_1, w_2)}{\partial w_1} & \dfrac{\partial C(w_1, w_2)}{\partial w_2} \end{bmatrix}}_{\left(\frac{\partial C}{\partial w}\right)^T_{|(w_{01}, w_{02})}} \underbrace{\begin{bmatrix} w_1 - w_{01} \\ w_2 - w_{02} \end{bmatrix}}_{\Delta w}$$

or

$$C(w) \approx C(w_0) + \left(\frac{\partial C}{\partial w}\right)^T_{|(w_{01}, w_{02})} \Delta w.$$

This approximation is valid for $||\Delta w|| = \sqrt{(\Delta w_1)^2 + (\Delta w_2)^2}$ "small".

With the constraint $||\Delta w|| \leq \epsilon$, $C(w)$ is minimized as a function of Δw by

$$\Delta w = -\epsilon \frac{1}{\|\partial C/\partial w\|} \frac{\partial C}{\partial w} \in \mathbb{R}^2.$$

Gradient Descent via the Hessian

Let $C(w_1, w_2)$ be a function of the two variables w_1, w_2.

A Taylor series expansion gives the (better) approximation

$$C(w_1,w_2) \approx C(w_{01},w_{02}) + \underbrace{\begin{bmatrix} \dfrac{\partial C(w_1,w_2)}{\partial w_1} & \dfrac{\partial C(w_1,w_2)}{\partial w_2} \end{bmatrix}}_{\left(\frac{\partial C}{\partial w}\right)^T_{|(w_{01},w_{02})}} \underbrace{\begin{bmatrix} w_1 - w_{01} \\ w_2 - w_{02} \end{bmatrix}}_{\Delta w}$$

$$+ \frac{1}{2!} \underbrace{\begin{bmatrix} w_1 - w_{01} & w_2 - w_{02} \end{bmatrix}}_{(\Delta w)^T} \underbrace{\begin{bmatrix} \dfrac{\partial^2 C(w_1,w_2)}{\partial^2 w_1} & \dfrac{\partial^2 C(w_1,w_2)}{\partial w_1 \partial w_2} \\ \dfrac{\partial^2 C(w_1,w_2)}{\partial w_2 \partial w_1} & \dfrac{\partial^2 C(w_1,w_2)}{\partial^2 w_2} \end{bmatrix}}_{H \triangleq \frac{\partial^2 C}{\partial w^2}{}_{|(w_{01},w_{02})}} \underbrace{\begin{bmatrix} w_1 - w_{01} \\ w_2 - w_{02} \end{bmatrix}}_{\Delta w}$$

or

$$C(w) \approx C(w_0) + \left(\frac{\partial C}{\partial w}\right)^T_{|(w_{01},w_{02})} \Delta w + \frac{1}{2!}(\Delta w)^T \underbrace{\frac{\partial^2 C(w_{01},w_{02})}{\partial w^2}}_{H} \Delta w.$$

This approximation is valid for $||\Delta w|| = \sqrt{(\Delta w_1)^2 + (\Delta w_2)^2}$ "small".

Gradient Descent via the Hessian

We have the Taylor series approximation

$$C(w, \Delta w) \approx C(w_0) + \left(\frac{\partial C}{\partial w}\right)^T_{|(w_{01}, w_{02})} \Delta w + \frac{1}{2!}(\Delta w)^T \frac{\partial^2 C}{\partial w^2}_{|(w_{01}, w_{02})} \Delta w.$$

This approximation is valid for $||\Delta w|| = \sqrt{(\Delta w_1)^2 + (\Delta w_2)^2}$ “small”.

Set

$$\frac{\partial C(w, \Delta w)}{\partial(\Delta w)} = \frac{\partial C(w_{01}, w_{02})}{\partial w} + \frac{\partial^2 C(w_{01}, w_{02})}{\partial w^2}\Delta w = 0_{2\times 1}$$

to obtain

$$\Delta w = -\left(\frac{\partial^2 C(w_{01}, w_{02})}{\partial w^2}\right)^{-1} \frac{\partial C(w_{01}, w_{02})}{\partial w} = -H^{-1}\nabla C \in \mathbb{R}^2$$

$$\text{Set } w^{(new)} = w^{(old)} - \eta H^{-1}\nabla C$$

where η is chosen so that $||\Delta w|| = \left\|\eta H^{-1}\nabla C\right\|$ is not too big.

Gradient Descent via the Hessian

$$\Delta w = -\left(\frac{\partial^2 C}{\partial w^2}\right)^{-1} \frac{\partial C}{\partial w} = -H^{-1}\nabla C$$

If there are n weights then the Hessian is

$$\frac{\partial^2 C(w_1, w_2, \ldots, w_n)}{\partial w^2} \in \mathbb{R}^{n\times n}.$$

This is a *symmetric* matrix.

n diagonal elements and $(n^2 - n)/2$ off diagonal elements to compute.

This requires a total of $n + (n^2 - n)/2 = n(n+1)/2$ computations.

E.g., if $n = 24,000$ then $H = \dfrac{\partial^2 C(w_1, w_2, \ldots, w_n)}{\partial w^2}$ requires 288×10^6 computations!

Further, we must then invert a $24,000 \times 24,000$ matrix

- Hessian approach is avoided due to the computational requirements.

Momentum Based Gradient Descent

- Hessian approach $\Delta w = -\left(\dfrac{\partial^2 C}{\partial w^2}\right)^{-1}\dfrac{\partial C}{\partial w}$ uses the gradient $\dfrac{\partial C}{\partial w}$ and the rate of change of the gradient (Hessian) $\dfrac{\partial^2 C}{\partial w^2}$.
- Momentum-based gradient descent uses similar idea, but avoids inversion of a large matrix.

Motivation for the Momentum Approach

Think of $C(w)$ as the potential function of a particle of mass m with w the particle's position.

Then $-\dfrac{\partial C}{\partial w}$ would be the force on the particle and $-\dfrac{1}{m}\dfrac{\partial C}{\partial w}$ its acceleration.

The change in velocity is $v^{(new)} = v^{(old)} - (\Delta t)\dfrac{1}{m}\dfrac{\partial C^{(old)}}{\partial w}$ where Δt is the time step.

The change in position is then $w^{(new)} = w^{(old)} + (\Delta t)v^{(new)}$

Momentum Based Gradient Descent

Cost as a Potential Function From previous slide:

$$v^{(new)} = v^{(old)} - (\Delta t)\frac{1}{m}\frac{\partial C^{(old)}}{\partial w}$$
$$w^{(new)} = w^{(old)} + (\Delta t)v^{(new)}$$

Momentum Based Gradient Descent (momentum $p = mv$)

$$v^{(new)} = \mu v^{(old)} - \eta\frac{\partial C^{(old)}}{\partial w} = v^{(old)}\underbrace{-(1-\mu)v^{(old)}}_{\text{viscous friction}} - \eta\frac{\partial C^{(old)}}{\partial w}, \quad \eta = (\Delta t)\frac{1}{m}$$

$$w^{(new)} = w^{(old)} + v^{(new)}, \quad \Delta t = 1$$

- $0 < \mu < 1$ is the **momentum coefficient** hyper-parameter.
- $1 - \mu$ represents the "viscous friction" coefficient.
- If $\mu = 0$ then we simply have

$$v^{(new)} = -\eta\frac{\partial C^{(old)}}{\partial w}$$
$$w^{(new)} = w^{(old)} - \eta\frac{\partial C^{(old)}}{\partial w}$$

Momentum Based Gradient Descent

Momentum Based Gradient Descent

$$\begin{aligned} v^{(new)} &= \mu v^{(old)} - \eta \frac{\partial C^{(old)}}{\partial w} \\ w^{(new)} &= w^{(old)} + v^{(new)} \end{aligned}$$

- Simple modification to the code to include momentum.

$$\begin{bmatrix} v^{(new)} \\ w^{(new)} \end{bmatrix} = \begin{bmatrix} \mu & 0 \\ \mu & 1 \end{bmatrix} \begin{bmatrix} v^{(old)} \\ w^{(old)} \end{bmatrix} - \eta \begin{bmatrix} \partial C^{(old)}/\partial w \\ \partial C^{(old)}/\partial w \end{bmatrix}$$

$$\Longrightarrow \begin{bmatrix} v^{(n)} \\ w^{(n)} \end{bmatrix} = \begin{bmatrix} \mu & 0 \\ \mu & 1 \end{bmatrix}^n \begin{bmatrix} v^{(0)} \\ w^{(0)} \end{bmatrix} - \sum_{i=0}^{n-1} \begin{bmatrix} \mu & 0 \\ \mu & 1 \end{bmatrix}^{n-1-i} \left(\eta \begin{bmatrix} \partial C^{(i)}/\partial w \\ \partial C^{(i)}/\partial w \end{bmatrix} \right)$$

where $\begin{bmatrix} \mu & 0 \\ \mu & 1 \end{bmatrix}^k = \begin{bmatrix} \mu^k & 0 \\ \sum_{i=1}^k \mu^i & 1 \end{bmatrix}$

- What is the problem with choosing $\mu > 1$?
- What is the problem with choosing $\mu < 0$?

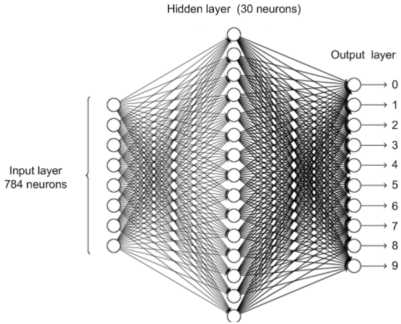
Softmax Activations

- Softmax is an activation function used only in the output layer.
- The net input $z_j^{(L)}$ variables to the activation of the **output layer** (layer L) are

$$z_j^{(L)} = \sum_{k=1}^{n_h} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

- Instead of a sigmoid, the **output activations** are given by

$$a_j^{(L)} \triangleq \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}}.$$



Softmax Activation

- Softmax is an activation function used only in the output layer.
- The weighted input z variables to the **output layer** (layer L) are

$$z_j^{(L)} = \sum_{k=1}^{n_h} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)} \quad \text{and} \quad a_j^{(L)} \triangleq \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}}.$$

- The hidden layers still use sigmoids.
- Note that

$$0 \le a_j^{(L)} = \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \le 1$$

and

$$\sum_{j=1}^{n_o} a_j^{(L)} = \sum_{j=1}^{n_o} \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} = \frac{1}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \sum_{j=1}^{n_o} e^{z_j^{(L)}} = 1$$

- We can consider the output activations as **probabilities**.

Softmax Activation

- Softmax is an activation function used only in the output layer.
- The weighted input z variables to the **output layer** (layer L) are

$$z_j^{(L)} = \sum_{k=1}^{n_h} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)} \quad \text{and} \quad a_j^{(L)} \triangleq \frac{e^{z_j^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}}.$$

- With **softmax** $a_j^{(L)}$ is now a function of $z_1^{(L)}, z_2^{(L)}, \ldots, z_{n_o}^{(L)}$.
- Compare with the **sigmoid** output where $a_j^{(L)} = \sigma(z_j^{(L)})$ is only a function of $z_j^{(L)}$!
- Exercise: Show

$$\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} > 0 \quad \text{while for } j \neq k \quad \frac{\partial a_j^{(L)}}{\partial z_k^{(L)}} < 0$$

- Exercise: Show

$$z_j^{(L)} = \ln(a_j^{(L)}) + \text{CONSTANT}$$

  where CONSTANT does not depend on j.

Maximum Likelihood Cost with Softmax

Let the input digit $x \in \mathbb{R}^{784}$ be a **seven** with corresponding label $j = 7$.
Then the one-hot output vector is

$$j = 7 \Longleftrightarrow y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & y_8 = 1 & 0 & 0 \end{bmatrix}^T.$$

The corresponding **output activations** are

$$\begin{bmatrix} a_1^{(L)}(x) & a_2^{(L)}(x) & a_3^{(L)}(x) & a_4^{(L)}(x) & a_5^{(L)}(x) & a_6^{(L)}(x) & a_7^{(L)}(x) & a_8^{(L)}(x) & a_9^{(L)}(x) & a_{10}^{(L)}(x) \end{bmatrix}^T.$$

- We want $a_8^{(L)}(x)$ to be the highest probability for this image.
- I.e., $a_8^{(L)}(x)$ is to be the **most likely** value or to have the **maximum likelihood**.
- The log-likelihood cost function defined by

$$C_j(x) \triangleq -\ln a_{j+1}^{(L)}(x).$$

Maximum Likelihood Cost with Softmax

With $j \in \{0, 1, 2, ..., 9\}$ the label for the image x, the log-likelihood cost for this image is

$$C_j(x) \triangleq -\ln a_{j+1}^{(L)}(x)$$

- $C_j(x) \geq 0$ as $0 \leq a_{j+1}^{(L)}(x) \leq 1$ and $C_j(x) = 0$ only if $a_{j+1}^{(L)}(x) = 1$.
- With one-hot encoding

$$C_j(x) = -\sum_{k=1}^{10} y_k \ln a_k^{(L)}(x) = -\ln a_{j+1}^{(L)}(x) \text{ as } y_{j+1} = 1 \text{ and } y_k = 0 \text{ for } k \neq j+1.$$

- The output activations $0 \leq a_k^{(L)} \leq 1$ and are normalized, i.e., $\sum_{k=1}^{10} a_k^{(L)} = 1$.
  So if $a_{j+1}^{(L)}(x)$ is close to one we must have $a_k^{(L)}(x)$ close to zero for $k \neq j+1$.
- Compare with **cross-entropy** which is

$$-\sum_{k=1}^{10}\Big(y_k \ln a_k^{(L)}(x) + (1-y_k)\ln(1-a_k^{(L)}(x))\Big) = -\ln a_{j+1}^{(L)}(x) - \sum_{k\neq j+1}^{10} \ln(1-a_k^{(L)}(x))$$

  as $y_{j+1} = 1$ and $y_k = 0$ for $k \neq j+1$.
- Recall that for cross-entropy the activations are sigmoids satisfying $0 \leq a_k^{(L)} \leq 1$.
  They are **not** normalized, i.e., $\sum_{k=1}^{10} a_k^{(L)}$ is not forced to be 1.
  The cost term $-\sum_{k\neq j+1}^{10} \ln(1-a_k^{(L)}(x))$ forces $a_k^{(L)}(x)$ to be close to zero for $k \neq j+1$.

Maximum Likelihood Cost with Softmax

Output Layer:

$$z_m^{(L)} = \sum_{\ell=1}^{n_h} w_{m\ell}^{(L)} a_\ell^{(L-1)} + b_m^{(L)} \quad \text{and} \quad a_m^{(L)} \triangleq e^{z_m^{(L)}} \Big/ \left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right) \quad \text{for} \quad m = 1, 2, \ldots, 10.$$

Let $j \in \{0, 1, 2, \ldots, 9\}$ be the label for the image x. Then

$$\begin{aligned}
\frac{\partial C_j}{\partial b_{j+1}^{(L)}} &= \frac{\partial}{\partial b_{j+1}^{(L)}} \left(-\ln a_{j+1}^{(L)} \right) \\
&= -\frac{1}{a_{j+1}^{(L)}} \frac{\partial}{\partial b_{j+1}^{(L)}} \left(\frac{e^{z_{j+1}^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \right) \\
&= -\frac{1}{a_{j+1}^{(L)}} \left(\frac{1}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \frac{\partial e^{z_{j+1}^{(L)}}}{\partial b_{j+1}^{(L)}} - \frac{e^{z_{j+1}^{(L)}}}{\left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right)^2} \frac{\partial}{\partial b_{j+1}^{(L)}} \left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right) \right) \\
&= -\frac{1}{a_{j+1}^{(L)}} \left(\frac{1}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \cdot e^{z_{j+1}^{(L)}} - \frac{e^{z_{j+1}^{(L)}}}{\left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right)^2} e^{z_{j+1}^{(L)}} \right)
\end{aligned}$$

Maximum Likelihood Cost with Softmax

Continue from previous slide:

$$\frac{\partial C_j}{\partial b_{j+1}^{(L)}} = \frac{\partial}{\partial b_{j+1}^{(L)}} \left(- \ln a_{j+1}^{(L)} \right)$$

$$= -\frac{1}{a_{j+1}^{(L)}} \left(\frac{1}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \cdot e^{z_{j+1}^{(L)}} - \frac{e^{z_{j+1}^{(L)}}}{\left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right)^2} e^{z_{j+1}^{(L)}} \right)$$

$$= -\frac{1}{a_{j+1}^{(L)}} \frac{e^{z_{j+1}^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \left(\underbrace{1}_{y_{j+1}} - \frac{e^{z_{j+1}^{(L)}}}{\sum_{k=1}^{n_o} e^{z_k^{(L)}}} \right)$$

$$= a_{j+1}^{(L)} - y_{j+1}$$

- **Note:** $\partial C_j / \partial z_{j+1}^{(L)} = \partial C_j / \partial b_{j+1}^{(L)}$

Maximum Likelihood Cost with Softmax

$$z_m^{(L)} = \sum_{k=1}^{n_h} w_{mk}^{(L)} a_k^{(L-1)} + b_m^{(L)} \quad \text{and} \quad a_m^{(L)} \triangleq e^{z_m^{(L)}} \Big/ \left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right) \quad \text{for} \quad m = 1, 2, ..., 10.$$

Let $j \in \{0, 1, 2, ..., 9\}$ be the label for the image x. Then

$$\begin{aligned}
\frac{\partial C_j}{\partial w_{j+1,k}^{(L)}} &= \frac{\partial}{\partial w_{j+1,k}^{(L)}} \left(-\ln a_{j+1}^{(L)} \right) = -\frac{1}{a_{j+1}^{(L)}} \frac{\partial}{\partial w_{j+1,k}^{(L)}} \left(\frac{e^{z_{j+1}^{(L)}}}{\sum_{i=1}^{n_o} e^{z_i^{(L)}}} \right) \\
&= -\frac{1}{a_{j+1}^{(L)}} \left(\frac{1}{\sum_{i=1}^{n_o} e^{z_i^{(L)}}} \frac{\partial}{\partial w_{j+1,k}^{(L)}} e^{z_{j+1}^{(L)}} - \frac{e^{z_{j+1}^{(L)}}}{\left(\sum_{i=1}^{n_o} e^{z_i^{(L)}} \right)^2} \frac{\partial}{\partial w_{j+1,k}^{(L)}} \left(\sum_{i=1}^{n_o} e^{z_i^{(L)}} \right) \right) \\
&= -\frac{1}{a_{j+1}^{(L)}} \left(\frac{1}{\sum_{i=1}^{n_o} e^{z_i^{(L)}}} \cdot e^{z_{j+1}^{(L)}} \frac{\partial z_{j+1}^{(L)}}{\partial w_{j+1,k}^{(L)}} - \frac{e^{z_{j+1}^{(L)}}}{\left(\sum_{i=1}^{n_o} e^{z_i^{(L)}} \right)^2} e^{z_{j+1}^{(L)}} \frac{\partial z_{j+1}^{(L)}}{\partial w_{j+1,k}^{(L)}} \right) \\
&= -\frac{1}{a_{j+1}^{(L)}} \frac{e^{z_{j+1}^{(L)}}}{\sum_{i=1}^{n_o} e^{z_i^{(L)}}} \frac{\partial z_{j+1}^{(L)}}{\partial w_{j+1,k}^{(L)}} \left(\underbrace{1}_{y_{j+1}} - \frac{e^{z_{j+1}^{(L)}}}{\sum_{i=1}^{n_o} e^{z_i^{(L)}}} \right) \\
&= a_k^{(L-1)} \left(a_{j+1}^{(L)} - y_{j+1} \right)
\end{aligned}$$

Maximum Likelihood Cost with Softmax

$$z_m^{(L)} = \sum_{\ell=1}^{n_h} w_{m\ell}^{(L)} a_\ell^{(L-1)} + b_m^{(L)} \text{ and } a_m^{(L)} \triangleq e^{z_m^{(L)}} \Big/ \left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right) \text{ for } m = 1, 2, ..., 10.$$

If the image x has the label $j = 7$ then the one-hot encoded output vector is

$$j = 7 \Longleftrightarrow y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & y_8 = 1 & 0 & 0 \end{bmatrix}^T.$$

$$C_j(x) = -\sum_{k=1}^{10} y_k \ln a_k^{(L)}(x) = -\ln a_{j+1}^{(L)} \text{ and we computed } \frac{\partial C_j}{\partial b_{j+1}^{(L)}} \;\&\; \frac{\partial C_j}{\partial w_{j+1,k}^{(L)}}.$$

For this same cost $C_j(x) = -\sum_{k=1}^{10} y_k \ln a_k^{(L)}(x) = -\ln a_{j+1}^{(L)}$ **you** can show that

$$\frac{\partial C_j}{\partial b_m^{(L)}} = \frac{\partial}{\partial b_m^{(L)}} \left(-\ln a_{j+1}^{(L)} \right) = a_m^{(L)} \quad \text{for } \; m \neq j+1$$

$$\frac{\partial C_j}{\partial w_{mk}^{(L)}} = \frac{\partial}{\partial w_{mk}^{(L)}} \left(-\ln a_{j+1}^{(L)} \right) = a_k^{(L-1)} a_m^{(L)} \quad \text{for } \; m \neq j+1 \text{ and } \; k = 1, ..., n_h.$$

Summary: Maximum Likelihood Cost with Softmax

Output Layer:

$$z_m^{(L)} = \sum_{\ell=1}^{n_h} w_{m\ell}^{(L)} a_\ell^{(L-1)} + b_m^{(L)} \quad \text{and} \quad a_m^{(L)} \triangleq e^{z_m^{(L)}} / \left(\sum_{k=1}^{n_o} e^{z_k^{(L)}} \right) \quad \text{for} \quad m = 1, 2, ..., 10.$$

$$j = 7 \Longleftrightarrow y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & y_8 = 1 & 0 & 0 \end{bmatrix}^T$$

The cost is

$$C_j(x) = -\sum_{k=1}^{10} y_k \ln a_k^{(L)}(x) = -\ln a_{j+1}^{(L)}.$$

The gradients for the output layer are

$$\frac{\partial C_j}{\partial b_m^{(L)}} = a_m^{(L)} - \delta_{m,j+1} \qquad \text{for} \quad m = 1, ..., n_o$$

$$\frac{\partial C_j}{\partial w_{mk}^{(L)}} = a_k^{(L-1)}(a_m^{(L)} - \delta_{m,j+1}) \quad \text{for} \quad m = 1, ..., n_o \quad \text{and} \quad k = 1, ..., n_h.$$

Total Cost Function

The **total cost** (average cost) is

$$C \triangleq \frac{1}{n}\sum_{(x,j)\in\text{data}} C_j(x) = \frac{1}{n}\sum_{(x,y)\in\text{data}}\left(-\sum_{k=1}^{10} y_k \ln a_k^{(L)}(x)\right) = -\frac{1}{n}\sum_{(x,j)\in\text{data}} \ln a_{j+1}^{(L)}(x)$$

where n is the length of the training *data* (or minibatch size).

The gradients of the **total cost** are simply

$$\frac{\partial C}{\partial b_m^{(L)}} = \frac{1}{n}\sum_{(x,j)\in\text{data}} \frac{\partial C_j}{\partial b_m^{(L)}} = \frac{1}{n}\sum_{(x,j)\in\text{data}}\left(a_m^{(L)}(x) - \delta_{m,j+1}\right) \qquad m = 1, \ldots, n_o$$

$$\frac{\partial C}{\partial w_{mk}^{(L)}} = \frac{1}{n}\sum_{(x,j)\in\text{data}} \frac{\partial C_j}{\partial w_{mk}^{(L)}} = \frac{1}{n}\sum_{(x,j)\in\text{data}} a_k^{(L-1)}(x)\left(a_m^{(L)}(x) - \delta_{m,j+1}\right) \qquad \begin{array}{l} m = 1, \ldots, n_o \\ k = 1, \ldots, n_h. \end{array}$$