A practical course on building and training neural networks using the Theano library

Why use neural networks

- Used for everyday tasks
  - Speech recognition
  - Face recognition
  - Character recognition
  - Handwriting recognition
- Employed by tech companies big and small

Google facebook Google DeepMind

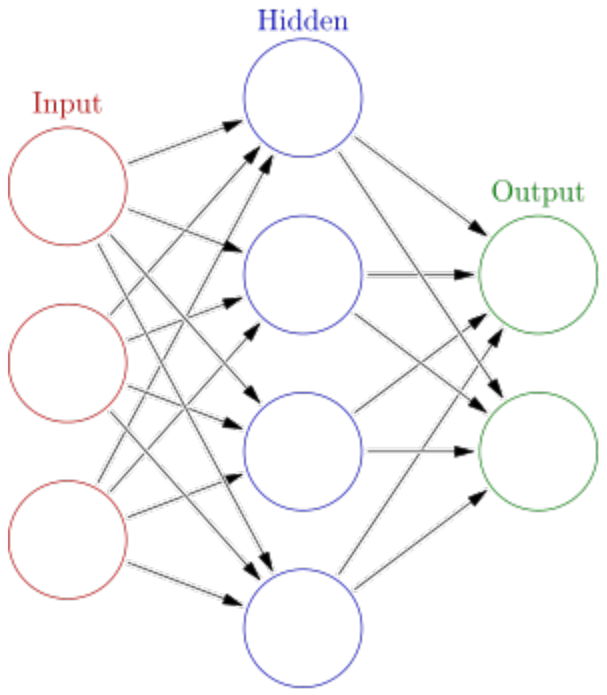
swype Jukedeck

Comparison with other ML methods

- PROs:
  - Work really well with lots of training data and computing power
  - Can be easily parallelized to take advantage of large computing clusters
- CONs:
  - Difficult to debug
  - Hard to impose prior beliefs about data
  - Require many tricks and trial and error to work well

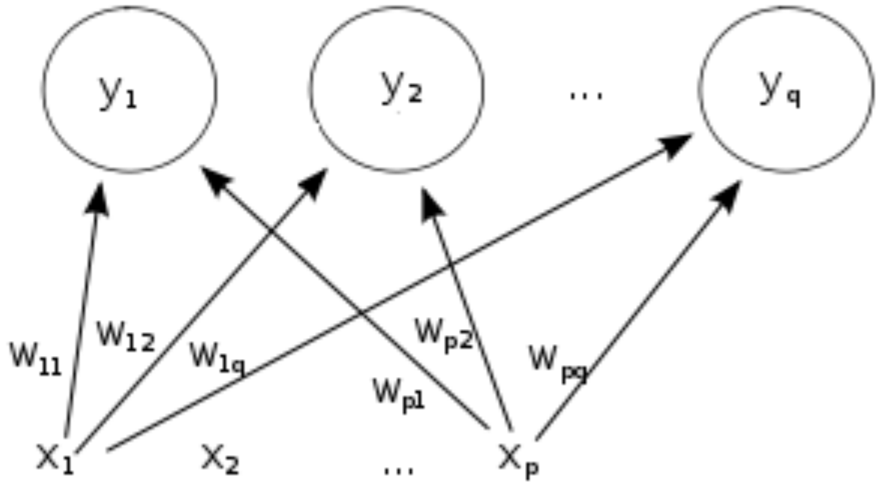
Model structure

- Layers of units (aka artificial neurons)
  - Input layer
    - Encodes the data
  - Output layer (supervised learning)
    - Encodes the label
  - Hidden layers
    - Increase the complexity of the model
- Weights between units
  - Model parameters
- Units take on scalar real values
- In neural nets, the model is a function approximator
- In belief nets, the units' values are probabilities
  - The activation of each unit has a probabilistic interpretation



Model structure: 2 layers

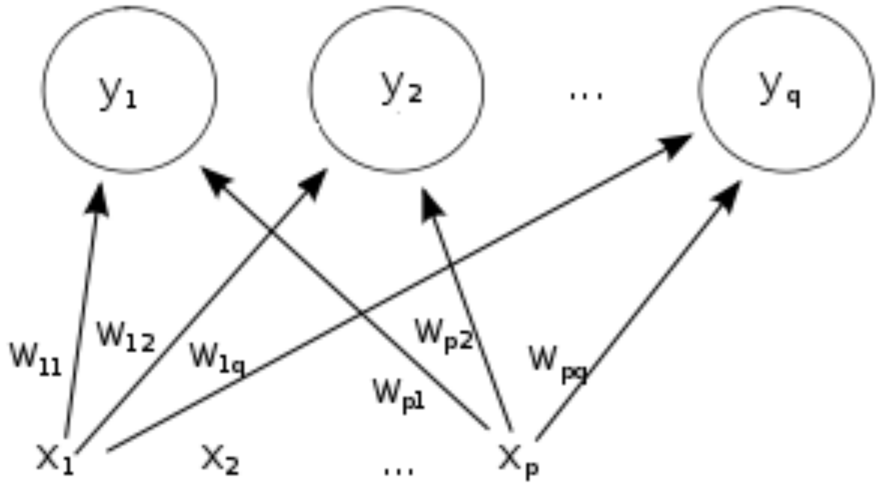
- Input units: x
- Output units: y
- Weights w, biases b
- Forward model
- Linear units:



$$\underset{\overset{\uparrow}{\text{output}}}{y_j} = \sum_i \overset{\overset{\text{weight}}{\downarrow}}{w_{ij}} \underset{\overset{\uparrow}{\text{input}}}{x_i} + \overset{\overset{\text{bias}}{\downarrow}}{b_j}$$

Model structure: 2 layers

- Input units: x
- Output units: y
- Weights w, biases b
- Linear units:



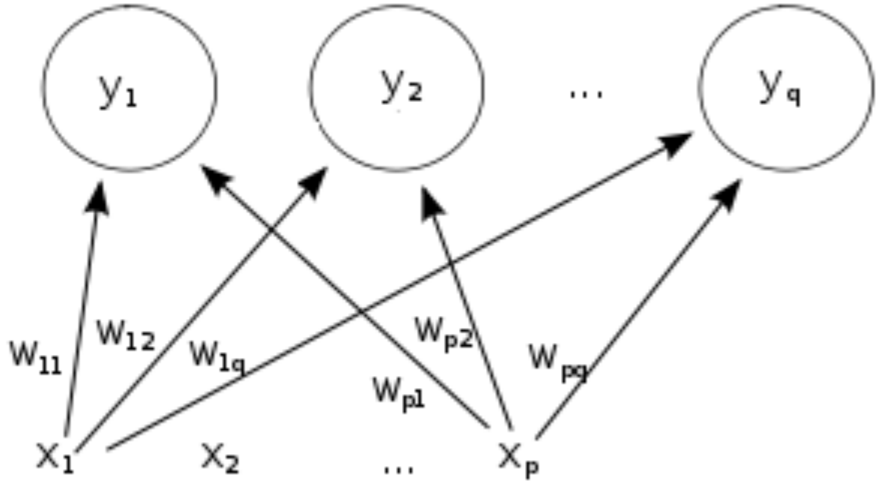
$$y_j = \sum_i w_{ij} x_i + b_j$$

$$y_j = \mathbf{w}_j^{\mathbf{T}} \mathbf{x} + b_j$$

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

Model structure: 2 layers

- Input units: x
- Output units: y
- Weights: w
- Non-linear units:



$$y_j = f\left(\sum_i w_{ij} x_i + b_j\right)$$

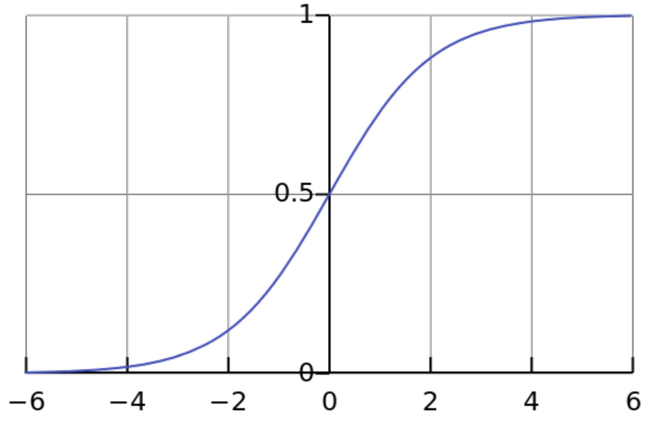
$$\mathbf{y} = f\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$

Common nonlinearities

- Most common nonlinear functions:
  - Sigmoid curve:

$$\sigma(t) = \frac{1}{1+e^{-t}}$$

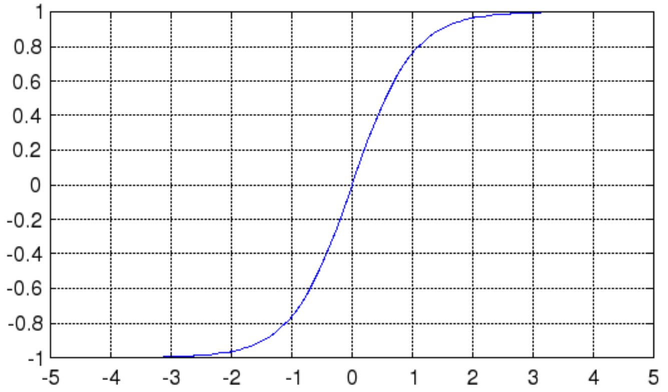
$$\sigma(t) \in [0,1]$$



  - Hyperbolic tangent:

$$\tanh(t) = \frac{e^{t} - e^{-t}}{e^{t} + e^{-t}}$$

$$\tanh(t) \in [-1,1]$$

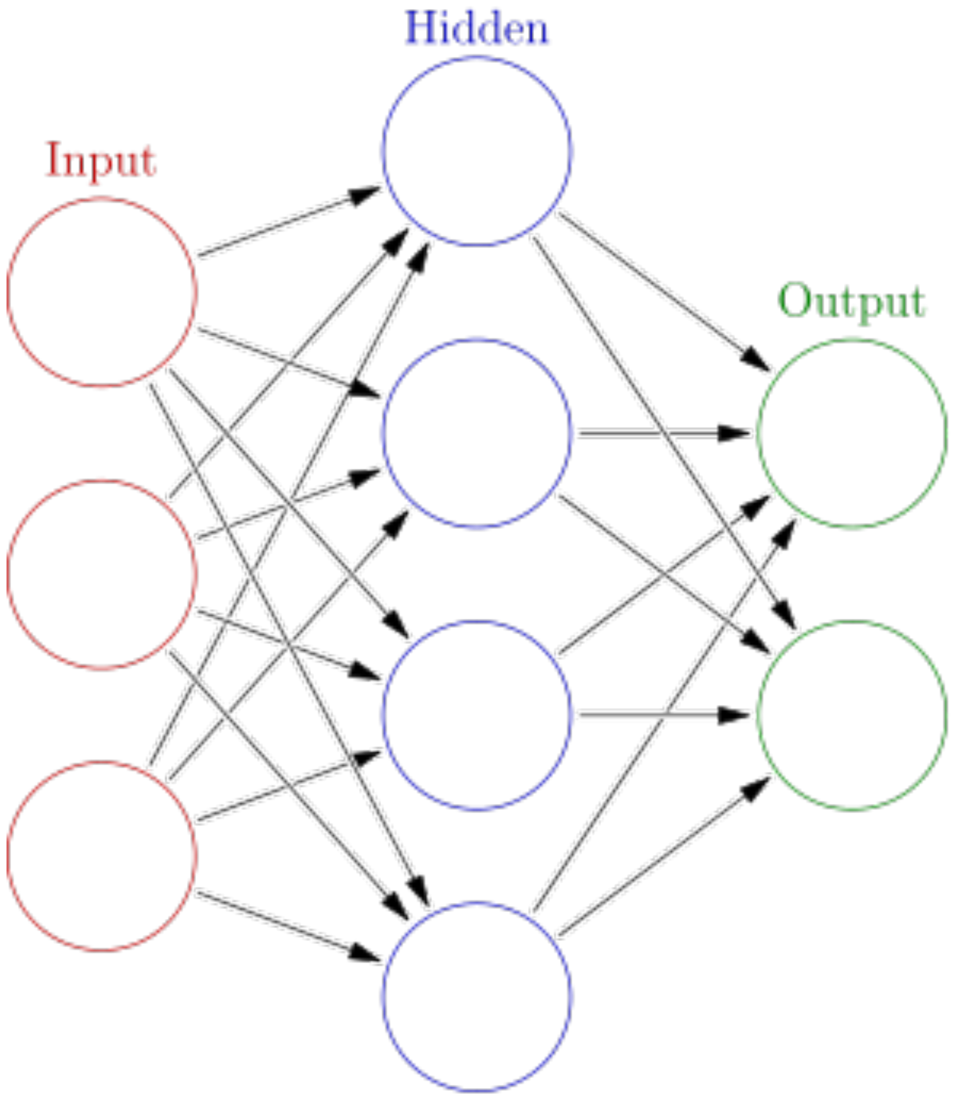


Model structure: 3 layers

- Input units: x
- Hidden units: h
- Output units: y
- Weights: $\mathbf{W}_{xh}$ $\mathbf{W}_{hy}$

$$\mathbf{h} = f_h\left(\mathbf{W}_{xh}\mathbf{x}\right)$$

$$\mathbf{y} = f_y\left(\mathbf{W}_{hy}\mathbf{h}\right)$$



Parameter estimation (learning)

- We want the model to perform as well as possible on the training set, so:
  - Define an error function that quantifies model performance on the training data, and minimize it
    - Squared error (regression): $E = \sum_i \left(\hat{y}_i - y_i \right)^2$
    - Negative log likelihood (classification): $E = -\sum_i \hat{y}_i \log y_i$
  - Minimize via stochastic gradient descent
  - Backpropagation algorithm: for 3+ layers, the calculus chain rule is used when computing the gradient
  - Theano computes the analytic gradients for us

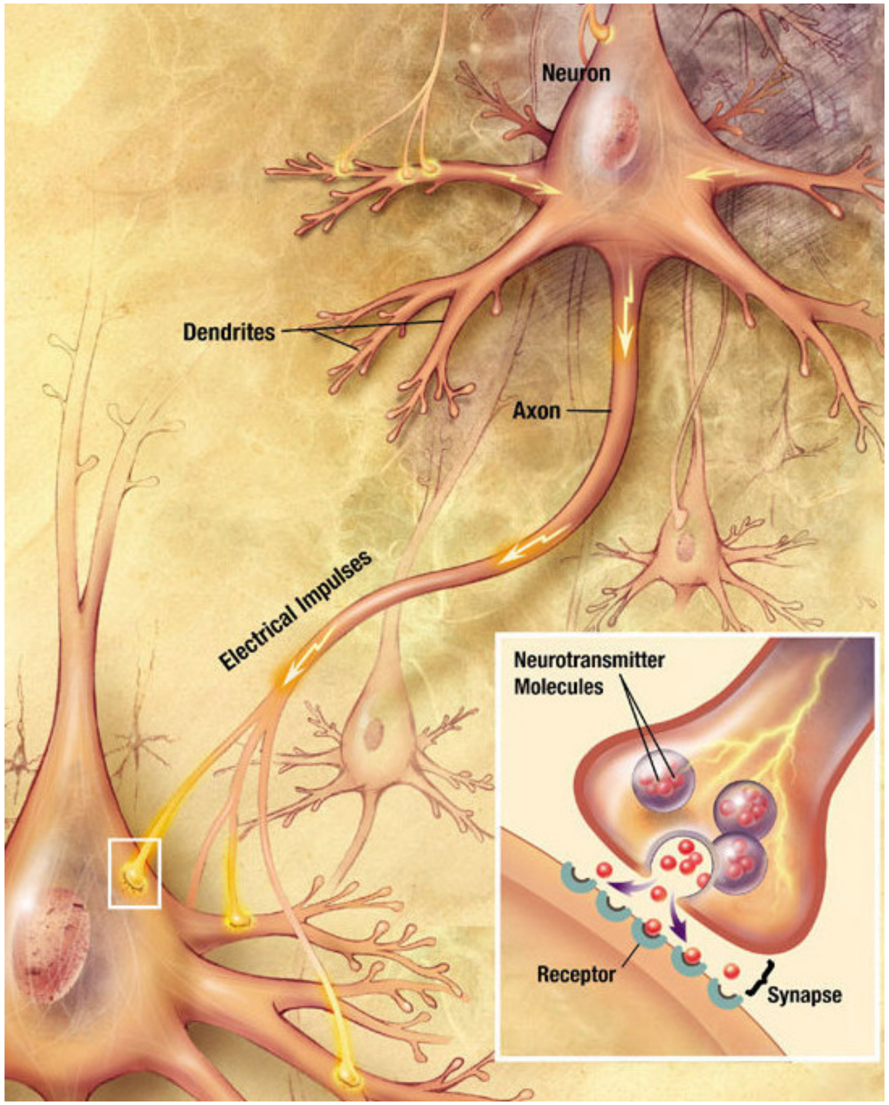
Logistic regression

- Prediction using logistic regression is equivalent to:
  - 2-layer neural network (input and output)
  - single output unit
  - sigmoid non-linearity

$$P\left(Y=1 \mid X=\mathbf{x}\right)=\sigma\left(\mathbf{w}^{\mathbf{T}}\mathbf{x}+\mathbf{b}\right)$$

  - Train by minimizing negative log-likelihood
  - We will implement this first, as it is a natural stepping stone to neural networks

Neural inspiration



Similarities:

- Biological neurons
  - Units (aka neurons)
- Axon
  - Unit activation (output)
- Dendrite
  - Unit input
- Synapses
  - Weights
- Probability of synaptic release
  - Weight value

Differences:

- Spiking, asynchronous computation
- Probably different learning algorithms
- Larger: ~10^{11} neurons, ~10^{15} synapses

History

1943	McCulloch and Pitts create a computational model of neural networks
1949	Hebbian learning: unsupervised algorithm based on neural plasticity
1958	Perceptron (linear 2-layer net)
1975	Backpropagation algorithm, multi-layer perceptron
1990-2009	Research stagnates as support vector machines overtake neural nets
2009-now	Thanks to larger datasets and more computational power, the same methods we had in 1975 are performing really well (+ a few tricks)
2009	Fast GPU implementations of neural nets

History

1943	McCulloch and Pitts create a computational model of neural networks
1949	Hebbian learning: unsupervised algorithm based on neural plasticity
1958	Perceptron (linear 2-layer net)
1975	Backpropagation algorithm, multi-layer perceptron
1990-2009	Research stagnates as support vector machines overtake neural nets
2009-now	Thanks to larger datasets and more computational power, the same methods we had in 1975 are performing really well (+ a few tricks)
2009	Fast GPU implementations of neural nets

Leading research labs

Geoff Hinton (Google, U of Toronto)

Yann LeCun (Facebook, NYU)

Yoshua Bengio (U of Montreal)

Logistic regression with Theano: MNIST handwritten digit recognition

Multiclass logistic regression

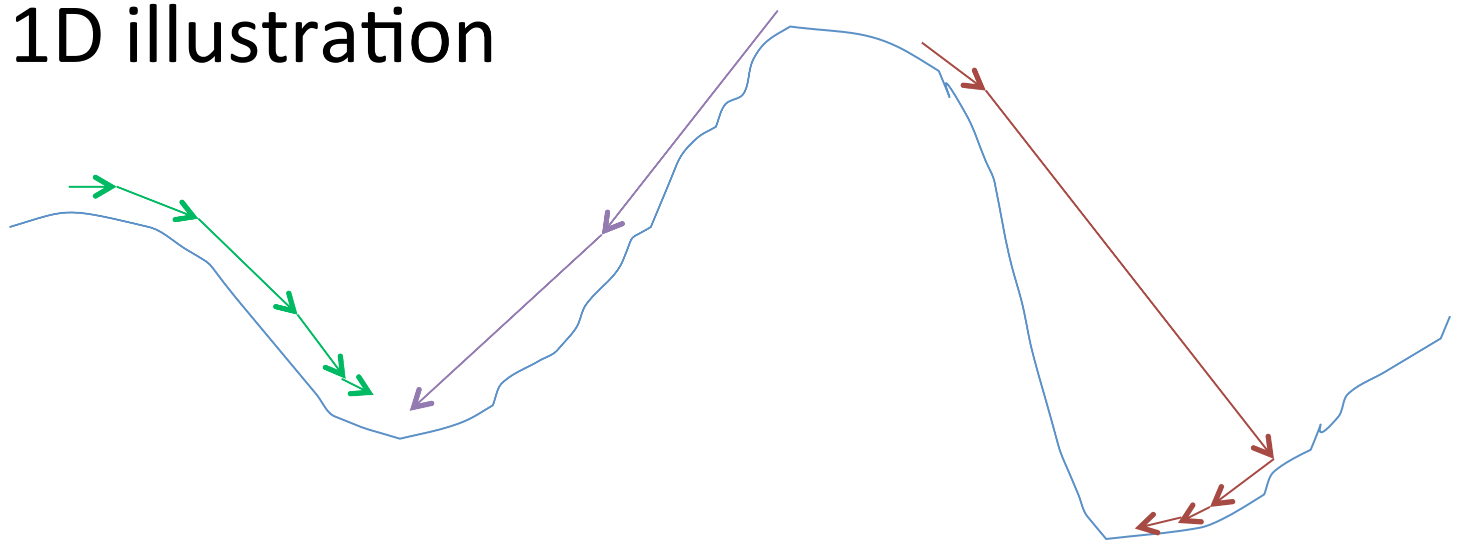
- Multiclass logistic regression is equivalent to:
  - 2-layer neural network (input and output)
  - one output unit for each class
  - sigmoid non-linearity
  - Output layer should be a distribution, therefore output units' activations must add up to 100%

$$P\left(Y_c = 1 \mid X = \mathbf{x}\right) = \frac{\sigma\left(\mathbf{w}_c{}^{\mathbf{T}}\mathbf{x} + b_c\right)}{\sum_{c'} \sigma\left(\mathbf{w}_{c'}{}^{\mathbf{T}}\mathbf{x} + b_{c'}\right)}$$

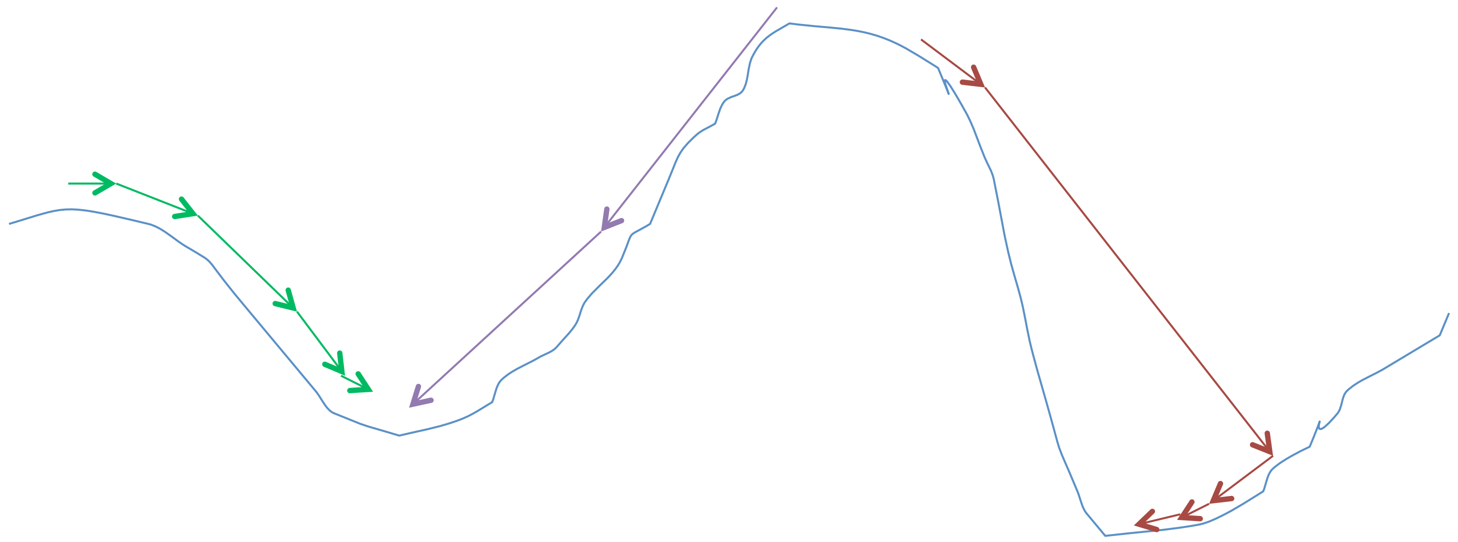
  - Train by minimizing negative log-likelihood via gradient descent

Gradient descent

- Function minimization algorithm
- Takes successive steps in the direction of steepest descent
- Steps are proportional to the slope
- 1D illustration



Gradient descent: pseudocode



Given a function $f : {}^\circ{}^{n} \to {}^\circ$

Choose an arbitrary starting point $\mathbf{x} \in {}^\circ{}^{n}$

Choose a learning rate λ

Update $\mathbf{x}$ until an ending condition is met:

$$\mathbf{x} \leftarrow \mathbf{x} - \lambda \nabla f$$

Theano

- Mathematical package for Python
- PROs:
  - Widely used in academia and small industry labs
  - Can compute gradients analytically given symbolic functions
  - Has computational optimizations for common neural net operations
  - Supports CUDA, allowing training on GPUs
- CONs:
  - Difficult to debug
  - Probably difficult to optimize for really complicated stuff
- Main alternative: Torch
  - Used by Google, Facebook, IBM, Yandex
  - Uses the Lua scripting language (lower level language similar to C)
  - Does not compute gradients
  - Probably best for larger research groups
- Other alternatives: OpenNN, Deeplearning4j

Theano: Installation

```
pip install theano
```

- If problems arise (on Windows), try

```
pip install theano==0.6.0
```

Theano: Hello World

```
>>> import theano
>>> import theano.tensor as T
>>> x = T.dscalar()
>>> y = T.dscalar()
>>> z = x**2 + y**3
>>> z.eval({x:2,y:2})
array(12.0)
>>> z.eval({x:3,y:1})
array(10.0)
```

$$z = x^2 + y^3$$

Theano: Computing derivatives

```
>>> import theano
>>> import theano.tensor as T
>>> x = T.dscalar()
>>> y = T.dscalar()
>>> z = x**2 + y**3
>>> T.grad(z,x).eval({x:2,y:2})
array(4.0)
>>> T.grad(z,x).eval({x:3,y:1})
array(6.0)
>>> T.grad(z,y).eval({x:2,y:2})
array(12.0)
```

$$z = x^2 + y^3$$

$$\frac{\partial z}{\partial x} = 2x$$

$$\frac{\partial z}{\partial y} = 3y^2$$

Theano: How differentiation works

- Theano differentiation is exact
- Takes advantage of the calculus chain rule

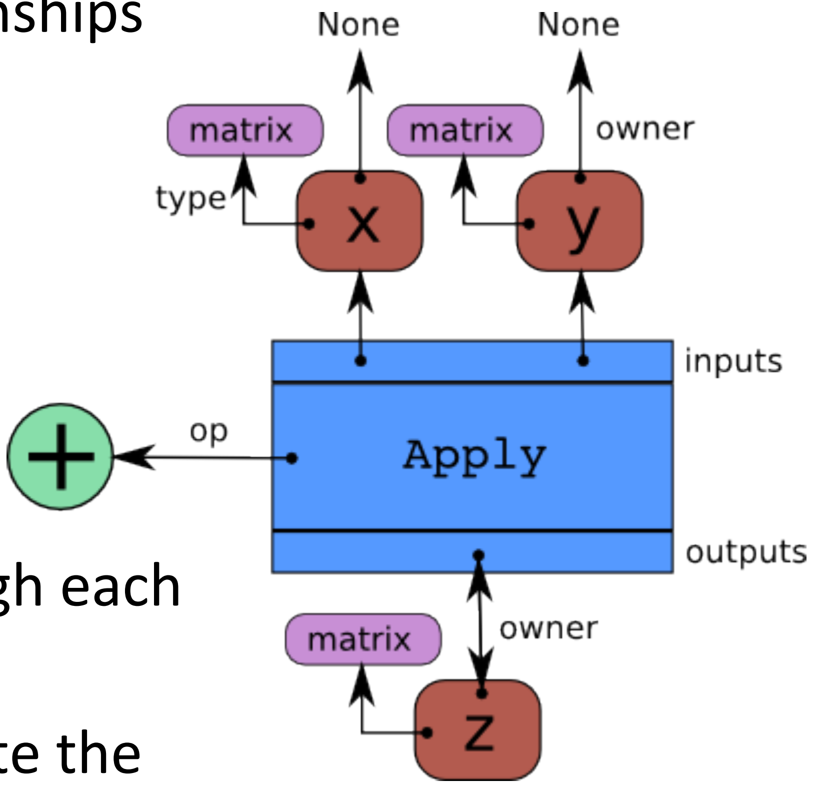
$$F(x) = f(g(x)) \rightarrow F'(x) = f'(g(x))g'(x)$$

$$\left.\begin{array}{l} y = g(x) \\ z = f(y) \end{array}\right\} \rightarrow \frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Theano: How differentiation works

- When variables are defined, theano also builds a graph of these variables and their relationships

```
>>> x = T.dmatrix()
>>> y = T.dmatrix()
>>> z = x + y
```



- To evaluate a function, go through each node and update the result
- To compute a derivative, compute the derivative of that node, then multiply all of the derivatives together

Theano: Compile to function

- Theano compiles the graph to C code every time we evaluate a variable
- For efficiency, we can create theano functions:

```
>>> x = T.dscalar()
>>> y = T.dscalar()
>>> z = x**2 + y**3
>>> f = theano.function(inputs=[x, y], outputs = z)
>>> f(2, 3)
array(31.0)
```

Theano: Linear Algebra and Gradients

```
>>> W = T.dmatrix()
>>> x = T.dvector()
>>> y = T.dot(x,T.dot(W,x))
>>> param_dictionary = {W:[[1,0],[1,1]], x:[3,5]}
>>> y.eval(param_dictionary)
array(49.0)
>>> T.grad(y,W).eval(param_dictionary)
array([[  9.,  15.],
       [ 15.,  25.]])
>>> T.grad(y,x).eval(param_dictionary)
array([ 11.,  13.])
```

$$y = \mathbf{x}^{\mathrm{T}}\mathbf{W}\mathbf{x}$$

$$\frac{\partial y}{\partial \mathbf{W}} = \mathbf{x}\mathbf{x}^{\mathrm{T}}$$

$$\frac{\partial y}{\partial \mathbf{x}} = \mathbf{x}^{\mathrm{T}}\left(\mathbf{W} + \mathbf{W}^{\mathrm{T}}\right)$$

Theano: Shared variables

- Create persistent variables
  - We need these for model parameters
  - Called “shared” because their value can be shared between multiple theano functions

```
>>> from numpy import array
>>> x = T.dvector()
>>> a = theano.shared(value = array([2,3]))
>>> y = T.dot(a, x)
>>> y.eval({x:[5,7]})
array(31.0)
>>> a.set_value(array([3,4]))
>>> y.eval({x:[5,7]})
array(43.0)
```

Theano: updates and givens

- `theano.function` has two other parameters that we will make use of:
  - `updates`
    - we will use this to update the model parameters
  - `givens`
    - we will use this to set the input and output variables to the values in our dataset

Theano: updates

- takes a list of pairs of the form `(shared_variable, new_expression)`
- after computing the function, it updates the values of the shared variables in the list
- we will use this to create a training function that takes a gradient descent step whenever we call it

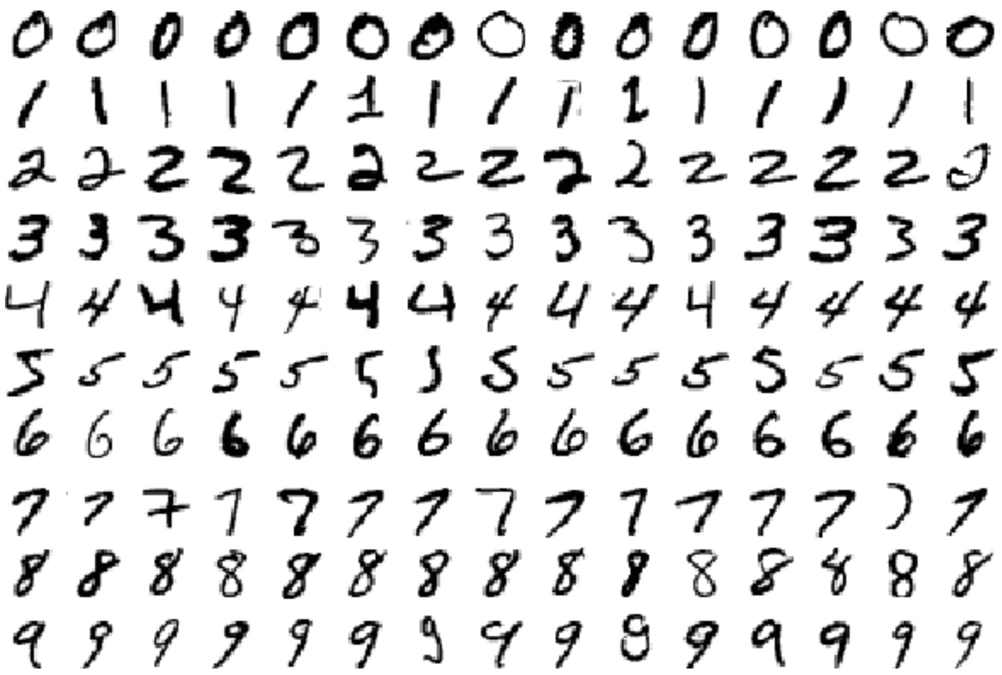
```
>>> x = T.dscalar()
>>> a = theano.shared(value = 0)
>>> f = theano.function(inputs=[x], outputs=a+x, updates=[[a,a+1]])
>>> a.get_value()
array(0)
>>> f(3)
array(3.0)
>>> a.get_value()
array(1)
>>> f(3)
array(4.0)
```

Theano: givens

- takes a dictionary of the form `{variable: value, …}`
- when computing the function, it uses the given values for the corresponding variables instead
- we will use this as follows:
  - we will have variables for the input and output of the model
  - when training, we will use `givens` to set both input and output to the desired values, when testing we will only set the input

```
>>> x = T.dscalar()
>>> y = T.dscalar()
>>> f = theano.function(inputs=[x], outputs=x+y, givens = {y: 2*x})
>>> f(3)
array(9.0)
```

Digit recognition: MNIST dataset



- Long tradition in neural networks research
  - Like using fruit flies for genetics research
- Labeled data: 28x28px grayscale images
- Split into 60,000 training and 10,000 test images

Loading the MNIST dataset

- Download Python gzipped pickle from:

```
http://www.iro.umontreal.ca/~lisa/deep/data/mnist/mnist.pkl.gz
```

- Load into Python:

```
import gzip
import cPickle
f = gzip.open('mnist.pkl.gz', 'rb')
train_set, valid_set, test_set = cPickle.load(f)
f.close()
```

- Here, the training data (60k samples) is split into 50k train and 10k validation
- Each of the three variables is an `(image, labels)` tuple, where `image` is an `(n_datapoints, 28`^{2}`)` `numpy.ndarray` and labels is an `(n_datapoints, )` `numpy.ndarray`

```
>>> train_set[0].shape
(50000L, 784L)
>>> train_set[1].shape
(50000L,)
```

MNIST historical performance

Year	Type	Classifier	Distortion	Preprocessing	Error rate
1998	Linear classifier	Pairwise linear classifier	None	Deskewing	7.6%
1998	Non-Linear Classifier	40 PCA + quadratic classifier	None	None	3.3%
2002	Support vector machine	Virtual SVM, deg-9 poly, 2-pixel jittered	None	Deskewing	0.56%
2003	Neural network	2-layer 784-800-10	None	None	1.6%
2003	Neural network	2-layer 784-800-10	elastic distortions	None	0.7%
2007	K-Nearest Neighbors	K-NN with non-linear deformation (P2DHMDM)	None	Shiftable edges	0.52%
2009	Boosted Stumps	Product of stumps on Haar features	None	Haar features	0.87%
2010	Deep neural network	6-layer 784-2500-2000-1500-1000-500-10	elastic distortions	None	0.35%
2012	Convolutional neural network	Committee of 35 conv. net, 1-20-P-40-P-150-10	elastic distortions	Width normalizations	0.23%

Putting it all together

- Load the dataset
- Create variables for
  - Digit image vector
  - Digit label
  - Shared variables: weights and biases, datasets
  - Cost function, classification error, intermediate variables
- Write a theano function for training that has:
  - Function input: the data point index
  - Function output: the cost function
  - Updates: gradient descent parameter update
  - Givens: image vector and digit label based on the index
- Write a theano function for testing that has:
  - Function input: the data point index
  - Function output: the digit label
  - Givens: image vector based on the index

Load the dataset

- First get the dataset into Python
- Also compute the number of training points, test points, dimensionality and classes

```
import gzip
import cPickle

f = gzip.open('C:/nnets/mnist.pkl.gz', 'rb')
train_set, valid_set, test_set = cPickle.load(f)
f.close()

n_train, n_test = map(lambda x:len(x[0]), [train_set, test_set])
dims = train_set[0].shape[1]
n_classes = len(set(train_set[1]))
```

Declare variables that are not computed

- X is a matrix where each row is a the digit image shaped as a vector
- y is the vector of labels corresponding to the rows of X
- W and b are the weights and biases
  - These are shared variables because they will be used for both training and testing

```
import numpy
import theano
import theano.tensor as T

X = T.dmatrix()
y = T.ivector()

W = theano.shared(numpy.zeros([dims,n_classes]))
b = theano.shared(numpy.zeros(n_classes))
```

Declare variables that are not computed

- Create theano shared variables for the training and test datasets
  - The data in our pickle file is using types `float32` and `int64` for the image and label respectively; we recast it to the default theano types
  - These will force theano to load all data into memory and be more efficient

```
prepare_data = (lambda x: (theano.shared(x[0].astype('float64')),
    theano.shared(x[1].astype('int32'))))
(training_X, training_y), (test_X, test_y) = map(prepare_data,
    [train_set, test_set])
```

Declare the other variables

- y_hat is the activation in the final layer
  - The estimated label probability distribution
  - We use the built-in theano function `softmax` that applies a sigmoid function to each vector entry, then normalizes the vector to sum to 1
  - The matrix multiplication is the transpose of what we saw earlier

$$P\left(Y \mid X = \mathbf{x}\right) = \mathrm{SoftMax}\left(\mathbf{X}_{\#data\_points \times \#features} \mathbf{W}_{\#features \times \#classes} + \mathbf{b}_{\#classes}\right)$$

```
y_hat = T.nnet.softmax(T.dot(X,W) + b)
```

Declare the other variables

- *y_pred* is the predicted class
  - We take the most likely class using the theano `argmax` function
- *test_error* is the proportion of misclassified digits
  - We use the theano `neq` and `mean` functions

```
y_pred = T.argmax(y_hat, axis=1)
test_error = T.mean(T.neq(y_pred, y))
```

Declare the other variables

- *training_error* is the negative log likelihood
  - We use the mean rather than the sum so that the gradient steps don't need to change as we change the size of our dataset
  - `[T.arange(y.shape[0]), y]` returns a vector containing the value in column `y` of each row

```
training_error = -T.mean(T.log(y_hat)[T.arange(y.shape[0]), y])
```

Declaring the gradient updates

- Implement the gradient descent updates

$$\mathbf{W} \leftarrow \mathbf{W} - \lambda \frac{\partial E}{\partial \mathbf{W}}$$

$$\mathbf{b} \leftarrow \mathbf{b} - \lambda \frac{\partial E}{\partial \mathbf{b}}$$

```
learning_rate = .5
updates = [
        [W, W - learning_rate * T.grad(training_error, W)],
        [b, b - learning_rate * T.grad(training_error, b)]
    ]
```

Compiling the theano functions

- Training function
  - No inputs, outputs the training_error, though this is not needed for the training
  - The updates parameter performs the gradient descent
  - The givens parameter substitutes the training set for the x and y variables
- Test function
  - No inputs, outputs the test_error, though this is not needed for the training
  - The givens parameter substitutes the test set for the x and y variables

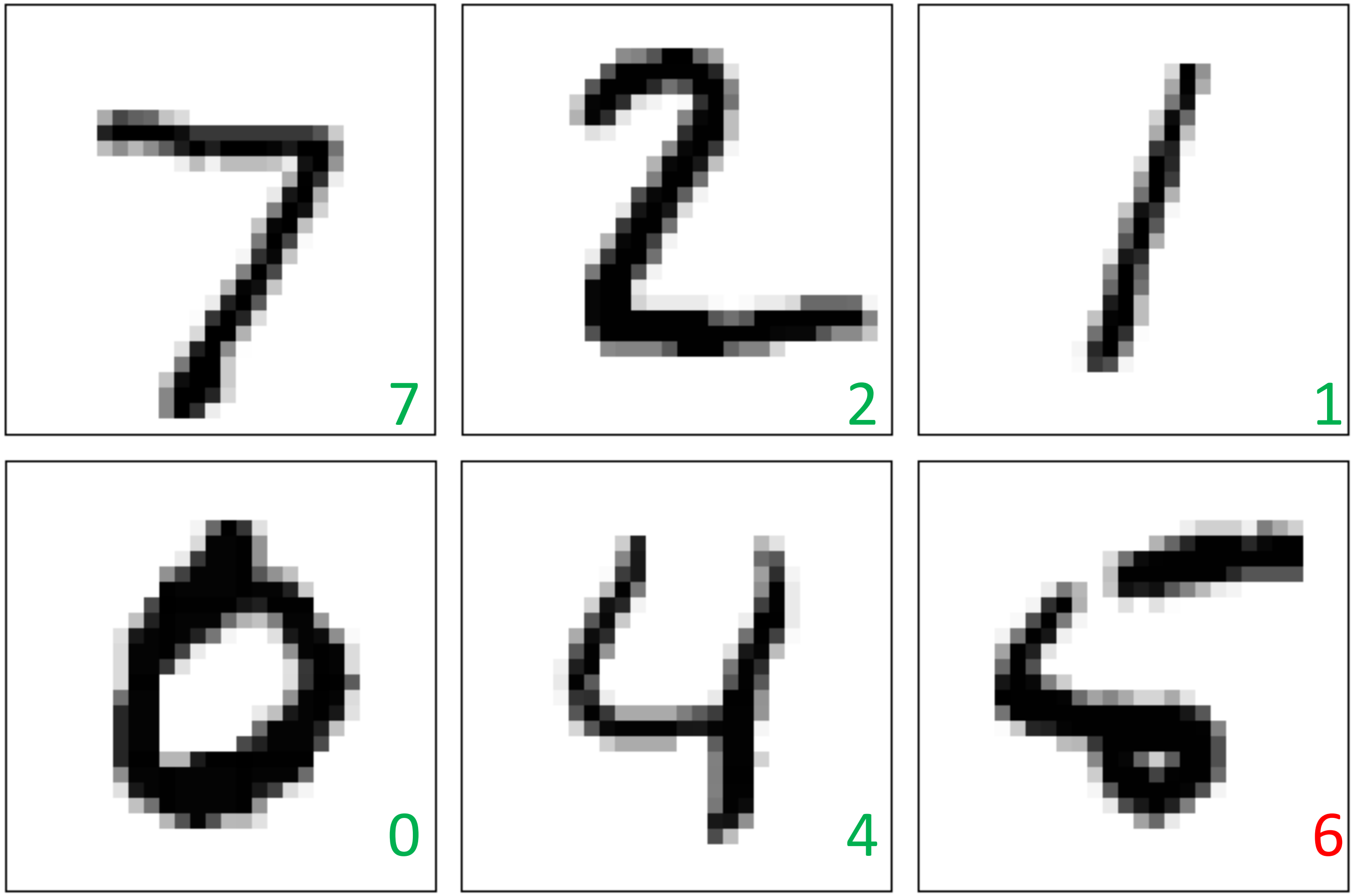
```
training_function = theano.function(
    inputs = [],
    outputs = training_error,
    updates = updates,
    givens = {X:training_x, y: training_y}
    )

test_function = theano.function(
    inputs = [],
    outputs = test_error,
    givens = {X: test_x, y: test_y}
    )
```

Running and testing

- Run for 100 training cycles (aka epochs)
  - Note that the mean negative log likelihood (the output of the training function) is computed **before** the gradient updates

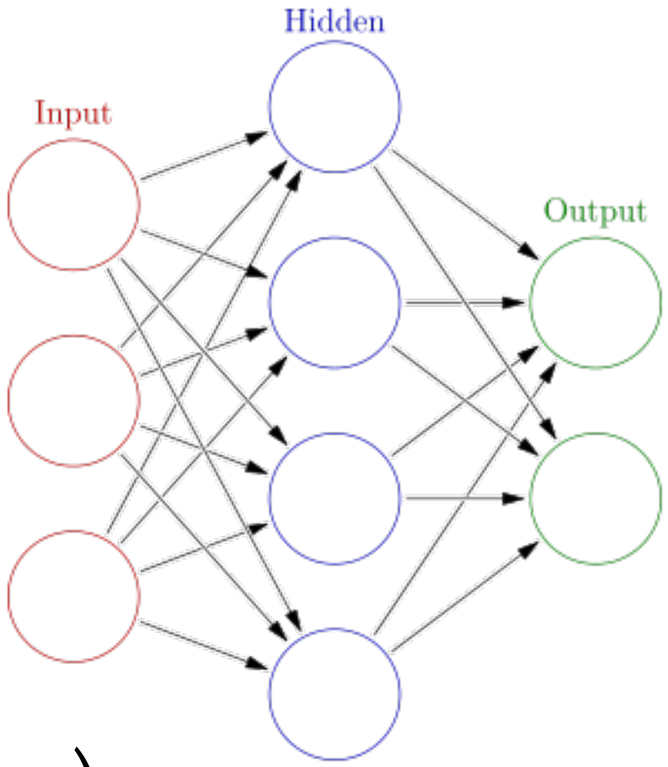
```
for i in range(100):
    print('Training set negative log-likelihood: %f' %
             training_function())
    print('Test set accuracy: %f' % test_function())
    print('')
```

Model performance (~8% error rate)



Adding a hidden layer

- Input units: $\mathbf{x}$
- Hidden units: $\mathbf{h}$
- Output units: $\mathbf{y}$



$$\mathbf{h}^{\mathrm{T}} = \tanh\left(\mathbf{x}^{\mathrm{T}}\mathbf{W}_{xh} + \mathbf{b}_{xh}\right)$$

$$\mathbf{y}^{\mathrm{T}} = \mathrm{SoftMax}\left(\mathbf{h}^{\mathrm{T}}\mathbf{W}_{hy} + \mathbf{b}_{hy}\right)$$

Adding a hidden layer: code changes

- We need two sets of weights and biases
  - Instead of creating W and b, create four variables: 2 Ws and 2 bs
  - Initializing the weights and biases between the input and hidden layer to zero will cause the gradient descent to be unable to move due to symmetry
    - Decide how many hidden units to use in the hidden layer
    - Initialize instead to small positive and negative random values

```
n_hidden_neurons = 20
W_xh = theano.shared(.01*numpy.random.randn(dims,n_hidden_neurons))
b_xh = theano.shared(numpy.zeros(n_classes))
W_hy = theano.shared(numpy.zeros([n_hidden_neurons,n_classes]))
b_hy = theano.shared(numpy.zeros(n_classes))
```

Adding a hidden layer: code changes

- Create a hidden layer
  - Declare a variable for it
  - Disconnect y_hat from X
  - Instead, connect X to h and h to y_hat
    - Use the new weights and biases
  - Use the theano function `tanh`

```
h = T.tanh(T.dot(X,W_xh) + b_xh)
y_hat = T.nnet.softmax(T.dot(h,W_hy) + b_hy)
```

Adding a hidden layer: code changes

- The gradient descent now needs to update all of the new parameters
  - Code is virtually unchanged
  - The learning rate has been increased (hack)

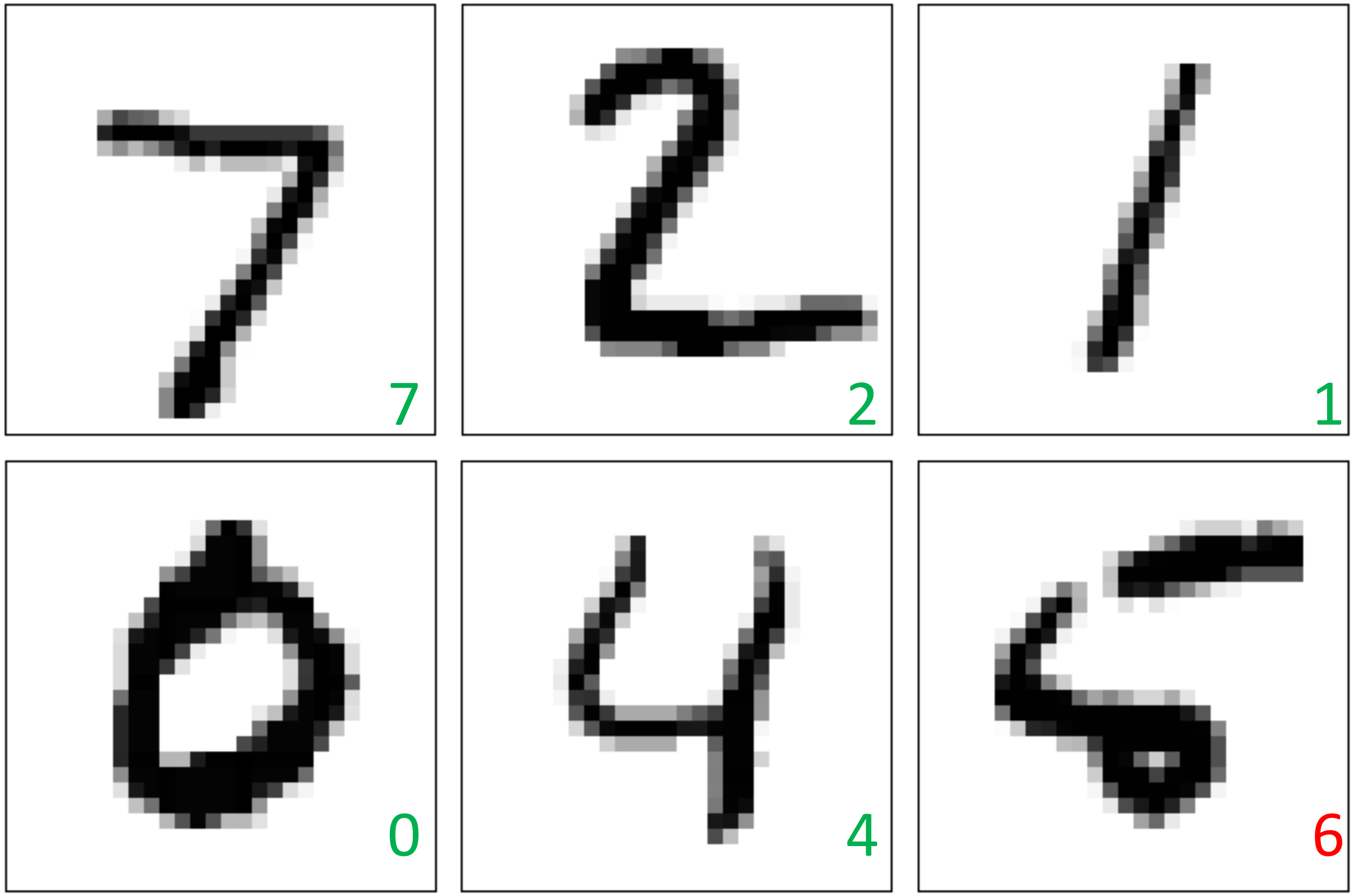
```
learning_rate = 2
updates = [
        [W_xh, W_xh - learning_rate * T.grad(training_error, W_xh)],
        [b_xh, b_xh - learning_rate * T.grad(training_error, b_xh)],
        [W_hy, W_hy - learning_rate * T.grad(training_error, W_hy)],
        [b_hy, b_hy - learning_rate * T.grad(training_error, b_hy)],
    ]
```

Because everything slows down…

- Neural network technology was rather useless until ~2009 due to lack of processing power
- And I have a tiny laptop for our demos…
- Let's only train and test using 5k data points

```
train_set = (train_set[0][:5000], train_set[1][:5000])
test_set = (test_set[0][:5000], test_set[1][:5000])
```

Model performance (~2% error rate)



Optimization tricks

- We can make huge gains in model performance and training speed by using smarter optimization algorithms
- Some questions to think about:
  - How can we make training epochs shorter?
  - How many epochs should we train for?
  - How should we initialize weights?
  - Is it a good idea to take gradient descent steps proportional to the gradient magnitude?
  - How do we choose the learning rate?
  - How can we avoid local optima?
  - How can we prevent overfitting?
  - How can we impose prior beliefs?
  - How do we choose the nonlinear activation functions?

How can we make training epochs shorter?

- Idea: use a small subset of the data (aka mini-batch) instead of the entire dataset when computing the gradient
  - Stochastic Gradient Descent (SGD or MSGD)
- Advantages:
  - Epochs take less time, so it's easier to tweak the training
  - If the subset is chosen randomly, then the optimization algorithm itself will be stochastic
  - This means it will have the ability to escape local minima
- Can be implemented in multiple ways
  - At each epoch choose a random subset
  - Deterministically split the data into equal chunks and cycle through it (outer-inner loops)
  - Same as above, but randomly permute the data in the outer loop
- Choosing mini-batch size
  - Between 20-1000 is a rough guideline, will need tweaking
  - Ideally have equal number of points from each class in each mini-batch

SGD: Implementation

- Modify the training function so that it takes as input a vector of dataset indexes
  - Add this vector variable to the graph
- Write a function that takes the data, applies a random permutation, and splits into mini-batches

```
idx = T.ivector()
training_function = theano.function(
    inputs = [idx],
    outputs = training_error,
    updates = updates,
    givens = {X:training_x[idx], y: training_y[idx]}
    )

getMiniBatches = (lambda n, colLen:
    numpy.reshape(numpy.random.permutation(n)[:n//colLen*colLen],
    [n//colLen, colLen]))
```

SGD: Implementation

- Rewrite the training loop to use mini-batches
  - Can switch back to gradient descent by setting the mini-batch size to the size of the dataset

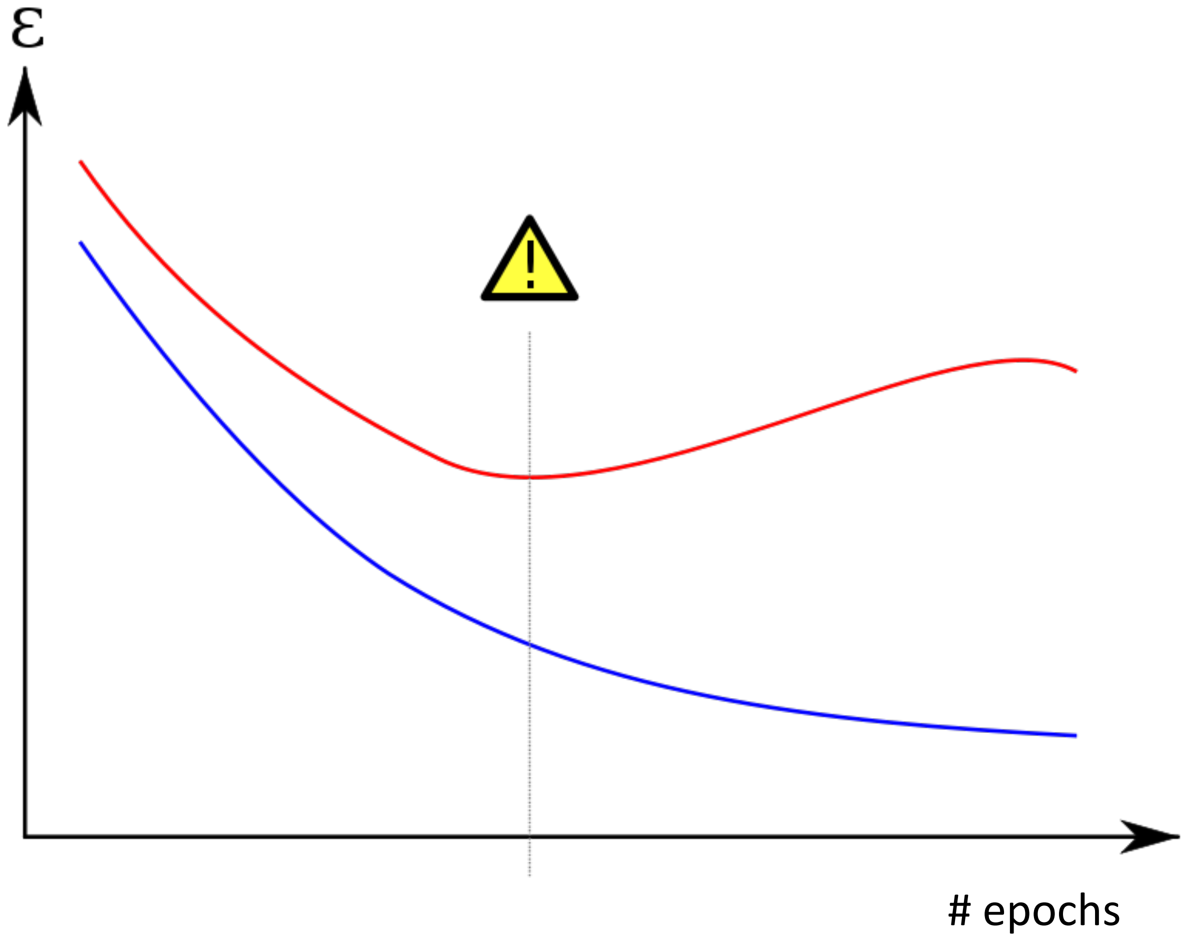
```
minibatchSize = 600
for dataset_cycles in range(50):
    for minibatch_idx in getMiniBatches(n_train, minibatchSize):
        training_function(minibatch_idx)
    print('Test set accuracy: %f' % test_function())
```

How many epochs should we train for?

- Idea #1: Train until our predictions stop getting better
  - When using SGD, often the error rate goes up from epoch to epoch
  - Wait until we haven't made an improvement **in a while**

- Idea #2: Train until our predictions stop getting better **on a validation dataset**
  - Prevents overfitting
  - Split dataset into training and validation

Overfitting: training vs validation error



Early Stopping

- Algorithm #1:
  - After training on the entire training set, compute the performance on the validation set
  - If a number of epochs has passed without improvement on the validation set, exit the loop
    - Return the parameters that performed best on the validation set
    - Optionally force a minimum number of training epochs
- Algorithm #2:
  - Declare a variable threshold on #data points to train on
  - When the validation performance improvement exceeds a threshold, update the threshold to 2 x #data points seen
    - But never decrease it
  - Stop when the number of data points exceeds the variable, and return the parameters that performed best on the validation set

Early Stopping: Implementation

- Get the validation data and massage it
  - We are already loading into memory the validation set
- Write a theano function for the validation error
  - Can use either classification error or log-likelihood

```
(training_x, training_y), (test_x, test_y), (validation_x, validation_y)
    = map(prepare_data, [train_set, test_set, valid_set])

validation_function = theano.function(
    inputs = [],
    outputs = test_error,
    givens = {X: validation_x, y: validation_y}
    )
```

Early Stopping: Implementation

```
nEpoch = 0
minEpochs = 50
patience = 0
maxPatience = 20

bestScore = numpy.inf
params = [W, b]
```

- Initialize variables for the early stopping loop
- Declare a list of all model parameters
  - Use to keep track of best parameters

```
while nEpoch < minEpochs or patience < maxPatience:
    for minibatch_idx in getMiniBatches(n_train, minibatchSize):
        training_function(minibatch_idx)
    validScore = validation_function()
    if validScore < bestScore:
        bestScore = validScore
        patience = 0
        bestParams = {i: i.get_value() for i in params}
    else: patience += 1
    nEpoch += 1
```

- `while` instead of `for` loop
- if validation score improves save parameters
- otherwise become more impatient

Early Stopping: Implementation

- Retrieve the best parameters after stopping

```
for var, val in bestParams.iteritems():
    var.set_value(val)
```

How should we initialize weights?

- For the weights going into the final layer, it's OK to initialize to 0
- For all other weights, initialize to small random numbers to break symmetry
  - Choose random numbers following Y. Bengio, X. Glorot, Understanding the difficulty of training deep feedforward neuralnetworks, AISTATS 2010
  - Sample from Uniform(-α, α), where α is:

$\sqrt{\frac{6}{n_{prev} + n_{cur}}}$ for tanh layer $4\sqrt{\frac{6}{n_{prev} + n_{cur}}}$ for sigmoid layers

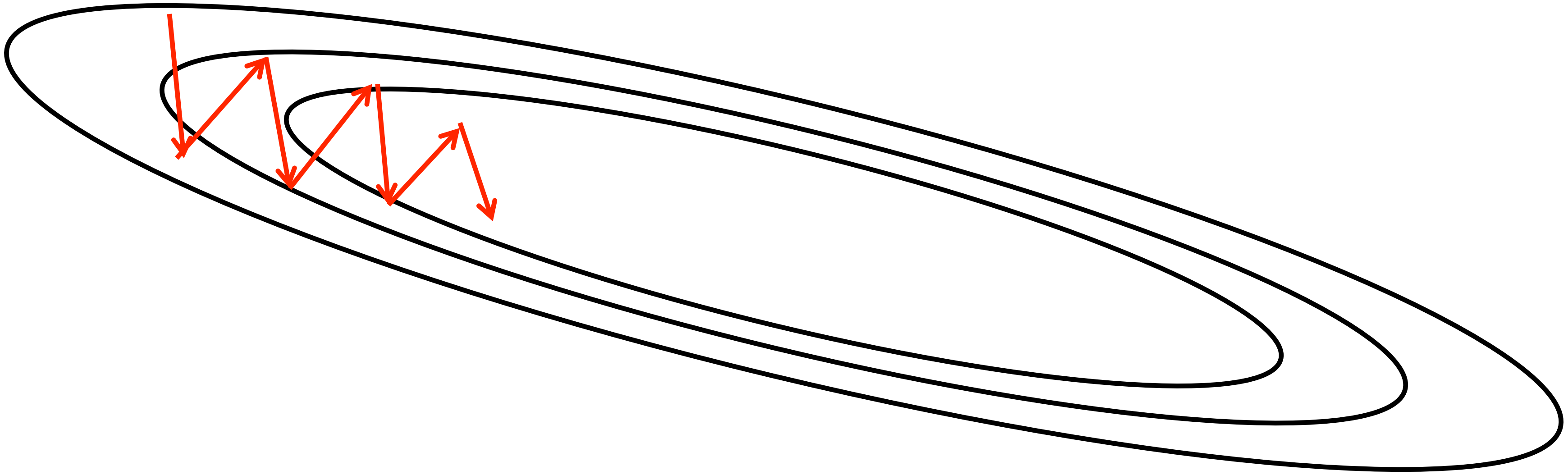
where n_{prev} and n_{cur} are the number of units in the previous and current layers respectively

```
alpha = numpy.sqrt(6.0 / (dims + n_hidden_neurons))
W_xh = theano.shared(numpy.random.uniform(-alpha,
           alpha, (dims,n_hidden_neurons)))
```

- Finally, it's OK to initialize biases to zero

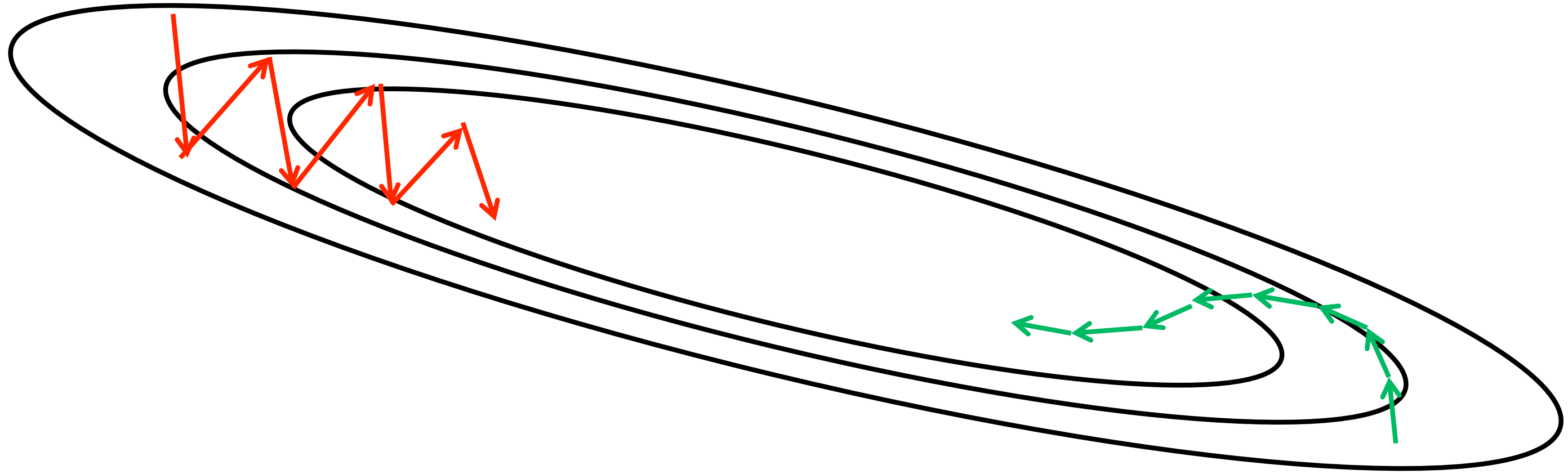
Is gradient descent a good idea?

- Is it a good idea to take gradient descent steps proportional to the gradient magnitude?
- When dealing with a linear neuron and squared error, the error surface is a quadratic bowl
- If this bowl is elongated, the gradient will be large in the direction of the small axes, and small in the direction of the large axes



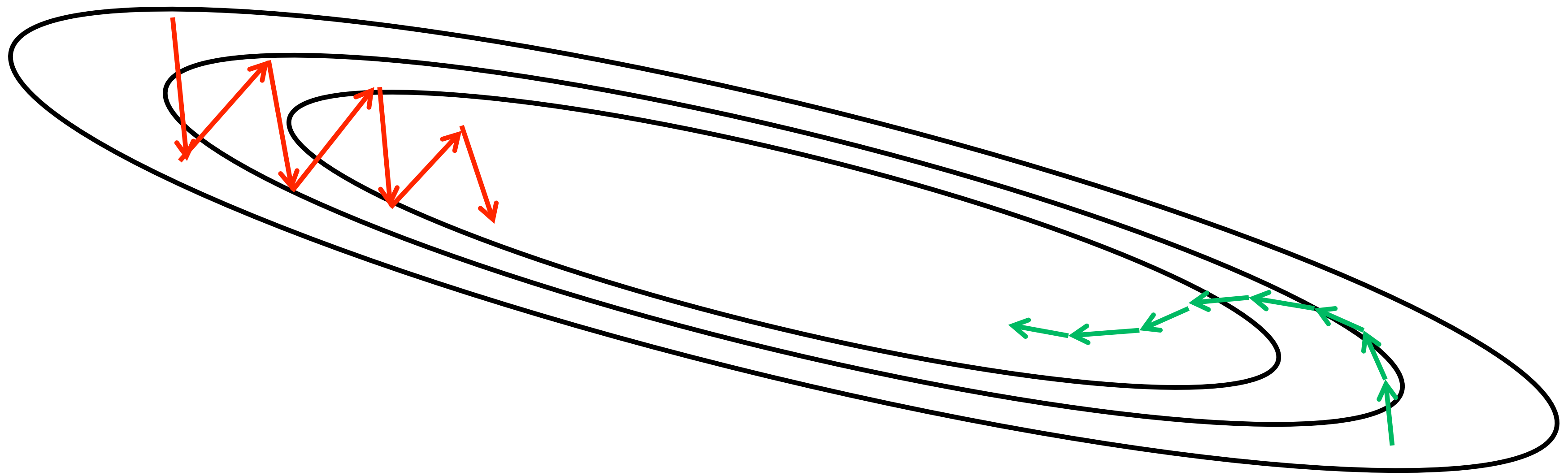
Optimizing by letting a ball roll down

- Imagine that instead we let a ball roll down
  - It would have a much more direct path towards the minimum
- Idea: simulate the physics of the ball, where the gradient gives the acceleration of the ball rather than the velocity



Momentum method

- This method is equivalent to smoothing the gradients with an exponentially-decaying filter
- For each model parameter, create an additional parameter that keeps track of the value of this filter



Momentum method: implementation

- Declare momentum variables
- Change the training function updates so that they now use momentum
  - The momentum variable itself needs updating
  - A new momentum parameter is introduced

```
beta = .8
params = [W, b]
updates = []
for p in params:
    m = theano.shared(0.*p.get_value())
    updates += [(p, p - learning_rate * m),
                (m, beta * m +
            (1 - beta) * T.grad(training_error, p))]
```

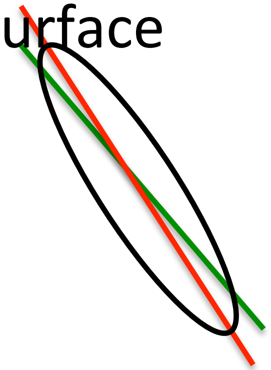
Other ways to deal with elongated bowls

- Having data that is not centered around zero or having features of different scales creates elongated bowls

Error surface

Color indicates data points

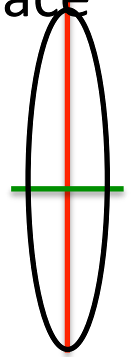
101, 101 -> 2
101, 99 -> 0



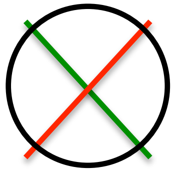
Error surface

Color indicates features

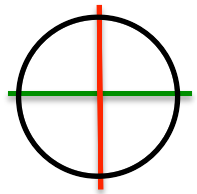
0.1, 10 → 2
0.1, -10 → 0



1, 1 -> 2
1, -1 -> 0



1, 1 → 2
1, -1 → 0



Whitening using PCA: Implementation

- Subtracts the mean, scales the dimensions, and rotates the data so that they are uncorrelated
- First, compute whitening parameters off the training data

```
def getWhiteningParameters(x):
    xMean = numpy.mean(x,axis=0)
    xBar = x-xMean
    sigma = numpy.dot(xBar.T, xBar) / x.shape[0]
    U,S,V = numpy.linalg.svd(sigma)
    epsilon = 1e-5
    nDims = sum(S > 1e-5)
    U = U[:,:nDims]
    S = S[:nDims]
    return xMean, U, S, epsilon
```

- Compute the data mean
- Compute the features scales and correlation
- Numerical correction
- Dimensionality reduction

Whitening using PCA: Implementation

- Implement a whitening function
- Whiten the datasets before creating variables

```
def whitenData(x, xMean, U, S, epsilon):
    xBar = x-xMean
    xRot = numpy.dot(xBar, U)
    xPCAwhite = 1./numpy.sqrt(S + epsilon) * xRot
    return xPCAwhite
```

- Remove the mean
- Remove correlations
- Rescale

```
dims = whiteningParams[1].shape[1]
prepare_data = lambda x: (theano.shared(whitenData(x[0],
    *whiteningParams).astype('float64')),
    theano.shared(x[1].astype('int32')))
```

- Get reduced dimensionality
- Add whitening to data preparation function

How should we set the learning rate?

- Observation #1: When we use high learning rates, the model performance jumps up and down a lot
- Observation #2: When we use low learning rates, the model performance decreases smoothly but slowly
- Observation #3: The performance jumps a lot towards the beginning of the training, but later tends to progress very slowly

Adaptive learning rates

- Idea: Instead of using a learning rate that is fixed throughout the entire training process, adjust it in response to changes in performance
- This can be done in multiple ways:
  - Use a single learning rate
  - <u>Have a different learning rate for each parameter</u>
    - a lot of research has been done on what works well, read the latest papers and tutorials from Hinton, LeCun, Bengio and their students
  - Use a 3rd party package
    - theano_lstm package has methods SGD, AdaGrad, AdaDelta

Adaptive learning rates: Algorithm details

- Like with momentum, introduce a new variable for the individual learning rates
- Get the gradient from the training function, and also keep track of the previous gradient
- If the gradients' signs are the same, increase the local weight additively, otherwise decrease multiplicatively
  - But do not allow the weight to escape a certain range (not implemented)

```
params = [W, b]
updates = []
ada_rates = {}
for p in params:
    ada_rates[p] = theano.shared(1.+0.*p.get_value())
    updates += [(p, p - learning_rate * ada_rates[p] *
T.grad(training_error, p))]
```

Adaptive learning rates: Implementation

- Update the training theano function to return all gradients
- In the inner training loop update the adaptive learning rate

```
training_function = theano.function(
    inputs = [idx],
    outputs = [training_error]+[T.grad(training_error, p) for p in params],
    updates = updates,
    givens = {X:training_x[idx], y: training_y[idx]}
    )
...
lastGradients = None; curGradients = None
...
 lastGradients = curGradients
        trFunOut = training_function(minibatch_idx)
        curGradients = dict(zip(params,trFunOut[1:]))
        if lastGradients is not None:
            for p in params:
                g = lastGradients[p] * curGradients[p]
                ar = numpy.copy(ada_rates[p].get_value())
                ar[g>0] += .05; ar[g<0] *= .95
                ada_rates[p].set_value(ar)
```

Other alternatives to gradient descent

- Only use the sign of the gradient, not its magnitude
  - When used with mini-batches it can cause bad behavior: 10 small positive gradients will overpower one very large negative gradient
- Using the sign of the gradient is equivalent to gradient descent where we divide the step by the magnitude of the gradient
  - Solution: Force the number we divide by to change slowly
- rmsprop algorithm: Compute a moving average of the squared gradient of each weight
- When taking a step, divide the gradient by the square root of the moving average
- Used in DeepMind's videogame-playing research, from an unpublished paper by Tijmen Tieleman

rmsprop: Implementation

- Similar to momentum: declare mean-square variables
- Change the training function updates so that they now use the mean-square variable
  - The mean-square variable itself needs updating

```
beta = .9
params = [W, b]
updates = []
for p in params:
    ms = theano.shared(1.+0.*p.get_value())
    updates += [(p, p-learning_rate*T.grad(training_error, p)/T.sqrt(ms)),
       (ms, beta * ms + (1 - beta) * T.sqr(T.grad(training_error, p)))]
```

Other alternatives to gradient descent

- Also check out:
  - Conjugate gradient
  - Nesterov momentum
  - Levenberg–Marquardt
  - L-BFGS
  - Hessian-free optimization

How can we prevent overfitting?

- Regularization
  - Sometimes models overfit by assigning strong weights to very rare features in a dataset
    - E.g. only one training case has a certain pixel on
  - One approach is to penalize large weights by adding to the cost function the L1-norm or L2-norm of the weights
- Dropout
  - Increase generalization power by forcing the network to perform well even when some of the units aren't used
  - During training, multiply the activations of each hidden unit by a random binary mask drawn from Bernoulli(.5)

Regularization: Implementation

- Create variables for the L1 and L2 terms
- Add these times constants to the training cost

```
L1 = 0; L2 = 0
for p in params:
    L1 += T.mean(abs(p))
    L2 += T.mean(p ** 2)

training_error = ((-T.mean(T.log(y_hat)[T.arange(y.shape[0]), y]))
    + .01 * L1 + .01 * L2)
```

Dropout: Implementation

- Create variables for the weights' binary masks
  - Use it to compute the final layer
- When training, choose a random mask
- When testing, use a uniform mask with the probability of the random mask

```
... # when constructing the network
drop_factor = .75
Wmask = theano.shared(numpy.zeros([dims,n_classes]))
y_hat = T.nnet.softmax(T.dot(X,W*Wmask) + b)
... # before calling the training function
 Wmask.set_value(numpy.random.binomial(1, drop_factor, W.get_value().shape))
... # before calling the validation and test functions
 Wmask.set_value(drop_factor * numpy.ones((W.get_value().shape)))
```

Dropout: Final thoughts

- It works really well: Alex Krizhevsky's object recognition made a dramatic improvement by using dropout
- Dropout can be seen as training multiple models with tied weights, and predicting via model averaging (geometric mean)
- Multiplying the weights by the mask probability at runtime is an approximation
  - Exact method would be to get samples using the random binary masks
- Dropout works for the input layer as well (but needs higher probability)
- A network trained with dropout can be much larger and not overfit
- With dropout, units can't count on other units being present, preventing complex co-adaptations that are only relevant on the training data
  - A unit that works well with different sets of co-workers is more likely to do something useful

How can we impose prior beliefs?

- Data enrichment is one widely used method to impose prior beliefs on the network
- An image that is distorted via translation or scaling usually represents the same thing
- The network is not aware of this fact
- We can tell it that by creating new labeled data points based on the ones in our dataset

How do we choose the nonlinear activation functions?

- Best approach is trial and error
  - sigmoid has a probabilistic interpretation
  - tanh is more likely to yield data centered around zero

Common neural network architectures

- Convolutional Neural Networks (CNNs)
- Restricted Boltzmann Machines (RBMs)
- Recurrent Neural Networks (RNNs)

Convolutional Networks

- Used mainly in computer vision
- Idea: lower layers of the network correspond to low-level image features
- Low-level features such as edges are local and look the same at every point in the picture
  - Use tied weights within the same layer
  - Reduces the number of parameters, and pools knowledge about features from all parts of the image
  - Equivalent to running a convolution operation on the image instead of the usual dot product

Convolutional Networks

- MaxPooling layer:
  - If a certain feature has been found in the image, it doesn't matter (within a displacement of a few pixels) where exactly it is
  - MaxPooling downsamples the image by taking the max activation of a certain filter over say a small 2x2 patch of filters
- ReLU layer:
  - Rectified Linear Units use the activation function $\max(0, x)$
  - Can lead to faster training than networks employing sigmoid or tanh

Convolution network: Layers

- Input layer
- One or more stacks of
  - Locally-connected convolutional layer
    - Looks only directly “below” and neighbors a certain distance
  - Locally-connected MaxPooling layer
    - For each filter, looks “below” and at neighbors
  - (optional) Locally-connected ReLU/sigmoid/tanh layer
- Fully connected layers
- Output: Each unit represents a class or object

Convolutional networks: Theano

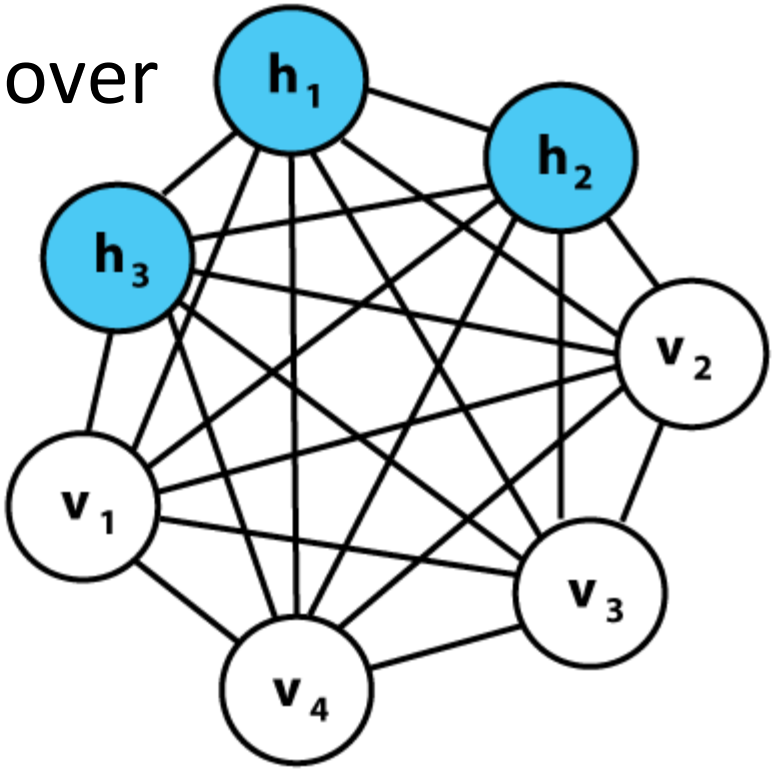
- Theano provides functions for image convolution, MaxPooling, and ReLU activation:
  - `theano.tensor.nnet.conv.conv2d`
  - `theano.tensor.signal.downsample.max_pool_2d`
  - `theano.tensor.nnet.relu`
- Training:
  - Same way as our other networks

Boltzmann Machines

- Unsupervised learning model
- Has a number of visible units (which model the data) and hidden units
- Defines a joint probability over all random variables:

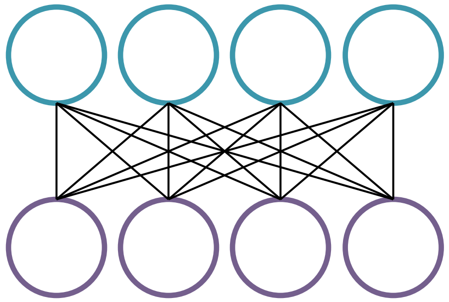
$$P(X) \propto e^{-\mathbf{x}^{\mathrm{T}}\mathbf{W}\mathbf{x}-\mathbf{x}^{\mathrm{T}}\mathbf{b}}$$

$\mathbf{x}$ includes both hidden and visible units



Restricted Boltzmann Machines

- Learning in Boltzmann Machines is intractable
  - Can be done via Markov Chain Monte Carlo (MCMC) but this involves computing $\langle x_i, x_j \rangle_{\text{data}}$, which is itself really hard
  - Learning in Boltzmann Machines does not work well
- Solution: Restricted Boltzmann Machines
  - Bipartite graph, each connection is between one hidden and one visible unit
  - $\langle v_i, h_j \rangle_{\text{data}}$ is easy to compute
  - Learning works well



$$P(V,H) \propto e^{-\mathbf{v}^{\mathbf{T}}\mathbf{W}\mathbf{h}-\mathbf{v}^{\mathbf{T}}\mathbf{b}_v-\mathbf{h}^{\mathbf{T}}\mathbf{b}_h}$$

Learning in RBMs: Contrastive Divergence

- Gradient descent in an RBM

$$\frac{\partial \log P(V)}{\partial w_{ij}} = \left\langle v_i h_j \right\rangle_{\text{data}} - \left\langle v_i h_j \right\rangle_{\text{model}}$$

- Intuition
  - First term makes the data more likely
    - Similar to neuroplasticity-inspired Hebbian learning
  - Second term makes model output less likely
    - Without this, learning that some data points are good could generalize to learning that **all** outputs are good

Learning in RBMs: Contrastive Divergence

- Gradient descent in an RBM

$$\frac{\partial \log P(V)}{\partial w_{ij}} = \left\langle v_i h_j \right\rangle_{\text{data}} - \left\langle v_i h_j \right\rangle_{\text{model}}$$

- Computation
  - First term can be computed in closed form
  - Second term is intractable
    - Compute by MCMC: start from an observation and repeat:
      - Sample hidden units given visible
      - Sample visible units given hidden
    - For convergence, repeat infinitely many times
    - In practice, algorithm works well with one MCMC step
      - As training approaches the end, increase number of steps to 2-5

Learning in RBMs: Contrastive Divergence

- Alternative for second term computation: Persistent Contrastive Divergence (PCD)
  - Instead of sampling one or more MCMC steps starting from data points, keep track of a number of particles (states of the entire model)
    - Particularly effective when multiple MCMC steps would need to be taken towards end of training
  - Take one MCMC step with each of these particles from wherever their last state was
    - Intuition: Parameters change slowly relative to MCMC mixing, so this is OK
  - Just a few particles (~100) may be enough to train well
    - Intuition: Particles would constantly be moving to states that are dissimilar from the data but likely under the model
    - Once a state is unlikely, the particle would move

Deep Belief Nets (DBN): Stacked RBMs

- Deep belief nets are created from stacked RBMs
- They can be used for supervised tasks
  - Can be trained with gradient descent
- Use Contrastive Divergence for pre-training
  - Intuition: Learn the structure of the data before learning to label it
- I feel DBNs have fallen out of fashion a bit
  - Most of the latest research is in ways of tweaking stochastic gradient descent

Recurrent Neural Networks

- Model time-series data (audio, video, finance)
- The hidden layer(s) at each time step receives input from layer(s) corresponding to past time steps
- Training algorithm: Backpropagation through time (BPTT)
  - No new idea here: apply calculus chain rule
- Long Short-Term Memory (LSTM) units:
  - RNNs' memory doesn't go far into the past
  - LSTM units have read/keep/write gates that control the information flow
  - The activation can be stored for many time steps
  - Designed so that its parameters can be learned by backpropagation

Questions?

Thank you

gabiteodoru@gmail.com