# Theano and Machine Learning

Martin Meier

June 14, 2016

# Outline

Theano

Single Neuron

Multilayer Perceptron

Denoising Autoencoder

# What is Theano?

- General linear algebra compiler
- Not only for machine learning
  - But that is our focus today
- Python based framework
- Good numpy integration

# What is Theano?

- Symbolic computation
  - Define variables and functions
  - obtain e.g. gradients without explicit definition
- Compile symbolic expressions to C or CUDA
- Optimizes functions before compilation

# Simple datatypes

- Data types:
  - scalar $x = theano.tensor.scalar()$
  - vector $x = theano.tensor.vector()$
  - matrix $x = theano.tensor.matrix()$
  - tensor $x = theano.tensor.tensor()$
- functions $y = x^2$
- Internally organized as graphs

# Installation (if you use linux)

```
mkdir theano
virtualenv `pwd`
pip install theano
source bin/active
```

# Scalar math and functions

```
import theano
x = theano.tensor.scalar()
y = x**2
# y
# Elemwise{pow,no_inplace}.0
f = theano.function(inputs=[x], outputs=y)
# f
# <theano.compile.function_module.Function at 0x7f449f7b7e
f(2)
# 4.0
```

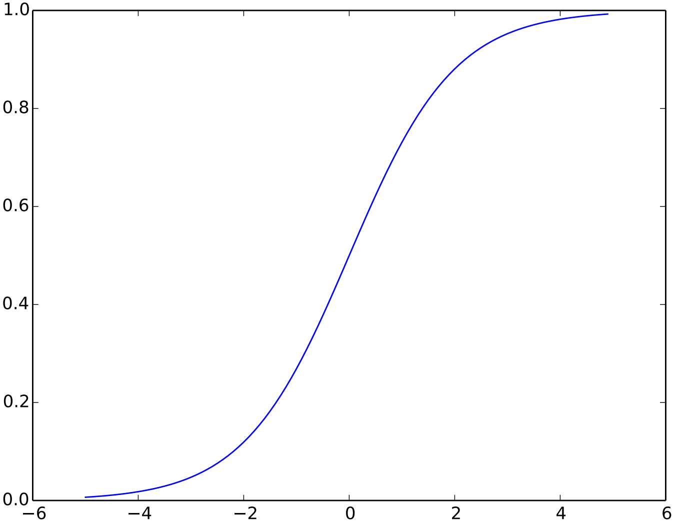
# What is it good for?

```
def f(x):
  return x**2

print f(2)
# 4
```

# Logistic function

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

x = T.scalar()
y = T.sum(1/(1+T.exp(-x)))
S = theano.function([x],y)

print S(2)
# 0.880797088146
```

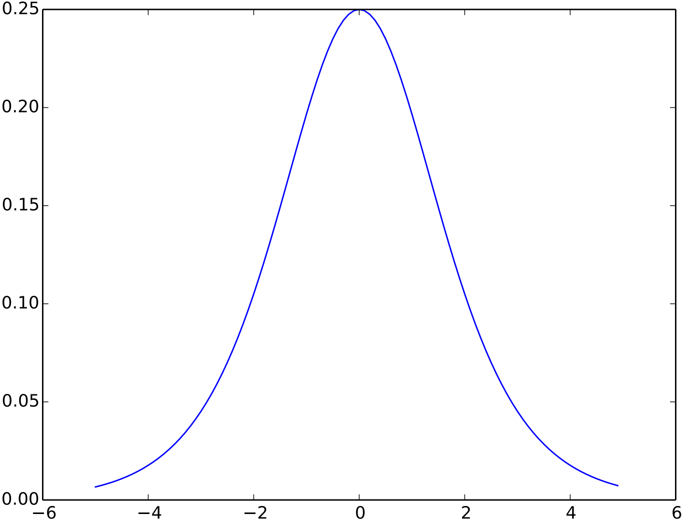
# Logistic function



# Computing Gradients

```
import theano
import theano.tensor as T

x = T.scalar()
y = 1/(1+T.exp(-x))
S = theano.function([x],y)
g = T.grad(y,x)
gS = theano.function([x],g)


print  S(2)
# 0.880797088146
print gS(2)
# 0.104993589222
```

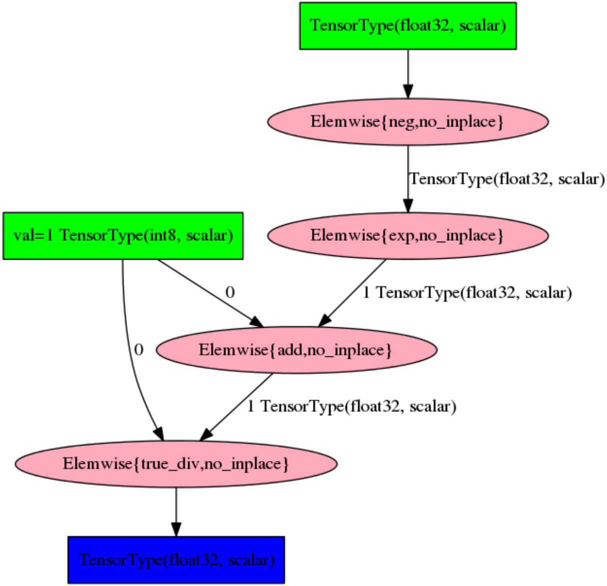
# Computed Gradient



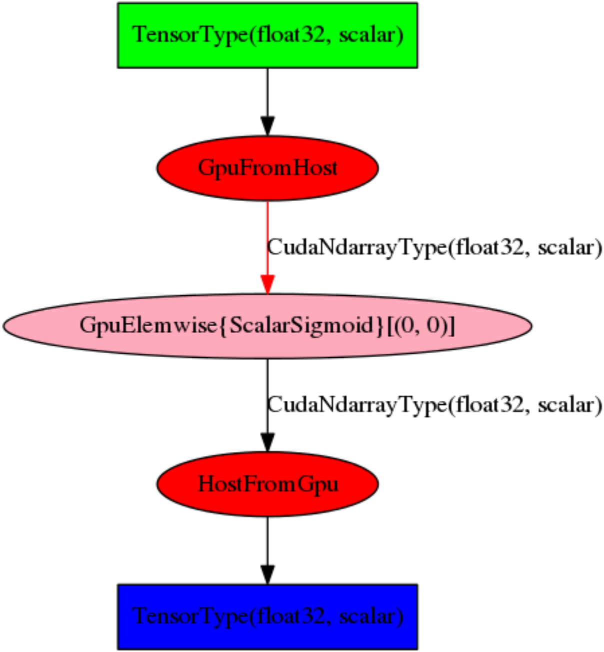
# Internal representations

```
import theano
import theano.tensor as T

x = T.scalar()
y = 1/(1+T.exp(-x))
S = theano.function([x],y)
g = T.grad(y,x)
gS = theano.function([x],g)

theano.printing.pydotprint(y,
  outfile="/tmp/y.png",
        var_with_name_simple=True)

theano.printing.pydotprint(S,
  outfile="/tmp/opty.png",
        var_with_name_simple=True)
```

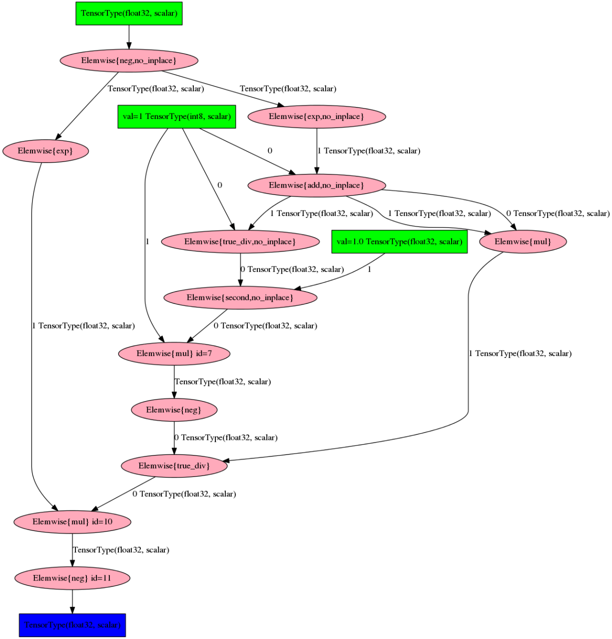
# Graph of y


TensorType(float32, scalar)
Elemwise{neg,no_inplace}
TensorType(float32, scalar)
val=1 TensorType(int8, scalar)
Elemwise{exp,no_inplace}
0
1 TensorType(float32, scalar)
0
Elemwise{add,no_inplace}
1 TensorType(float32, scalar)
Elemwise{true_div,no_inplace}
TensorType(float32, scalar)

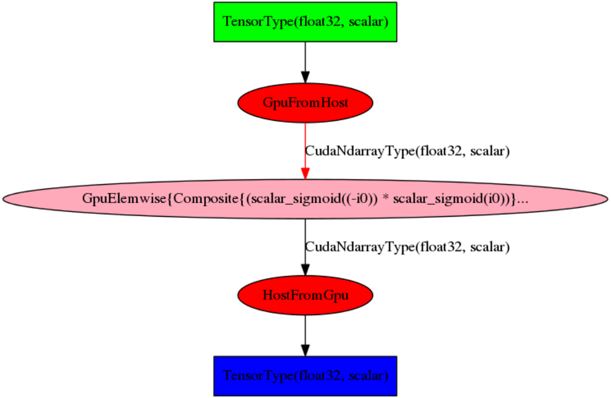
# Graph of S = theano.function(x,y)



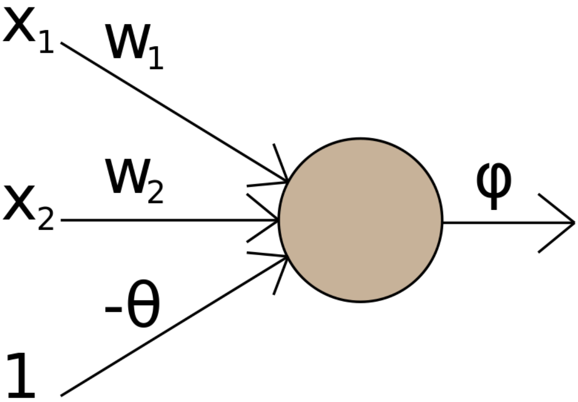
# Graph of g


TensorType(float32, scalar)
Elemwise{neg,no_inplace}
TensorType(float32, scalar)
TensorType(float32, scalar)
val=1 TensorType(int8, scalar)
Elemwise{exp,no_inplace}
Elemwise{exp}
0
1 TensorType(float32, scalar)
0
Elemwise{add,no_inplace}
1 TensorType(float32, scalar)
1 TensorType(float32, scalar)
0 TensorType(float32, scalar)
1
Elemwise{true_div,no_inplace}
val=1.0 TensorType(float32, scalar)
Elemwise{mul}
0 TensorType(float32, scalar)
1
Elemwise{second,no_inplace}
1 TensorType(float32, scalar)
0 TensorType(float32, scalar)
Elemwise{mul} id=7
1 TensorType(float32, scalar)
TensorType(float32, scalar)
Elemwise{neg}
0 TensorType(float32, scalar)
Elemwise{true_div}
0 TensorType(float32, scalar)
Elemwise{mul} id=10
TensorType(float32, scalar)
Elemwise{neg} id=11
TensorType(float32, scalar)

# Graph of gS = theano.function(x,g)



# Implementing a single neuron




$$y_k = \varphi\left(\sum_{j=0}^{m} w_{kj} x_j\right) + b$$

## Single neuron without update

```
import theano
import theano.tensor as T
import random

x = T.vector()          # input
w = T.vector()          # weights
b = T.scalar()          # bias
z = T.dot(w,x) + b      # summation
y = 1/(1+T.exp(-z))     # activation

neuron = theano.function(inputs=[x,w,b],outputs=[y])
w = [-1,1]
b = 0

for i in range (100):
        x = [random.random(), random.random()]
        print x
        print neuron(x,w,b)
```

# Shared Variables

- Only **x** should be an input
- **w** and **b** are model parameters
- In theano, these are represented as shared variables

# Single neuron with shared variables

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

x = T.vector()
w = theano.shared(np.array([1.,1.]))
b = theano.shared(0.)
z = T.dot(w,x) + b
y = 1/(1+T.exp(-z))

neuron = theano.function(inputs=[x],outputs=[y])
print w.get_value()
w.set_value([-1,1]) # set theano.shared
```

# Single neuron - Training

- to train the neuron, we need to adapt the model parameters
- Requires a cost function

# Adding a cost function

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

x = T.vector()
w = theano.shared(np.array([-1.,1.]))
b = theano.shared(0.)
z = T.dot(w,x) + b
y = 1/(1+T.exp(-z))

neuron = theano.function(inputs=[x],outputs=[y])

y_hat = T.scalar() # desired output
cost = T.sum((y-y_hat)**2)
dw,db = T.grad(cost, [w,b])

gradient = theano.function([x,y_hat], [dw,db])
```

# Updating parameters

```
# [snip]
y_hat = T.scalar() # desired output
cost = T.sum((y-y_hat)**2)
dw,db = T.grad(cost, [w,b])

gradient = theano.function([x,y_hat], [dw,db])

x = [1,-1]
y_hat = 1
lr = 0.01 # learning rate
for i in range(1000):
        dw,db = gradient(x, y_hat)
        w.set_value(w.get_value() - lr * dw)
        b.set_value(b.get_value() - lr * bw)
```

# Updating parameters - the easy way

```
# [snip]
y_hat = T.scalar() # desired output
cost = T.sum((y-y_hat)**2)
dw,db = T.grad(cost, [w,b])

x = [1,-1]
y_hat = 1
lr = 0.01 # learning rate

#easier
gradient = theano.function([x,y_hat], [dw,db],
updates=[(w,w-lr*dw), (b,b-lr*db)] )

for i in range(1000):
        dw,db = gradient(x, y_hat)
```

# Putting neurons together - Multilayer Perceptron

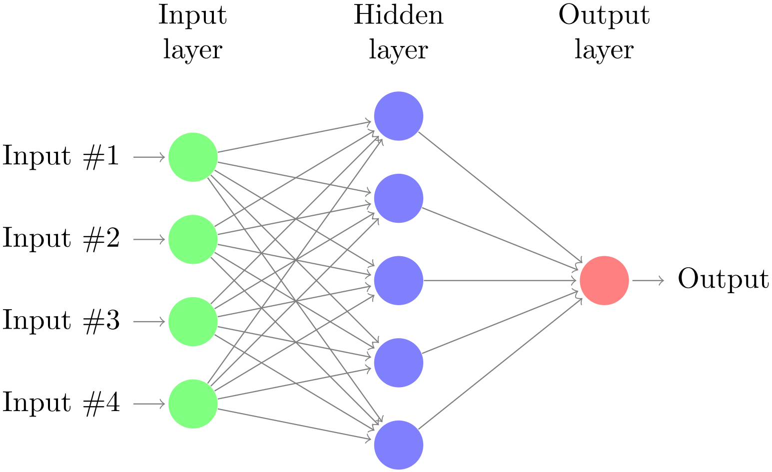



image from www.texample.net/tikz/examples/neural-network/

# MLP - Ingredients

- Layers
  - Input
  - Hidden
  - Output
  - Connected by weights
  - Weights have to be initialized and updated
- Cost function
- Forward pass
- Backpropagation

# Input to hidden layer

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

def init_hidden_weights(n_in, n_hidden):
  rng = numpy.random.RandomState(1111)
  weights = numpy.asarray( # Xavier initialization
    rng.uniform(
      low=-numpy.sqrt(6. / (n_in + n_hidden)),
      high=numpy.sqrt(6. / (n_in + n_hidden)),
      size=(n_in, n_hidden)
    )
  bias = numpy.zeros(n_hidden,)
  return (
    theano.shared(value=weights, name='W', borrow=True),
    theano.shared(value=bias, name='b', borrow=True)
  )
```

# Hidden to output layer

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

def init_output_weights(n_hidden, n_out):
  weights = numpy.zeros(n_hidden, n_out)
  bias = numpy.zeros(n_out,)
  return (
    theano.shared(value=weights, name='W', borrow=True),
    theano.shared(value=bias, name='b', borrow=True)
  )
```

# Connecting layers

```
n_in = 50
n_hidden = 30
n_out = 10

h_w, h_b = init_hidden_weights(n_in, n_hidden)
o_w, o_b = init_output_weights(n_hidden, n_out)
```

# Cost function and regularization

- Needed to adapt model parameters $w$ and $b$
- Do forward pass and acquire error
- Square error cost function
- With regularization
  - L1/L2 regularization
  - used to prevent overfitting

# Forward pass and regularization

```
def L1(L1_reg, w1, w2):
  return L1_reg * (abs(w1).sum() + abs(w2).sum())

def L2(L2_reg, w1, w2):
  return L2_reg * ((w1 ** 2).sum() + (w2 ** 2).sum())

def feed_forward(activation, weights, bias, input_):
  return activation(T.dot(input_, weights) + bias)
```

# Cost function and gradient decent

```
def feed_forward(activation, weights, bias, input_):
  return activation(T.dot(input_, weights) + bias)

# how good is our current model
# theano also provides some convenient nn functions
p_y_x = feed_forward(T.nnet.softmax, o_w, o_b,
          feed_forward(T.tanh, h_w, h_b, x))

cost = (
  -T.log(p_y_x[0, y]) # -log likelihood of desired label
  + L1(L1_reg, o_w, h_w) + L2(L2_reg, o_w, h_w)
)

# theano calculates the gradient, param are the weights
def gradient_step(param, cost, lr):
  return param - (lr * T.grad(cost, param))
```

# Training and evaluation

```
train = theano.function(inputs=[x, y],
  outputs=cost, # output depends on cost
  updates=[
    (o_w, gradient_step(o_w, cost, lr)),
    (o_b, gradient_step(o_b, cost, lr)),
    (h_w, gradient_step(h_w, cost, lr)),
    (h_b, gradient_step(h-b, cost, lr)),
  ])

evaluate = theano.function(inputs=[x, y],
  outputs=T.neq(y, T.argmax(p_y_x[0])),
)
```

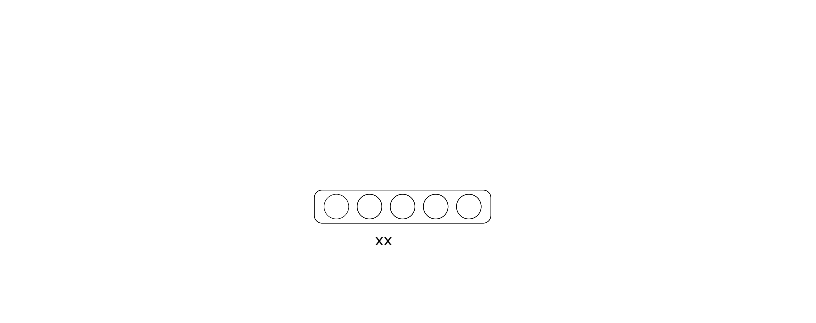
# Putting everything together

```
lr = 0.01
L1_reg = 0.0001
L2_reg = 0.0001
n_hidden = 100

for epoch in range(1, 1000):
  for x, y in examples:
    train(x, y)
    error = np.mean(
      [evaluate(x, y) for x, y in examples]
    )
    print('epoch %i, error %f %%' % (epoch, error * 100))
```

# Denoising Autoencoder

- Learning of good features is important for deep architectures
- For example convolutional layers
- Deal with noisy inputs (missing/wrong inputs)

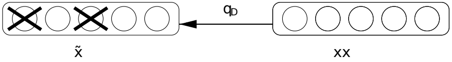
# Denoising Autoencoder




- We have an input $x$

images from *Extracting and composing robust features with denoising autoencoders, P. Vincent et. al. ICML 2008*

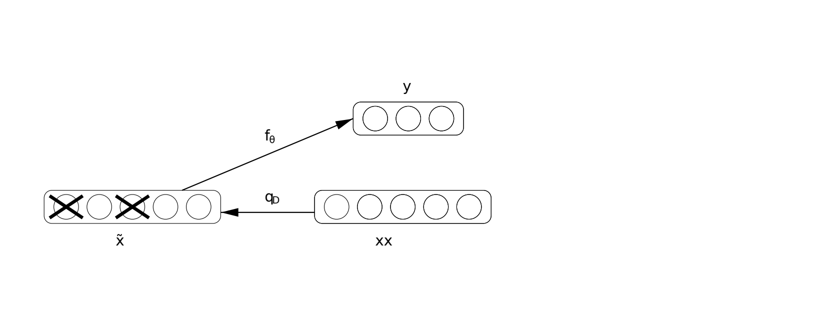
# Denoising Autoencoder




- Input x is corrupted: $\tilde{x} \approx q_D(\tilde{x}|x)$

images from *Extracting and composing robust features with denoising autoencoders, P. Vincent et. al. ICML 2008*

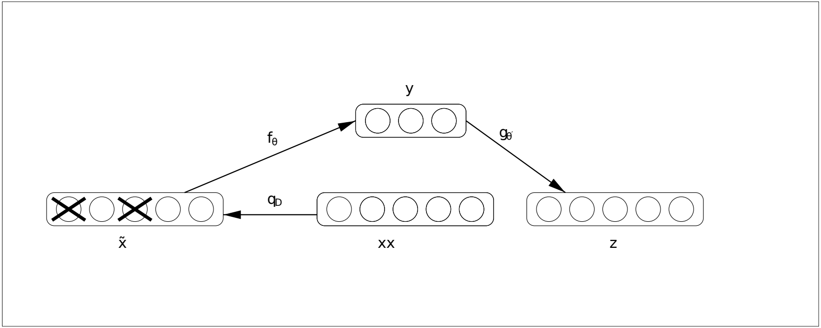
# Denoising Autoencoder




- $\tilde{x}$ is mapped to hidden representation $y = f_\theta(x)$

images from *Extracting and composing robust features with denoising autoencoders, P. Vincent et. al. ICML 2008*

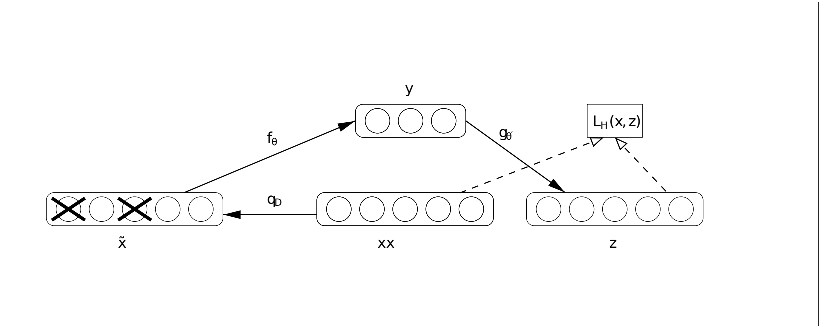
# Denoising Autoencoder




- $y$ is used to reconstruct $z = g_{\theta'}(y)$

images from *Extracting and composing robust features with denoising autoencoders, P. Vincent et. al. ICML 2008*

# Denoising Autoencoder




- Minimize reconstruction error $L_H(x, z)$

images from *Extracting and composing robust features with denoising autoencoders, P. Vincent et. al. ICML 2008*

# More formally

randomly remove data with mapping $q_D$

$$\tilde{x} \approx q_D(\tilde{x}|x)$$

projection to latent space

$$y = s(W\tilde{x} + b)$$

reconstruction of the input

$$x = s(W'y + b')$$

reconstruction error

$$L_H(x, z) = -sum_{k=1}^{d}[x_k \log z_k + (1 - x_k) \log(1 - z_k)]$$

# Weights of an autoencoder

```
def init_weights(n_in, h_hidden):
  w_init = numpy.asarray(
    numpy_rng.uniform(
      low=-4 * numpy.sqrt(6. / (n_hidden + n_in)),
      high=4 * numpy.sqrt(6. / (n_hidden + n_in)),
      size=(n_in, n_hidden)
    ))
  w = theano.shared(value=w_init, name='W', borrow=True)
        return w

def corrupt_input(input, corruption_level):
  return self.theano_rng.binomial(size=input.shape, n=1,
  p=1 - corruption_level) * input
```

# Reconstruction and costs

```
def hidden_values(input, w):
  return T.nnet.sigmoid(T.dot(input, w))

def reconstruct_input(hidden, w):
  return T.nnet.sigmoid(T.dot(hidden, w.T))

def cost_update(x, w, corruption_level, learning_rate):
  tilde_x = corrupt_input(x, corruption_level)
  y = hidden_values(tilde_x)
  z = reconstruct_input(y)
  L = - T.sum(x * T.log(z) + (1 - x) * T.log(1 - z), axis=1
  cost = T.mean(L)
  g = T.grad(cost, w)
  updates = [(w, w - learning_rate * g)]
  return (cost, updates)
```

# Putting it together

```
x = get_input()
w = init_weights(50,40)
corruption_level = 0.2
lr = 0.001
cost, updates = cost_update(x, w, corruption_level, lr)
train = theano.function(x, cost, updates=updates)
```