Introduction to Machine Learning Methods in Condensed Matter Physics

LECTURE 1, FALL 2021

Yi Zhang (张亿)

International Center for Quantum Materials, School of Physics
Peking University, Beijing, 100871, China

Email: frankzhangyi@pku.edu.cn

Course information

- **Instructor:** Yi Zhang (张亿) *frankzhangyi@pku.edu.cn*
- **Lectures:** Mon 18:40-20:30,

  School of Physics, Room W563
- **Teaching assistant:** Pei-lin Zheng (郑沛林), *peilinzheng@pku.edu.cn*
- **Mass communication, notifications and course slides:** available on course.pku.edu.cn
- **Reference:** Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015, also available online at: http://neuralnetworksanddeeplearning.com/

  Further references (mainly academic publications) will be added on the go.
- **Course grade**: Simple homework routines (40%) + final project (60%)
- Practice is key! Be both well-grounded and adventurous at the same time.

Syllabus

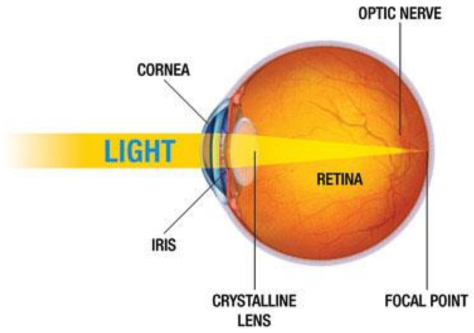
Machine learning methods	Condensed matter physics
➢ Supervised machine learning (artificial neural network, deep learning, adversarial attacks …) ➢ Unsupervised machine learning ➢ Reinforcement learning ➢ Generative modelling (restricted Boltzmann machine, generative adversarial network …) ➢ Quantum neural network	➢ Phases and properties ➢ Experimental and numerical data analysis ➢ Monte Carlo methods ➢ Renormalization group ➢ Control, time evolution, and dynamics ➢ Many-body state and entanglement ➢ …
➢ Common ground: abundant microscopic degrees of freedom versus (emergent properties of) a few collective degrees of freedom	

- I wish you all a fruitful semester!

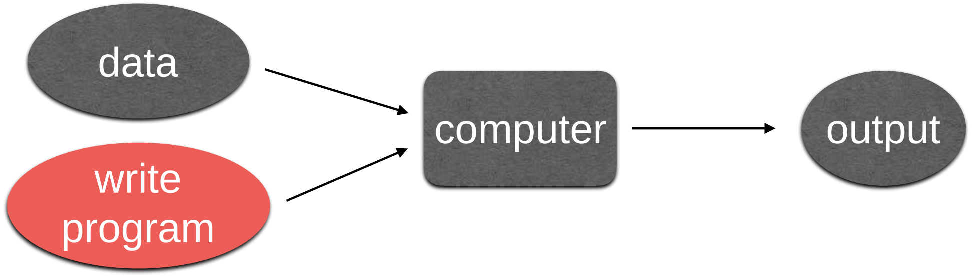
More is different. – P. W. Anderson

The image recognition problem task

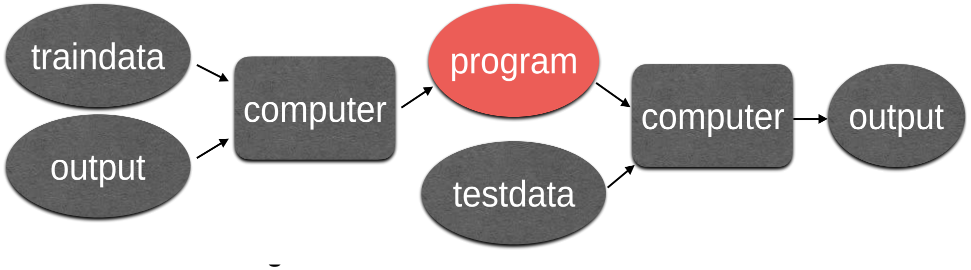
- The MNIST data set:



Conventional programing:

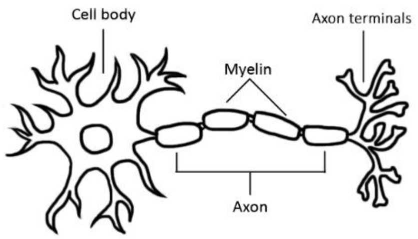


Machine learning:



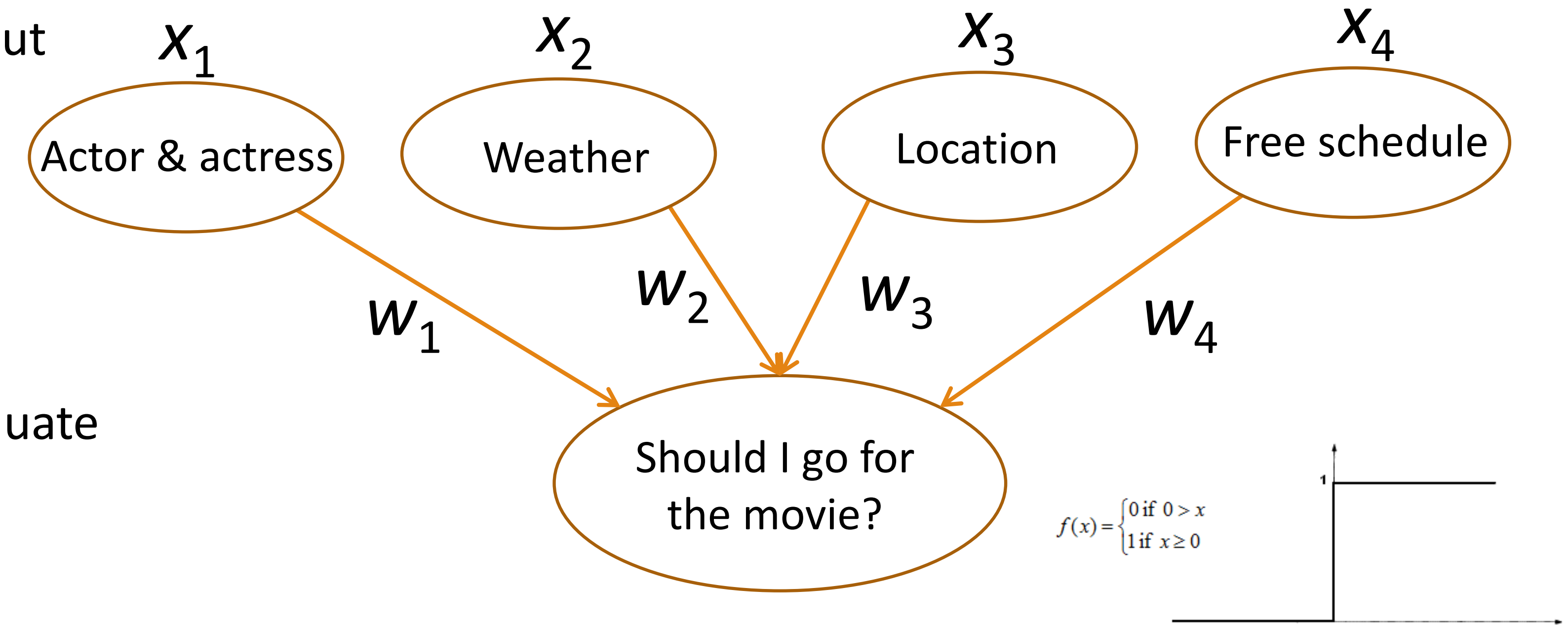
Perceptron

- Biological neural network:

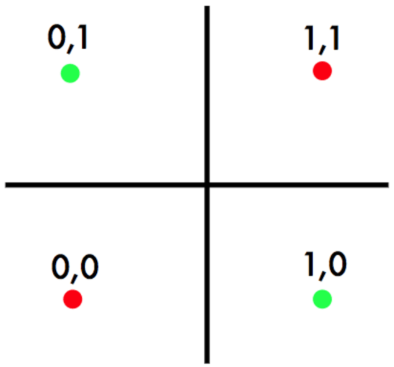


Perceptron for decision making: (1960s)

(1) Input



- Issue with XOR: (first AI winter)



(2) Evaluate

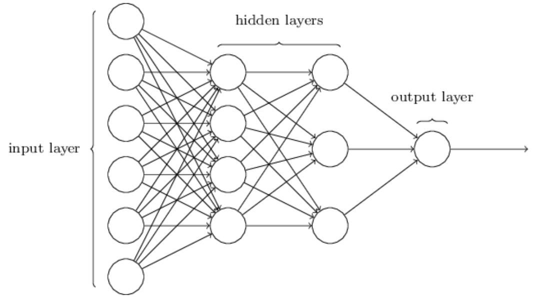
(3) Decision

Yes! $(\boldsymbol{w}\cdot\boldsymbol{x}+b>0)$ or No. $(\boldsymbol{w}\cdot\boldsymbol{x}+b<0)$

Artificial neural network

- Perceptron as NAND gates → multiple layers could help

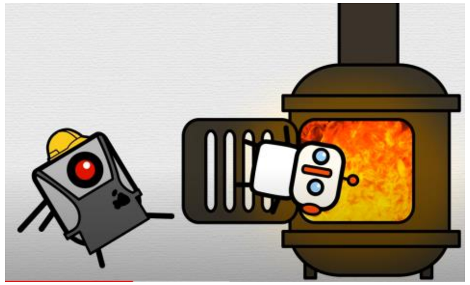
  (Fully-connected feed-forward neural network)



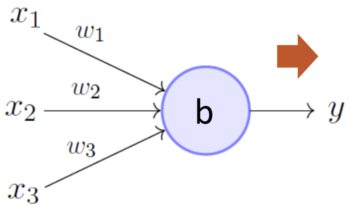
- Powerful non-linear expressions

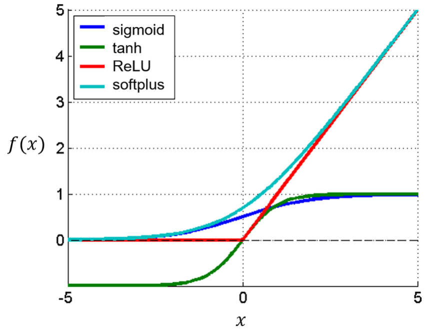
ANN with a single hidden layer of sufficient* width can express any nonlinear function within a given error.

- Optimization: (how does machine learn?)
- 1st generation: evolution;



- 2nd generation: smooth functions, finite differences;

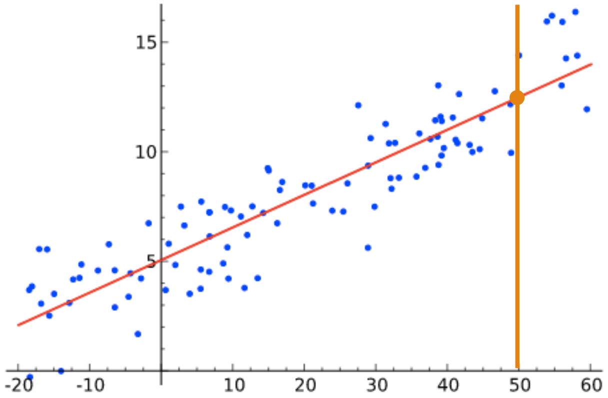




- 3rd generation: the gradient-descent family with back propagation

Perceptron revisited

Learn a function from data
(linear regression = least squares fit)

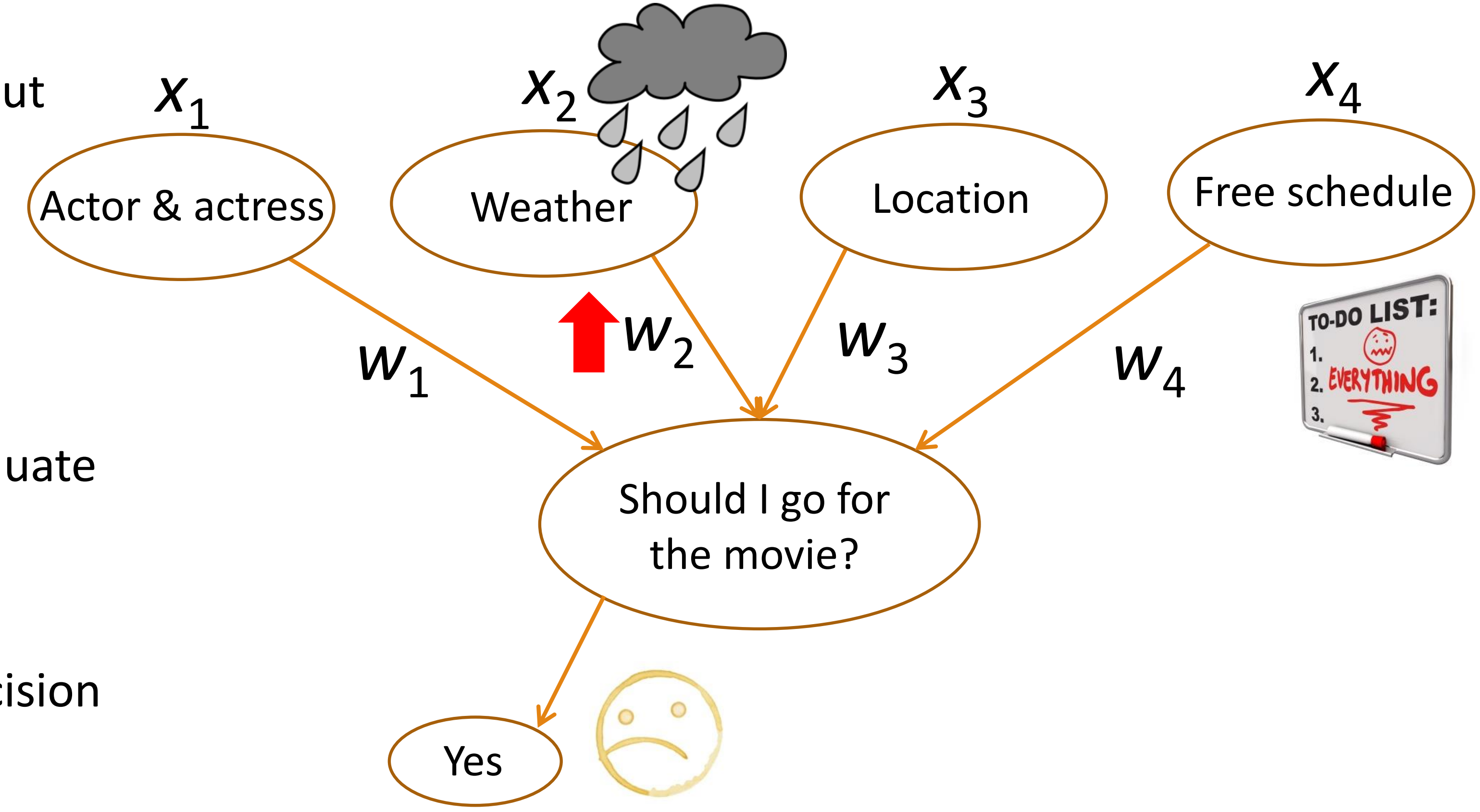


(1) Input

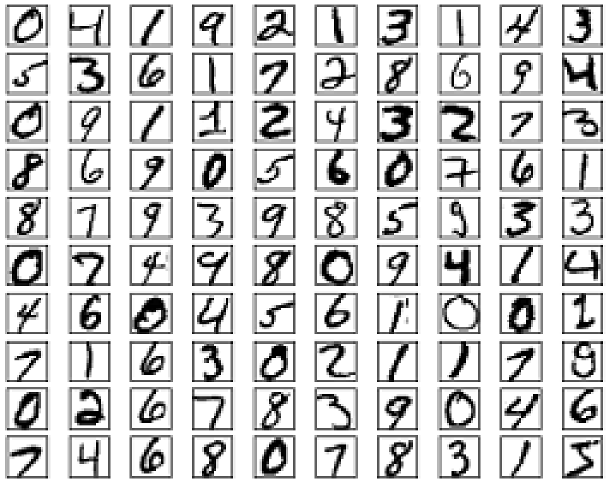
(2) Evaluate

(3) Decision

(4) Learn

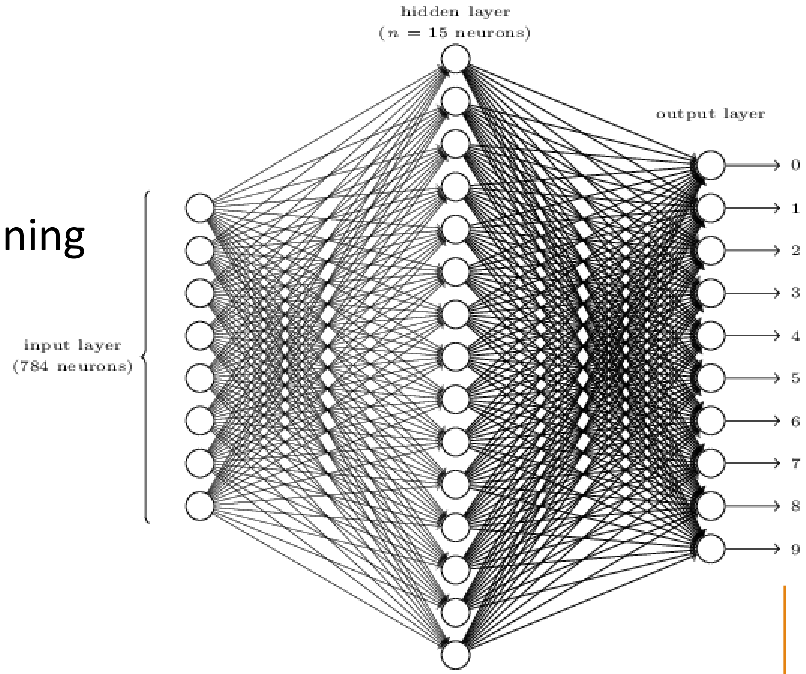


Artificial neural network learning



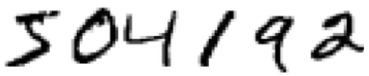
Established examples:
the ***training group***

1. Training



Optimal ***weights*** and ***biases***

2. Application



Minimize cost function: $C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$

ANN outputs: $a(x) = (0.1, 0.2, 0.1, 0.42, \dots)^T$

e.g. for image labeled '6': $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$

Gradient descent optimization

Minimize cost function: $$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

learning rate

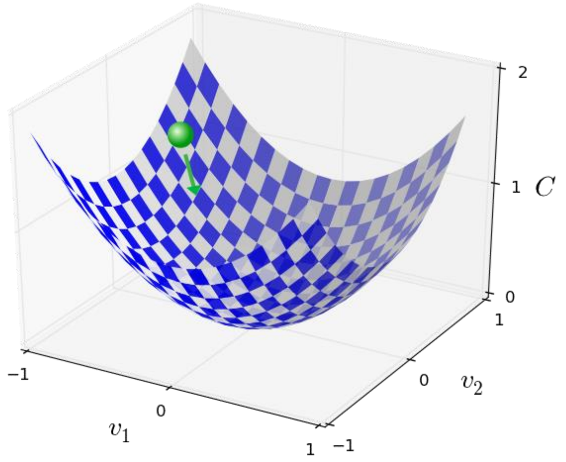
Adjust parameters v as: $$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \qquad b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \leq 0$$

Average over samples x: $$C = \frac{1}{n} \sum_x C_x \qquad C_x \equiv \frac{\|y(x)-a\|^2}{2}$$

Stochastic gradient descent:
Mini-batch of size m $$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}$$

Epoch: every sample is covered once in training.



$$v \rightarrow v' = v - \eta \nabla C$$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m} \right)^T$$

Back propagation

- Notations: from one layer to the next: $z^l \equiv w^l a^{l-1} + b^l$

  nonlinear activation function: $a^l = \sigma(z^l)$

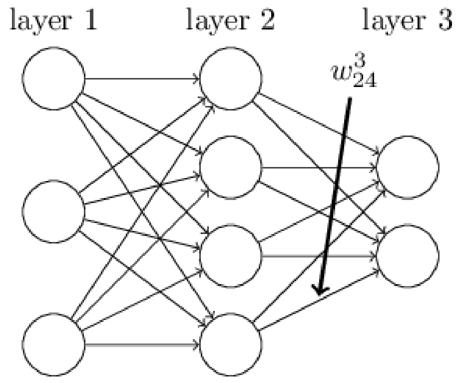
- Chain rule of derivatives:

(1) $$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

(2) $$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'(z_j^l)$$

(3) $$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

(4) $$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$



1. **Input x:** Set the corresponding activation a^1 for the input layer.
2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.
3. **Output error δ^L:** Compute the vector δ^L .
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute δ^l.
5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

- Usually applicable to a full mini-batch in matrix form.

A first ML code

```python
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], 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
```

```python
    def evaluate(self, test_data):
        test_results = [(np.argmax(self.feedforward(x)), y)
                        for (x, y) in test_data]
        return sum(int(x == y) for (x, y) in test_results)
```

```python
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))
```

```python
>>> net = network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

```python
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] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation)+b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-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)
```

```python
    def cost_derivative(self, output_activations, y):
        return (output_activations-y)
```

```python
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    if test_data: n_test = len(test_data)
    n = len(training_data)
    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)
        if test_data:
            print "Epoch {0}: {1} / {2}".format(
                j, self.evaluate(test_data), n_test)
        else:
            print "Epoch {0} complete".format(j)

def update_mini_batch(self, mini_batch, eta):
    """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)``, and ``eta``
    is the learning rate."""
    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 = [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)]
```