MNIST MLP

January 21, 2016

In [87]:
```
import numpy as np

class Layer:
    # A layer has no parameters by default
    parameters = []

    def update_parameters(self, input_, gradient_wrt_output, learning_rate):
        updates = self.parameter_gradients(input_, gradient_wrt_output)
        for parameter, update in zip(self.parameters, updates):
            parameter -= learning_rate * update

    def parameter_gradients(self, input_, gradient_wrt_output):
        return []

class Linear(Layer):
    def __init__(self, input_dim, output_dim):
        W = np.random.randn(input_dim, output_dim)
        b = np.random.randn(output_dim)
        self.parameters = [W, b]

    def forward_propagation(self, input_):
        W, b = self.parameters
        return input_.dot(W) + b

    def backward_propagation(self, input_, gradient_wrt_output):
        W, _ = self.parameters
        return gradient_wrt_output.dot(W.T)

    def parameter_gradients(self, input_, gradient_wrt_output):
        W, b = self.parameters
        return input_.T.dot(gradient_wrt_output), gradient_wrt_output.sum(axis=0)

class Sigmoid(Layer):
    def forward_propagation(self, input_):
        self.output = 1 / (1 + np.exp(-input_))
        return self.output

    def backward_propagation(self, input_, gradient_wrt_output):
        return gradient_wrt_output * self.output * (1 - self.output)

class Softmax(Layer):
    def forward_propagation(self, input_):
        exp_input = np.exp(input_ - input_.max(axis=1, keepdims=True))
```

```python
            self.output = exp_input / exp_input.sum(axis=1, keepdims=True)
            return self.output

        def backward_propagation(self, input_, gradient_wrt_output):
            gx = self.output * gradient_wrt_output
            gx -= self.output * gx.sum(axis=1, keepdims=True)
            return gx
```

In [88]:
```python
class MLP(Layer):
    def __init__(self, layers):
        self.layers = layers

    def forward_propagation(self, input_):
        # We remember the inputs for each layer so that we can use
        # them during backpropagation
        self.inputs = []
        for layer in self.layers:
            self.inputs.append(input_)
            output = layer.forward_propagation(input_)
            input_ = output
        return output

    def backward_propagation(self, input_, gradient_wrt_output):
        # We remember the gradients so that we can use them for the parameter updates
        self.gradients_wrt_output = []
        for input_, layer in zip(self.inputs[::-1], self.layers[::-1]):
            self.gradients_wrt_output.append(gradient_wrt_output)
            gradient_wrt_input = layer.backward_propagation(input_, gradient_wrt_output)
            gradient_wrt_output = gradient_wrt_input
        self.gradients_wrt_output = self.gradients_wrt_output[::-1]
        return gradient_wrt_input

    def update_parameters(self, input_, gradient_wrt_output, learning_rate):
        for input_, gradient_wrt_output, layer in zip(self.inputs, self.gradients_wrt_output, s
            layer.update_parameters(input_, gradient_wrt_output, learning_rate)
```

In [89]:
```python
class CrossEntropy(object):
    def cost(self, activations, targets):
        target_activations = activations[np.arange(activations.shape[0]), targets]
        return -np.log(target_activations).mean()

    def gradient(self, activations, targets):
        g = np.zeros_like(activations)
        g[np.arange(g.shape[0]), targets] = -1 / activations[np.arange(g.shape[0]), targets]
        return g

class Classification(object):
    def cost(self, activations, targets):
        decisions = activations.argmax(axis=1)
        return (decisions == targets).mean()
```

In [90]:
```python
import gzip
import pickle

def load_data():
```

```
        with gzip.open('mnist.pkl.gz', 'rb') as f:
            return pickle.load(f, encoding='latin-1')
```

In [91]:
```
# Let's construct our MLP
mlp = MLP([Linear(784, 100), Sigmoid(), Linear(100, 10), Softmax()])
```

In [92]:
```
train_set, valid_set, test_set = load_data()
train_X, train_y = train_set
valid_X, valid_y = valid_set
test_X, test_y = test_set

num_epochs = 30
batch_size = 100
num_batches = int(train_set[0].shape[0] / batch_size)

for epoch in range(1, num_epochs + 1):
    y_hat = mlp.forward_propagation(valid_X)
    cost = Classification().cost(y_hat, valid_y)
    print(cost)
    for i in range(num_batches):
        start = batch_size * i
        stop = batch_size * (i + 1)
        X = train_X[start:stop]
        T = train_y[start:stop]

        y_hat = mlp.forward_propagation(X)
        gradient = CrossEntropy().gradient(y_hat, T)
        mlp.backward_propagation(X, gradient)
        mlp.update_parameters(X, gradient, 0.01)
```

```
0.0758
0.8898
0.9105
0.9214
0.9276
0.9325
0.9353
0.9375
0.9389
0.9408
0.9416
0.9427
0.9442
0.9446
0.9443
0.9449
0.9456
0.9462
0.9466
0.9476
0.9484
0.9479
0.9485
0.9494
0.9498
```

```
0.9504
0.9507
0.9513
0.9514
0.9513
```

In [ ]: