6CS012: AI and Machine Learning

Lecture 07:
Neural Network and Hyper-Parameters Optimization

Dr. Ahsan Adeel

Lecture 6 Review
(Pay attention to the whiteboard)

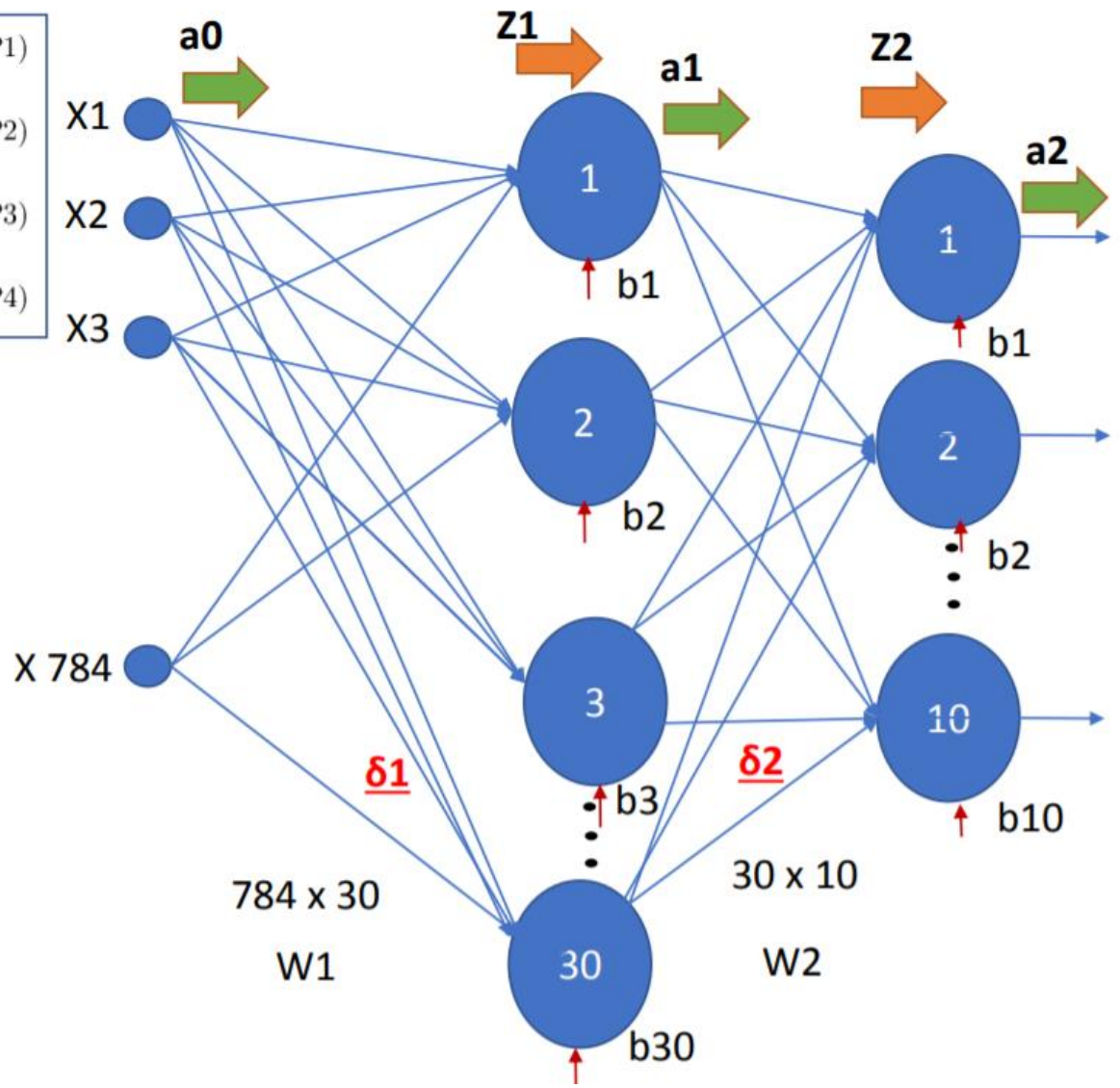
Let's use multiple neurons to recognize handwritten digits

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{(BP4)}$$



δ1 = ?
δ2 = ?
δ3 = ?

$\nabla J(W1) = \delta 1 \cdot a0^T$
$\nabla J(W2) = \delta 2 \cdot a1^T$
$\nabla J(b1) = \delta 1$
$\nabla J(b2) = \delta 2$

W1 = W1 - $\eta \nabla J(W1)$
W2 = W3 - $\eta \nabla J(W2)$
b1 = b1 - $\eta \nabla J(b1)$
b2 = b2 - $\eta \nabla J(b2)$

Typo: its W2

δ3 = a2 - y

$Z^l = W^l a^{l-1} + b^l$

$a^l = \sigma(Z^l)$

MLP executable code will be provided in the Lab

```
>>> 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,
```

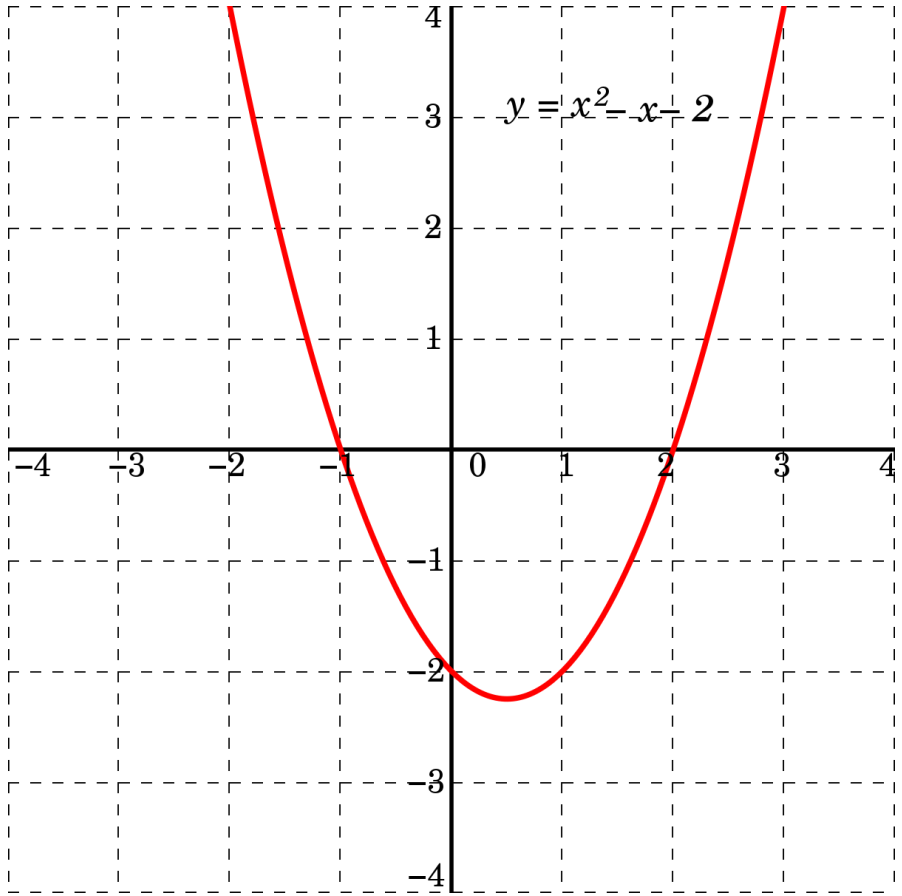
def SGD(self, **training_data**, **epochs**, **mini_batch_size**, **eta**, **test_data**)

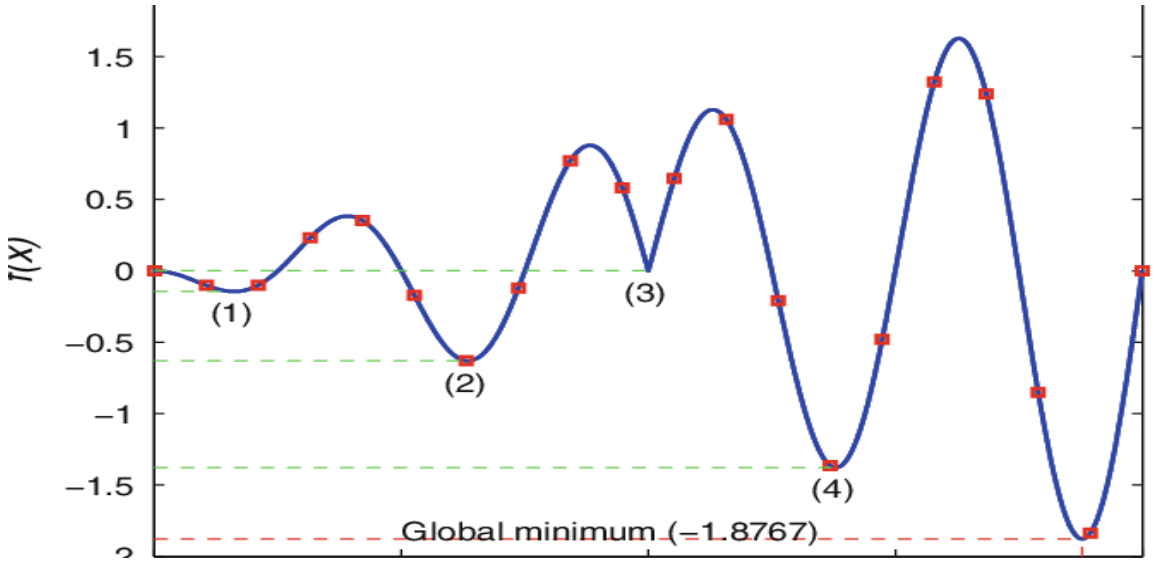
Hyper-Parameters Optimization

- Learning rate
- Number of hidden neurons/layers
- Cost function
- Overfitting/Regularization
- Dropout
- Weight Initialization

- **These are some of the inherent hyper-parameters of the neural network that we need to choose carefully.**
- **Their settings vary problem to problem.**
- **There are ways to get them right which also depends on the experience.**

Learning rate





Cost function: Quadratic cost function and saturated neuron issue

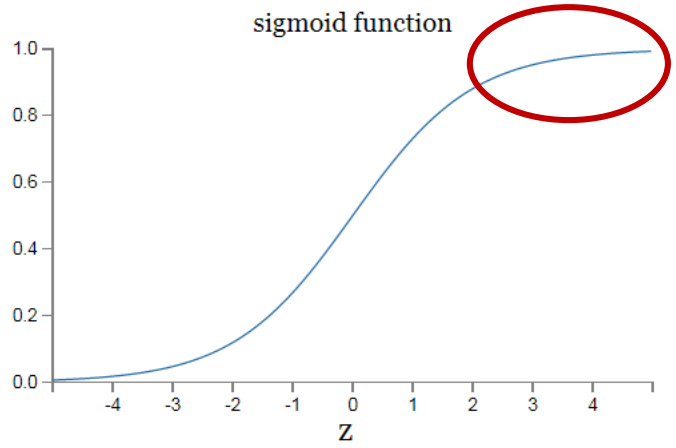
Quadratic cost function

$$C = \frac{(y-a)^2}{2}$$ Where, a = $\boldsymbol{\sigma}(z) = \boldsymbol{\sigma}(wx + b)$

Quadratic cost function derivative

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$ Where x=1 and y=0 for simplicity.

Learning slow down problem



Please note that when the neuron's output is close to 1, the curve gets very flat, and so σ'(z) gets very small. This is the origin of learning slow down.

Cost function: Cross-entropy cost function and solution to saturated neuron problem

Cross-entropy cost function

From Lecture 4

$$C = -\frac{1}{n}\sum_x [y \ln a + (1-y)\ln(1-a)]$$

Cross-entropy cost function derivative

$$\frac{\partial C}{\partial w_j} = \frac{1}{n}\sum_x x_j(\sigma(z) - y).$$

Addresses learning slow down problem

- The rate at which the weight learns is controlled by the error in the output neuron i.e. σ(z)−y
- No derivative
- It means the larger the error, the faster the neuron learning.
- How humans learn when the error is large?

Note that only one of the two terms in the summation is non-zero for each training example (depending on whether the label y(i) is 0 or 1). When y(i)=1, minimizing the cost function means we need to make first term large, and when y(i)=0 we want to make the second term large.

Overfitting

- Overfitting is a major problem in neural networks.
- It is important to detect overfitting for effective training.
- There are several techniques for reducing the effects of overfitting.
- One obvious way is to detect overfitting by keeping track of accuracy on the test data as our network trains. This is called **early stopping**

Overfitting

- Let’s construct a scenario where the network does a bad job in terms of generalization.
- Using the same network but not with all 50,000 MNIST training images. Instead, only first 1,000 training images.
- This will make the generalization problem much more evident.
- We will use cross-entropy cost function, learning rate of 0.5, and a mini-batch size of 10.

Overfitting

Note:

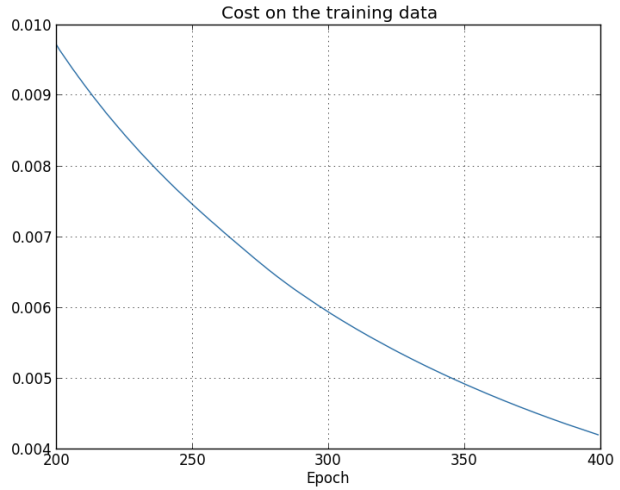
- A smooth decrease in the cost.

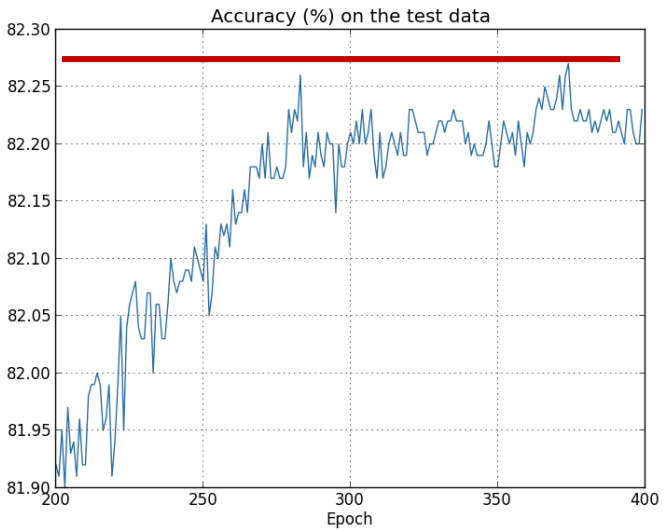
- The classification accuracy on the test data: Learning stops improving around 280 epochs with fluctuations.

- **Cost vs accuracy:** Cost on the training data continues to smoothly drop (i.e. model is still getting better), whereas the test accuracy says it's a delusion.

- What our network learns after epoch 280 is no longer generalizes to the test data – Hence, not useful learning. **What's going wrong here?**

This is called overfitting or overtraining beyond epoch 280.





Solution to overfitting: Regularization

- Increasing the amount of training data is one way of reducing overfitting.
- Another possible approach is to reduce the size of our network. However, large networks have the potential to be more powerful than small networks, and so this is an option we'd only adopt reluctantly.
- Fortunately, there are other techniques which can reduce overfitting, even when we have a fixed network and fixed training data. These are known as **regularization techniques.**
- One of the most commonly used regularization techniques, known as weight decay or L2 regularization.
- The idea of L2 regularization is to add an extra term to the cost function, a term called the regularization term. Here's the regularized cross-entropy:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right] + \frac{\lambda}{2n}\sum_w w^2$$

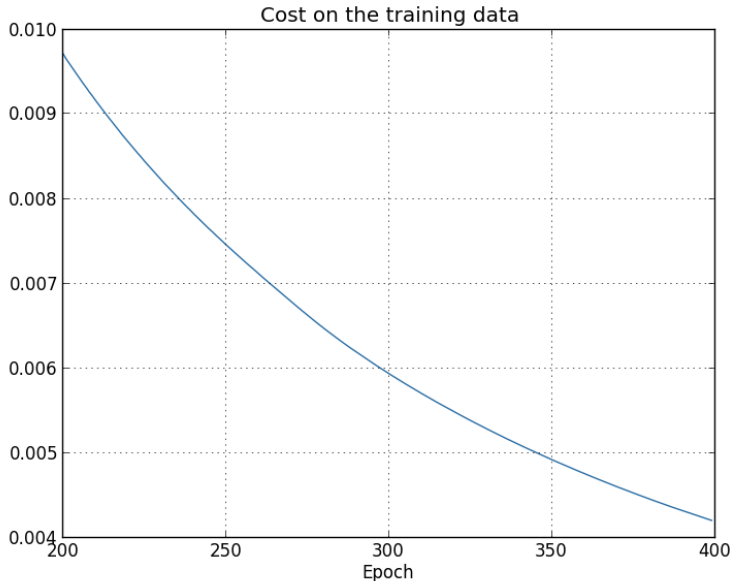
penalizing large weights, and tending to make the network prefer small weights.

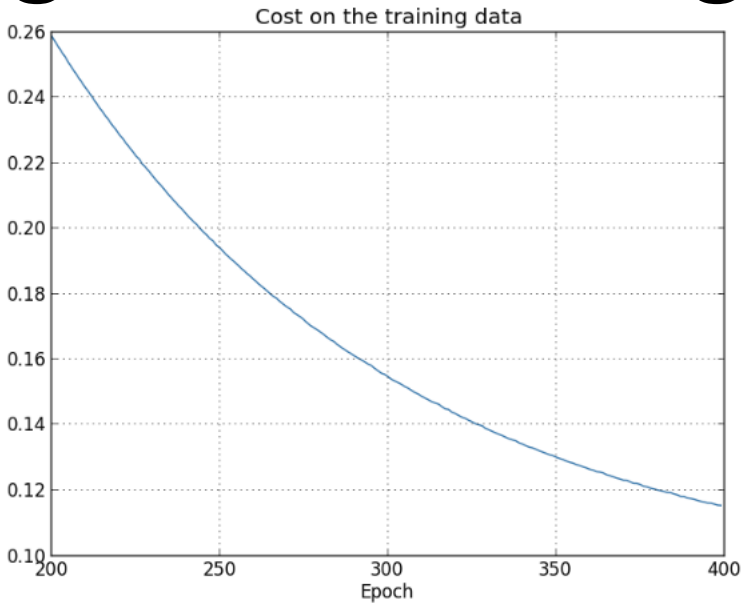
Solution to overfitting: Regularization

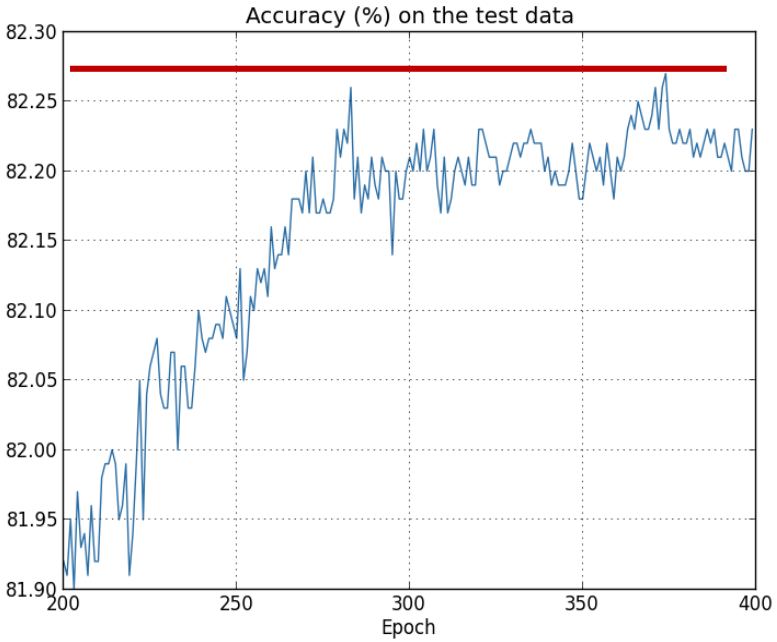
- Of course, it's possible to regularize other cost functions, such as the quadratic cost. This can be done in a similar way:

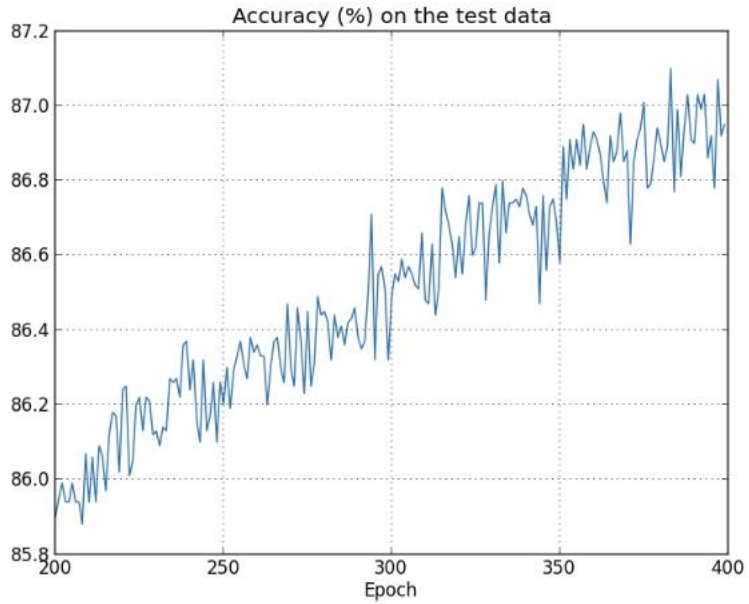
$$C = \frac{1}{2n} \sum_x \|y - a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2.$$

Non-Regularized vs Regularized





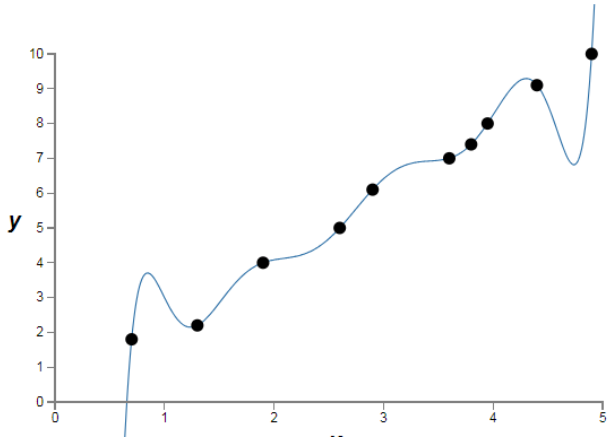




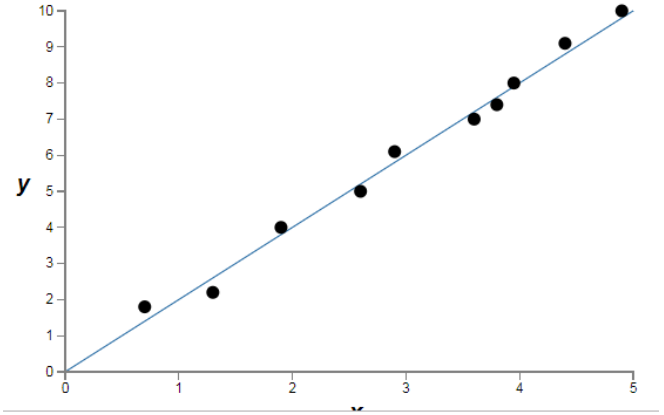
- The use of regularization has addressed overfitting.
- The accuracy has increased considerably achieving a peak accuracy of 87.1% as compared to 82.27% achieved by the unregularized model.

Why does regularization help reduce overfitting?

- We've seen empirically that regularization helps reduce overfitting but why?



Ten points graph modelled with a unique 9th-order polynomial $y=a_0x^9+a_1x^8+\ldots+a_9$ (Exact fit)



y=mx + noise

What is happening? The model y=mx+noise seems much simpler than 9th order polynomial model which is really just learning the effects of local noise i.e. not important in this case

Code Overview

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```

```
>>> import network
>>> net = network.Network([784, 30, 10])
```

```
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)
```

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):
    """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.  If ``test_data`` is provided then the
    network will be evaluated against the test data after each
    epoch, and partial progress printed out.  This is useful for
    tracking progress, but slows things down substantially."""
    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)]
```

```
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_derivative(activations[-1], y) * \
        sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in Chapter 2 of the book.  Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on.  It's a renumbering of the
    # scheme in the book, used here to take advantage of the fact
    # that Python can use negative indices in lists.
    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)
```

```
#### Miscellaneous functions
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))
```

```
def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial C_x /
    \partial a for the output activations."""
    return (output_activations-y)
```

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

95.42% classification accuracy

```python
def evaluate(self, test_data):
    """Return the number of test inputs for which the neural
    network outputs the correct result. Note that the neural
    network's output is assumed to be the index of whichever
    neuron in the final layer has the highest activation."""
    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 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
```

WS7

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

1. Download mnist.pkl.gz
2. Download the file ‘MNIST_MLP_first.py’
3. Download ‘mnist_loader.py’
4. Open ‘mnist_loader.py’
5. Change the path/replace with the MNIST dataset location i.e. f = gzip.open(‘**File_Path/mnist.pkl.gz**', 'rb’)
6. Save all
7. Open the new terminal
8. Go to the saved directory where ‘MNIST_MLP_first.py’ is saved
9. Write python ‘MNIST_MLP_first.py’

WS7 – Task1

(network structure and learning rate)

1.1. Change the number of hidden neurons to 100 from 30 and explain the results in terms of accuracy and processing time.

1.2. Compare your results with the student sitting next to you and explain your results are different?

1.3. Change the learning rate to 0.1 and explain the results.

WS7 – Task1

(network structure and learning rate)

1.4. Set the learning rate to 1 and epochs to 10, observe the behaviour and suggest possible changes to the hyper-parameters.

1.5. Keep increasing the learning rate up to 3 and observe the network behaviour.

1.6. Now, increase the learning rate to 100. Observe and explain the behaviour

WS7 – Task2

(Cost function)

- Download networ2.py
- Download cross_entropy_cost_function.py

2.1. Observe and explain the improvement in accuracy as compared to network1.py (Quadratic function)

2.2. Observe the network behaviour, why there is a fluctuation in learning?

2.3. How could we address the fluctuation to stabilise the network?

WS7 – Task3

(Regularization)

- Download overfitting.py
- Create a new text file in the same directory name it, ‘overfitting_regularizaiton_results’
- Run the code
- Enter:
  1. Text file path
  2. Training cost (x-axis range) = 0
  3. Training accuracy (x-axis range) = 0
  4. Test cost (x-axis range) = 0
  5. Test accuracy (x-axis range) = 0
  6. Training set size = 100
  7. Regularization parameter = 0.05

WS7 – Task3

(Regularization)

- net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost())
- net.large_weight_initializer()
- **test_cost**, **test_accuracy**, **training_cost**, **training_accuracy** \

= net.SGD(training_data[:**training_set_size**], num_epochs, 10, 0.5, evaluation_data=test_data)

WS7 – Task3

(Regularization)

Let's see how regularization changes the performance of our neural network. We'll use a network with 30 hidden neurons, a mini-batch size of 10, a learning rate of 0.5, and the cross-entropy cost function. Use regularization parameter:

- lmbda=0.0
- lmbda=0.1
- lmbda=0.5

3.1. Explain the differences in results for different values of lmbda

References

- https://www.axonoptics.com/2015/06/light-triggers-migraines/
- Huff, Trevor, and Scott C. Dulebohn. "Neuroanatomy, Visual Cortex." (2017).
- Ian Goodfellow and Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press, 2016, url: http://www.deeplearningbook.org
- 3blue1brown: https://www.3blue1brown.com/
- http://neuralnetworksanddeeplearning.com/index.html