Student Names and IDs:

- Newton Kwan, nk150
- Joyce Choi, jc515
- Ashka Stevens, aas74

Homework 10

Part 1: Correlation

Problem 1.1

Let

$$\mathbf{x} = [1, 0, 2, -1, 3]^T \quad \text{and} \quad \mathbf{g} = [-1, 2, 1]^T \ .$$

Give the valid correlation $\mathbf{z}_v$ and the shape-preserving correlation $\mathbf{z}_s$ of signal $\mathbf{x}$ with kernel $\mathbf{g}$.

Answer

One dimensional correlation of input x with kernel g

$$z_i = \sum_{a=0}^{k-1} g_a x_{i+a} \text{ for } i = 0, \ldots, e-1 = d-k$$

Signal $\mathbf{x}$

- $x \in \mathbb{R}^d$
- d = 5
- k = 3

In this specific case,

$$z_i = \sum_{a=0}^{2} g_a x_{i+a} \text{ for } i = 0, 1, 2$$

$$\begin{aligned} z_0 &= g_0 x_0 + g_1 x_1 + g_2 x_2 \\ z_1 &= g_0 x_1 + g_1 x_2 + g_2 x_3 \\ z_2 &= g_0 x_2 + g_1 x_3 + g_2 x_4 \end{aligned}$$

Valid correlation

$$\mathbf{z_v} = [1, 3, -1]$$

Padded signal $\mathbf{x}' = [1, 0, 2, -1, 3, 0, 0]$

- $x' \in \mathbb{R}^{d+k-1}$
- d+k-1 = 7
- k=3

In this specific case,

$$z_i = \sum_{a=0}^{2} g_a x_{i+a} \text{ for } i = 0, 1, 2, 3, 4$$

$$\begin{aligned} z_0 &= g_0 x_0 + g_1 x_1 + g_2 x_2 \\ z_1 &= g_0 x_1 + g_1 x_2 + g_2 x_3 \\ z_2 &= g_0 x_2 + g_1 x_3 + g_2 x_4 \\ z_3 &= g_0 x_3 + g_1 x_4 + g_2 x_5 \\ z_4 &= g_0 x_4 + g_1 x_5 + g_2 x_6 \end{aligned}$$

Shape-preserving or padded correlation

$$\mathbf{z_s} = [1, 3, -1, 7, -3]$$

Problem 1.2

Using the values in the previous problem, give the matrices V_v and V_s such that

$$\mathbf{z}_v = V_v \mathbf{x} \quad \text{and} \quad \mathbf{z}_s = V_s \mathbf{x} \; .$$

Answer

$$V_v = \begin{bmatrix} g_0 & g_1 & g_2 & 0 & 0 \\ 0 & g_0 & g_1 & g_2 & 0 \\ 0 & 0 & g_0 & g_1 & g_2 \end{bmatrix} = \begin{bmatrix} -1 & 2 & 1 & 0 & 0 \\ 0 & -1 & 2 & 1 & 0 \\ 0 & 0 & -1 & 2 & 1 \end{bmatrix}$$

$$V_s = \begin{bmatrix} g_0 & g_1 & g_2 & 0 & 0 & 0 & 0 \\ 0 & g_0 & g_1 & g_2 & 0 & 0 & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & 0 & 0 \\ 0 & 0 & 0 & g_0 & g_1 & g_2 & 0 \\ 0 & 0 & 0 & 0 & g_0 & g_1 & g_2 \end{bmatrix} = \begin{bmatrix} -1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 2 & 1 \end{bmatrix}$$

Problem 1.3

Let

$$X = \begin{bmatrix} 2 & 1 & 3 & -1 & 4 \\ 0 & -2 & 1 & 0 & 3 \end{bmatrix} \quad \text{and} \quad G = \begin{bmatrix} 1 & 2 \\ -3 & 0 \end{bmatrix} \; .$$

Give the valid correlation Z_v and the shape-preserving correlation Z_s of image X with kernel G.

Answer

$$X = \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} & x_{04} \\ x_{10} & x_{11} & x_{12} & x_{13} & x_{14} \end{bmatrix} \quad \text{and} \quad G = \begin{bmatrix} g_{00} & g_{01} \\ g_{10} & g_{11} \end{bmatrix} \; .$$

In general, the two-dimensional correlation of input X with kernel G is

$$z_{ij} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} g_{ab} x_{i+a,j+b} \text{ for } i = 0, \ldots, e_1 - 1 = d_1 - k_1 \text{ and } j = 0, \ldots, e_2 - 1 = d_2 - k_2$$

For our example, G has the following specifications

- $G \in \mathbb{R}^{2\times 2}$
- $k_1 = 2$
- $k_2 = 2$

For the valid correlation Z_v

- $X \in \mathbb{R}^{2\times 5}$
- $d_1 = 2$
- $d_2 = 5$
- $d_1 - k_1 = 0$
- $d_2 - k_2 = 3$
- $e_1 = 1$
- $e_2 = 4$

$$z_{ij} = \sum_{a=0}^{1}\sum_{b=0}^{1} g_{ab}x_{i+a,j+b} \text{ for } i = 0 \text{ and } j = 0,\ldots,3$$

$$\begin{aligned}
z_{00} &= g_{00}x_{00} + g_{01}x_{01} + g_{10}x_{10} + g_{11}x_{11} \text{ for } i = 0, j = 0\\
z_{01} &= g_{00}x_{01} + g_{01}x_{02} + g_{10}x_{11} + g_{11}x_{12} \text{ for } i = 0, j = 1\\
z_{02} &= g_{00}x_{02} + g_{01}x_{03} + g_{10}x_{12} + g_{11}x_{13} \text{ for } i = 0, j = 2\\
z_{03} &= g_{00}x_{03} + g_{01}x_{04} + g_{10}x_{13} + g_{11}x_{14} \text{ for } i = 0, j = 3
\end{aligned}$$

$$Z_v = [4, 13, -2, 7]$$

For the shape preserving correlation Z_s

Padding is computed using

$$p_m = k_m - 1$$

- $p_1 = k_1 - 1 = 1$
- $p_2 = k_2 - 1 = 1$

We add a row to the bottom and a column to the right of X to get X'

- $X' \in \mathbb{R}^{3\times 6}$
- $d_1 = 3$
- $d_2 = 6$
- $d_1 - k_1 = 1$
- $d_2 - k_2 = 4$
- $e_1 = 2$
- $e_2 = 5$

$$z_{ij} = \sum_{a=0}^{1}\sum_{b=0}^{1} g_{ab}x_{i+a,j+b} \text{ for } i = 0, 1 \text{ and } j = 0, \ldots, 4$$

$$\begin{aligned}
z_{00} &= g_{00}x_{00} + g_{01}x_{01} + g_{10}x_{10} + g_{11}x_{11} \text{ for } i = 0, j = 0\\
z_{01} &= g_{00}x_{01} + g_{01}x_{02} + g_{10}x_{11} + g_{11}x_{12} \text{ for } i = 0, j = 1\\
z_{02} &= g_{00}x_{02} + g_{01}x_{03} + g_{10}x_{12} + g_{11}x_{13} \text{ for } i = 0, j = 2\\
z_{03} &= g_{00}x_{03} + g_{01}x_{04} + g_{10}x_{13} + g_{11}x_{14} \text{ for } i = 0, j = 3\\
z_{04} &= g_{00}x_{04} + g_{01}x_{05} + g_{10}x_{14} + g_{11}x_{15} \text{ for } i = 0, j = 4
\end{aligned}$$

$$\begin{aligned}
z_{10} &= g_{00}x_{10} + g_{01}x_{11} + g_{10}x_{20} + g_{11}x_{21} \text{ for } i = 1, j = 0\\
z_{11} &= g_{00}x_{11} + g_{01}x_{12} + g_{10}x_{21} + g_{11}x_{22} \text{ for } i = 1, j = 1\\
z_{12} &= g_{00}x_{12} + g_{01}x_{13} + g_{10}x_{22} + g_{11}x_{23} \text{ for } i = 1, j = 2\\
z_{13} &= g_{00}x_{13} + g_{01}x_{14} + g_{10}x_{23} + g_{11}x_{24} \text{ for } i = 1, j = 3\\
z_{14} &= g_{00}x_{14} + g_{01}x_{15} + g_{10}x_{24} + g_{11}x_{25} \text{ for } i = 1, j = 4
\end{aligned}$$

$$Z_s = \begin{bmatrix} z_{00} & z_{01} & z_{02} & z_{03} & z_{04} \\ z_{10} & z_{11} & z_{12} & z_{13} & z_{14} \end{bmatrix}$$

$$Z_s = \begin{bmatrix} 4 & 13 & -2 & 7 & -5 \\ -4 & 0 & 1 & 6 & 3 \end{bmatrix}$$

Problem 1.4

Give the image Z_v as defined in the previous problem, except that stride 2 is used.

Answer

Stride = s_m = 2

- $d_1 = 2$
- $d_2 = 5$
- $s_1 = 2$
- $s_2 = 2$
- $e_1 \approx d_1/s_1 \approx 1$
- $e_2 \approx d_2/s_2 \approx 2$
- $Z_v \in \mathbb{R}^{1\times 2}$

$$z_{00} = g_{00}x_{00} + g_{01}x_{01} + g_{10}x_{10} + g_{11}x_{11}$$
$$z_{01} = g_{00}x_{02} + g_{01}x_{03} + g_{10}x_{12} + g_{11}x_{13}$$

$$Z_v = [4, -2]$$

Part 2: Layers

In [424]:
```
import numpy as np

def tensorize(x):
    return np.squeeze(np.asfarray(x))


class Layer:

    def __init__(self, parms, f, dfdx):
        self.parms = [tensorize(p) for p in parms]
        self.f = f
        self.dfdx = dfdx
        self.x = None


    def reset(self, r=None):
        self.x = None


    def getWeights(self):
        if len(self.parms) == 0:
            return []
        else:
            return np.concatenate([p.flatten() for p in self.parms])


    def setWeights(self, w):
        if len(w) > 0:
            w = tensorize(w)
            for k in range(len(self.parms)):
                s = self.parms[k].shape
                n = 1 if len(s) == 0 else np.prod(s)
                self.parms[k] = np.reshape(w[:n], s)
                w = w[n:]


    def dfdw(self):
        assert self.x is not None, 'dfdw called before f'
        return np.empty((len(self.x), 0))


class FCLayer(Layer):

    def __init__(self, V, b):
        V, b = tensorize(V), tensorize(b)

        def f(x):
            self.x = tensorize(x)
            return np.dot(self.parms[0], self.x) + self.parms[1]

        def dfdx():
            assert self.x is not None, 'dfdx called before f'
            return self.parms[0]

        Layer.__init__(self, [V, b], f, dfdx)
```

```
    def dfdw(self):
        assert self.x is not None, 'dfdw called before f'
        m, n = self.parms[0].shape
        D = np.zeros((m, m * (n + 1)))
        js, je = 0, n
        for i in range(m):
            D[i][js:je] = self.x
            js, je = js + n, je + n
        D[:, (m * n):] = np.diag(np.ones(m))
        return D


    def __initialWeights(m, n, r=None):
        if r is None:
            r = np.sqrt(2/m) # Formula by He et al.
        V = np.random.randn(n, m) * r
        b = np.zeros(n)
        return V, b


    @classmethod
    def ofShape(cls, m, n, r=None):
        V, b = FCLayer.__initialWeights(m, n, r)
        return cls(V, b)


    def reset(self, r=None):
        self.x = None
        n, m = self.parms[0].shape
        V, b = FCLayer.__initialWeights(m, n, r)
        self.parms = [V, b]
```

Problem 2.1

Write an explicit formula for the Jacobian matrix J_ρ of

$$\mathbf{z} = \rho(\mathbf{x})$$

where ρ is the ReLU and $\mathbf{x} = [a, b]^T$.

Answer

In general,

$$J_\rho = \frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial z_1}{\partial x_1} & \frac{\partial z_1}{\partial x_2} \\ \frac{\partial z_2}{\partial x_1} & \frac{\partial z_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(1 + sign(x_1)) & 0 \\ 0 & \frac{1}{2}(1 + sign(x_2)) \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(1 + sign(a)) & 0 \\ 0 & \frac{1}{2}(1 + sign(b)) \end{bmatrix}$$

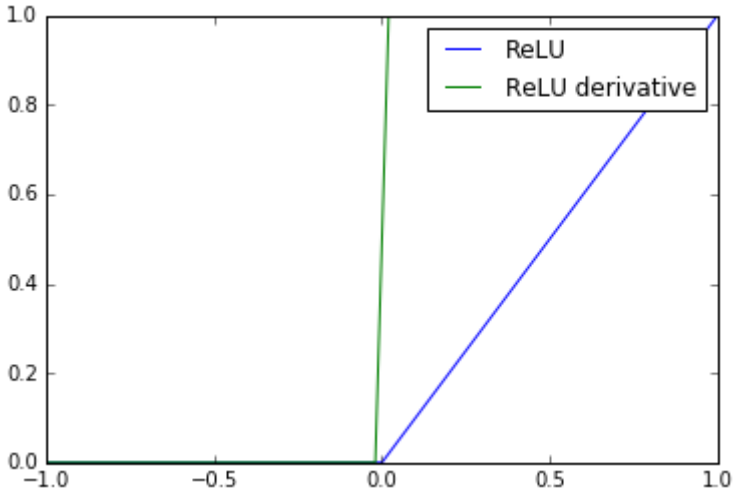
Problem 2.2

The code below is a partial implementation of a ReLU layer. Replace the two `pass` commands with code so that the `ReLULayer` class works correctly.

Answer (when Completed)

In [425]:
```
class ReLULayer(Layer):

    def __init__(self):

        def f(x):
            self.x = tensorize(x)
            return np.maximum(0, self.x)

        def dfdx():
            assert self.x is not None, 'dfdx called before f'
            diagVals = 0.5*(1+np.sign(self.x))
            diagValsTen = tensorize(diagVals)
            if diagValsTen.size >= 2:
                dfdxMatrix = np.diag(diagValsTen)
                return dfdxMatrix
            return diagValsTen

        Layer.__init__(self, [], f, dfdx)
```

In [426]:
```
try:
    xs = np.linspace(-1, 1, 101)
    relu = ReLULayer()
    out = [(relu.f(x), relu.dfdx()) for x in xs]
    y = [a[0] for a in out]
    dydx = [a[1] for a in out]

    import matplotlib.pyplot as plt
    %matplotlib inline

    plt.figure()
    plt.plot(xs, y, label='ReLU')
    plt.plot(xs, dydx, label='ReLU derivative')
    plt.legend()
    plt.show()
except:
    pass
```



Problem 2.3

Show that if $\mathbf{s} = \sigma(\mathbf{x})$ then

$$J_\sigma = \text{diag}(\mathbf{s}) - \mathbf{s}\,\mathbf{s}^T$$

where $\text{diag}(\mathbf{s})$ is a square matrix with the entries of $\mathbf{s}$ on its main diagonal.

Answer

The index in the numerator is the row and the index in the denominator is the column.

Generic entry on the diagonal

$$\frac{\partial \sigma_i}{\partial x_i} = \frac{e^{x_i}(\sum_{k=0}^{d-1} e^{x_k}, i \neq k)}{\sum_{k=0}^{d-1}(e^{x_k})^2}$$

Generic entry on the off diagonal

$$\frac{\partial \sigma_i}{\partial x_j} = -\frac{e^{x_i} e^{x_j}}{\sum_{k=0}^{d-1}(e^{x_k})^2}$$

Problem 2.4

Show that if c is any real number and $\sigma(\mathbf{x})$ is the `softmax` function, then

$$\sigma(\mathbf{x} - c) = \sigma(\mathbf{x}) \ .$$

Subtracting a scalar from a vector subtracts the scalar from each entry of the vector.

Answer

Looking at the ith entry of $\sigma(\mathbf{x} - c)$,

$$\sigma_i(\mathbf{x} - c) = \frac{e^{x_i - c}}{e^{x_0 - c} + \ldots + e^{x_i - c} + \ldots + e^{x_{d-1} - c}}$$

Rewrite exponentials,

$$\sigma_i(\mathbf{x} - c) = \frac{e^{x_i}\, e^{-c}}{e^{x_0}\, e^{-c} + \ldots + e^{x_i}\, e^{-c} + \ldots + e^{x_{d-1}}\, e^{-c}}$$

Factoring out e^{-c},

$$\sigma_i(\mathbf{x} - c) = \frac{e^{-c}}{e^{-c}} \frac{e^{x_i}}{e^{x_0} + \ldots + e^{x_i} + \ldots + e^{x_{d-1}}}$$

The constant is equal to 1,

$$\sigma_i(\mathbf{x} - c) = \frac{e^{x_i}}{e^{x_0} + \ldots + e^{x_i} + \ldots + e^{x_{d-1}}}$$

and

$$\sigma_i(\mathbf{x}) = \frac{e^{x_i}}{e^{x_0} + \ldots + e^{x_i} + \ldots + e^{x_{d-1}}}$$

We have shown this in the general case for the ith entry. Therefore,

$$\sigma(\mathbf{x} - c) = \sigma(\mathbf{x})$$

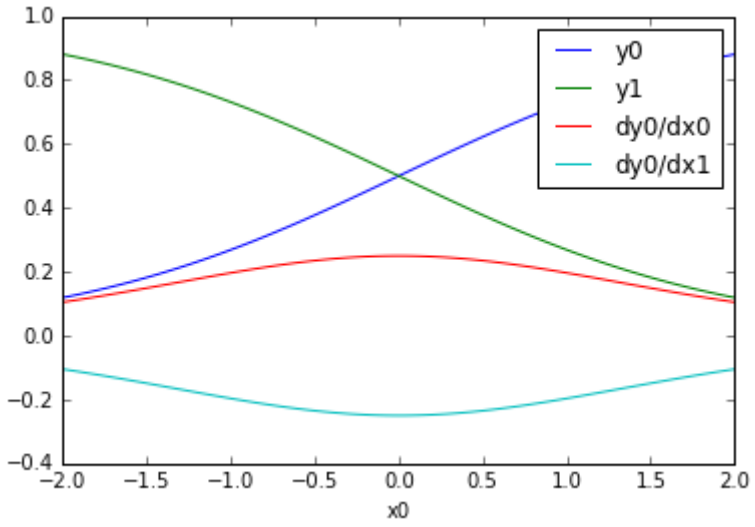
Problem 2.5

The code below is a partial implementation of a `softmax` layer. Replace the `pass` command with code so that the `SoftmaxLayer` class works correctly.

Answer (when Completed)

In [427]:
```
class SoftmaxLayer(Layer):

    def __softmax(x):
        e = np.exp(x - np.max(x))
        return e / np.sum(e)

    def __init__(self, n):

        def f(x):
            self.x = tensorize(x)
            return SoftmaxLayer.__softmax(self.x)

        def dfdx():
            assert self.x is not None, 'dfdx called before f'
            s = SoftmaxLayer.__softmax(self.x)
            if s.size >= 2:
                diagMat = np.diag(s)
            else:
                diagMat = s
            sProd = np.outer(s, s)
            return diagMat-sProd


        Layer.__init__(self, [], f, dfdx)
```

In [428]:
```
try:
    ps = np.linspace(-2, 2, 101)
    q = 0
    softmax = SoftmaxLayer(2)
    out = [(softmax.f((p, q)), softmax.dfdx()) for p in ps]
    #print(out)

    y0 = [a[0][0] for a in out]
    y1 = [a[0][1] for a in out]
    dydx00 = [a[1][0, 0] for a in out]
    dydx01 = [a[1][0, 1] for a in out]

    plt.figure()
    plt.plot(ps, y0, label='y0')
    plt.plot(ps, y1, label='y1')
    plt.plot(ps, dydx00, label='dy0/dx0')
    plt.plot(ps, dydx01, label='dy0/dx1')
    plt.xlabel('x0')
    plt.legend()
    plt.show()
except:
    pass
```



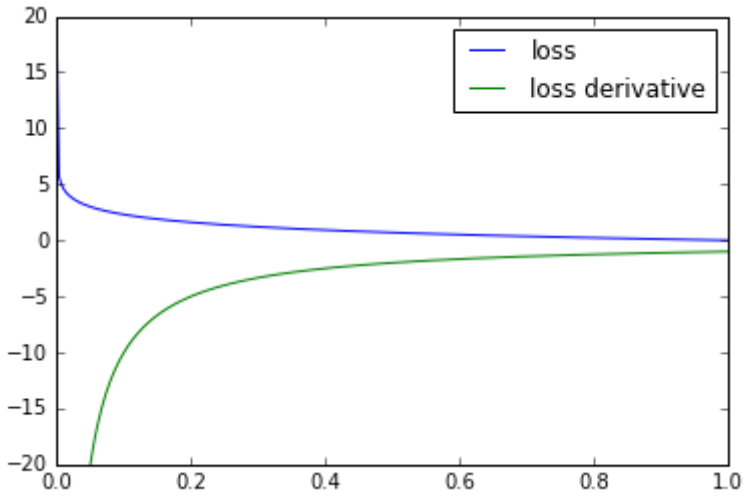
Problem 2.6

The code below is a partial implementation of a `Loss` class that embodies the cross-entropy loss and its Jacobian. Replace the two `pass` commands with code so that the `Loss` class works correctly.

Answer (when Completed)

In [429]:
```
class Loss:
    def __init__(self):
        self.small = 1e-8

    def f(self, y, p):
        self.p = tensorize(p)
        py = self.p[int(y)]
        if py < self.small: py = self.small
        return -np.log(py)

    def dfdx(self, y):
        assert self.p is not None, 'dfdx called before f'
        y = int(y)
        d = np.zeros(len(self.p))
        py = self.p[y]
        if py < self.small: py = self.small
        d[y] = -1/py
        return d
```

In [430]:
```
try:
    loss = Loss()
    l, d, y = [], [], 0
    xs = np.linspace(0, 1, 301)
    for x in xs:
        p = (x, 1-x)
        l.append(loss.f(y, p))
        dfdx = loss.dfdx(y)
        if dfdx is not None: d.append(dfdx[y])

    plt.figure()
    plt.plot(xs, l, label='loss')
    plt.plot(xs, d, label='loss derivative')
    plt.ylim((-20, 20))
    plt.legend()
    plt.show()
except:
    pass
```



Part 3: Back-Propagation and SGD

Problem 3.1

The code below is a partial implementation of a neural network. The only missing code is part of the `backprop` method that implements back-propagation. As usual, this part is replaced by a `pass` statement. Write code in place of this `pass` so that `backprop` computes its output by back-propagation.

Answer (when Completed)

In [431]:
```
class Network:

    def __init__(self, sizes):
        self.layers = []
        for i in range(len(sizes) - 1):
            self.layers.append(FCLayer.ofShape(sizes[i], sizes[i+1]))
            self.layers.append(ReLULayer())
        self.layers.append(SoftmaxLayer(sizes[-1]))
        self.p = None

    def reset(self, r=None):
        for layer in self.layers: layer.reset(r)
        self.p = None

    def getWeights(self):
        return np.concatenate([layer.getWeights() for layer in self.laye
rs])

    def setWeights(self, w):
        for layer in self.layers:
            n = len(layer.getWeights())
            layer.setWeights(w[:n])
            w = w[n:]

    def f(self, x):
        x = tensorize(x)
        for layer in self.layers: x = layer.f(x)
        self.p = x
        return self.p

    def backprop(self, x, y, loss):
        '''
        Performs backpropagation for a single training example
        returns a pair (loss, gradient of loss)
        '''
        L = loss.f(y, self.f(x))  # forward pass
        dldx = loss.dfdx(y)       # gradient of loss wrt x^(k) = p, wher
e k is the last layer. This is a softmax layer.
        gradL = []                # initialize the list of gradients of
 the loss wrt to the weights

        # iterate from the kth layer to the 1st layer
        # Note: the indexing in theory is K ... 1
        # but we the indexing is different in Python
        for k in range(len(self.layers)-1, -1, -1):
            if self.layers[k].dfdw().size != 0: # only calculate gradien
ts if the layer has weights
                dfdw = self.layers[k].dfdw()    # gradient of x^(k) wrt
 to w^(k)
                dldw = np.dot(dldx, dfdw)       # gradient of loss wrt t
o w^(k)
                dldw = np.flip(dldw, 0)         # reverse the order of t
he weights of layer k along axis 0
                gradL.append(dldw)              # append dldw to the lis
t of the gradients of the loss
            dfdx = self.layers[k].dfdx()        # gradient of x^(k) wrt
```

```
 to x^(k-1)
            dldx = np.dot(dldx, dfdx)           # gradient of loss wrt t
o x^(k-1)

        gradL = np.concatenate(gradL)           # concatenate the arrays
 in the list of the gradients of the loss
        gradL = np.flip(gradL, 0)               # reverse the order of t
he list of the gradients of the loss

        return (L, gradL)
```

In [432]:
```
try:
    net = Network((3, 2, 3))
    L, x, y, w = Loss(), [0.1, -0.2, 0.3], 0, range(len(net.getWeights
()))
    net.setWeights(w)

    def Lw(w):
        net.setWeights(w)
        return L.f(y, net.f(x))

    def Jacobian(f, z, delta=1e-5):
        return np.transpose([(f(z + delta * e) - f(z - delta * e)) / (2
* delta)\
                             for e in np.eye(len(z))])

    net.setWeights(w)
    backprop = net.backprop(x, y, L)
    numerical = Jacobian(Lw, w)
    backprop = backprop[1]
    print('Gradient gap for Network:', np.linalg.norm(backprop - numeric
al))
except:
    print("error")
    pass
```

```
Gradient gap for Network: 2.058849846383457e-17
```

Problem 3.2

Write a function with header

```
def sgd(net, loss, T, batch_size=1, max_iter=1, learning_rate_init=1e-3,
    tol=1e-6, n_iter_no_change=10):
```

that uses your `Network.backprop` function to implement stochastic gradient descent without momentum:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - r\nabla\mathbf{w}_t \ .$$

where the gradient is computed on mini-batches.

Place your code in a separate notebook cell before the test code.

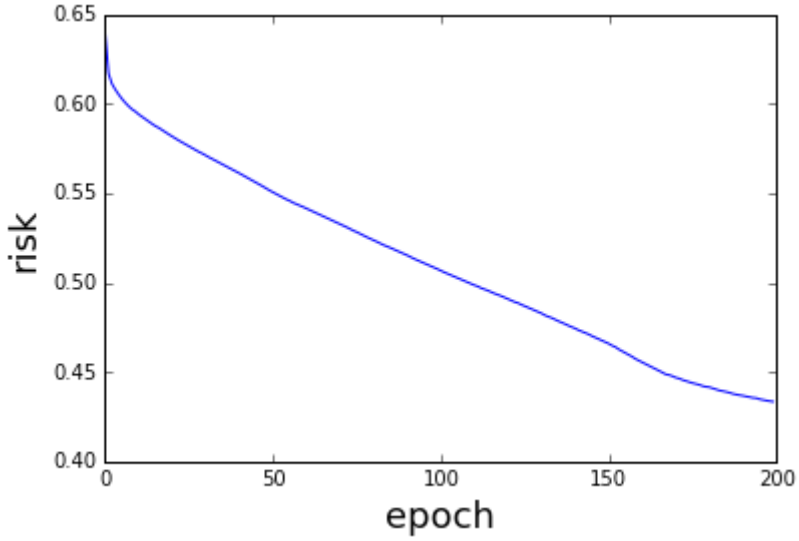
Answer

In [433]:
```
# Place your code here
import random

def sgd(net, loss, T, batch_size=1, max_iter=1, learning_rate_init=1e-3,
 tol=1e-6, n_iter_no_change=10):
    '''
    net = instance of Network class
    loss = instance of Loss class
    T = training set specified with keywords 'x' and 'y'
    batch_size = size of minibatches for stochastic optimizers
    max_iter = Maximum number of iterations. The solver iterates until c
onvergence (determined by ‘tol’) or this number of iterations
    learning_rate_init = The initial learning rate used. It controls the
 step-size in updating the weights
    tol = Tolerance for the optimization. when loss is not improving by
 at least tol, training stops
    n_iter_no_change = Maximum number of epochs to not meet tol improvem
ent
    returns a list or array of the training risk L_T at the end of every
 epoch
    '''
    # generate an array of scrambled indices into T
    n = len(T['y'])                       # the length of the training s
et
    idx = np.argsort(np.random.random(n)) # creates a list of indexes of
 length n in a random order

    r = learning_rate_init                # this learning rate r never c
hanges
    L_T = []                              # training risk with size max_
iter = iterations = epochs
    w = net.getWeights()                  # initialize weights for the n
etwork

    # go through B = batch_size number of training examples and then tak
e a step in the negative gradient
    # After going through the entire training set once = one epoch, appe
nd the risk to L_T

    # goes through max_iter epochs
    for epoch_num in range(max_iter): # iterate through the training set
 'max_iter' = num. of epochs times
        # used to print progress
        if epoch_num % 20 == 0:
            print("Starting epoch", epoch_num)

        tracker = 0                               # keeps track of how ma
ny training examples we've looked at
        epoch_loss = 0                            # initialize epoch loss


        # goes through (n / batch_size) batches
        for i in range(int(n / batch_size)):     # loop over the number
 of batches used per epoch
            gradient_losses = np.zeros((len(w))) # initialize the gradie
nts of loss wrt to weights to 0 for the batch
```

```
            # goes through one batch of size batch_size
            for j in range(tracker, tracker + batch_size):
                index = idx[j]    # value of idx at index j
                x = T['x'][index] # the value of T['x'] at the index of
 the value of idx at index j
                y = T['y'][index] # the value of T['y'] at the index of
 the value of idx at index j
                backprop = net.backprop(x, y, loss) # do backprop, retur
ns tuple
                bp_loss = backprop[0]               # compute loss from
 backprop
                bp_gradients = backprop[1]          # gradient of the lo
ss wrt to weights from backprop
                epoch_loss += bp_loss               # add loss to batch_
loss (not normalized yet)
                gradient_losses += bp_gradients     # add gradient of th
e loss wrt to the weights for the batch
                tracker += 1                        # increase the track
er by one
            w = w - r*gradient_losses               # do gradient descen
t for the batch
            net.setWeights(w)                       # set the neural net
work weights to the new w
        epoch_risk = (1 / n )* epoch_loss

        L_T.append(epoch_risk) # append the risk of the epoch to L_T
        if epoch_num % 20 == 0:
            print("Epoch", epoch_num, "risk:", epoch_risk)

    return L_T
```

In [434]:
```
try:
    def readArray(filename):
        with open(filename, 'r') as file:
            X = np.array([[float(a) for a in line.strip().split()] for l
ine in file])
        return X

    X = readArray('x')
    y = readArray('y').flatten()
    T = {'x': X, 'y': y}

    net = Network((2, 16, 16, 2))
    loss = Loss()
    LT = sgd(net, loss, T, batch_size=20, max_iter=200)
    plt.figure()
    plt.plot(LT)
    plt.xlabel('epoch', fontsize=18)
    plt.ylabel('risk', fontsize=18)
    plt.show()
except:
    print("error")
    pass
```

```
Starting epoch 0
Epoch 0 risk: 0.6399355561502225
Starting epoch 20
Epoch 20 risk: 0.5816260729180347
Starting epoch 40
Epoch 40 risk: 0.5610776491653381
Starting epoch 60
Epoch 60 risk: 0.541395885809135
Starting epoch 80
Epoch 80 risk: 0.5237168321119043
Starting epoch 100
Epoch 100 risk: 0.5066155383428247
Starting epoch 120
Epoch 120 risk: 0.4907855530308446
Starting epoch 140
Epoch 140 risk: 0.4743842993479527
Starting epoch 160
Epoch 160 risk: 0.4553895573378711
Starting epoch 180
Epoch 180 risk: 0.44150291262763963
```

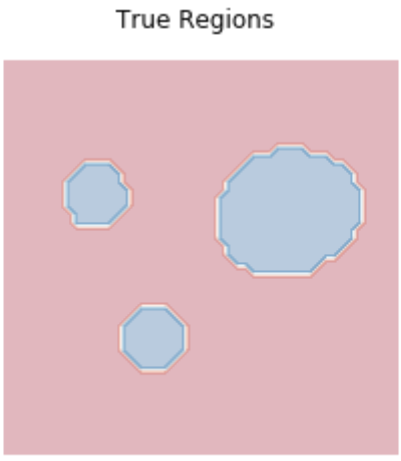


Part 4: Training a Small Network

In [435]:
```
def readArray(filename):
    with open(filename, 'r') as file:
        X = np.array([[float(a) for a in line.strip().split()] for line
in file])
    return X

X = readArray('x')
y = readArray('y').flatten()
XTest = readArray('xTest')
yTest = readArray('yTest').flatten()

T = {'x': X, 'y': y}
S = {'x': XTest, 'y': yTest, 'shape': (50, 50)}
SGrid = {'x0': S['x'][:, 0].reshape(S['shape']),
         'x1': S['x'][:, 1].reshape(S['shape']),
         'y': S['y'].reshape(S['shape'])}

def showRegions(grid, title, s = None):
    if s:
        plt.subplot(2, 3, s)
    else:
        plt.figure()
    plt.contourf(grid['x0'], grid['x1'], grid['y'],
                 cmap=plt.cm.RdBu, alpha=0.3)
    plt.axis('equal')
    plt.axis('off')
    plt.title(title)

plt.figure(1)
showRegions(SGrid, 'True Regions')
```

```
/anaconda3/lib/python3.6/site-packages/numpy/ma/core.py:6442: MaskedArr
ayFutureWarning: In the future the default for ma.minimum.reduce will b
e axis=0, not the current None, to match np.minimum.reduce. Explicitly
pass 0 or None to silence this warning.
  return self.reduce(a)
/anaconda3/lib/python3.6/site-packages/numpy/ma/core.py:6442: MaskedArr
ayFutureWarning: In the future the default for ma.maximum.reduce will b
e axis=0, not the current None, to match np.maximum.reduce. Explicitly
pass 0 or None to silence this warning.
  return self.reduce(a)
```

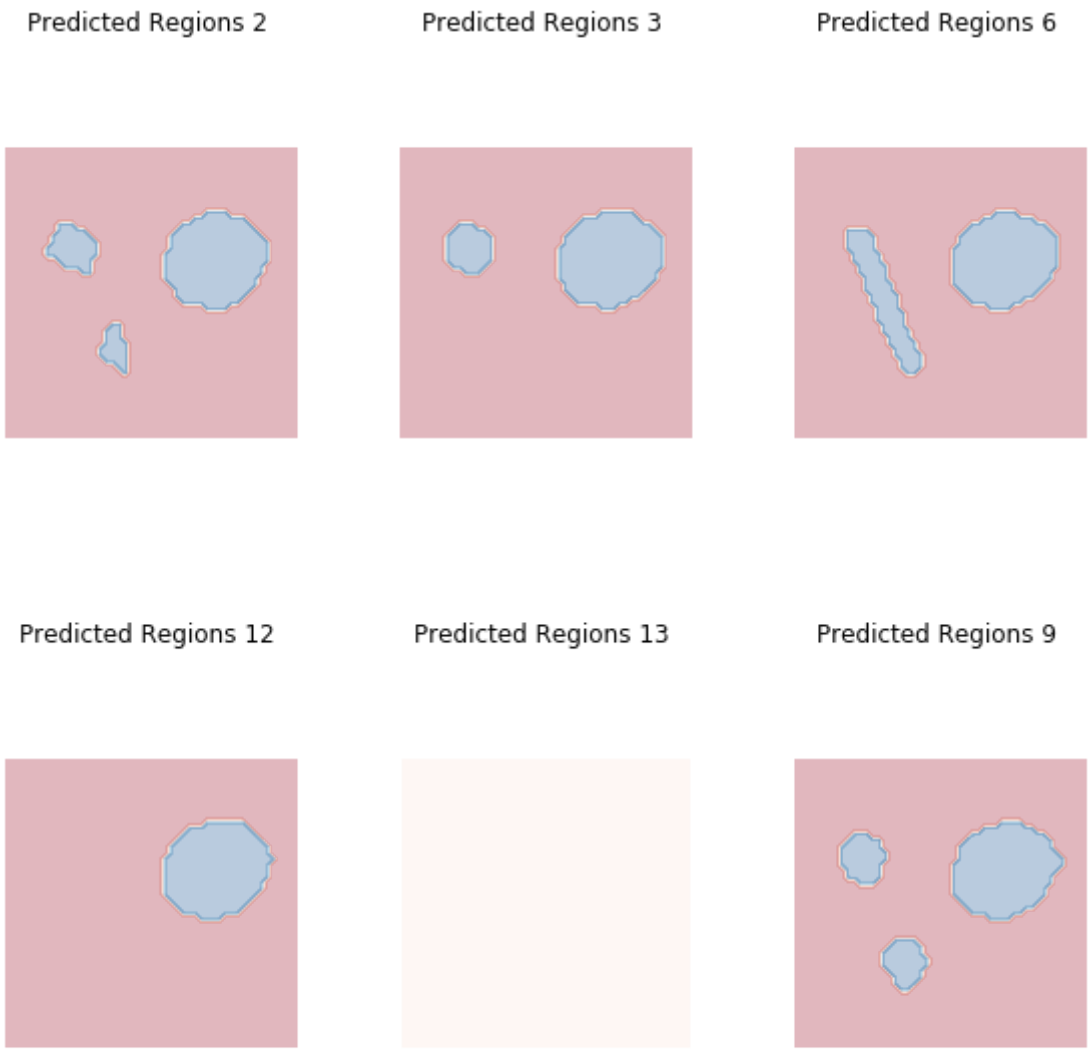
```
<matplotlib.figure.Figure at 0x113535b38>
```



Problem 4.1

In [436]:
```
from sklearn.neural_network import MLPClassifier

layerSizes = (16, 16, 2)
SHatGrid = SGrid
state = (2, 3, 6, 12, 13, 9)
lossCurve = []
plt.figure(figsize=(10,10))
for k in range(len(state)):
    network = MLPClassifier(hidden_layer_sizes = layerSizes, tol=1e-6,
                            max_iter = 10000, random_state=state[k], verbose
=False)
    network.fit(T['x'], T['y'])
    lossCurve.append(network.loss_curve_)
    SHatGrid['y'] = network.predict(S['x']).reshape(S['shape'])
    showRegions(SHatGrid, 'Predicted Regions ' + str(state[k]), k+1)
plt.show()
```

```
/anaconda3/lib/python3.6/site-packages/numpy/ma/core.py:6442: MaskedArr
ayFutureWarning: In the future the default for ma.minimum.reduce will b
e axis=0, not the current None, to match np.minimum.reduce. Explicitly
pass 0 or None to silence this warning.
  return self.reduce(a)
/anaconda3/lib/python3.6/site-packages/numpy/ma/core.py:6442: MaskedArr
ayFutureWarning: In the future the default for ma.maximum.reduce will b
e axis=0, not the current None, to match np.maximum.reduce. Explicitly
pass 0 or None to silence this warning.
  return self.reduce(a)
```



Comment briefly on the importance of initialization when training a neural network. Why is this an issue in terms of advancing our empirical understanding of neural networks? Keep in mind that for larger networks training can take days or weeks, even on a cluster of GPUs.

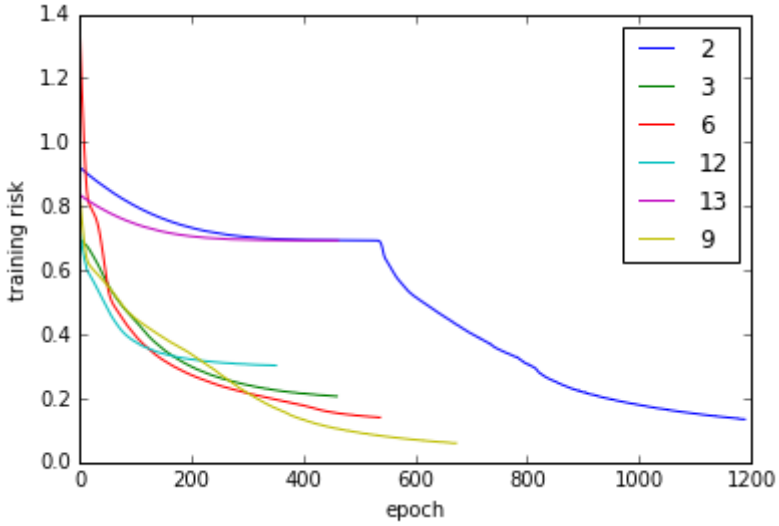
Answer

The six graphs show that neural networks, depending on its specific initialization, will give different results, even if they share some similarities. The reason we get different results from different initializations is that the loss function with respect to its weights is usually not convex. Since the weights are initialized differently in each plot, the results of computing the gradients of the loss and doing gradient descent can result in different local minimums. In addition, it is not clear a priori which initialization will give the lowest training risk or the fastest drop in risk over a given number of epochs.

One issue that arises in terms of advancing our empirical understanding of neural networks is that needing the exact initialization is a problem in the reproducibility of results, for example if the initialization is not published or unclear even to the researcher. In the case where a network takes days or weeks to train, the cost of trying to verify results is extremely high. Moreover, if a network takes days or weeks to train, it may not be possible to compare the network across many different initializations. We need to find another statistically significant factor to compare the accuracy of a model because it may not be possible to have the same network trained on different initializations.

Problem 4.2

In [437]:
```
plt.figure()
for k in range(len(lossCurve)):
    plt.plot(lossCurve[k], label=str(state[k]))
plt.xlabel('epoch')
plt.ylabel('training risk')
plt.legend()
plt.show()
```



In what way are these plots consistent with the decision regions displayed earlier? Comment in particular about the experiment with `random_state=9`.

Answer

The predicted decision boundaries shown do not always correspond to how quickly a network is able to minimize the training risk and find a minimum. Random_state = 9 looks graphically the most similar to the true region plot and has on average the lowest training risk. Random_state = 9 also looks the most similar to the training risk found in 3.2. Interestingly, by the end, random_state = 2 has a decision boundary that looks similar to the true region plot, but has a higher training risk than most networks before epoch 600. Random_state = 3, 6, and 12 all had lower training risks than random_state = 2 before epoch 600, but random_state = 2 eventually reduced its training risk to something comparable to random_state = 9. Generally, it seems that it takes networks with different initializations different amounts of time to converge to a minimum. It seems like adaptive step size training helped achieve better training risk for random_state = 2, which has a noticeable drop starting at around epoch 600. From this plot, we can conclude that it is important to not only look at the risk at a certain epoch, but also the rate at which the training risk is decreasing.