Exercise 2

Deadline: 18.05.2016

Organizational:

- Please send your solutions to `mlhd1516@gmail.com` in a single zip file containing (*well documented*) code, results and plots in the following format:
  Subject line: *"[Ex01] Ghahramani, Hinton, Koller"*
  Attachement: `ex01_ghahramani_hinton_koller.zip`
  (If your group consists of Zoubin Ghahramani, Geoffrey Hinton and Daphne Koller)
- There won't be an exercise group this Friday (13.05). Instead there will be two regular lectures.

1 A Neural Playground(0 Points)

Have a look at `http://playground.tensorflow.org/` and play around with it for a while to get some feeling for neural networks.

This is not an official exercise, so you don't need to hand in anything.

2 Classification Capacity(15 Points)

In this exercise we want to proof that neural networks can classify an arbitrary training set with zero training error. First, we will construct single layer networks that fulfill specific tasks. Second, we combine them to a multilayer network that can classify a given dataset "flawlessly". For each task draw a sketch of the network, specify which activation functions are used and how the weights must be chosen.

2.1 Simple Networks(10 Points)

First, design one-layer neural networks that perform the following tasks:

1. logical OR operation on a binary input vector $z \in \{0,1\}^m$

$$\text{i.e. } z \to f(z) = \begin{cases} 0 & z_i = 0 \ \forall i \\ 1 & \text{otherwise} \end{cases}$$

2. for an arbitrary but fixed binary vector $c \in \{0,1\}^m$ map the input vector $z \in \{0,1\}^m$ to

$$z \to f(z) = \begin{cases} 1 & z = c \\ 0 & \text{otherwise} \end{cases}$$

3. for the dataset X, Y displayed in Figure 1 (with feature vectors X_i and associated classes $Y_i \in \{$ red minus, blue plus, green circle $\}$) map every X_i onto the corners of a hypercube $\{0,1\}^m$ such that each corner contains only one class (you can map one class to multiple corners, but not multiple classes onto one corner). The dimension m of the hypercube is not a priori fixed and may be adjusted to fit the dataset. Draw the decision boundaries of your network in Figure 1 and sketch the mapping of the dataset to the hypercube (you do not need to specifying the input weights for this task). How can this be generalized to arbitrary many input dimensions and arbitrary (non degenerate) class distributions?

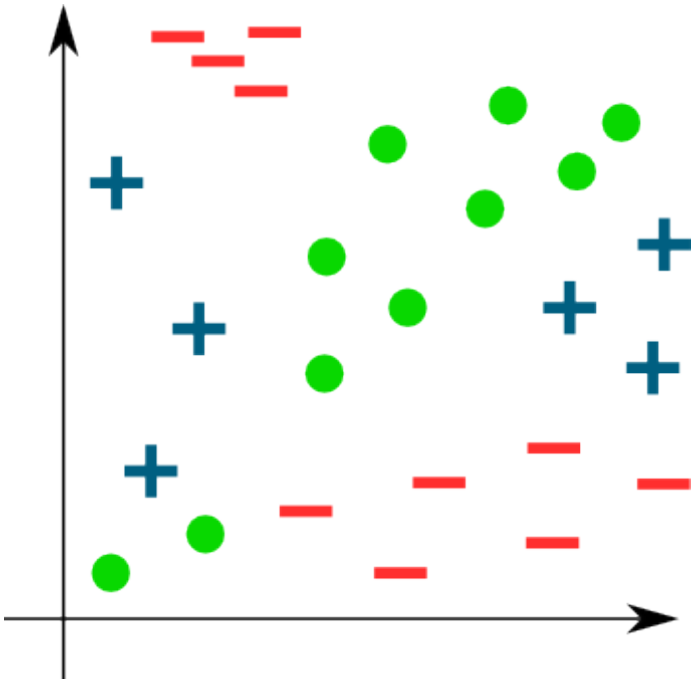

Abbildung 1: Example dataset with three classes (red minus, blue plus, green circle). Sketch the decision boundaries of your network here and draw the matching labeled hypercube.

2.2 3-layer Universal Classifier (5 Points)

Combine the networks from above into a universal classifier which classifies an arbitrary training set with zero training error. Draw a sketch of the network and explain in a few sentences how it works. Do you see any problems with this zero training loss classifier?

3 Theory(15 Points)

3.1 Linear Activation Function(5 Points)

For a feed forward network the output of each layer l is calculated iteratively by

$$Z_0 = \begin{bmatrix} 1 \\ \mathbf{X}^T \end{bmatrix} \tag{1}$$

$$\tilde{Z}_l = B_l Z_{l-1} \tag{2}$$

$$Z_l = \phi_l(\tilde{Z}_l) \tag{3}$$

Proof that if ϕ_l is a linear mapping then any network (with depth $L > 1$) is equivalent to a 1-layer neuronal network.

3.2 Weight Decay(10 Points)

One can show that doing gradient descent on a Loss function that includes a regularization term

$$\text{Loss}(\mathbf{w}) = L_0(\mathbf{w}) + L_{reg}(\mathbf{w}).$$

is equivalent to the weight decay optimization method. For the proof we introduce the vector $\mathbf{w}$ as concatenation of the columns vectors B_{lh}^T. This makes it possible to use vector multiplication inside of the regularisation term.

1. Show that performing gradient descent on the error function with $L_{reg} = \frac{\lambda}{2N}\mathbf{w}^T\mathbf{w}$ leads to the weight decay update rule $\mathbf{w} \to (1-\epsilon)\mathbf{w} - \tau\partial L_0/\partial\mathbf{w}$. To this end,
   - derive the equation for $\partial\text{Loss}/\partial\mathbf{w}$
   - find the modified weight $\mathbf{w}$ update rule
   - identify the terms with the usual update step and express ϵ in terms of λ, N and the learning rate τ.
2. Describe qualitatively how the weight decay methods affect the network.
3. Likewise, derive a weight update rule for $L_{reg} = \frac{\lambda}{2N}|\mathbf{w}|_1$
4. What would happen if the bias weights were not affected by the scaling and only the feature weights decay? Do you think this would be beneficial?

4 Application(10 Points)

In this exercise we want to implement a simple Multi-Layer-Network classifier using `numpy`. The following python code defines an MLP class. You can download the code as a separate .py file from the lecture website. This code was kindly provided by Marius Felix Killinger.

```
class MLP(object):
  def __init__(self, layer_sizes, nin):
    self.layers     = []
    self.last_grads = None
    n_lay           = len(layer_sizes)
    for i in xrange(n_lay-1):
      print "Adding layer (#in %i, #out %i)" % (nin, layer_sizes[i])
      self.layers.append(PerceptronLayer(nin, layer_sizes[i]))
      nin = layer_sizes[i]

    print "Adding layer (#in %i, #out %i)" % (nin, layer_sizes[-1])
    self.layers.append(PerceptronLayer(nin,layer_sizes[-1], act_func='lin'))

    self.softmax = Softmax()
    self.loss = NLL(layer_sizes[-1])

  def forward(self, X):
    bs = X.shape[0]
    X = X.reshape(bs, -1)
    result = X
    for lay in self.layers:
      result = lay.forward(result)
    return result

  def class_prob(self, X):
    out = self.forward(X)
    return self.softmax.forward(out)

  def get_loss(self, X, Y):
    pred = self.class_prob(X)
    loss = self.loss.forward(pred, Y)
    cls  = pred.argmax(axis=1)
    acc  = 1-np.mean(np.equal(cls, Y))
    return loss, acc


  def gradients(self, X, Y):
    class_prob = self.class_prob(X)
    loss    = self.loss.forward(class_prob, Y)

    top_err = self.loss.backward()
    top_err = self.softmax.backward(top_err)
    grads   = []
    for lay in self.layers[::-1]:
```

```
      new_err = lay.backward(top_err)
      grad_W, grad_b = lay.grad(top_err)
      grads.append(grad_b)
      grads.append(grad_W)
      top_err = new_err
    self.last_grads = grads
    return grads[::-1], loss, class_prob

  def update(self, lr):
    for lay in self.layers:
      lay.GD_update(lr)
```

To use the Network you will have to implement the Perceptron, Softmax and Negative Log Likelihood layer class. Therefore, complete the following code:

```
import numpy as np
from sklearn import datasets

class Node(object):
  def __init__(self):
    self.input   = None
  def forward(self, input):
    return input
  def backward(self, top_error):
    return top_error

class Tanh(Node):
  def __init__(self):
    self.input   = None
  def forward(self, input):
    self.input = input
    return ...
  def backward(self, top_error):
    """ return d out / d in """
    return ...


class PerceptronLayer(Node):
  def __init__(self, nin, nout):
    self.nin  = nin
    self.nout = nout
    self.W    = np.random.uniform(low=-np.sqrt(6. / (nin + nout)),
                          high=np.sqrt(6. / (nin + nout)), size=(nin, nout)).astype
                              (np.float32)
    self.b    = np.random.uniform(-1e-8,1e-8,(nout,)).astype(np.float32)

    self.lin  = None   # stores the dot product of w and the last input
    self.act  = None   # stores the mapping of lin with the activation function
    self.input  = None # stores the last input
    self.grad_b = None
    self.grad_W = None
    self.act_func = Tanh()


  def forward(self, input):
    """ (bs, n_in) --> (bs, n_out) """
    self.lin =   ...
    self.act =   ...
    self.input = ...
    return self.act

  def backward(self, top_error):
    """ d out / d in """
    self.act_error = ...
    err = ...
    return err

  def grad(self, top_error):
```

```
    """ d out / d W """
    grad_b = ...
    grad_W = ...
    self.grad_b = grad_b
    self.grad_W = grad_W
    return grad_W, grad_b

  def GD_update(self, lr):
    self.W = self.W - (lr*self.grad_W)
    self.b = self.b - (lr*self.grad_b)


class Softmax(Node):
  def __init__(self):
    self.input   = None

  def forward(self, input):
    self.input = input
    """ return softmax function to input vector"""
    return ...

  def backward(self, top_error):
        """ return the back propagation error """
    return ...


class NLL(object):
  def __init__(self, n_lab):
    self.input   = None
    self.classes = np.arange(n_lab, dtype=np.int)[None,:]

  def forward(self, input, Y):
    self.input = input
    self.n     = Y.shape[0]
    self.active_class = np.equal(self.classes,Y[:,None])
    return ...

  def backward(self):
    return ...
```

Please train the network on the moon shaped sklearn dataset with

```
import numpy as np
from sklearn import datasets

if __name__=="__main__":

  n = 2000

  X, Y = datasets.make_moons(n, noise=0.05)
  X_test   = np.meshgrid(np.linspace(-1,1,20), np.linspace(-1,1,20))
  X_test   = np.vstack((X_test[0].flatten(), X_test[1].flatten())).T
  X -= X.min(axis=0)
  X /= X.max(axis=0)
  X  = (X - 0.5) * 2

  nin = 2
  bs  = 200
  lr  = 0.05

  trace = dict(X=X, Y=Y, W1=[], b1=[], a1=[], W2=[], b2=[], a2=[], dec=[])
  nn = MLP([3,2], nin)

  pos = 0
  perm = np.random.permutation(n)

  nn = MLP([30,30,10], nin)
  for i in xrange(10000):
```

```
    grads, loss, pred = nn.gradients(X[perm[pos:pos+bs]], Y[perm[pos:pos+bs]])
    nn.update(lr)
    if i%1000==0:
      valid_loss, valid_err = nn.get_loss(X, Y)
      print "Loss:",loss,"Valid Loss:",valid_loss,"Valid Error:",valid_err
```

Compare the validation errors for the networks

```
MLP([2,2,2], nin)
MLP([3,3,2], nin)
MLP([5,5,2], nin)
MLP([30,30,2], nin)
```

Regulations

Please hand in the python code, figures and explanations (describing clearly which belongs to which). Non-trivial sections of your code should be explained with short comments, and variables should have self-explanatory names. Plots should have informative axis labels, legends and captions. Please enclose all results into a single .pdf document and hand in the .py files that created the results. Please email the solutions to `mlhd1516@gmail.com` before the deadline. You may hand in the exercises in teams of maximally three people, which must be clearly named on the solution sheet (one email is sufficient). Discussions between different teams about the exercises are encouraged, but the code must not be copied verbatim (the same holds for any implementations which may be available on the WWW). Please respect particularly this rule, otherwise we cannot give you a passing grade. If you have 50% or more points in the end of the semester you will be allowed to take the exam.