In [1]:

```
import numpy as np
```

Module is an abstract class which defines fundamental methods necessary for a training a neural network. You do not need to change anything here, just read the comments.

In [2]:

```
class Module(object):
    """
    Basically, you can think of a module as of a something (black box)
    which can process `input` data and produce `ouput` data.
    This is like applying a function which is called `forward`:

        output = module.forward(input)

    The module should be able to perform a backward pass: to differentiate the `f
    More, it should be able to differentiate it if is a part of chain (chain rule
    The latter implies there is a gr adient from previous step of a chain rule.

        gradInput = module.backward(input, gradOutput)
    """
    def __init__ (self):
        self.output = None
        self.gradInput = None
        self.training = True

    def forward(self, input):
        """
        Takes an input object, and computes the corresponding output of the modul
        """
        return self.updateOutput(input)

    def backward(self, input, gradOutput):
        """
        Performs a backpropagation step through the module, with respect to the g

        This includes
         - computing a gradient w.r.t. `input` (is needed for further backprop),
         - computing a gradient w.r.t. parameters (to update parameters while opt
        """
        self.updateGradInput(input, gradOutput)
        self.accGradParameters(input, gradOutput)
        return self.gradInput


    def updateOutput(self, input):
        """
        Computes the output using the current parameter set of the class and inpu
        This function returns the result which is stored in the `output` field.

        Make sure to both store the data in `output` field and return it.
        """
```

```python
        # The easiest case:

        # self.output = input
        # return self.output

        pass

    def updateGradInput(self, input, gradOutput):
        """
        Computing the gradient of the module with respect to its own input.
        This is returned in `gradInput`. Also, the `gradInput` state variable is

        The shape of `gradInput` is always the same as the shape of `input`.

        Make sure to both store the gradients in `gradInput` field and return it.
        """

        # The easiest case:

        # self.gradInput = gradOutput
        # return self.gradInput

        pass

    def accGradParameters(self, input, gradOutput):
        """
        Computing the gradient of the module with respect to its own parameters.
        No need to override if module has no parameters (e.g. ReLU).
        """
        pass

    def zeroGradParameters(self):
        """
        Zeroes `gradParams` variable if the module has params.
        """
        pass

    def getParameters(self):
        """
        Returns a list with its parameters.
        If the module does not have parameters return empty list.
        """
        return []

    def getGradParameters(self):
        """
        Returns a list with gradients with respect to its parameters.
        If the module does not have parameters return empty list.
        """
        return []

    def training(self):
        """
        Sets training mode for the module.
        Training and testing behaviour differs for Dropout, BatchNorm.
        """
        self.training = True
```

```python
    def evaluate(self):
        """
        Sets evaluation mode for the module.
        Training and testing behaviour differs for Dropout, BatchNorm.
        """
        self.training = False

    def __repr__(self):
        """
        Pretty printing. Should be overrided in every module if you want
        to have readable description.
        """
        return "Module"
```

Sequential container

Define a forward and backward pass procedures.

In [3]:

```python
class Sequential(Module):
    """
         This class implements a container, which processes `input` data sequenti

         `input` is processed by each module (layer) in self.modules consecutivel
         The resulting array is called `output`.
    """

    def __init__ (self):
        super().__init__()
        self.modules = []

    def add(self, module):
        """
        Adds a module to the container.
        """
        self.modules.append(module)

    def updateOutput(self, input):
        """
        Basic workflow of FORWARD PASS:

            y_0    = module[0].forward(input)
            y_1    = module[1].forward(y_0)
            ...
            output = module[n-1].forward(y_{n-2})


        Just write a little loop.
        """

        self.output = [input]
        for module in self.modules:
            self.output.append(module.updateOutput(self.output[-1]))
```

```python
        return self.output[-1]

    def backward(self, input, gradOutput):
        """
        Workflow of BACKWARD PASS:

            g_{n-1} = module[n-1].backward(y_{n-2}, gradOutput)
            g_{n-2} = module[n-2].backward(y_{n-3}, g_{n-1})
            ...
            g_1 = module[1].backward(y_0, g_2)
            gradInput = module[0].backward(input, g_1)


        !!!

        To ech module you need to provide the input, module saw while forward pas
        it is used while computing gradients.
        Make sure that the input for `i-th` layer the output of `module[i]` (just
        and NOT `input` to this Sequential module.

        !!!

        """
        self.gradInput = gradOutput
        for module, current_input in zip(reversed(self.modules), reversed(self.ou
            self.gradInput = module.backward(current_input, self.gradInput)
        return self.gradInput

    def zeroGradParameters(self):
        for module in self.modules:
            module.zeroGradParameters()

    def getParameters(self):
        """
        Should gather all parameters in a list.
        """
        return [x.getParameters() for x in self.modules]

    def getGradParameters(self):
        """
        Should gather all gradients w.r.t parameters in a list.
        """
        return [x.getGradParameters() for x in self.modules]

    def __repr__(self):
        string = "".join([str(x) + '\n' for x in self.modules])
        return string

    def __getitem__(self,x):
        return self.modules.__getitem__(x)
```

Layers

- input: **`batch_size x n_feats1`**

- output: **batch_size x n_feats2**

In [12]:

```
class Linear(Module):
    """
    A module which applies a linear transformation
    A common name is fully-connected layer, InnerProductLayer in caffe.

    The module should work with 2D input of shape (n_samples, n_feature).
    """
    def __init__(self, n_in, n_out):
        super().__init__()

        # This is a nice initialization
        stdv = 1.0 / np.sqrt(n_in)
        self.W = np.random.uniform(-stdv, stdv, size = (n_out, n_in))
        self.b = np.random.uniform(-stdv, stdv, size = n_out)

        self.gradW = np.zeros_like(self.W)
        self.gradb = np.zeros_like(self.b)

    def updateOutput(self, input):
        self.output = np.dot(input, self.W.T) + self.b
        return self.output

    def updateGradInput(self, input, gradOutput):
        self.gradInput = np.dot(gradOutput, self.W)
        return self.gradInput

    def accGradParameters(self, input, gradOutput):
        self.gradW = np.dot(gradOutput.T, input)
        self.gradb = np.sum(gradOutput, axis=0)

    def zeroGradParameters(self):
        self.gradW.fill(0)
        self.gradb.fill(0)

    def getParameters(self):
        return [self.W, self.b]

    def getGradParameters(self):
        return [self.gradW, self.gradb]

    def __repr__(self):
        s = self.W.shape
        q = 'Linear {} -> {}'.format(s[1],s[0])
        return q
```

This one is probably the hardest but as others only takes 5 lines of code in total.

- input: **batch_size x n_feats**
- output: **batch_size x n_feats**

In [5]:

```
class SoftMax(Module):
    def __init__(self):
         super().__init__()

    def updateOutput(self, input):
        # start with normalization for numerical stability
        self.output = np.subtract(input, input.max(axis=1, keepdims=True))
        self.output = (np.exp(self.output).T / np.sum(np.exp(self.output), axis=1
        return self.output

    def updateGradInput(self, input, gradOutput):
        butch_size, n_feats = self.output.shape
        matrix_1 = self.output.reshape(butch_size, n_feats, -1)
        matrix_2 = self.output.reshape(butch_size, -1, n_feats)
        self.gradInput = np.multiply(
            gradOutput.reshape(butch_size, -1, n_feats),
            np.subtract(np.multiply(np.eye(n_feats), matrix_1), np.multiply(matri
        ).sum(axis=2)
        return self.gradInput

    def __repr__(self):
        return "SoftMax"
```

Implement **dropout** (https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf). The idea and implementation is really simple: just multimply the input by $Bernoulli(p)$ mask.

This is a very cool regularizer. In fact, when you see your net is overfitting try to add more dropout.

While training (`self.training == True`) it should sample a mask on each iteration (for every batch). When testing this module should implement identity transform i.e. `self.output = input`.

- input: **`batch_size x n_feats`**
- output: **`batch_size x n_feats`**

In [6]:

```
class Dropout(Module):
    def __init__(self, p=0.5):
        super().__init__()

        self.p = p
        self.mask = None

    def updateOutput(self, input):
        self.output = input
        if self.training:
            self.mask = np.random.rand(*input.shape) < self.p
            self.output *= self.mask
        return self.output

    def updateGradInput(self, input, gradOutput):
        self.gradInput = gradOutput * self.mask
        return self.gradInput

    def __repr__(self):
        return "Dropout"
```

Activation functions

Here's the complete example for the **Rectified Linear Unit** non-linearity (aka **ReLU**):

In [7]:

```
class ReLU(Module):
    def __init__(self):
         super().__init__()

    def updateOutput(self, input):
        self.output = np.maximum(input, 0)
        return self.output

    def updateGradInput(self, input, gradOutput):
        self.gradInput = np.multiply(gradOutput, input > 0)
        return self.gradInput

    def __repr__(self):
        return "ReLU"
```

Implement **Leaky Rectified Linear Unit** (http://en.wikipedia.org/wiki%2FRectifier_%28neural_networks%29%23Leaky_ReLUs). Expriment with slope.

In [8]:

```
class LeakyReLU(Module):
    def __init__(self, slope = 0.03):
        super().__init__()
        self.slope = slope

    def updateOutput(self, input):
        self.output = np.maximum(input, self.slope * input)
        return  self.output

    def updateGradInput(self, input, gradOutput):
        self.gradInput = np.multiply(gradOutput, (np.sign(input) - 1) / 2 * (1 -
        return self.gradInput

    def __repr__(self):
        return "LeakyReLU"
```

Criterions

Criterions are used to score the models answers.

In [9]:

```
class Criterion(object):
    def __init__ (self):
        self.output = None
        self.gradInput = None

    def forward(self, input, target):
        """
            Given an input and a target, compute the loss function
            associated to the criterion and return the result.

            For consistency this function should not be overrided,
            all the code goes in `updateOutput`.
        """
        return self.updateOutput(input, target)

    def backward(self, input, target):
        """
            Given an input and a target, compute the gradients of the loss functi
            associated to the criterion and return the result.

            For consistency this function should not be overrided,
            all the code goes in `updateGradInput`.
        """
        return self.updateGradInput(input, target)

    def updateOutput(self, input, target):
        """
        Function to override.
        """
        return self.output

    def updateGradInput(self, input, target):
        """
        Function to override.
        """
        return self.gradInput

    def __repr__(self):
        """
        Pretty printing. Should be overrided in every module if you want
        to have readable description.
        """
        return "Criterion"
```

The **MSECriterion**, which is basic L2 norm usually used for regression, is implemented here for you.

In [10]:

```
class MSECriterion(Criterion):
    def __init__(self):
        super().__init__()

    def updateOutput(self, input, target):
        self.output = np.sum(np.power(input - target,2)) / input.shape[0]
        return self.output

    def updateGradInput(self, input, target):
        self.gradInput  = (input - target) * 2 / input.shape[0]
        return self.gradInput

    def __repr__(self):
        return "MSECriterion"
```

You task is to implement the **ClassNLLCriterion**. It should implement [multiclass log loss (http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss)](http://scikit-learn.org/stable/modules/model_evaluation.html#log-loss). Nevertheless there is a sum over `y` (target) in that formula, remember that targets are one-hot encoded. This fact simplifies the computations a lot. Note, that criterions are the only places, where you divide by batch size.

In [11]:

```
class ClassNLLCriterion(Criterion):
    def __init__(self):
        super().__init__()

    def updateOutput(self, input, target):
        # Use this trick to avoid numerical errors
        eps = 1e-15
        input_clamp = np.clip(input, eps, 1 - eps)
        self.output = -np.mean(np.log(input_clamp[range(np.shape(input)[0]), targ
        return self.output

    def updateGradInput(self, input, target):
        # Use this trick to avoid numerical errors
        input_clamp = np.maximum(1e-15, np.minimum(input, 1 - 1e-15))
        self.gradInput = np.zeros(np.shape(input))
        self.gradInput[range(np.shape(input)[0]), target] += (
            -1 / (np.shape(input)[0] * input_clamp[range(np.shape(input)[0]), tar
        )
        return self.gradInput

    def __repr__(self):
        return "ClassNLLCriterion"
```