Exercise 6

1) Neural Network Classifier from Scratch (10p.)

In this exercise we will implement a small neural network from scratch, i.e., only using numpy. This is nothing you would do "in real life" but it is a good exercise to deepen understanding.

The network will consist of an arbitrary number of hidden layers with ReLU activation, a sigmoid output layer (as we are doing binary classification) and we will train it using the binary cross entropy (negative bernoulli likelihood). Ok, so lets start by importing and loading what we need.

In [25]:
```
import numpy as np
import matplotlib.pyplot as plt
from typing import List, Tuple

# Load our two moons (I promise we will get a new dataset in the next exercise)
train_data = dict(np.load("two_moons.npz", allow_pickle=True))
test_data = dict(np.load("two_moons_test.npz", allow_pickle=True))
# we need to reshape our labels so that they are [N, 1] and not [N] anymore
train_samples, train_labels = train_data["samples"], train_data["labels"][:, Non
test_samples, test_labels = test_data["samples"], test_data["labels"][:, None]
```

1.1.) Auxillary Functions (3 p.)

We start with implementing some auxillary functions we are going to need later. The sigmoid and relu activation functions, the binary cross entropy loss as well as their derviatives.

The binary cross entropy loss is given as $ - \dfrac{1}{N} \sum_{i=1}^N (y_i \log (p_i) + (1 - y_i) \log (1 - p_i)) $ where y_i denotes the ground truth label and p_i the network prediction for sample i.

Hint all derivatives where derived/implemented during the lecture or previous exercise - so feel free to borrow them from there.

In [26]:
```
a = np.array([-2,-1,0,1,2])
np.maximum([0],a)
```

Out[26]:
```
array([0, 0, 0, 1, 2])
```

In [27]:
```
def relu(x: np.ndarray) -> np.ndarray:
    """
    elementwise relu activation function
    :param x: input to function [shape: arbitrary]
    :return : relu(x) [shape: same as x]
    """
    ### DONE #########################
    return np.maximum([0],x)
    ##################################


def d_relu(x: np.ndarray) -> np.ndarray:
```

```
    """
    elementwise gradient of relu activation function
    :param x: input to function [shape: arbitrary]
    :return : d relu(x) / dx [shape: same as x]
    """
    ### DONE #########################
    x[x<=0] = 0
    x[x>0] = 1
    return x
    ##################################


def sigmoid(x: np.ndarray) -> np.ndarray:
    """
    elementwise sigmoid activation function
    :param x: input to function [shape: arbitrary]
    :return : d sigmoid(x) /dx [shape: same as x]
    """
    ### DONE #########################
    return (1/(1+np.exp(-x)))
    ##################################


def d_sigmoid(x: np.ndarray) -> np.ndarray:
    """
    elementwise sigmoid activation function
    :param x: input to function [shape: arbitrary]
    :return : sigmoid(x) [shape: same as x]
    """
    # --- Is this correct?
    # --- Yes, seems so (https://towardsdatascience.com/derivative-of-the-sigmoi
    ### DONE #########################
    return sigmoid(x) * (1-sigmoid(x))
    ##################################


def binary_cross_entropy(predictions: np.ndarray, labels: np.ndarray) -> float:
    """
    binary cross entropy loss (negative bernoulli ll)
    :param predictions: predictions by model (shape [N])
    :param labels: class labels corresponding to train samples, (shape: [N])
    :return binary cross entropy
    """
    ### DONE #########################
    N = labels.shape[0]
    loss = np.zeros((N,1))
    loss[labels == 1] = - np.log(predictions[labels == 1])
    loss[labels == 0] = - np.log(1-predictions[labels == 0])

    # optional: divide by number of predictions/ labels
    return 1/N * np.sum(loss)
    ##################################


def d_binary_cross_entropy(predictions: np.ndarray, labels: np.ndarray) -> np.nd
    """
    gradient of the binary cross entropy loss
    :param predictions: predictions by model (shape [N])
    :param labels: class labels corresponding to train samples, (shape [N])
    :return gradient of binary cross entropy, w.r.t. the predictions (shape [N])
    """
```

```
    ### DONE ##########################
    N = predictions.shape[0]
    gradient = np.zeros(N)
    gradient = (labels / predictions - ((1-labels) / (1-predictions)))

    # optional: divide by number of predictions/ labels
    return -1/N * gradient
    ###################################
```

General Setup & Intialization

Next we are going to set up the Neural Network. We will represent it as a list of weight matrices and a list of bias vectors. Each list has one entry for each layer.

In [28]:
```
def init_weights(neurons_per_hidden_layer: List[int], input_dim: int, output_dim
        -> Tuple[List[np.ndarray], List[np.ndarray]]:
    """
    :param neurons_per_hidden_layer: list of numbers, indicating the number of n
    :param input_dim: input dimension of the network
    :param output_dim: output dimension of the network
    :param seed: seed for random number generator
    :return list of weights and biases as specified by dimensions and hidden lay
    """
    # seed random number generator
    rng = np.random.RandomState(seed)
    scale_factor = 1.0
    prev_n = input_dim
    weights = []
    biases = []

    # hidden layers
    for n in neurons_per_hidden_layer:
        # initialize weights with gaussian noise
        weights.append(scale_factor * rng.normal(size=[prev_n, n]))
        # initialize bias with zeros
        biases.append(np.zeros([1, n]))
        prev_n = n

    # output layer
    weights.append(scale_factor * rng.normal(size=[prev_n, output_dim]))
    biases.append(np.zeros([1, output_dim]))

    return weights, biases
```

NOTE As NNs are non-convex, initialization plays a very important role in NN training and there is a lot of work into how to initialize them properly - this here is not a very good initialization, but sufficient for our small example.

1.2) Forward Pass (3 p.)

Next step is the forward pass, i.e., propagate a batch of samples through the network to get the final prediciton. But that's not all - to compute the gradietns later we also need to store all necessary quantities, here those are:

- The input to every layer (here called h's)

- The "pre-activation" of every layer, i.e., the qantity that is fed into the non-linearity (here called z's)

In [29]:
```
def forward_pass(x: np.ndarray, weights: List[np.ndarray], biases: List[np.ndarr
        -> Tuple[np.ndarray, List[np.ndarray], List[np.ndarray]]:
    """
    propagate input through network
    :param x: input: (shape, [N x input_dim])
    :param weights: weight parameters of the layers
    :param biases: bias parameters of the layers
    :return: - Predictions of the network (shape, [N x out_put_dim])
             - hs: output of each layer (input + all hidden layers) (length: len
             - zs: preactivation of each layer (all hidden layers + output) (len
    """

    hs = []  # list to store all inputs
    zs = []  # list to store all pre-activations

    # input to first hidden layer is just the input to the network
    h = x
    hs.append(h)

    ### DONE #########################
    # pass "h" to all hidden layers
    # record all inputs and pre-activations in the lists
    for layer, w in enumerate(weights):
        b = biases[layer]

        # weight shape: [2, 64], [64,64], [64, 1]
        # data shape:   [100,2], [64,64], [64,64]
        # bias shape:   [1, 64], [1, 64], [1, 1]
        print("Layer", layer, "Weights", w.shape, "Data", h.shape)
        s = np.sum(w.T @ h.T,axis=0)
        z = (s + b.T).T
        print(s.shape, z.shape, b.shape)
        zs.append(z)

        h = relu(z)
        print("H:", h.shape)
        hs.append(h)

    ##################################
    # has to have same shape as labels, i.e. [N,1]
    y = sigmoid(z)   # z denotes the pre-activation of the output layer here. Fe

    return y, hs[:-1], zs
```

1.3) Backward Pass (4 p.)

For training by gradient descent we need - well - gradients. Those are computed using backpropagation during the so called "backward pass". We will use the chain rule to propagate the gradient back through the network and at every layer, compute the gradients for the weights and biases at that layer. The initial gradient is given by the gradient of the loss function w.r.t. the network output.

In [30]:
```
def backward_pass(loss_grad: np.ndarray,
                  hs: List[np.ndarray], zs: List[np.ndarray],
```

```
                  weights: List[np.ndarray], biases: List[np.ndarray]) -> \
    Tuple[List[np.ndarray], List[np.ndarray]]:
    """
    propagate gradient backwards through network
    :param loss_grad: gradient of the loss function w.r.t. the network output (s
    :param hs: values of all hidden layers during forward pass
    :param zs: values of all preactivations during forward pass
    :param weights: weight paramameters of the layers
    :param biases: bias parameters of the layers
    :return: d_weights: List of weight gradients - one entry with same shape for
             d_biases: List of bias gradients - one entry with same shape for ea
    """

    # return gradients as lists - we pre-initialize the lists as we iterate back
    d_weights = [None] * len(weights)
    d_biases  = [None] * len(biases)

    ### TODO #########################
    depth = len(weights)
    hs_grad = [None] * (depth+1)
    zs_grad = [None] * depth

    print(f'Weights: {depth} with shapes: {[w.shape for w in weights]}')
    print(f'Biases: {depth} with shapes: {[b.shape for b in biases ]}')

    hs_grad[depth] = loss_grad # np.sum(loss_grad).reshape(-1,1)

    # backwards trough network -- [2, 1, 0] for 3 layers
    for layer in range(depth-1, -1, -1):

        print("values:", hs_grad[layer+1].shape, "grad preact:", d_sigmoid(zs[la
        zs_grad[layer] = hs_grad[layer+1] * d_relu(zs[layer])
        print("preactivation grad:", zs_grad[layer].shape)
        d_weights[layer] = np.outer(zs_grad[layer],hs[layer])
        d_biases[layer] = zs_grad[layer]

        # update hs_grad for next round -- like giving the loss to the subnetwor
        print("Weights:", weights[layer].shape, "Preact Grad:", zs_grad[layer].s
        hs_grad[layer] = (weights[layer] @ zs_grad[layer].T).T

    print(f'Weight Grad: {depth} with shapes: {[w.shape for w in d_weights]}')
    print(f'Bias Grad: {depth} with shapes: {[b.shape for b in d_biases]}')

    ##################################

    return d_weights, d_biases
```

Tying Everything Together

Finally we can tie everything together and train our network.

In [31]:

```
N = train_samples.shape[0]

# hyper parameters
layers = [64, 64]
learning_rate = 1e-2

# init model
weights, biases = init_weights(layers, input_dim=2, output_dim=1, seed=42)
```

```python
#book keeping
train_losses = []
test_losses = []

# Here we work with a simple gradient descent implementation, using the whole da
# You can modify it to stochastic gradient descent or a batch gradient descent p
for i in range(1000):

    # predict network outputs and record intermediate quantities using the forwa
    prediction, hs, zs = forward_pass(train_samples, weights, biases)
    # print("Labels:", train_labels.shape, "vs. predictions", prediction.shape)
    train_losses.append(binary_cross_entropy(prediction, train_labels))

    # compute gradients
    loss_grad = d_binary_cross_entropy(prediction, train_labels)
    w_grads, b_grads = backward_pass(loss_grad, hs, zs, weights, biases)

    # apply gradients
    for i in range(len(w_grads)):
        weights[i] -= learning_rate * w_grads[i]
        biases[i] -= learning_rate * b_grads[i]

    test_losses.append(binary_cross_entropy(forward_pass(test_samples, weights,

# plotting
plt.title("Loss")
plt.semilogy(train_losses)
plt.semilogy(test_losses)
plt.legend(["Train Loss", "Test Loss"])

def plt_solution(samples, labels):
    plt_range = np.arange(-1.5, 2.5, 0.01)
    plt_grid = np.stack(np.meshgrid(plt_range, plt_range), axis=-1)
    plt_grid_shape = plt_grid.shape[:2]
    pred_grid = np.reshape(forward_pass(plt_grid, weights, biases)[0], plt_grid_
    plt.contour(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=[0.5], col
    plt.contourf(plt_grid[..., 0], plt_grid[..., 1], pred_grid, levels=10)
    plt.colorbar()
    s0 = plt.scatter(x=samples[labels[:, 0] == 0, 0], y=samples[labels[:, 0] ==
                     label="c=0", c="blue")
    s1 = plt.scatter(x=samples[labels[:, 0] == 1, 0], y=samples[labels[:, 0] ==
                     label="c=1", c="orange")
    plt.legend([s0, s1], ["c0", "c1"])
    plt.xlim(-1.5, 2.5)
    plt.ylim(-1.5, 1.5)


plt.figure()
plt.title("Trained Network - with train samples")
plt_solution(train_samples, train_labels)

plt.figure()
plt.title("Trained Network - with test samples")
plt_solution(test_samples, test_labels)
plt.show()
```

```
Layer 0 Weights (2, 64) Data (100, 2)
(100,) (100, 64) (1, 64)
H: (100, 64)
```

```
Layer 1 Weights (64, 64) Data (100, 64)
(100,) (100, 64) (1, 64)
H: (100, 64)
Layer 2 Weights (64, 1) Data (100, 64)
(100,) (100, 1) (1, 1)
H: (100, 1)
Weights: 3 with shapes: [(2, 64), (64, 64), (64, 1)]
Biases: 3 with shapes: [(1, 64), (1, 64), (1, 1)]
values: (100, 1) grad preact: (100, 1)
preactivation grad: (100, 1)
Weights: (64, 1) Preact Grad: (100, 1)

values: (100, 64) grad preact: (100, 64)
preactivation grad: (100, 64)
/home/vincent/.local/lib/python3.6/site-packages/ipykernel_launcher.py:32: Runti
meWarning: overflow encountered in exp
/home/vincent/.local/lib/python3.6/site-packages/ipykernel_launcher.py:77: Runti
meWarning: invalid value encountered in true_divide
Weights: (64, 64) Preact Grad: (100, 64)

values: (100, 64) grad preact: (100, 64)
preactivation grad: (100, 64)
Weights: (2, 64) Preact Grad: (100, 64)

Weight Grad: 3 with shapes: [(6400, 200), (6400, 6400), (100, 6400)]
Bias Grad: 3 with shapes: [(100, 64), (100, 64), (100, 1)]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-31-84f03f73ba91> in <module>
     28     # apply gradients
     29     for i in range(len(w_grads)):
---> 30         weights[i] -= learning_rate * w_grads[i]
     31         biases[i] -= learning_rate * b_grads[i]
     32 

ValueError: operands could not be broadcast together with shapes (2,64) (6400,20
0) (2,64)
```

In [ ]:

```
a = np.ones((100,64))
np.sum(a,axis=0).reshape(.shape
```

2.) MNIST Classifier with PyTorch (10 p.)

Modern deep learning approaches are mostly implemented using special libraries, providing functionality such as automatic differentiation, common SGD Optimiziers, easy usage of GPUs and so on. We will use PyTorch, at the moment the, arguably, most common framework (for research).

Getting Started

You can find a documentation of the PyTorch API here https://pytorch.org/docs/stable/torch.html# . Don't worry if it seems a lot, we will point out the relevant bits during the exercise as we go along

Installation You can find installation instructions here https://pytorch.org/ . Take the most recent stable version (1.7.X). We won't use GPUs here so you can take the cuda-free installation.

We also don't need torchvision nor torchaudio so those don't need to be installed.

Data We finally use a new dataset. The classical MNIST Handwritten Digit Classification set. It consists of grayscale images of size 28x28 of handwritten digits. Let's load it and visualize some of the images. We also do some preprocessing.

In [ ]:

```
import torch
import torch.nn as nn
import numpy as np

data_dict = dict(np.load("mnist.npz"))

# prepare data:
# - images are casted to float 32 (from uint8) mapped in interval (0,1) and a "f
#   torch uses "NCHW"-layout for 2d convolutions. (i.e., a batch of images is re
#   where the first axis (N) is the batch dimension, the second the (color) **C*
#   and a **W**idth axis). As we have grayscale images there is only 1 color cha
# - targets are mapped to one hot encoding - torch does that for us


with torch.no_grad():
    # YTA: We don't need torch to calculate gradients here since we only want to
    # YTA: reshape() because of: https://pytorch.org/tutorials/beginner/blitz/ne
    train_samples = torch.from_numpy(data_dict["train_samples"].astype(np.float3
    train_labels = torch.nn.functional.one_hot(torch.from_numpy(data_dict["train
    test_samples = torch.from_numpy(data_dict["test_samples"].astype(np.float32)
    test_labels = torch.nn.functional.one_hot(torch.from_numpy(data_dict["test_l


# plot first 25 images in train setp
plt.figure(figsize=(25, 1))
for i in range(25):
    plt.subplot(1, 25, i + 1)
    # drop channel axis for plotting
    plt.imshow(train_samples[i, 0], cmap="gray", interpolation="none")
    plt.gca().axis("off")
```

2.1) Specifiying Networks (4 p.)

The first step in training a neural network is specifying its architecture. Here we will actually build two networks

- classifier_fc: A classifier consisting only of fully connected layers
- classifier_conv: A classifier combining, convolutional layers, pooling and fully connected layers

In the torch API under torch.nn you can find everything you need. Take a look at the classes "Linear", "ReLU", "Softmax", "Conv2d", "MaxPool2d" and "Sequential"

In [ ]:

```
layers_fc = [
    torch.nn.Flatten(), # Flatten image into vector
    ## TODO ##
    # Hidden Layer 1: 256 neurons, Relu activation
    nn.Linear(28**2,256),
    nn.ReLU(),
```

```
    # Hidden Layer 2: 128 neurons, Relu activation
    nn.Linear(256,128),
    nn.ReLU(),

    # Outputlayer: 10 neurons (one for each class), softmax activation
    nn.Linear(128,10),
    nn.Softmax()
    ###########
]
classifier_fc = torch.nn.Sequential(
    *layers_fc # unpack layers
)

# YTA: THIS IS NOT DONE
layers_conv = [
    # Conv Layer 1: 8 filters of 3x3 size, ReLU, Max Pool with size 2x2 and stri
    nn.Conv2d(8, 33, 3, stride=2), # 1 auf 8
    nn.ReLU(),
    nn.MaxPool2d(22,2),

    # Conv Layer 2: 16 filters of 3x3 size, ReLU, Max Pool with size 2x2 and str
    nn.Conv2d(16, 33, 3, stride=2), # 8 auf 16
    nn.ReLU(),
    nn.MaxPool2d(22,2),

    # Flatten
    nn.Flatten(),

    # Fully Connected Layer 1: 64 Neurons, ReLU
    nn.Linear(400, 64),
    nn.ReLU(),

    # Outputlayer: 10 neurons (one for each class), softmax activation
    nn.Linear(64,10),
    nn.Softmax()
]
classifier_conv = torch.nn.Sequential(
    *layers_conv
)
```

From now on we going to use both the classifiers interchangable, pick one here and the rest should work with both models

In [ ]:

```
classifier = classifier_fc
#classifier = classifier_conv
```

2.2) Optimizer and Loss (2 p.)

Next we need to specify an optimizer and a loss function. For the optimizer we will use Adam (look at torch.optim) with default parameters and as a loss function we will use the cross-entropy

In [36]:

```
from torch.optim import Adam
from torch.nn import CrossEntropyLoss

optimizer:torch.optim.Adam = Adam(classifier.parameters(), lr=1e-4)
```

```
def cross_entropy_loss(labels: torch.Tensor, predictions: torch.Tensor) -> torch
    """ Cross entropy Loss:
    :param labels: Ground truth class labels (shape; [N, num_classes])
    :param predictions: predicted class labels (shape: [N, num_classes])
    :return: cross entropy (scalar)
    """
    # YTA: I am ~80% sure we should implement this ourselves but I am learning P
    loss = CrossEntropyLoss()
    print(predictions.shape, labels.shape)
    return loss(predictions, labels)
```

2.3) Data Loader (2 p.)

For batch gradient descent we need to shuffle and batch the data, PyTorch provdies some functionality for that in form of the "DataLoader" (Look at torch.utils.data). In the simplest form used here, it simply shuffles and batches the data but you can also build more complex preprocessing pipelines. We also need a loader for the test data.

In [37]:

```
from torch.utils.data import DataLoader, TensorDataset

batch_size = 64

#TODO
train_data   = TensorDataset(train_samples, train_labels)
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_w
test_data    = TensorDataset(test_samples, test_labels)
test_loader  = DataLoader(test_data, batch_size=batch_size, shuffle=True, num_wo
#####
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-37-9c783d532527> in <module>
      4 
      5 #TODO
----> 6 train_data   = TensorDataset(train_samples, train_labels)
      7 train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=Tru
e, num_workers=0)
      8 test_data    = TensorDataset(test_samples, test_labels)

~/.local/lib/python3.6/site-packages/torch/utils/data/dataset.py in __init__(sel
f, *tensors)
    164 
    165     def __init__(self, *tensors: Tensor) -> None:
--> 166         assert all(tensors[0].size(0) == tensor.size(0) for tensor in te
nsors)
    167         self.tensors = tensors
    168 

~/.local/lib/python3.6/site-packages/torch/utils/data/dataset.py in <genexpr>(.
0)
    164 
    165     def __init__(self, *tensors: Tensor) -> None:
--> 166         assert all(tensors[0].size(0) == tensor.size(0) for tensor in te
nsors)
    167         self.tensors = tensors
    168 

TypeError: 'int' object is not callable
```

2.4) Training (2 p.)

We now have all ingredients and can implement our train loop and a evaluation procedure. You should get a test set accuracy of > 0.95 with both architectures in 2 epochs.

In [35]:

```
epochs = 2  # small number of epochs should be sufficient to get descent perform

train_losses = []
test_losses = []


for i in range(epochs):
    print("Epoch {:03d}".format(i + 1))
    for batch in train_loader:
        #TODO##################
        # forward pass         #
        samples, labels = batch
        predictions = classifier(samples)

        # backward pass        #

        # YTA: note for the future: https://discuss.pytorch.org/t/runtimeerror-m
        loss = cross_entropy_loss(labels, predictions)
        optimizer.zero_grad() # clear gradient buffers
        loss.backward()

        # update step
        optimizer.step()

        #######################
        train_losses.append(loss.detach().numpy())

# Evaluate (we still need batching as evaluating all test points at once would p
avg_loss = avg_acc = 0
for batch in test_loader:
    samples, labels = batch
    predictions = classifier(samples)
    loss = cross_entropy_loss(labels, predictions)
    acc = torch.count_nonzero(predictions.argmax(dim=-1) == labels.argmax(dim=-1

    avg_acc += acc / len(test_loader)
    avg_loss += loss / len(test_loader)


print("Test Set Accuracy: {:.3f}, Test Loss {:.3f}".format(avg_acc.detach().nump

plt.figure()
plt.semilogy(train_losses)
plt.show()
```

```
Epoch 001
torch.Size([64, 10]) torch.Size([64, 10])
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-35-7f29c3f5b91f> in <module>
     16 
     17         # YTA: note for the future: https://discuss.pytorch.org/t/runtim
eerror-multi-target-not-supported-newbie/10216/2
---> 18         loss = cross_entropy_loss(labels, predictions)
     19         optimizer.zero_grad() # clear gradient buffers
```

```
     20         loss.backward()

<ipython-input-33-055bca1060a8> in cross_entropy_loss(labels, predictions)
     13     loss = CrossEntropyLoss()
     14     print(predictions.shape, labels.shape)
---> 15     return loss(predictions, labels)

~/.local/lib/python3.6/site-packages/torch/nn/modules/module.py in _call_impl(se
lf, *input, **kwargs)
    725             result = self._slow_forward(*input, **kwargs)
    726         else:
--> 727             result = self.forward(*input, **kwargs)
    728         for hook in itertools.chain(
    729                 _global_forward_hooks.values(),

~/.local/lib/python3.6/site-packages/torch/nn/modules/loss.py in forward(self, i
nput, target)
    960     def forward(self, input: Tensor, target: Tensor) -> Tensor:
    961         return F.cross_entropy(input, target, weight=self.weight,
--> 962                                ignore_index=self.ignore_index, reduction
=self.reduction)
    963
    964

~/.local/lib/python3.6/site-packages/torch/nn/functional.py in cross_entropy(inp
ut, target, weight, size_average, ignore_index, reduce, reduction)
   2466     if size_average is not None or reduce is not None:
   2467         reduction = _Reduction.legacy_get_string(size_average, reduce)
-> 2468     return nll_loss(log_softmax(input, 1), target, weight, None, ignore_
index, None, reduction)
   2469
   2470

~/.local/lib/python3.6/site-packages/torch/nn/functional.py in nll_loss(input, t
arget, weight, size_average, ignore_index, reduce, reduction)
   2262                          .format(input.size(0), target.size(0)))
   2263     if dim == 2:
-> 2264         ret = torch._C._nn.nll_loss(input, target, weight, _Reduction.ge
t_enum(reduction), ignore_index)
   2265     elif dim == 4:
   2266         ret = torch._C._nn.nll_loss2d(input, target, weight, _Reduction.
get_enum(reduction), ignore_index)

RuntimeError: 1D target tensor expected, multi-target not supported
```

In [ ]:

In [ ]: