Exercise 6: Solution

Activation Functions

Leaky ReLU – Forward

```python
def forward(self, x):
    """
    Computes forward pass for a LeakyReLu layer.
    :param x: Inputs, of any shape

    :return out: Output, of the same shape as x
    :return cache: Cache, for backward computation, of the same shape as x
    """
    outputs = np.zeros(x.shape)
    cache = np.zeros(x.shape)
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass of LeakyRelu activation function          #
    ########################################################################
    cache = np.copy(x)
    outputs = np.copy(x)
    outputs[x <= 0] *= self.slope
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return outputs, cache
```

Remark:
What is different from Relu is, when input output is not 0, but (0.01 by default).

Leaky ReLU – Backward

```
def backward(self, dout, cache):
    """
    Computes backward pass for a LeakyReLu layer.
    :param dout: Upstream derivative
    :param cache: Cache from forward() function, of the same
    shape than input to forward() function

    :return: dx: the gradient w.r.t. input X
    """
    dx = np.zeros((cache*dout).shape)
    ########################################################################
    # TODO:                                                                #
    # Implement the backward pass of LeakyRelu activation function         #
    ########################################################################
    x = cache
    d = np.ones_like(x)
    d[x <= 0] = self.slope
    dx = d * dout
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return dx
```

Remark:
What is different from Relu is, when the cache , the gradient is not 0 but the slope.

Tanh – Forward

```python
def forward(self, x):
    """
    Computes the forward pass for a Tanh layer.
    :param x: Inputs, of any shape

    :return out: Output, of the same shape as x
    :return cache: Cache, for backward computation, of the same shape as x
    """
    outputs = np.ones(x.shape)
    cache = np.ones(x.shape)
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass of Tanh activation function               #
    ########################################################################
    outputs = (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))
    cache = outputs
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return outputs, cache
```

Remark:
Forward pass of Tanh is

Optional:
You may also restore input as cache.

Tanh – Backward

```python
def backward(self, dout, cache):
    """
    Computes the backward pass of a Tanh layer.
    :param dout: Upstream derivative
    :param cache: Output of the forward pass

    :return: dx: the gradient w.r.t. input X
    """
    dx = np.ones((cache*dout).shape)
    ########################################################################
    # TODO:                                                                #
    # Implement the backward pass of Tanh activation function             #
    ########################################################################
    x = cache
    dx = 1 - x ** 2
    dx = dx * dout
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return dx
```

Remark:
The backward pass of Tanh is

Random Search

A feasible set of range of hyperparameters

```
from exercise_code.networks import MyOwnNetwork, ClassificationNet


model_type = ClassificationNet
#model_type = MyOwnNetwork

########################################################################
# TODO:                                                                #
# Implement your own neural network and find suitable hyperparameters  #
# Be sure to edit the MyOwnNetwork class in the following code snippet #
# to upload the correct model! Or just use the given                   #
# "ClassificationNet".                                                 #
#                                                                      #
# Note: the pickling cell expects your model to be named "best_model". #
# Unless you change it there, naming the best model in any other way   #
# will result in an unknown behavior.                                  #
########################################################################
from exercise_code.hyperparameter_tuning import random_search

best_model, best_config, results  = random_search(dataloaders['train_small'], dataloaders['val_500files'],
                                         random_search_spaces = {
                                           "learning_rate": ([1e-3, 1e-4], 'log'),
                                           "lr_decay": ([0.8, 1.0], 'float'),
                                           "reg": ([1e-3, 1e-5], "log"), # [1e-4, 1e-6]
                                           "std": ([1e-2, 1e-5], "log"), # [1e-4, 1e-6]
                                           "hidden_size": ([150, 250], "int"),
                                           "num_layer": ([2, 4], "int"), # [2, 5]
                                           "activation": ([Relu], "item"),
                                           "optimizer": ([Adam], "item"),
                                           "loss_func": ([CrossEntropyFromLogits], "item")
                                          }, num_search = 3, epochs=20, patience=3,
                                   model_class=ClassificationNet)

best_model.reset_weights()
solver = Solver(best_model, dataloaders['train'], dataloaders['val'], **best_config)
solver.train(epochs=25, patience=5)
########################################################################
#                            END OF YOUR CODE                          #
########################################################################
```

Pick the best set of hyperparameters

```
Search done. Best Val Loss = 1.944257451949427
Best Config: {'learning_rate': 0.0006969479654498323, 'lr_decay': 0.9653239284975306, 'reg': 2.0619039489725804e-05,
'std': 4.040993722890476e-05, 'hidden_size': 157, 'num_layer': 3, 'activation': <class 'exercise_code.networks.laye
r.Relu'>, 'optimizer': <class 'exercise_code.networks.optimizer.Adam'>, 'loss_func': <class 'exercise_code.networks.
loss.CrossEntropyFromLogits'>}
```

3.5 Checking the validation accuracy

```python
from exercise_code.tests.base_tests import bcolors

labels, pred, acc = best_model.get_dataset_prediction(dataloaders['train'])
res = bcolors.colorize("green", acc * 100) if acc * 100 > 48 else bcolors.colorize("red", acc * 100)
print("Train Accuracy: {}%".format(res))
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['val'])
res = bcolors.colorize("green", acc * 100) if acc * 100 > 48 else bcolors.colorize("red", acc * 100)
print("Validation Accuracy: {}%".format(res))
```

```
Train Accuracy: 70.18563034188034%
Validation Accuracy: 51.20192307692307%
```

```python
# comment this part out to see your model's performance on the test set.
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['test'])
res = bcolors.colorize("green", acc * 100) if acc * 100 > 48 else bcolors.colorize("red", acc * 100)
print("Test Accuracy: {}%".format(res))
```

```
Test Accuracy: 51.53245192307693%
```

Questions? Piazza ☺