# 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

```python
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

best_model = ClassificationNet()
#best_model = 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!                                         #
########################################################################
from exercise_code.hyperparameter_tuning import random_search

best_model, results = random_search(dataloaders['train'], dataloaders['val'],
                                    random_search_spaces = {
                                      "learning_rate": ([1e-3, 1e-4], 'log'),
                                      "lr_decay": ([0.8, 0.9], 'float'),
                                      "reg": ([1e-4, 1e-6], "log"),
                                      "std": ([1e-4, 1e-6], "log"),
                                      "hidden_size": ([50, 100], "int"),
                                      "num_layer": ([2], "int"),
                                      "activation": ([Relu()], "item"),
                                      "optimizer": ([Adam], "item"),
                                      "loss_func": ([CrossEntropyFromLogits()], "item")
                                     }, num_search = 5, epochs=20, patience=5,
                      model_class=ClassificationNet)
########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

# Pick the best set of hyperparameters

```
Search done. Best Val Loss = 1.4614823323760282
Best Config: {'learning_rate': 0.0009363255745516442, 'lr_decay': 0.8106866888065208, 'reg': 3.5115962843695404e-05, 'std': 1.0074810757234067e-06, 'hidden_size': 96, 'num_layer': 2, 'activation': <exercise_code.networks.layer.Relu object at 0x7f3a256d52b0>, 'optimizer': <class 'exercise_code.networks.optimizer.Adam'>, 'loss_func': <exercise_code.networks.loss.CrossEntropyFromLogits object at 0x7f3a4cb2da00>}
```

## Checking the validation accuracy

```
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['train'])
print("Train Accuracy: {}%".format(acc*100))
labels, pred, acc = best_model.get_dataset_prediction(dataloaders['val'])
print("Validation Accuracy: {}%".format(acc*100))
```

```
Train Accuracy: 57.85590277777778%
Validation Accuracy: 49.23878205128205%
```

```
# comment this part out to see your model's performance on the test set.

labels, pred, acc = best_model.get_dataset_prediction(dataloaders['test'])
print("Test Accuracy: {}%".format(acc*100))
```

```
Test Accuracy: 49.31891025641025%
```

# Questions? Piazza ☺