Exercise 06 — Solution Reference

I2DL — Introduction to Deep Learning

exercise_code/networks/classification_net.py

```
########################################################################
# TODO:  Your initialization here                                      #
########################################################################

self.activation = activation()
self.reg_strength = reg

self.cache = None

self.memory = 0
self.memory_forward = 0
self.memory_backward = 0
self.num_operation = 0

self.num_layer = num_layer
self.params = {'W1': std * np.random.randn(input_size, hidden_size),
               'b1': np.zeros(hidden_size)}

for i in range(num_layer - 2):
    self.params['W' + str(i + 2)] = std * np.random.randn(hidden_size,
                                                          hidden_size)
    self.params['b' + str(i + 2)] = np.zeros(hidden_size)

self.params['W' + str(num_layer)] = std * np.random.randn(hidden_size,
                                                          num_classes)
self.params['b' + str(num_layer)] = np.zeros(num_classes)

self.grads = {}
self.reg = {}
for i in range(num_layer):
    self.grads['W' + str(i + 1)] = 0.0
    self.grads['b' + str(i + 1)] = 0.0

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

```
########################################################################
# TODO:  Your forward here                                             #
########################################################################

# Something like sandwich layer here
self.cache = {}
```

```
self.reg = {}
X = X.reshape(X.shape[0], -1)
# Unpack variables from the params dictionary
for i in range(self.num_layer - 1):
    W, b = self.params['W' + str(i + 1)], self.params['b' + str(i + 1)]

    # Forward i_th layer
    X, cache_affine = affine_forward(X, W, b)
    self.cache["affine" + str(i + 1)] = cache_affine

    # Activation function
    X, cache_sigmoid = self.activation.forward(X)
    self.cache["sigmoid" + str(i + 1)] = cache_sigmoid

    # Store the reg for the current W
    self.reg['W' + str(i + 1)] = np.sum(W ** 2) * self.reg_strength

# last layer contains no activation functions
W, b = self.params['W' + str(self.num_layer)],\
       self.params['b' + str(self.num_layer)]
out, cache_affine = affine_forward(X, W, b)
self.cache["affine" + str(self.num_layer)] = cache_affine
self.reg['W' + str(self.num_layer)] = np.sum(W ** 2) * self.reg_strength

############################################################################
#                           END OF YOUR CODE                               #
############################################################################
```

```
############################################################################
# TODO:  Your backward here                                                #
############################################################################

# Note that last layer has no activation
cache_affine = self.cache['affine' + str(self.num_layer)]
dh, dW, db = affine_backward(dy, cache_affine)
self.grads['W' + str(self.num_layer)] = \
    dW + 2 * self.reg_strength * self.params['W' + str(self.num_layer)]
self.grads['b' + str(self.num_layer)] = db

# The rest sandwich layers
for i in range(self.num_layer - 2, -1, -1):
    # Unpack cache
    cache_sigmoid = self.cache['sigmoid' + str(i + 1)]
    cache_affine = self.cache['affine' + str(i + 1)]

    # Activation backward
    dh = self.activation.backward(dh, cache_sigmoid)

    # Affine backward
    dh, dW, db = affine_backward(dh, cache_affine)

    # Refresh the gradients
    self.grads['W' + str(i + 1)] = \
        dW + 2 * self.reg_strength * self.params['W' + str(i + 1)]
    self.grads['b' + str(i + 1)] = db

grads = self.grads
############################################################################
#                           END OF YOUR CODE                               #
############################################################################
```

```
##############################################################################
```

exercise_code/networks/layer.py

```
##############################################################################
# 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                               #
##############################################################################
```

```
##############################################################################
# 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                               #
##############################################################################
```

```
##############################################################################
# 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                               #
##############################################################################
```

```
##############################################################################
# TODO:                                                                      #
# Implement the backward pass of Tanh activation function              #
##############################################################################
x = cache
dx = 1 - x ** 2
dx = dx * dout
##############################################################################
```

```
#                             END OF YOUR CODE                                 #
################################################################################
```

1_cifar10_classification.ipynb

```
################################################################################
# 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

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"),
}

best_model, best_config, results = random_search(
    dataloaders["train_small"],
    dataloaders["val_500files"],
    random_search_spaces=random_search_spaces,
    num_search=3,
    epochs=10,
    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                                 #
################################################################################
```