Problem 1: Basics of Neural Networks

- **Learning Objective:** In this problem, you are asked to implement a basic multi-layer fully connected neural network from scratch, including forward and backward passes of certain essential layers, to perform an image classification task on the CIFAR100 dataset. You need to implement essential functions in different indicated python files under directory `lib`.
- **Provided Code:** We provide the skeletons of classes you need to complete. Forward checking and gradient checkings are provided for verifying your implementation as well.
- **TODOs:** You are asked to implement the forward passes and backward passes for standard layers and loss functions, various widely-used optimizers, and part of the training procedure. And finally we want you to train a network from scratch on your own. Also, there are inline questions you need to answer. See `README.md` to set up your environment.

```
from lib.mlp.fully_conn import *
from lib.mlp.layer_utils import *
from lib.datasets import *
from lib.mlp.train import *
from lib.grad_check import *
from lib.optim import *
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Loading the data (CIFAR-100 with 20 superclasses)

In this homework, we will be classifying images from the CIFAR-100 dataset into the 20 superclasses. More information about the CIFAR-100 dataset and the 20 superclasses can be found [here](#).

Download the CIFAR-100 data files [here](#), and save the `.mat` files to the `data/cifar100` directory.

Load the dataset.

```python
data = CIFAR100_data('data/cifar100/')
for k, v in data.items():
    if type(v) == np.ndarray:
        print ("Name: {} Shape: {}, {}".format(k, v.shape, type(v)))
    else:
        print("{}: {}".format(k, v))
label_names = data['label_names']
mean_image = data['mean_image'][0]
std_image = data['std_image'][0]
```

```
Name: data_train Shape: (40000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_train Shape: (40000,), <class 'numpy.ndarray'>
Name: data_val Shape: (10000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_val Shape: (10000,), <class 'numpy.ndarray'>
Name: data_test Shape: (10000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_test Shape: (10000,), <class 'numpy.ndarray'>
label_names: ['aquatic_mammals', 'fish', 'flowers', 'food_containers',
'fruit_and_vegetables', 'household_electrical_devices', 'household_furniture', 'insects',
'large_carnivores', 'large_man-made_outdoor_things', 'large_natural_outdoor_scenes',
'large_omnivores_and_herbivores', 'medium_mammals', 'non-insect_invertebrates', 'people',
'reptiles', 'small_mammals', 'trees', 'vehicles_1', 'vehicles_2']
Name: mean_image Shape: (1, 1, 1, 3), <class 'numpy.ndarray'>
Name: std_image Shape: (1, 1, 1, 3), <class 'numpy.ndarray'>
```

Implement Standard Layers

You will now implement all the following standard layers commonly seen in a fully connected neural network (aka multi-layer perceptron, MLP). Please refer to the file `lib/mlp/layer_utils.py`. Take a look at each class skeleton, and we will walk you through the network layer by layer. We provide results of some examples we pre-computed for you for checking the forward pass, and also the gradient checking for the backward pass.

FC Forward [2pt]

In the class skeleton `flatten` and `fc` in `lib/mlp/layer_utils.py`, please complete the forward pass in function `forward`. The input to the `fc` layer may not be of dimension (batch size, features size), it could be an image or any higher dimensional data. We want to convert the input to have a shape of (batch size, features size). Make sure that you handle this dimensionality issue.

```python
%reload_ext autoreload

# Test the fc forward function
input_bz = 3 # batch size
input_dim = (7, 6, 4)
output_dim = 4
```

```python
input_size = input_bz * np.prod(input_dim) # 504
weight_size = output_dim * np.prod(input_dim) # 672

flatten_layer = flatten(name="flatten_test")
single_fc = fc(np.prod(input_dim), output_dim, init_scale=0.02, name="fc_test")

x = np.linspace(-0.1, 0.4, num=input_size).reshape(input_bz, *input_dim) # (3, 7, 6, 4)
w = np.linspace(-0.2, 0.2, num=weight_size).reshape(np.prod(input_dim), output_dim) #
(168, 4)
b = np.linspace(-0.3, 0.3, num=output_dim) # (4, )

single_fc.params[single_fc.w_name] = w
single_fc.params[single_fc.b_name] = b

out = single_fc.forward(flatten_layer.forward(x)) #(3, 4)

correct_out = np.array([[0.63910291, 0.83740057, 1.03569824, 1.23399591],
                        [0.61401587, 0.82903823, 1.04406058, 1.25908294],
                        [0.58892884, 0.82067589, 1.05242293, 1.28416997]])

# Compare your output with the above pre-computed ones.
# The difference should not be larger than 1e-8
print ("Difference: ", rel_error(out, correct_out))
```

```
Difference:  4.0260162945880345e-09
```

FC Backward [2pt]

Please complete the function `backward` as the backward pass of the `flatten` and `fc` layers. Follow the instructions in the comments to store gradients into the predefined dictionaries in the attributes of the class. Parameters of the layer are also stored in the predefined dictionary.

```python
%reload_ext autoreload

# Test the fc backward function
inp = np.random.randn(15, 2, 2, 3)
w = np.random.randn(12, 15)
b = np.random.randn(15)
dout = np.random.randn(15, 15)

flatten_layer = flatten(name="flatten_test")
x = flatten_layer.forward(inp) # flatten x: (15, 2, 2, 3) -> (15, 12)
single_fc = fc(np.prod(x.shape[1:]), 15, init_scale=5e-2, name="fc_test") # (input_dim,
output_dim) = (12, 15)
```

```
single_fc.params[single_fc.w_name] = w # (12, 15)
single_fc.params[single_fc.b_name] = b # (15, )

dx_num = eval_numerical_gradient_array(lambda x: single_fc.forward(x), x, dout) # (15, 12)
dw_num = eval_numerical_gradient_array(lambda w: single_fc.forward(x), w, dout) # (12, 15)
db_num = eval_numerical_gradient_array(lambda b: single_fc.forward(x), b, dout) # (15, )

out = single_fc.forward(x) # (15, 15)
dx = single_fc.backward(dout) # # (15, 12)
dw = single_fc.grads[single_fc.w_name] # (12, 15)
db = single_fc.grads[single_fc.b_name] # (15, )
dinp = flatten_layer.backward(dx) # (15, 2, 2, 3)

# The error should be around 1e-9
print("dx Error: ", rel_error(dx_num, dx))
# The errors should be around 1e-10
print("dw Error: ", rel_error(dw_num, dw))
print("db Error: ", rel_error(db_num, db))
# The shapes should be same
print("dinp Shape: ", dinp.shape, inp.shape)
```

```
dx Error:  9.16324855129798e-09
dw Error:  1.970794888679424e-09
db Error:  2.57067060494361e-11
dinp Shape:  (15, 2, 2, 3) (15, 2, 2, 3)
```

GeLU Forward [2pt]

In the class skeleton `gelu` in `lib/mlp/layer_utils.py`, please complete the `forward` pass.

GeLU is a smooth version of ReLU and it's used in pre-training LLMs such as GPT-3(maybe chatGPT) and BERT.

$$\mathrm{GeLU}(x) = x\Phi(x) \approx 0.5x(1 + \tanh(\sqrt{2/\pi}(x + 0.044715x^3)))$$

Where $\Phi(x)$ is the CDF for standard Gaussian random variables. You should use the approximate version to compute forward and backward pass.

```
%reload_ext autoreload

# Test the leaky_relu forward function
x = np.linspace(-1.5, 1.5, num=12).reshape(3, 4)
gelu_f = gelu(name="gelu_f")

out = gelu_f.forward(x)
correct_out = np.array([[-0.10042842, -0.13504766, -0.16231757, -0.1689214 ],
```

```
                        [-0.13960493, -0.06078651,  0.07557713,  0.26948598],
                        [ 0.51289678,  0.79222788,  1.09222506,  1.39957158]])

# Compare your output with the above pre-computed ones.
# The difference should not be larger than 1e-7
print ("Difference: ", rel_error(out, correct_out))
```

```
Difference:  1.8037541876132445e-08
```

GeLU Backward [2pt]

Please complete the `backward` pass of the class `gelu`.

```
%reload_ext autoreload

# Test the relu backward function
x = np.random.randn(15, 15)
dout = np.random.randn(*x.shape)
gelu_b = gelu(name="gelu_b")

dx_num = eval_numerical_gradient_array(lambda x: gelu_b.forward(x), x, dout)

out = gelu_b.forward(x)
dx = gelu_b.backward(dout)

# The error should not be larger than 1e-4, since we are using an approximate version of
GeLU activation.
print ("dx Error: ", rel_error(dx_num, dx))
```

```
dx Error:  4.8150860216957875e-09
```

Dropout Forward [2pt]

In the class `dropout` in `lib/mlp/layer_utils.py`, please complete the `forward` pass.
Remember that the dropout is **only applied during training phase**, you should pay attention to this while implementing the function.

Important Note1: The probability argument input to the function is the "keep probability": probability that each activation is kept.

Important Note2: If the keep_prob is set to 1, make it as no dropout.

```
%reload_ext autoreload

x = np.random.randn(100, 100) + 5.0

print ("----------------------------------------------------------------")
for p in [0, 0.25, 0.50, 0.75, 1]:
    dropout_f = dropout(keep_prob=p)
    out = dropout_f.forward(x, True)
    out_test = dropout_f.forward(x, False)

    # Mean of output should be similar to mean of input
    # Means of output during training time and testing time should be similar
    print ("Dropout Keep Prob = ", p)
    print ("Mean of input: ", x.mean())
    print ("Mean of output during training time: ", out.mean())
    print ("Mean of output during testing time: ", out_test.mean())
    print ("Fraction of output set to zero during training time: ", (out == 0).mean())
    print ("Fraction of output set to zero during testing time: ", (out_test == 0).mean())
    print ("----------------------------------------------------------------")
```

```
----------------------------------------------------------------
Dropout Keep Prob =  0
Mean of input:  5.005474277864162
Mean of output during training time:  5.005474277864162
Mean of output during testing time:  5.005474277864162
Fraction of output set to zero during training time:  0.0
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  0.25
Mean of input:  5.005474277864162
Mean of output during training time:  4.951419466067096
Mean of output during testing time:  5.005474277864162
Fraction of output set to zero during training time:  0.753
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  0.5
Mean of input:  5.005474277864162
Mean of output during training time:  4.894524816642579
Mean of output during testing time:  5.005474277864162
Fraction of output set to zero during training time:  0.5102
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  0.75
```

```
Mean of input:  5.005474277864162
Mean of output during training time:  4.982551780652704
Mean of output during testing time:  5.005474277864162
Fraction of output set to zero during training time:  0.253
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  1
Mean of input:  5.005474277864162
Mean of output during training time:  5.005474277864162
Mean of output during testing time:  5.005474277864162
Fraction of output set to zero during training time:  0.0
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
```

Dropout Backward [2pt]

Please complete the `backward` pass. Again remember that the dropout is only applied during training phase, handle this in the backward pass as well.

```
%reload_ext autoreload

x = np.random.randn(5, 5) + 5
dout = np.random.randn(*x.shape)

keep_prob = 0.75
dropout_b = dropout(keep_prob, seed=100)
out = dropout_b.forward(x, True, seed=1)
dx = dropout_b.backward(dout)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_b.forward(xx, True, seed=1), x,
dout)

# The error should not be larger than 1e-10
print ('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  3.003117117962552e-11
```

Testing cascaded layers: FC + GeLU [2pt]

Please find the `TestFCGeLU` function in `lib/mlp/fully_conn.py`.

You only need to complete a few lines of code in the TODO block.

Please design an `Flatten -> FC -> GeLU` network where the parameters of them match the given x, w, and b.

Please insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively. Here you only modify the param_name part, the `_w`, and `_b` are automatically assigned during network setup

```
%reload_ext autoreload

x = np.random.randn(3, 5, 3)  # the input features
w = np.random.randn(15, 5)   # the weight of fc layer
b = np.random.randn(5)       # the bias of fc layer
dout = np.random.randn(3, 5) # the gradients to the output, notice the shape

tiny_net = TestFCGeLU() # input: 15 -> output: 5

###################################################
# TODO: param_name should be replaced accordingly #
###################################################
tiny_net.net.assign("fc_tiny_w", w)
tiny_net.net.assign("fc_tiny_b", b)
###################################################
#                 END OF YOUR CODE                #
###################################################

out = tiny_net.forward(x)
dx = tiny_net.backward(dout)

###################################################
# TODO: param_name should be replaced accordingly #
###################################################
dw = tiny_net.net.get_grads("fc_tiny_w")
db = tiny_net.net.get_grads("fc_tiny_b")
###################################################
#                 END OF YOUR CODE                #
###################################################

dx_num = eval_numerical_gradient_array(lambda x: tiny_net.forward(x), x, dout)
dw_num = eval_numerical_gradient_array(lambda w: tiny_net.forward(x), w, dout)
db_num = eval_numerical_gradient_array(lambda b: tiny_net.forward(x), b, dout)

# The errors should not be larger than 1e-7
print ("dx error: ", rel_error(dx_num, dx))
print ("dw error: ", rel_error(dw_num, dw))
print ("db error: ", rel_error(db_num, db))
```

```
dx error:  8.731538036587429e-10
dw error:  4.765131662361882e-10
db error:  2.3802939941946996e-11
```

SoftMax Function and Loss Layer [2pt]

In the `lib/mlp/layer_utils.py`, please first complete the function `softmax`, which will be used in the function `cross_entropy`. Then, implement `corss_entropy` using `softmax`.
Please refer to the lecture slides of the mathematical expressions of the cross entropy loss function, and complete its forward pass and backward pass. You should also take care of `size_average` on whether or not to divide by the batch size.

```
%reload_ext autoreload

num_classes, num_inputs = 6, 100
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

test_loss = cross_entropy()

dx_num = eval_numerical_gradient(lambda x: test_loss.forward(x, y), x, verbose=False)

loss = test_loss.forward(x, y)
dx = test_loss.backward()

# Test softmax_loss function. Loss should be around 1.792
# and dx error should be at the scale of 1e-8 (or smaller)
print ("Cross Entropy Loss: ", loss)
print ("dx error: ", rel_error(dx_num, dx))
```

```
Cross Entropy Loss:  1.7917815336510288
dx error:  6.097059491682977e-09
```

Test a Small Fully Connected Network [2pt]

Please find the `SmallFullyConnectedNetwork` function in `lib/mlp/fully_conn.py`.

Again you only need to complete few lines of code in the TODO block.

Please design an `FC --> GeLU --> FC` network where the shapes of parameters match the given shapes.

Please insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively.

Here you only modify the param_name part, the `_w`, and `_b` are automatically assigned during network setup.

```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

model = SmallFullyConnectedNetwork()
loss_func = cross_entropy()

N, D, = 4, 4  # N: batch size, D: input dimension
H, C  = 30, 7 # H: hidden dimension, C: output dimension
std = 0.02
x = np.random.randn(N, D)
y = np.random.randint(C, size=N)

print ("Testing initialization ... ")

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
w1_std = abs(model.net.get_params("fc1_w").std() - std)
b1 = model.net.get_params("fc1_b").std()
w2_std = abs(model.net.get_params("fc2_w").std() - std)
b2 = model.net.get_params("fc2_b").std()
###################################################
#                 END OF YOUR CODE                #
###################################################

assert w1_std < std / 10, "First layer weights do not seem right"
assert np.all(b1 == 0), "First layer biases do not seem right"
assert w2_std < std / 10, "Second layer weights do not seem right"
assert np.all(b2 == 0), "Second layer biases do not seem right"
print ("Passed!")

print ("Testing test-time forward pass ... ")
w1 = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
w2 = np.linspace(-0.2, 0.2, num=H*C).reshape(H, C)
b1 = np.linspace(-0.6, 0.2, num=H)
b2 = np.linspace(-0.9, 0.1, num=C)

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
model.net.assign("fc1_w", w1)
model.net.assign("fc1_b", b1)
```

```
model.net.assign("fc1_b", b1)
model.net.assign("fc2_w", w2)
model.net.assign("fc2_b", b2)
###################################################
#                 END OF YOUR CODE                #
###################################################

feats = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.forward(feats)
correct_scores = np.asarray([[-2.33881897, -1.92174121, -1.50466344, -1.08758567,
-0.6705079, -0.25343013,  0.16364763],
                             [-1.57214916, -1.1857013 , -0.79925345, -0.41280559,
-0.02635774, 0.36009011,  0.74653797],
                             [-0.80178618, -0.44604469, -0.0903032 ,  0.26543829,
 0.62117977, 0.97692126,  1.33266275],
                             [-0.00331319,  0.32124836,  0.64580991,  0.97037146,
 1.29493301, 1.61949456,  1.94405611]])
scores_diff = np.sum(np.abs(scores - correct_scores))
assert scores_diff < 1e-6, "Your implementation might be wrong!"
print ("Passed!")

print ("Testing the loss ...",)
y = np.asarray([0, 5, 1, 4])
loss = loss_func.forward(scores, y)
dLoss = loss_func.backward()
correct_loss = 2.4248995879903195
print(loss - correct_loss)
assert abs(loss - correct_loss) < 1e-10, "Your implementation might be wrong!"
print ("Passed!")

print ("Testing the gradients (error should be no larger than 1e-6) ...")
din = model.backward(dLoss)
for layer in model.net.layers:
    if not layer.params:
        continue
    for name in sorted(layer.grads):
        f = lambda _: loss_func.forward(model.forward(feats), y)
        grad_num = eval_numerical_gradient(f, layer.params[name], verbose=False)
        print ('%s relative error: %.2e' % (name, rel_error(grad_num, layer.grads[name])))
```

```
Testing initialization ...
Passed!
Testing test-time forward pass ...
Passed!
Testing the loss ...
0.0
Passed!
Testing the gradients (error should be no larger than 1e-6) ...
fc1_b relative error: 3.69e-09
fc1_w relative error: 9.50e-09
fc2_b relative error: 4.01e-10
fc2_w relative error: 2.50e-08
```

Test a Fully Connected Network regularized with Dropout [2pt]

Please find the `DropoutNet` function in `fully_conn.py` under `lib/mlp` directory.

For this part you don't need to design a new network, just simply run the following test code.

If something goes wrong, you might want to double check your dropout implementation.

```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

N, D, C = 3, 15, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for keep_prob in [0, 0.25, 0.5]:
    np.random.seed(seed=seed)
    print ("Dropout p =", keep_prob)
    model = DropoutNet(keep_prob=keep_prob, seed=seed)
    loss_func = cross_entropy()
    output = model.forward(X, True, seed=seed)
    loss = loss_func.forward(output, y)
    dLoss = loss_func.backward()
    dX = model.backward(dLoss)
    grads = model.net.grads

    print ("Error of gradients should be around or less than 1e-3")
    for name in sorted(grads):

        if name not in model.net.params.keys():
```

```
            continue
        f = lambda _: loss_func.forward(model.forward(X, True, seed=seed), y)
        grad_num = eval_numerical_gradient(f, model.net.params[name], verbose=False, h=1e-
5)
        print ("{} relative error: {}".format(name, rel_error(grad_num, grads[name])))
    print ()
```

```
Dropout p = 0
Error of gradients should be around or less than 1e-3
fc1_b relative error: 9.824168294355846e-08
fc1_w relative error: 4.706355822908009e-06
fc2_b relative error: 1.133402768221828e-08
fc2_w relative error: 3.167223138534255e-05
fc3_b relative error: 2.05181811870711e-10
fc3_w relative error: 2.253362152699091e-06

Dropout p = 0.25
Error of gradients should be around or less than 1e-3
fc1_b relative error: 1.1070571846756998e-07
fc1_w relative error: 8.179318698085921e-06
fc2_b relative error: 1.1076481195535879e-08
fc2_w relative error: 2.372456507086429e-05
fc3_b relative error: 2.4432007409083205e-10
fc3_w relative error: 8.853121862924033e-07

Dropout p = 0.5
Error of gradients should be around or less than 1e-3
fc1_b relative error: 1.162731518399566e-07
fc1_w relative error: 1.1142454913520794e-06
fc2_b relative error: 2.2533227779864148e-08
fc2_w relative error: 1.5836547834856484e-06
fc3_b relative error: 3.463291656092594e-10
fc3_w relative error: 6.610872364354695e-06
```

Training a Network

In this section, we defined a `TinyNet` class for you to fill in the TODO block in `lib/mlp/fully_conn.py`.

- Here please design a two layer fully connected network with Leaky ReLU activation (`Flatten --> FC --> GeLU --> FC`).
- You can adjust the number of hidden neurons, batch_size, epochs, and learning rate decay parameters.
- Please read the `lib/train.py` carefully and complete the TODO blocks in the `train_net` function first.

  Codes in "Test a Small Fully Connected Network" can be helpful

Codes in "Test a Small Fully Connected Network" can be helpful.

- Implement SGD in `lib/optim.py`, you will be asked to complete weight decay and Adam in the later sections.

```
# Arrange the data
data_dict = {
    "data_train": (data["data_train"], data["labels_train"]),
    "data_val": (data["data_val"], data["labels_val"]),
    "data_test": (data["data_test"], data["labels_test"])
}
```

```
print("Data shape:", data["data_train"].shape)
print("Flattened data input size:", np.prod(data["data_train"].shape[1:]))
print("Number of data classes:", max(data['labels_train']) + 1)
```

```
Data shape: (40000, 32, 32, 3)
Flattened data input size: 3072
Number of data classes: 20
```

Now train the network to achieve at least 30% validation accuracy [5pt]

You may only adjust the hyperparameters inside the TODO block

```
%autoreload
```

```
%reload_ext autoreload

seed = 123
np.random.seed(seed=seed)

model = TinyNet()
loss_f = cross_entropy()
optimizer = SGD(model.net, 0.1)

results = None
#############################################################################
# TODO: Use the train_net function you completed to train a network         #
#############################################################################

batch_size = 256
epochs = 20
lr_decay = 0.95
lr_decay_every = 50
```

```
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
# input_dim = 3072, output_dim = 10
results = train_net(data_dict, model, loss_f, optimizer, batch_size, epochs,
                    lr_decay, lr_decay_every, show_every=10000, verbose=True)
opt_params, loss_hist, train_acc_hist, val_acc_hist = results
```

```
  1%|▏         | 2/156 [00:00<00:20,  7.45it/s]

(Iteration 1 / 3120) Average loss: 5.833234980703788
```

```
100%|██████████| 156/156 [00:21<00:00,  7.42it/s]
  1%|          | 1/156 [00:00<00:22,  6.81it/s]

(Epoch 1 / 20) Training Accuracy: 0.327275, Validation Accuracy: 0.2762
```

```
100%|██████████| 156/156 [00:20<00:00,  7.48it/s]
  1%|          | 1/156 [00:00<00:21,  7.33it/s]

(Epoch 2 / 20) Training Accuracy: 0.374625, Validation Accuracy: 0.2865
```

```
100%|██████████| 156/156 [00:20<00:00,  7.62it/s]
  1%|          | 1/156 [00:00<00:21,  7.30it/s]

(Epoch 3 / 20) Training Accuracy: 0.396675, Validation Accuracy: 0.2897
```

```
100%|██████████| 156/156 [00:20<00:00,  7.57it/s]
  1%|          | 1/156 [00:00<00:22,  6.93it/s]

(Epoch 4 / 20) Training Accuracy: 0.466575, Validation Accuracy: 0.3106
```

```
100%|██████████| 156/156 [00:20<00:00,  7.55it/s]
  1%|          | 1/156 [00:00<00:21,  7.22it/s]

(Epoch 5 / 20) Training Accuracy: 0.5021, Validation Accuracy: 0.3147
```

```
100%|██████████| 156/156 [00:20<00:00,  7.56it/s]
  1%|          | 1/156 [00:00<00:21,  7.23it/s]

(Epoch 6 / 20) Training Accuracy: 0.49515, Validation Accuracy: 0.3164
```

```
100%|██████████| 156/156 [00:20<00:00,  7.49it/s]
  1%|          | 1/156 [00:00<00:21,  7.23it/s]

(Epoch 7 / 20) Training Accuracy: 0.55485, Validation Accuracy: 0.327
```

```
100%|██████████| 156/156 [00:20<00:00,  7.44it/s]
  1%|          | 1/156 [00:00<00:21,  7.21it/s]

(Epoch 8 / 20) Training Accuracy: 0.541525, Validation Accuracy: 0.3118
```

```
100%|██████████| 156/156 [00:20<00:00,  7.45it/s]
  1%|          | 1/156 [00:00<00:21,  7.21it/s]

(Epoch 9 / 20) Training Accuracy: 0.49885, Validation Accuracy: 0.2743
```

```
100%|██████████| 156/156 [00:20<00:00,  7.44it/s]
  1%|          | 1/156 [00:00<00:28,  5.39it/s]

(Epoch 10 / 20) Training Accuracy: 0.544825, Validation Accuracy: 0.2863
```

```
100%|██████████| 156/156 [00:21<00:00,  7.25it/s]
  1%|          | 1/156 [00:00<00:22,  6.95it/s]

(Epoch 11 / 20) Training Accuracy: 0.619975, Validation Accuracy: 0.3247
```

```
100%|██████████| 156/156 [00:21<00:00,  7.26it/s]
  1%|          | 1/156 [00:00<00:27,  5.73it/s]

(Epoch 12 / 20) Training Accuracy: 0.608325, Validation Accuracy: 0.3079
```

```
100%|██████████| 156/156 [00:21<00:00,  7.23it/s]
  1%|          | 1/156 [00:00<00:22,  6.99it/s]

(Epoch 13 / 20) Training Accuracy: 0.6469, Validation Accuracy: 0.3287
```

```
100%|██████████| 156/156 [00:21<00:00,  7.22it/s]
  1%|          | 1/156 [00:00<00:21,  7.06it/s]

(Epoch 14 / 20) Training Accuracy: 0.635475, Validation Accuracy: 0.3119
```

```
100%|██████████| 156/156 [00:21<00:00,  7.14it/s]
  1%|          | 1/156 [00:00<00:22,  7.03it/s]

(Epoch 15 / 20) Training Accuracy: 0.6255, Validation Accuracy: 0.3055
```

```
100%|██████████| 156/156 [00:21<00:00,  7.38it/s]
  1%|          | 1/156 [00:00<00:21,  7.13it/s]

(Epoch 16 / 20) Training Accuracy: 0.64515, Validation Accuracy: 0.3239
```

```
100%|██████████| 156/156 [00:21<00:00,  7.38it/s]
  1%|          | 1/156 [00:00<00:21,  7.06it/s]

(Epoch 17 / 20) Training Accuracy: 0.7014, Validation Accuracy: 0.3174
```

```
100%|██████████| 156/156 [00:21<00:00,  7.31it/s]
  1%|          | 1/156 [00:00<00:21,  7.09it/s]

(Epoch 18 / 20) Training Accuracy: 0.696325, Validation Accuracy: 0.3131
```

```
100%|██████████| 156/156 [00:21<00:00,  7.31it/s]
  1%|          | 1/156 [00:00<00:21,  7.05it/s]

(Epoch 19 / 20) Training Accuracy: 0.732175, Validation Accuracy: 0.3201
```

```
100%|██████████| 156/156 [00:21<00:00,  7.21it/s]
```

```
(Epoch 20 / 20) Training Accuracy: 0.72455, Validation Accuracy: 0.323
```

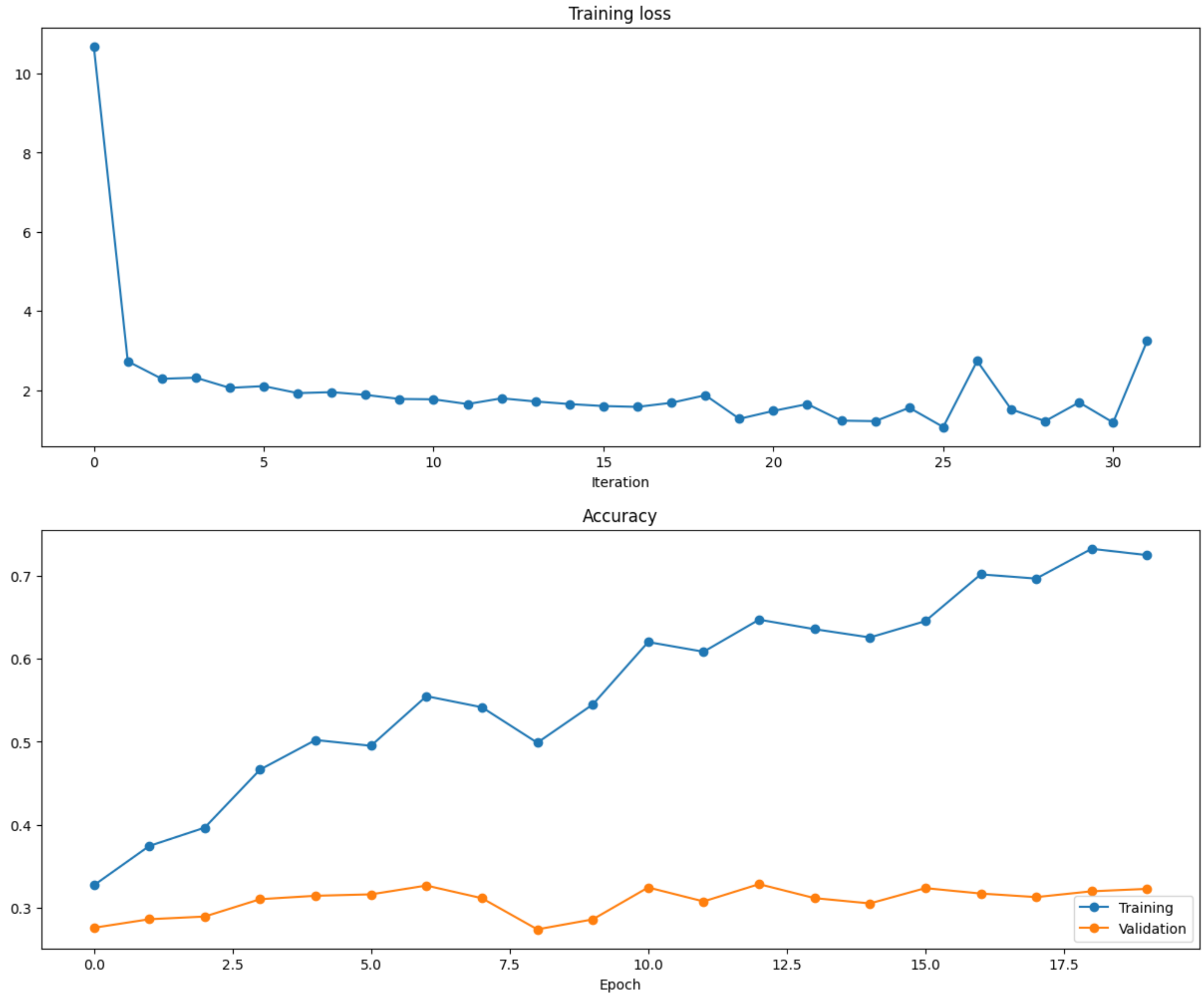
```python
# Take a look at what names of params were stored
print (opt_params.keys())
```

```
dict_keys(['fc1_w', 'fc1_b', 'fc2_w', 'fc2_b'])
```

```python
# Demo: How to load the parameters to a newly defined network
model = TinyNet()
model.net.load(opt_params)
val_acc = compute_acc(model, data["data_val"], data["labels_val"])
print ("Validation Accuracy: {}%".format(val_acc*100))
test_acc = compute_acc(model, data["data_test"], data["labels_test"])
print ("Testing Accuracy: {}%".format(test_acc*100))
```

```
Loading Params: fc1_w Shape: (3072, 1024)
Loading Params: fc1_b Shape: (1024,)
Loading Params: fc2_w Shape: (1024, 20)
Loading Params: fc2_b Shape: (20,)
Validation Accuracy: 32.300000000000004%
Testing Accuracy: 33.03%
```

```python
# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss')
loss_hist_ = loss_hist[1::100] # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(train_acc_hist, '-o', label='Training')
plt.plot(val_acc_hist, '-o', label='Validation')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



Different Optimizers and Regularization Techniques

There are several more advanced optimizers than vanilla SGD, and there are many regularization tricks. You'll implement them in this section.
Please complete the TODOs in the `lib/optim.py`.

SGD + Weight Decay [2pt]

The update rule of SGD plus weigh decay is as shown below:
\begin{align}
\theta{t+1} &= \theta_t - \eta \nabla{\theta}J(\theta_t) - \lambda \theta_t
\end{align}
Update the `SGD()` function in `lib/optim.py`, and also incorporate weight decay options.

```
%reload_ext autoreload
```

```
%reload_ext autoreload

# Test the implementation of SGD with Momentum
seed = 1234
np.random.seed(seed=seed)

N, D = 4, 5
test_sgd = sequential(fc(N, D, name="sgd_fc"))

w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)

test_sgd.layers[0].params = {"sgd_fc_w": w}
test_sgd.layers[0].grads = {"sgd_fc_w": dw}


test_sgd_wd = SGD(test_sgd, 1e-3, 1e-4)
test_sgd_wd.step()

updated_w = test_sgd.layers[0].params["sgd_fc_w"]


expected_updated_w = np.asarray([
       [-0.39936   , -0.34678632, -0.29421263, -0.24163895, -0.18906526],
       [-0.13649158, -0.08391789, -0.03134421,  0.02122947,  0.07380316],
       [ 0.12637684,  0.17895053,  0.23152421,  0.28409789,  0.33667158],
       [ 0.38924526,  0.44181895,  0.49439263,  0.54696632,  0.59954   ]])


print ('The following errors should be around or less than 1e-6')
print ('updated_w error: ', rel_error(updated_w, expected_updated_w))
```

```
The following errors should be around or less than 1e-6
updated_w error:  8.677112905190533e-08
```

The accuracy meets the requirements.

Comparing SGD and SGD with Weight Decay [2pt]

Run the following code block to train a multi-layer fully connected network with both SGD and SGD plus Weight Decay.
You are expected to see Weight Decay have better validation accuracy than vinilla SGD.

```
seed = 1234
```

```
# Arrange a small data
num_train = 20000
small_data_dict = {
    "data_train": (data["data_train"][:num_train], data["labels_train"][:num_train]),
    "data_val": (data["data_val"], data["labels_val"]),
    "data_test": (data["data_test"], data["labels_test"])
}

reset_seed(seed=seed)
model_sgd      = FullyConnectedNetwork()
loss_f_sgd     = cross_entropy()
optimizer_sgd  = SGD(model_sgd.net, 0.01)
print ("Training with Vanilla SGD...")
results_sgd = train_net(small_data_dict, model_sgd, loss_f_sgd, optimizer_sgd,
batch_size=100,
                        max_epochs=50, show_every=10000, verbose=True)

reset_seed(seed=seed)
model_sgdw     = FullyConnectedNetwork()
loss_f_sgdw    = cross_entropy()
optimizer_sgdw = SGD(model_sgdw.net, 0.01, 1e-4)
print ("\nTraining with SGD plus Weight Decay...")
results_sgdw = train_net(small_data_dict, model_sgdw, loss_f_sgdw, optimizer_sgdw,
batch_size=100,
                         max_epochs=50, show_every=10000, verbose=True)

opt_params_sgd,  loss_hist_sgd,  train_acc_hist_sgd,  val_acc_hist_sgd  = results_sgd
opt_params_sgdw, loss_hist_sgdw, train_acc_hist_sgdw, val_acc_hist_sgdw = results_sgdw

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd, 'o', label="Vanilla SGD")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd, '-o', label="Vanilla SGD")
plt.subplot(3, 1, 3)
```

```
plt.plot(val_acc_hist_sgd, '-o', label="Vanilla SGD")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgdw, 'o', label="SGD with Weight Decay")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgdw, '-o', label="SGD with Weight Decay")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgdw, '-o', label="SGD with Weight Decay")

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
  2%|          | 4/200 [00:00<00:05, 37.11it/s]

Training with Vanilla SGD...
(Iteration 1 / 10000) Average loss: 3.333215453908898
```

```
100%|██████████| 200/200 [00:05<00:00, 39.42it/s]
  2%|          | 4/200 [00:00<00:05, 37.64it/s]

(Epoch 1 / 50) Training Accuracy: 0.15095, Validation Accuracy: 0.1474
```

```
100%|██████████| 200/200 [00:04<00:00, 40.03it/s]
  2%|          | 3/200 [00:00<00:07, 27.42it/s]

(Epoch 2 / 50) Training Accuracy: 0.18815, Validation Accuracy: 0.1805
```

```
100%|██████████| 200/200 [00:04<00:00, 40.44it/s]
  2%|          | 4/200 [00:00<00:04, 39.21it/s]

(Epoch 3 / 50) Training Accuracy: 0.2107, Validation Accuracy: 0.2029
```

```
100%|██████████| 200/200 [00:04<00:00, 40.63it/s]
  2%|▏         | 5/200 [00:00<00:05, 36.40it/s]

(Epoch 4 / 50) Training Accuracy: 0.2314, Validation Accuracy: 0.212
```

```
100%|██████████| 200/200 [00:05<00:00, 39.98it/s]
```

```
  2%|▏         | 5/200 [00:00<00:04, 40.53it/s]

(Epoch 5 / 50) Training Accuracy: 0.23915, Validation Accuracy: 0.2197
```

```
100%|██████████| 200/200 [00:04<00:00, 41.34it/s]
  2%|▏         | 5/200 [00:00<00:04, 41.35it/s]

(Epoch 6 / 50) Training Accuracy: 0.2552, Validation Accuracy: 0.2298
```

```
100%|██████████| 200/200 [00:04<00:00, 40.54it/s]
  2%|▏         | 5/200 [00:00<00:04, 41.68it/s]

(Epoch 7 / 50) Training Accuracy: 0.26645, Validation Accuracy: 0.2403
```

```
100%|██████████| 200/200 [00:04<00:00, 41.91it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.02it/s]

(Epoch 8 / 50) Training Accuracy: 0.27555, Validation Accuracy: 0.2414
```

```
100%|██████████| 200/200 [00:04<00:00, 40.87it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.03it/s]

(Epoch 9 / 50) Training Accuracy: 0.28185, Validation Accuracy: 0.2413
```

```
100%|██████████| 200/200 [00:04<00:00, 41.83it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.12it/s]

(Epoch 10 / 50) Training Accuracy: 0.2944, Validation Accuracy: 0.252
```

```
100%|██████████| 200/200 [00:04<00:00, 41.77it/s]
  2%|▏         | 3/200 [00:00<00:07, 27.77it/s]

(Epoch 11 / 50) Training Accuracy: 0.29735, Validation Accuracy: 0.2543
```

```
100%|██████████| 200/200 [00:04<00:00, 40.62it/s]
  2%|▏         | 5/200 [00:00<00:04, 40.58it/s]

(Epoch 12 / 50) Training Accuracy: 0.3021, Validation Accuracy: 0.2587
```

```
100%|██████████| 200/200 [00:04<00:00, 41.00it/s]
```

```
  2%|▏         | 3/200 [00:00<00:08, 24.13it/s]

(Epoch 13 / 50) Training Accuracy: 0.31105, Validation Accuracy: 0.2641
```

```
100%|██████████| 200/200 [00:04<00:00, 41.42it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.89it/s]

(Epoch 14 / 50) Training Accuracy: 0.3168, Validation Accuracy: 0.2653
```

```
100%|██████████| 200/200 [00:04<00:00, 42.58it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.22it/s]

(Epoch 15 / 50) Training Accuracy: 0.3217, Validation Accuracy: 0.2681
```

```
100%|██████████| 200/200 [00:04<00:00, 40.99it/s]
  2%|▏         | 4/200 [00:00<00:06, 29.74it/s]

(Epoch 16 / 50) Training Accuracy: 0.3307, Validation Accuracy: 0.2699
```

```
100%|██████████| 200/200 [00:04<00:00, 41.60it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.75it/s]

(Epoch 17 / 50) Training Accuracy: 0.33835, Validation Accuracy: 0.2696
```

```
100%|██████████| 200/200 [00:04<00:00, 41.90it/s]
  2%|▏         | 3/200 [00:00<00:06, 28.64it/s]

(Epoch 18 / 50) Training Accuracy: 0.34565, Validation Accuracy: 0.2737
```

```
100%|██████████| 200/200 [00:04<00:00, 41.51it/s]
  2%|▎         | 5/200 [00:00<00:04, 41.90it/s]

(Epoch 19 / 50) Training Accuracy: 0.3495, Validation Accuracy: 0.2729
```

```
100%|██████████| 200/200 [00:04<00:00, 42.16it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.75it/s]

(Epoch 20 / 50) Training Accuracy: 0.35565, Validation Accuracy: 0.2758
```

```
100%|██████████| 200/200 [00:04<00:00, 42.33it/s]
```

```
  2%|▏         | 5/200 [00:00<00:04, 42.90it/s]

(Epoch 21 / 50) Training Accuracy: 0.35825, Validation Accuracy: 0.2729
```

```
100%|██████████| 200/200 [00:04<00:00, 42.41it/s]
  2%|▏         | 4/200 [00:00<00:05, 39.14it/s]

(Epoch 22 / 50) Training Accuracy: 0.36895, Validation Accuracy: 0.278
```

```
100%|██████████| 200/200 [00:04<00:00, 41.45it/s]
  2%|▏         | 3/200 [00:00<00:06, 29.83it/s]

(Epoch 23 / 50) Training Accuracy: 0.3734, Validation Accuracy: 0.2783
```

```
100%|██████████| 200/200 [00:04<00:00, 41.46it/s]
  2%|▏         | 5/200 [00:00<00:04, 39.53it/s]

(Epoch 24 / 50) Training Accuracy: 0.3756, Validation Accuracy: 0.2768
```

```
100%|██████████| 200/200 [00:04<00:00, 41.39it/s]
  2%|▏         | 5/200 [00:00<00:04, 43.04it/s]

(Epoch 25 / 50) Training Accuracy: 0.38495, Validation Accuracy: 0.278
```

```
100%|██████████| 200/200 [00:04<00:00, 41.28it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.49it/s]

(Epoch 26 / 50) Training Accuracy: 0.38415, Validation Accuracy: 0.2757
```

```
100%|██████████| 200/200 [00:04<00:00, 41.92it/s]
  2%|▏         | 5/200 [00:00<00:04, 43.15it/s]

(Epoch 27 / 50) Training Accuracy: 0.40365, Validation Accuracy: 0.2804
```

```
100%|██████████| 200/200 [00:04<00:00, 42.40it/s]
  2%|▏         | 5/200 [00:00<00:04, 40.48it/s]

(Epoch 28 / 50) Training Accuracy: 0.40105, Validation Accuracy: 0.2812
```

```
100%|██████████| 200/200 [00:04<00:00, 42.89it/s]
```

```
  2%|▏         | 5/200 [00:00<00:04, 42.89it/s]

(Epoch 29 / 50) Training Accuracy: 0.40885, Validation Accuracy: 0.2773
```

```
100%|██████████| 200/200 [00:04<00:00, 41.80it/s]
  2%|▏         | 4/200 [00:00<00:05, 33.56it/s]

(Epoch 30 / 50) Training Accuracy: 0.4163, Validation Accuracy: 0.2803
```

```
100%|██████████| 200/200 [00:04<00:00, 41.56it/s]
  2%|▎         | 5/200 [00:00<00:04, 41.18it/s]

(Epoch 31 / 50) Training Accuracy: 0.41745, Validation Accuracy: 0.2838
```

```
100%|██████████| 200/200 [00:04<00:00, 41.72it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.84it/s]

(Epoch 32 / 50) Training Accuracy: 0.42125, Validation Accuracy: 0.2758
```

```
100%|██████████| 200/200 [00:04<00:00, 41.34it/s]
  2%|▏         | 4/200 [00:00<00:06, 31.72it/s]

(Epoch 33 / 50) Training Accuracy: 0.433, Validation Accuracy: 0.2777
```

```
100%|██████████| 200/200 [00:04<00:00, 41.63it/s]
  2%|▎         | 5/200 [00:00<00:04, 43.08it/s]

(Epoch 34 / 50) Training Accuracy: 0.4322, Validation Accuracy: 0.2782
```

```
100%|██████████| 200/200 [00:04<00:00, 41.11it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.62it/s]

(Epoch 35 / 50) Training Accuracy: 0.44095, Validation Accuracy: 0.2753
```

```
100%|██████████| 200/200 [00:04<00:00, 43.14it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.46it/s]

(Epoch 36 / 50) Training Accuracy: 0.4517, Validation Accuracy: 0.2783
```

```
100%|██████████| 200/200 [00:04<00:00, 40.66it/s]
```

```
  2%|▌         | 5/200 [00:00<00:04, 42.57it/s]

(Epoch 37 / 50) Training Accuracy: 0.4583, Validation Accuracy: 0.2759
```

```
100%|██████████| 200/200 [00:04<00:00, 42.51it/s]
  2%|▏         | 4/200 [00:00<00:05, 38.97it/s]

(Epoch 38 / 50) Training Accuracy: 0.4637, Validation Accuracy: 0.2815
```

```
100%|██████████| 200/200 [00:04<00:00, 41.75it/s]
  2%|▌         | 5/200 [00:00<00:04, 41.62it/s]

(Epoch 39 / 50) Training Accuracy: 0.4642, Validation Accuracy: 0.2808
```

```
100%|██████████| 200/200 [00:04<00:00, 41.40it/s]
  2%|▏         | 4/200 [00:00<00:05, 37.82it/s]

(Epoch 40 / 50) Training Accuracy: 0.47055, Validation Accuracy: 0.2784
```

```
100%|██████████| 200/200 [00:04<00:00, 42.77it/s]
  2%|▌         | 5/200 [00:00<00:04, 42.78it/s]

(Epoch 41 / 50) Training Accuracy: 0.4684, Validation Accuracy: 0.2747
```

```
100%|██████████| 200/200 [00:04<00:00, 42.92it/s]
  2%|▏         | 4/200 [00:00<00:05, 34.64it/s]

(Epoch 42 / 50) Training Accuracy: 0.4795, Validation Accuracy: 0.2758
```

```
100%|██████████| 200/200 [00:04<00:00, 42.68it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.29it/s]

(Epoch 43 / 50) Training Accuracy: 0.48745, Validation Accuracy: 0.2793
```

```
100%|██████████| 200/200 [00:04<00:00, 42.74it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.38it/s]

(Epoch 44 / 50) Training Accuracy: 0.49715, Validation Accuracy: 0.2751
```

```
100%|██████████| 200/200 [00:04<00:00, 43.23it/s]
```

```
  2%|▌         | 5/200 [00:00<00:04, 43.37it/s]

(Epoch 45 / 50) Training Accuracy: 0.49545, Validation Accuracy: 0.2736
```

```
100%|██████████| 200/200 [00:04<00:00, 43.51it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.32it/s]

(Epoch 46 / 50) Training Accuracy: 0.50175, Validation Accuracy: 0.2767
```

```
100%|██████████| 200/200 [00:04<00:00, 42.66it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.24it/s]

(Epoch 47 / 50) Training Accuracy: 0.51565, Validation Accuracy: 0.2704
```

```
100%|██████████| 200/200 [00:04<00:00, 43.63it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.22it/s]

(Epoch 48 / 50) Training Accuracy: 0.51875, Validation Accuracy: 0.2786
```

```
100%|██████████| 200/200 [00:04<00:00, 42.95it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.41it/s]

(Epoch 49 / 50) Training Accuracy: 0.5235, Validation Accuracy: 0.2818
```

```
100%|██████████| 200/200 [00:04<00:00, 43.02it/s]
  2%|▌         | 5/200 [00:00<00:04, 41.12it/s]

(Epoch 50 / 50) Training Accuracy: 0.52375, Validation Accuracy: 0.2778

Training with SGD plus Weight Decay...
(Iteration 1 / 10000) Average loss: 3.333215453908898
```

```
100%|██████████| 200/200 [00:04<00:00, 41.51it/s]
  2%|▍         | 4/200 [00:00<00:05, 38.40it/s]

(Epoch 1 / 50) Training Accuracy: 0.148, Validation Accuracy: 0.1458
```

```
100%|██████████| 200/200 [00:04<00:00, 41.73it/s]
```

```
  2%|▍         | 5/200 [00:00<00:04, 42.40it/s]

(Epoch 2 / 50) Training Accuracy: 0.186, Validation Accuracy: 0.1822
```

```
100%|██████████| 200/200 [00:05<00:00, 36.97it/s]
  2%|▍         | 5/200 [00:00<00:04, 41.67it/s]

(Epoch 3 / 50) Training Accuracy: 0.2073, Validation Accuracy: 0.2027
```

```
100%|██████████| 200/200 [00:04<00:00, 41.71it/s]
  2%|▍         | 5/200 [00:00<00:04, 42.29it/s]

(Epoch 4 / 50) Training Accuracy: 0.22575, Validation Accuracy: 0.2101
```

```
100%|██████████| 200/200 [00:04<00:00, 42.41it/s]
  2%|▏         | 4/200 [00:00<00:04, 39.77it/s]

(Epoch 5 / 50) Training Accuracy: 0.2345, Validation Accuracy: 0.2223
```

```
100%|██████████| 200/200 [00:04<00:00, 43.05it/s]
  2%|▍         | 5/200 [00:00<00:04, 43.34it/s]

(Epoch 6 / 50) Training Accuracy: 0.24915, Validation Accuracy: 0.2338
```

```
100%|██████████| 200/200 [00:04<00:00, 43.60it/s]
  2%|▍         | 5/200 [00:00<00:04, 43.55it/s]

(Epoch 7 / 50) Training Accuracy: 0.2584, Validation Accuracy: 0.2451
```

```
100%|██████████| 200/200 [00:04<00:00, 43.31it/s]
  2%|▍         | 5/200 [00:00<00:04, 43.79it/s]

(Epoch 8 / 50) Training Accuracy: 0.2651, Validation Accuracy: 0.2488
```

```
100%|██████████| 200/200 [00:04<00:00, 43.74it/s]
  2%|▍         | 5/200 [00:00<00:04, 43.86it/s]

(Epoch 9 / 50) Training Accuracy: 0.2648, Validation Accuracy: 0.2471
```

```
100%|██████████| 200/200 [00:04<00:00, 43.98it/s]
```

```
  2%|▏         | 5/200 [00:00<00:04, 42.20it/s]

(Epoch 10 / 50) Training Accuracy: 0.27685, Validation Accuracy: 0.2558
```

```
100%|██████████| 200/200 [00:04<00:00, 40.83it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.36it/s]

(Epoch 11 / 50) Training Accuracy: 0.2792, Validation Accuracy: 0.2583
```

```
100%|██████████| 200/200 [00:04<00:00, 44.39it/s]
  2%|▏         | 5/200 [00:00<00:04, 44.60it/s]

(Epoch 12 / 50) Training Accuracy: 0.28575, Validation Accuracy: 0.2646
```

```
100%|██████████| 200/200 [00:04<00:00, 45.11it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.00it/s]

(Epoch 13 / 50) Training Accuracy: 0.2879, Validation Accuracy: 0.2657
```

```
100%|██████████| 200/200 [00:04<00:00, 44.46it/s]
  2%|▏         | 5/200 [00:00<00:04, 45.35it/s]

(Epoch 14 / 50) Training Accuracy: 0.28865, Validation Accuracy: 0.2664
```

```
100%|██████████| 200/200 [00:04<00:00, 45.03it/s]
  2%|▏         | 5/200 [00:00<00:04, 45.27it/s]

(Epoch 15 / 50) Training Accuracy: 0.29545, Validation Accuracy: 0.2705
```

```
100%|██████████| 200/200 [00:04<00:00, 45.76it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.75it/s]

(Epoch 16 / 50) Training Accuracy: 0.2964, Validation Accuracy: 0.2737
```

```
100%|██████████| 200/200 [00:04<00:00, 45.67it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.92it/s]

(Epoch 17 / 50) Training Accuracy: 0.30345, Validation Accuracy: 0.2752
```

```
100%|██████████| 200/200 [00:04<00:00, 45.08it/s]
```

```
  2%|▌         | 5/200 [00:00<00:05, 38.18it/s]

(Epoch 18 / 50) Training Accuracy: 0.30555, Validation Accuracy: 0.276
```

```
100%|██████████| 200/200 [00:04<00:00, 45.83it/s]
  2%|▌         | 5/200 [00:00<00:04, 43.12it/s]

(Epoch 19 / 50) Training Accuracy: 0.30715, Validation Accuracy: 0.2821
```

```
100%|██████████| 200/200 [00:04<00:00, 43.60it/s]
  2%|▌         | 5/200 [00:00<00:04, 46.19it/s]

(Epoch 20 / 50) Training Accuracy: 0.31265, Validation Accuracy: 0.2799
```

```
100%|██████████| 200/200 [00:04<00:00, 45.88it/s]
  2%|▌         | 5/200 [00:00<00:04, 46.52it/s]

(Epoch 21 / 50) Training Accuracy: 0.31315, Validation Accuracy: 0.2787
```

```
100%|██████████| 200/200 [00:04<00:00, 46.78it/s]
  2%|▌         | 5/200 [00:00<00:04, 46.97it/s]

(Epoch 22 / 50) Training Accuracy: 0.31755, Validation Accuracy: 0.2836
```

```
100%|██████████| 200/200 [00:04<00:00, 46.20it/s]
  2%|▌         | 5/200 [00:00<00:04, 47.83it/s]

(Epoch 23 / 50) Training Accuracy: 0.3192, Validation Accuracy: 0.2833
```

```
100%|██████████| 200/200 [00:04<00:00, 47.78it/s]
  2%|▌         | 5/200 [00:00<00:04, 48.05it/s]

(Epoch 24 / 50) Training Accuracy: 0.31905, Validation Accuracy: 0.2837
```

```
100%|██████████| 200/200 [00:04<00:00, 48.09it/s]
  2%|▌         | 5/200 [00:00<00:04, 48.13it/s]

(Epoch 25 / 50) Training Accuracy: 0.32525, Validation Accuracy: 0.2894
```

```
100%|██████████| 200/200 [00:04<00:00, 47.69it/s]
```

```
  2%|▏         | 5/200 [00:00<00:04, 48.01it/s]

(Epoch 26 / 50) Training Accuracy: 0.3238, Validation Accuracy: 0.2895
```

```
100%|██████████| 200/200 [00:04<00:00, 47.91it/s]
  2%|▏         | 5/200 [00:00<00:04, 45.23it/s]

(Epoch 27 / 50) Training Accuracy: 0.33645, Validation Accuracy: 0.2944
```

```
100%|██████████| 200/200 [00:04<00:00, 47.98it/s]
  2%|▏         | 5/200 [00:00<00:04, 48.02it/s]

(Epoch 28 / 50) Training Accuracy: 0.33645, Validation Accuracy: 0.2941
```

```
100%|██████████| 200/200 [00:04<00:00, 46.83it/s]
  2%|▏         | 5/200 [00:00<00:04, 48.35it/s]

(Epoch 29 / 50) Training Accuracy: 0.33695, Validation Accuracy: 0.2953
```

```
100%|██████████| 200/200 [00:04<00:00, 48.47it/s]
  2%|▏         | 5/200 [00:00<00:04, 48.36it/s]

(Epoch 30 / 50) Training Accuracy: 0.3425, Validation Accuracy: 0.3
```

```
100%|██████████| 200/200 [00:04<00:00, 48.51it/s]
  2%|▏         | 5/200 [00:00<00:03, 48.89it/s]

(Epoch 31 / 50) Training Accuracy: 0.3406, Validation Accuracy: 0.2982
```

```
100%|██████████| 200/200 [00:04<00:00, 48.86it/s]
  2%|▏         | 5/200 [00:00<00:04, 46.17it/s]

(Epoch 32 / 50) Training Accuracy: 0.34505, Validation Accuracy: 0.2949
```

```
100%|██████████| 200/200 [00:04<00:00, 48.79it/s]
  2%|▏         | 3/200 [00:00<00:07, 27.99it/s]

(Epoch 33 / 50) Training Accuracy: 0.34595, Validation Accuracy: 0.3011
```

```
100%|██████████| 200/200 [00:04<00:00, 49.26it/s]
```

```
  2%|▏         | 5/200 [00:00<00:03, 49.47it/s]

(Epoch 34 / 50) Training Accuracy: 0.34755, Validation Accuracy: 0.301
```

```
100%|██████████| 200/200 [00:04<00:00, 49.37it/s]
  2%|▏         | 5/200 [00:00<00:03, 49.49it/s]

(Epoch 35 / 50) Training Accuracy: 0.3548, Validation Accuracy: 0.3012
```

```
100%|██████████| 200/200 [00:04<00:00, 49.66it/s]
  2%|▏         | 5/200 [00:00<00:04, 42.88it/s]

(Epoch 36 / 50) Training Accuracy: 0.3552, Validation Accuracy: 0.2995
```

```
100%|██████████| 200/200 [00:04<00:00, 49.19it/s]
  2%|▏         | 5/200 [00:00<00:04, 46.65it/s]

(Epoch 37 / 50) Training Accuracy: 0.35525, Validation Accuracy: 0.3034
```

```
100%|██████████| 200/200 [00:03<00:00, 50.48it/s]
  3%|▎         | 6/200 [00:00<00:03, 50.56it/s]

(Epoch 38 / 50) Training Accuracy: 0.3593, Validation Accuracy: 0.3017
```

```
100%|██████████| 200/200 [00:04<00:00, 42.92it/s]
  3%|▎         | 6/200 [00:00<00:03, 50.78it/s]

(Epoch 39 / 50) Training Accuracy: 0.3648, Validation Accuracy: 0.3048
```

```
100%|██████████| 200/200 [00:03<00:00, 50.48it/s]
  3%|▎         | 6/200 [00:00<00:04, 48.48it/s]

(Epoch 40 / 50) Training Accuracy: 0.36665, Validation Accuracy: 0.311
```

```
100%|██████████| 200/200 [00:03<00:00, 50.48it/s]
  2%|▏         | 5/200 [00:00<00:04, 46.74it/s]

(Epoch 41 / 50) Training Accuracy: 0.35765, Validation Accuracy: 0.3068
```

```
100%|██████████| 200/200 [00:03<00:00, 50.86it/s]
```

```
  2%|▌         | 5/200 [00:00<00:04, 47.62it/s]

(Epoch 42 / 50) Training Accuracy: 0.36375, Validation Accuracy: 0.302
```

```
100%|██████████| 200/200 [00:03<00:00, 50.82it/s]
  2%|▌         | 5/200 [00:00<00:04, 48.10it/s]

(Epoch 43 / 50) Training Accuracy: 0.3702, Validation Accuracy: 0.3062
```

```
100%|██████████| 200/200 [00:03<00:00, 52.16it/s]
  3%|▌         | 6/200 [00:00<00:03, 51.41it/s]

(Epoch 44 / 50) Training Accuracy: 0.37215, Validation Accuracy: 0.306
```

```
100%|██████████| 200/200 [00:03<00:00, 51.18it/s]
  3%|▌         | 6/200 [00:00<00:03, 52.16it/s]

(Epoch 45 / 50) Training Accuracy: 0.37475, Validation Accuracy: 0.3037
```

```
100%|██████████| 200/200 [00:03<00:00, 52.11it/s]
  3%|▌         | 6/200 [00:00<00:03, 52.08it/s]

(Epoch 46 / 50) Training Accuracy: 0.37205, Validation Accuracy: 0.3089
```

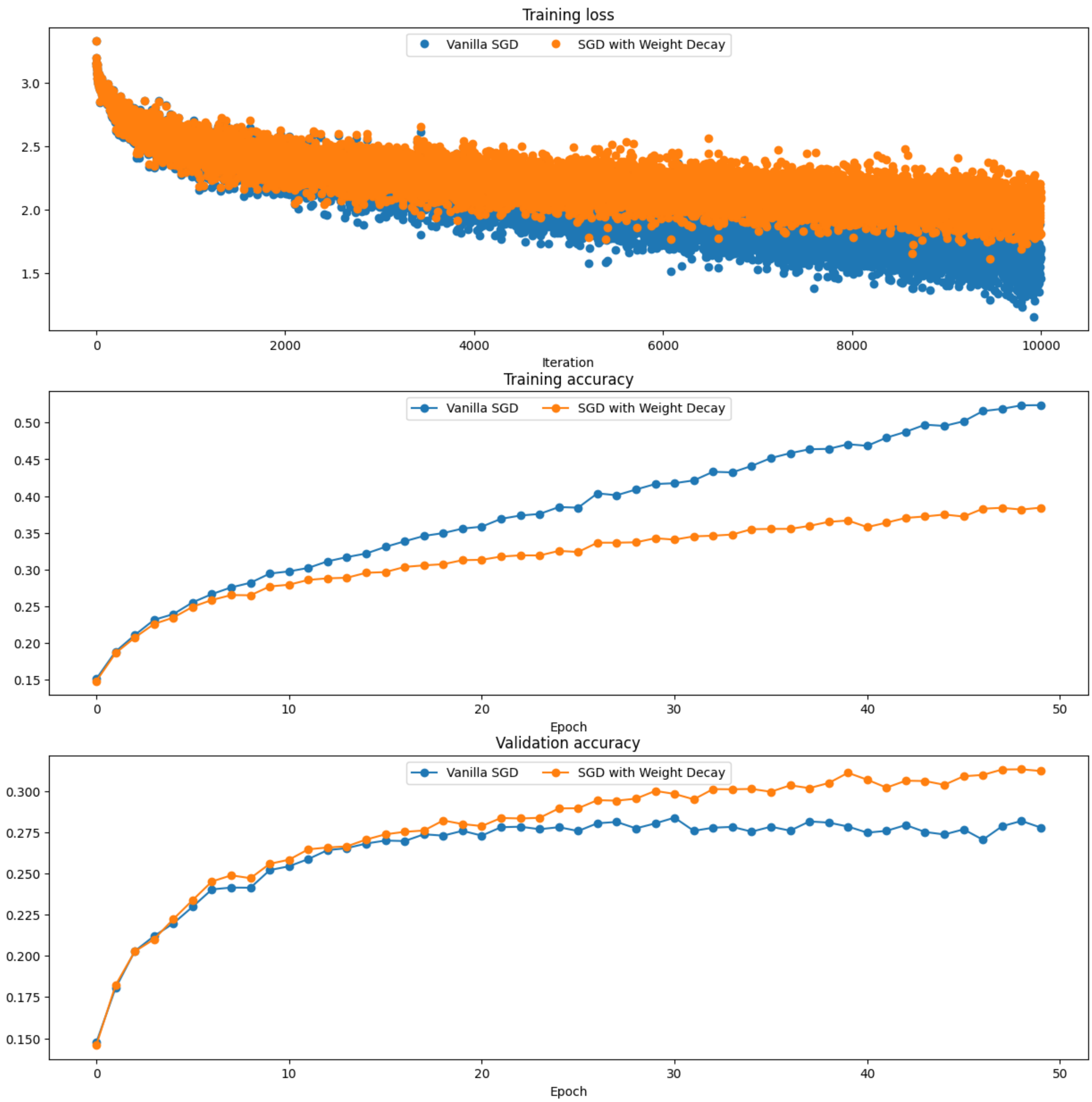
```
100%|██████████| 200/200 [00:03<00:00, 51.81it/s]
  3%|▌         | 6/200 [00:00<00:03, 52.40it/s]

(Epoch 47 / 50) Training Accuracy: 0.3827, Validation Accuracy: 0.3097
```

```
100%|██████████| 200/200 [00:03<00:00, 51.65it/s]
  2%|▏         | 4/200 [00:00<00:05, 37.04it/s]

(Epoch 48 / 50) Training Accuracy: 0.38395, Validation Accuracy: 0.313
```

```
100%|██████████| 200/200 [00:03<00:00, 52.26it/s]
  3%|▌         | 6/200 [00:00<00:03, 52.91it/s]

(Epoch 49 / 50) Training Accuracy: 0.38155, Validation Accuracy: 0.3131
```

```
100%|██████████| 200/200 [00:03<00:00, 53.13it/s]
```

```
(Epoch 50 / 50) Training Accuracy: 0.38415, Validation Accuracy: 0.3121
```



As shown in figure, SGD with weight decay has relatively higher loss, lower training accuracy, but has higher validation accuracy than vanilla SGD.

SGD with L1 Regularization [2pts]

With L1 Regularization, your regularized loss becomes $\tilde{J}_{\ell_1}(\theta)$ and it's defined as

$$\tilde{J}_{\ell_1}(\theta) = J(\theta) + \lambda \|\theta\|_{\ell_1}$$

where

$$\|\theta\|_{\ell_1} = \sum_{l=1}^{n} \sum_{k=1}^{n_l} |\theta_{l,k}|$$

Please implmemt TODO block of `apply_l1_regularization` in `lib/layer_utils`. Such regularization funcationality is called after gradient gathering in the `backward` process.

```
reset_seed(seed=seed)
model_sgd_l1     = FullyConnectedNetwork()
loss_f_sgd_l1    = cross_entropy()
optimizer_sgd_l1 = SGD(model_sgd_l1.net, 0.01)

print ("\nTraining with SGD plus L1 Regularization...")
results_sgd_l1 = train_net(small_data_dict, model_sgd_l1, loss_f_sgd_l1, optimizer_sgd_l1,
batch_size=100,
                         max_epochs=50, show_every=10000, verbose=True,
regularization="l1", reg_lambda=1e-3)

opt_params_sgd_l1, loss_hist_sgd_l1, train_acc_hist_sgd_l1, val_acc_hist_sgd_l1=
results_sgd_l1

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd, 'o', label="Vanilla SGD")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd, '-o', label="Vanilla SGD")

plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgd, '-o', label="Vanilla SGD")
```

```python
plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd_l1, 'o', label="SGD with L1 Regularization")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd_l1, '-o', label="SGD with L1 Regularization")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgd_l1, '-o', label="SGD with L1 Regularization")

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
  2%|          | 4/200 [00:00<00:05, 35.21it/s]
```

```
Training with SGD plus L1 Regularization...
(Iteration 1 / 10000) Average loss: 3.333215453908898
```

```
100%|██████████| 200/200 [00:05<00:00, 36.96it/s]
  2%|          | 4/200 [00:00<00:05, 37.35it/s]

(Epoch 1 / 50) Training Accuracy: 0.1491, Validation Accuracy: 0.1457
```

```
100%|██████████| 200/200 [00:05<00:00, 37.60it/s]
  2%|          | 4/200 [00:00<00:05, 37.53it/s]

(Epoch 2 / 50) Training Accuracy: 0.1854, Validation Accuracy: 0.1806
```

```
100%|██████████| 200/200 [00:05<00:00, 37.57it/s]
  2%|          | 4/200 [00:00<00:05, 38.02it/s]

(Epoch 3 / 50) Training Accuracy: 0.20755, Validation Accuracy: 0.2014
```

```
100%|██████████| 200/200 [00:05<00:00, 38.27it/s]
  2%|          | 4/200 [00:00<00:05, 38.16it/s]

(Epoch 4 / 50) Training Accuracy: 0.22465, Validation Accuracy: 0.2111
```

```
100%|██████████| 200/200 [00:05<00:00, 38.61it/s]
  2%|          | 4/200 [00:00<00:05, 38.24it/s]
```

```
  2%|▏         | 4/200 [00:00<00:05, 38.24it/s]

(Epoch 5 / 50) Training Accuracy: 0.2331, Validation Accuracy: 0.2212
```

```
100%|██████████| 200/200 [00:05<00:00, 38.85it/s]
  2%|▏         | 4/200 [00:00<00:05, 33.34it/s]

(Epoch 6 / 50) Training Accuracy: 0.24735, Validation Accuracy: 0.2337
```

```
100%|██████████| 200/200 [00:05<00:00, 38.98it/s]
  2%|▏         | 4/200 [00:00<00:05, 35.87it/s]

(Epoch 7 / 50) Training Accuracy: 0.25725, Validation Accuracy: 0.2395
```

```
100%|██████████| 200/200 [00:05<00:00, 38.31it/s]
  2%|▏         | 4/200 [00:00<00:05, 38.64it/s]

(Epoch 8 / 50) Training Accuracy: 0.26245, Validation Accuracy: 0.2431
```

```
100%|██████████| 200/200 [00:05<00:00, 39.56it/s]
  2%|▏         | 4/200 [00:00<00:04, 39.49it/s]

(Epoch 9 / 50) Training Accuracy: 0.26185, Validation Accuracy: 0.2449
```

```
100%|██████████| 200/200 [00:05<00:00, 39.89it/s]
  2%|▏         | 4/200 [00:00<00:04, 39.51it/s]

(Epoch 10 / 50) Training Accuracy: 0.27205, Validation Accuracy: 0.251
```

```
100%|██████████| 200/200 [00:05<00:00, 39.65it/s]
  2%|▏         | 4/200 [00:00<00:05, 36.93it/s]

(Epoch 11 / 50) Training Accuracy: 0.27515, Validation Accuracy: 0.2582
```

```
100%|██████████| 200/200 [00:05<00:00, 39.94it/s]
  2%|▎         | 5/200 [00:00<00:04, 40.50it/s]

(Epoch 12 / 50) Training Accuracy: 0.282, Validation Accuracy: 0.2606
```

```
100%|██████████| 200/200 [00:04<00:00, 40.11it/s]
  2%|▎         | 5/200 [00:00<00:04, 40.49it/s]
```

```
  2%|▌         | 5/200 [00:00<00:04, 40.49it/s]

(Epoch 13 / 50) Training Accuracy: 0.2838, Validation Accuracy: 0.267
```

```
100%|██████████| 200/200 [00:05<00:00, 39.89it/s]
  2%|▍         | 4/200 [00:00<00:04, 39.97it/s]

(Epoch 14 / 50) Training Accuracy: 0.28535, Validation Accuracy: 0.2645
```

```
100%|██████████| 200/200 [00:04<00:00, 40.60it/s]
  2%|▌         | 5/200 [00:00<00:04, 40.86it/s]

(Epoch 15 / 50) Training Accuracy: 0.2883, Validation Accuracy: 0.2655
```

```
100%|██████████| 200/200 [00:04<00:00, 41.19it/s]
  2%|▍         | 4/200 [00:00<00:06, 29.96it/s]

(Epoch 16 / 50) Training Accuracy: 0.2926, Validation Accuracy: 0.2676
```

```
100%|██████████| 200/200 [00:05<00:00, 38.91it/s]
  2%|▌         | 5/200 [00:00<00:04, 41.23it/s]

(Epoch 17 / 50) Training Accuracy: 0.296, Validation Accuracy: 0.2742
```

```
100%|██████████| 200/200 [00:04<00:00, 41.22it/s]
  2%|▌         | 5/200 [00:00<00:04, 39.27it/s]

(Epoch 18 / 50) Training Accuracy: 0.2991, Validation Accuracy: 0.2715
```

```
100%|██████████| 200/200 [00:04<00:00, 41.31it/s]
  2%|▌         | 5/200 [00:00<00:04, 41.75it/s]

(Epoch 19 / 50) Training Accuracy: 0.30085, Validation Accuracy: 0.2734
```

```
100%|██████████| 200/200 [00:04<00:00, 41.40it/s]
  2%|▌         | 5/200 [00:00<00:04, 41.60it/s]

(Epoch 20 / 50) Training Accuracy: 0.30465, Validation Accuracy: 0.2756
```

```
100%|██████████| 200/200 [00:04<00:00, 41.31it/s]
  2%|▌         | 5/200 [00:00<00:04, 39.34it/s]
```

```
  2%|▌          | 5/200 [00:00<00:04, 39.54it/s]

(Epoch 21 / 50) Training Accuracy: 0.30195, Validation Accuracy: 0.271
```

```
100%|██████████| 200/200 [00:04<00:00, 41.28it/s]
  2%|▌          | 5/200 [00:00<00:04, 41.81it/s]

(Epoch 22 / 50) Training Accuracy: 0.3069, Validation Accuracy: 0.2785
```

```
100%|██████████| 200/200 [00:04<00:00, 41.25it/s]
  2%|▌          | 5/200 [00:00<00:04, 42.22it/s]

(Epoch 23 / 50) Training Accuracy: 0.30985, Validation Accuracy: 0.2776
```

```
100%|██████████| 200/200 [00:04<00:00, 42.05it/s]
  2%|▏          | 4/200 [00:00<00:09, 20.69it/s]

(Epoch 24 / 50) Training Accuracy: 0.30745, Validation Accuracy: 0.2768
```

```
100%|██████████| 200/200 [00:05<00:00, 39.80it/s]
  2%|▌          | 5/200 [00:00<00:04, 43.66it/s]

(Epoch 25 / 50) Training Accuracy: 0.3103, Validation Accuracy: 0.2814
```

```
100%|██████████| 200/200 [00:04<00:00, 42.97it/s]
  2%|▌          | 5/200 [00:00<00:04, 43.64it/s]

(Epoch 26 / 50) Training Accuracy: 0.3091, Validation Accuracy: 0.2778
```

```
100%|██████████| 200/200 [00:04<00:00, 43.37it/s]
  2%|▌          | 5/200 [00:00<00:04, 43.58it/s]

(Epoch 27 / 50) Training Accuracy: 0.31465, Validation Accuracy: 0.2853
```

```
100%|██████████| 200/200 [00:04<00:00, 44.03it/s]
  2%|▌          | 5/200 [00:00<00:04, 43.08it/s]

(Epoch 28 / 50) Training Accuracy: 0.31695, Validation Accuracy: 0.2851
```

```
100%|██████████| 200/200 [00:04<00:00, 43.76it/s]
  2%|▌          | 5/200 [00:00<00:04, 44.27it/s]
```

```
  2%|▌          | 5/200 [00:00<00:04, [illegible]it/s]

(Epoch 29 / 50) Training Accuracy: 0.3157, Validation Accuracy: 0.2819
```

```
100%|██████████| 200/200 [00:04<00:00, 42.39it/s]
  2%|▌          | 5/200 [00:00<00:04, 44.28it/s]

(Epoch 30 / 50) Training Accuracy: 0.31705, Validation Accuracy: 0.2901
```

```
100%|██████████| 200/200 [00:04<00:00, 44.54it/s]
  2%|▌          | 5/200 [00:00<00:04, 44.41it/s]

(Epoch 31 / 50) Training Accuracy: 0.3152, Validation Accuracy: 0.2835
```

```
100%|██████████| 200/200 [00:04<00:00, 44.13it/s]
  2%|▌          | 5/200 [00:00<00:04, 44.69it/s]

(Epoch 32 / 50) Training Accuracy: 0.3168, Validation Accuracy: 0.2843
```

```
100%|██████████| 200/200 [00:04<00:00, 44.12it/s]
  2%|▌          | 5/200 [00:00<00:04, 41.48it/s]

(Epoch 33 / 50) Training Accuracy: 0.31745, Validation Accuracy: 0.2843
```

```
100%|██████████| 200/200 [00:04<00:00, 44.08it/s]
  2%|▌          | 5/200 [00:00<00:04, 41.95it/s]

(Epoch 34 / 50) Training Accuracy: 0.31705, Validation Accuracy: 0.2855
```

```
100%|██████████| 200/200 [00:04<00:00, 44.49it/s]
  2%|▌          | 5/200 [00:00<00:04, 42.06it/s]

(Epoch 35 / 50) Training Accuracy: 0.32255, Validation Accuracy: 0.287
```

```
100%|██████████| 200/200 [00:04<00:00, 43.94it/s]
  2%|▌          | 5/200 [00:00<00:04, 44.84it/s]

(Epoch 36 / 50) Training Accuracy: 0.3215, Validation Accuracy: 0.2873
```

```
100%|██████████| 200/200 [00:04<00:00, 45.48it/s]
  2%|▌          | 5/200 [00:00<00:04, 45.47it/s]
```

```
(Epoch 37 / 50) Training Accuracy: 0.32235, Validation Accuracy: 0.2887
```

```
100%|██████████| 200/200 [00:04<00:00, 45.39it/s]
  2%|▎         | 5/200 [00:00<00:04, 45.38it/s]

(Epoch 38 / 50) Training Accuracy: 0.3196, Validation Accuracy: 0.2845
```

```
100%|██████████| 200/200 [00:04<00:00, 44.19it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.52it/s]

(Epoch 39 / 50) Training Accuracy: 0.32645, Validation Accuracy: 0.2928
```

```
100%|██████████| 200/200 [00:04<00:00, 44.52it/s]
  2%|▎         | 5/200 [00:00<00:04, 44.57it/s]

(Epoch 40 / 50) Training Accuracy: 0.32535, Validation Accuracy: 0.2926
```

```
100%|██████████| 200/200 [00:04<00:00, 44.31it/s]
  2%|▎         | 5/200 [00:00<00:04, 44.87it/s]

(Epoch 41 / 50) Training Accuracy: 0.3185, Validation Accuracy: 0.2867
```

```
100%|██████████| 200/200 [00:04<00:00, 43.29it/s]
  2%|▎         | 5/200 [00:00<00:04, 42.81it/s]

(Epoch 42 / 50) Training Accuracy: 0.3197, Validation Accuracy: 0.2841
```

```
100%|██████████| 200/200 [00:04<00:00, 43.90it/s]
  2%|▎         | 5/200 [00:00<00:04, 44.90it/s]

(Epoch 43 / 50) Training Accuracy: 0.32515, Validation Accuracy: 0.2906
```

```
100%|██████████| 200/200 [00:04<00:00, 44.92it/s]
  2%|▎         | 5/200 [00:00<00:04, 45.32it/s]

(Epoch 44 / 50) Training Accuracy: 0.3239, Validation Accuracy: 0.2868
```

```
100%|██████████| 200/200 [00:04<00:00, 45.46it/s]
  0%|          | 1/200 [00:00<00:21,  9.40it/s]
```

```
(Epoch 45 / 50) Training Accuracy: 0.32375, Validation Accuracy: 0.2884
```

```
100%|██████████| 200/200 [00:04<00:00, 45.55it/s]
  2%|▏         | 5/200 [00:00<00:04, 46.42it/s]

(Epoch 46 / 50) Training Accuracy: 0.3223, Validation Accuracy: 0.289
```

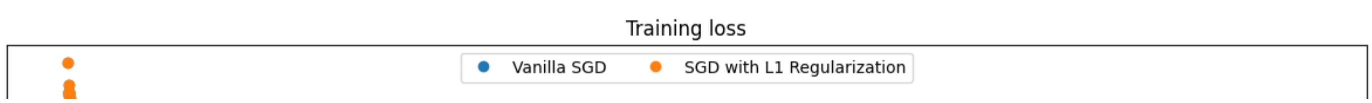
```
100%|██████████| 200/200 [00:04<00:00, 47.03it/s]
  2%|▏         | 5/200 [00:00<00:04, 43.48it/s]

(Epoch 47 / 50) Training Accuracy: 0.32585, Validation Accuracy: 0.2897
```

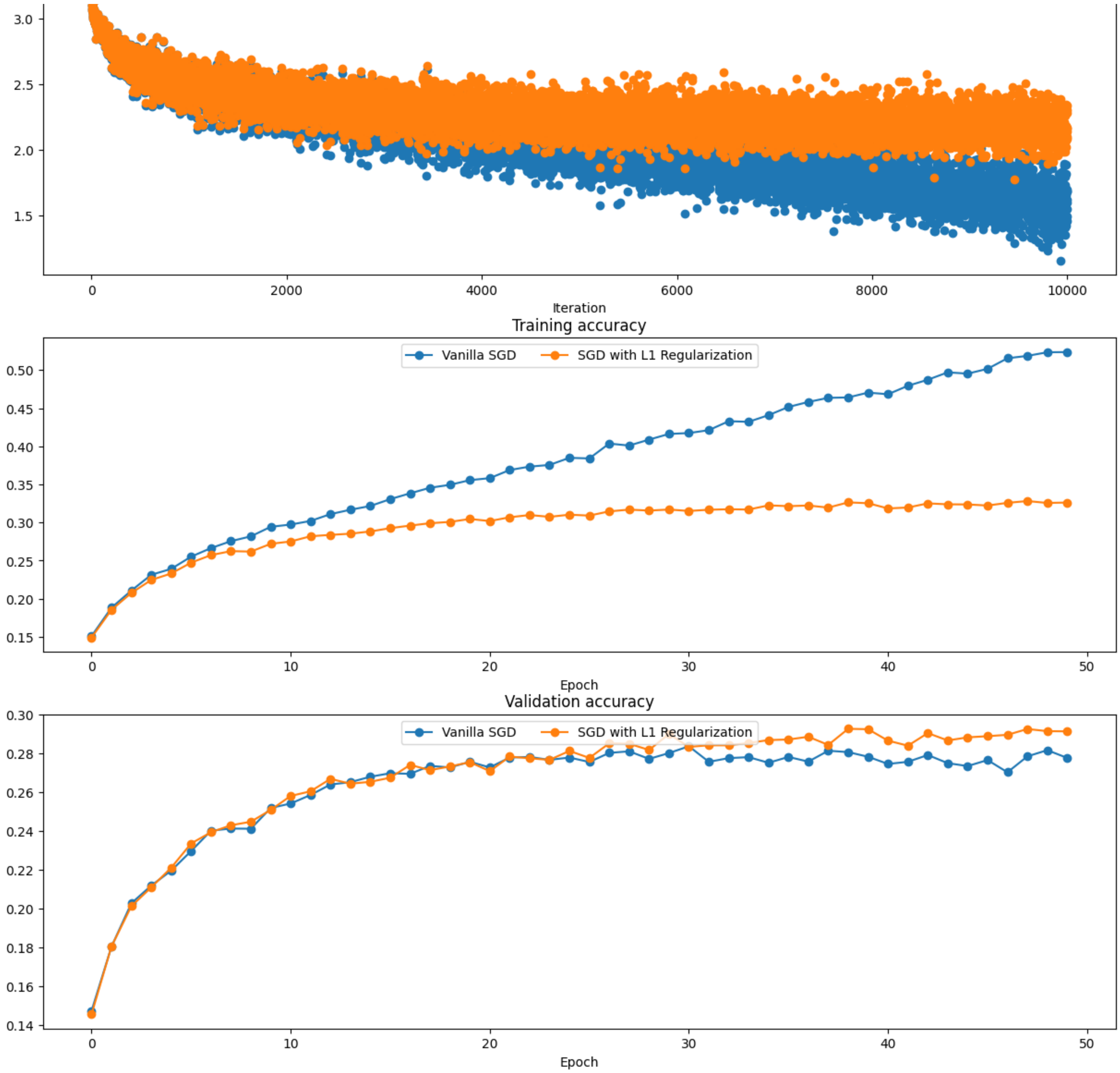
```
100%|██████████| 200/200 [00:04<00:00, 46.34it/s]
  2%|▏         | 5/200 [00:00<00:04, 46.84it/s]

(Epoch 48 / 50) Training Accuracy: 0.3282, Validation Accuracy: 0.2927
```

```
100%|██████████| 200/200 [00:04<00:00, 46.77it/s]
  2%|▏         | 5/200 [00:00<00:04, 46.67it/s]

(Epoch 49 / 50) Training Accuracy: 0.3257, Validation Accuracy: 0.2916
```

```
100%|██████████| 200/200 [00:04<00:00, 46.19it/s]
```

```
(Epoch 50 / 50) Training Accuracy: 0.32625, Validation Accuracy: 0.2915
```





As shown in figure, SGD with L1 regularization has relatively higher loss, lower training accuracy, but higher validation accuracy than vanilla SGD.

SGD with L2 Regularization [2pts]

With L2 Regularization, your regularized loss becomes $\tilde{J}_{\ell_2}(\theta)$ and it's defined as

$$\tilde{J}_{\ell_2}(\theta) = J(\theta) + \lambda\|\theta\|_{\ell_2}^2$$

where

where

$$\|\theta\|_{\ell_2}^2 = \sum_{l=1}^{n} \sum_{k=1}^{n_l} \theta_{l,k}^2$$

Similarly, implmemt TODO block of `apply_l2_regularization` in `lib/layer_utils`.
For SGD, you're also asked to find the λ for L2 Regularization such that it achives the EXACTLY SAME effect as weight decay in the previous cells. As a reminder, learning rate is the same as previously, and the weight decay paramter was 1e-4.

```
reset_seed(seed=seed)
model_sgd_l2     = FullyConnectedNetwork()
loss_f_sgd_l2    = cross_entropy()
optimizer_sgd_l2 = SGD(model_sgd_l2.net, 0.01)
######################################################################
#### Find lambda for L2 regularization so that                   ####
#### it achieves EXACTLY THE SAME learning curve as weight decay ####
# 2 * lambda * lr = weight_decay = 1e-4 -> lambda = 0.5e-2
l2_lambda = 0.5e-2
######################################################################

print ("\nTraining with SGD plus L2 Regularization...")
results_sgd_l2 = train_net(small_data_dict, model_sgd_l2, loss_f_sgd_l2, optimizer_sgd_l2,
batch_size=100,
                           max_epochs=50, show_every=10000, verbose=False,
regularization="l2", reg_lambda=l2_lambda)

opt_params_sgd_l2, loss_hist_sgd_l2, train_acc_hist_sgd_l2, val_acc_hist_sgd_l2 =
results_sgd_l2

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgdw, 'o', label="SGD with Weight Decay")

plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgdw, '-o', label="SGD with Weight Decay")
```

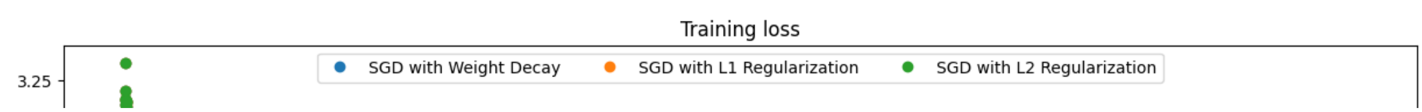
```
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgdw, '-o', label="SGD with Weight Decay")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd_l1, 'o', label="SGD with L1 Regularization")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd_l1, '-o', label="SGD with L1 Regularization")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgd_l1, '-o', label="SGD with L1 Regularization")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd_l2, 'o', label="SGD with L2 Regularization")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd_l2, '-o', label="SGD with L2 Regularization")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgd_l2, '-o', label="SGD with L2 Regularization")

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

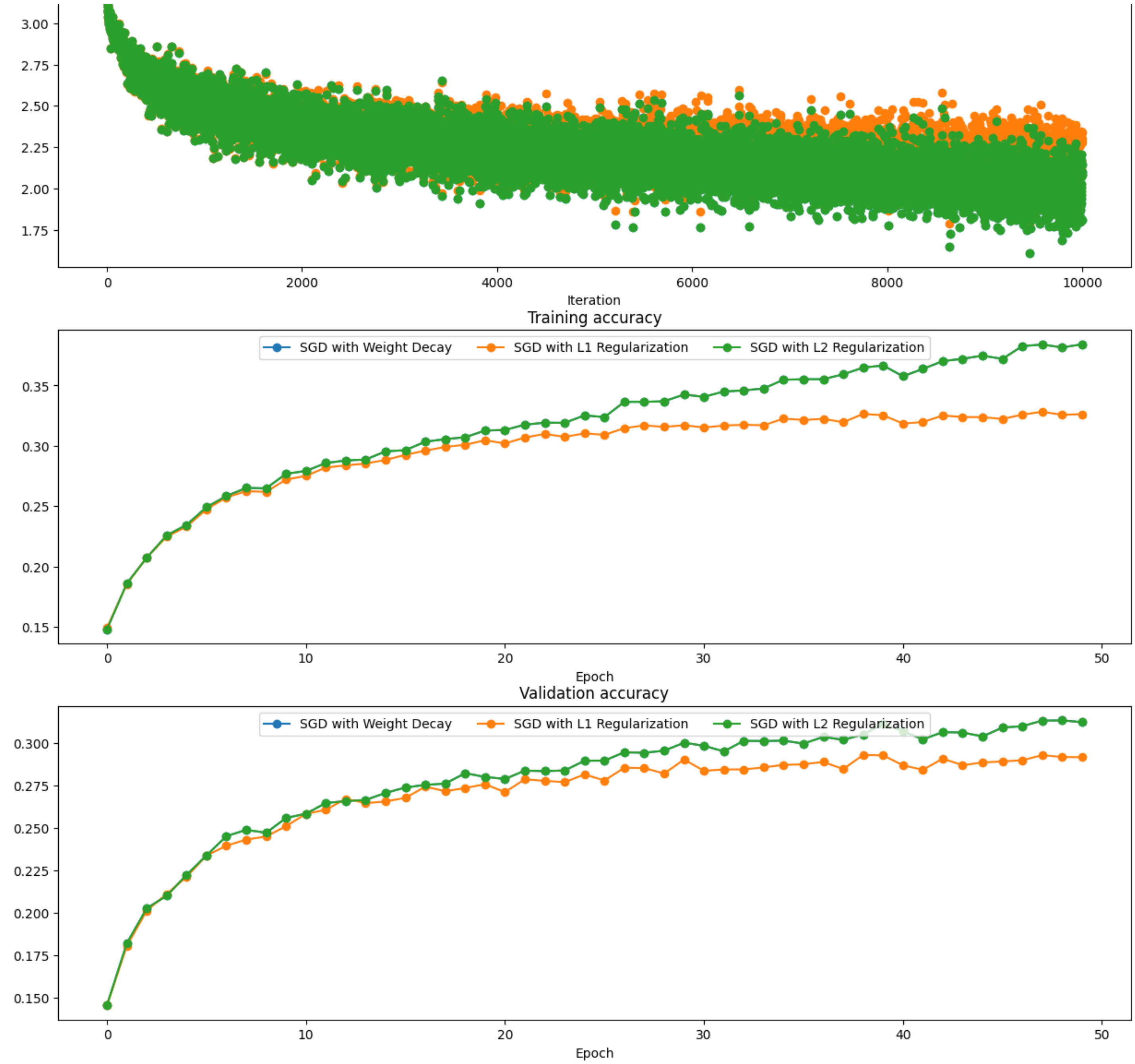
```
  2%|▏         | 4/200 [00:00<00:05, 36.26it/s]
```

```
Training with SGD plus L2 Regularization...
```

```
100%|██████████| 200/200 [00:05<00:00, 39.47it/s]
100%|██████████| 200/200 [00:04<00:00, 40.01it/s]
100%|██████████| 200/200 [00:05<00:00, 39.49it/s]
100%|██████████| 200/200 [00:05<00:00, 39.70it/s]
100%|██████████| 200/200 [00:04<00:00, 40.09it/s]
100%|██████████| 200/200 [00:04<00:00, 40.33it/s]
100%|██████████| 200/200 [00:04<00:00, 40.68it/s]
100%|██████████| 200/200 [00:05<00:00, 38.50it/s]
100%|██████████| 200/200 [00:04<00:00, 41.23it/s]
100%|██████████| 200/200 [00:04<00:00, 41.45it/s]
100%|██████████| 200/200 [00:04<00:00, 41.93it/s]
100%|██████████| 200/200 [00:04<00:00, 41.77it/s]
100%|██████████| 200/200 [00:04<00:00, 40.94it/s]

100%|██████████| 200/200 [00:04<00:00, 42.03it/s]
100%|██████████| 200/200 [00:04<00:00, 42.60it/s]
100%|██████████| 200/200 [00:04<00:00, 42.42it/s]
```

```
100%|██████████| 200/200 [00:04<00:00, 42.42it/s]
100%|██████████| 200/200 [00:04<00:00, 42.32it/s]
100%|██████████| 200/200 [00:04<00:00, 42.66it/s]
100%|██████████| 200/200 [00:04<00:00, 42.73it/s]
100%|██████████| 200/200 [00:04<00:00, 42.92it/s]
100%|██████████| 200/200 [00:04<00:00, 42.51it/s]
100%|██████████| 200/200 [00:04<00:00, 43.69it/s]
100%|██████████| 200/200 [00:04<00:00, 43.62it/s]
100%|██████████| 200/200 [00:04<00:00, 43.34it/s]
100%|██████████| 200/200 [00:04<00:00, 42.36it/s]
100%|██████████| 200/200 [00:04<00:00, 44.46it/s]
100%|██████████| 200/200 [00:04<00:00, 44.52it/s]
100%|██████████| 200/200 [00:04<00:00, 45.00it/s]
100%|██████████| 200/200 [00:04<00:00, 44.96it/s]
100%|██████████| 200/200 [00:04<00:00, 44.70it/s]
100%|██████████| 200/200 [00:04<00:00, 46.18it/s]
100%|██████████| 200/200 [00:04<00:00, 46.32it/s]
100%|██████████| 200/200 [00:04<00:00, 46.66it/s]
100%|██████████| 200/200 [00:04<00:00, 40.59it/s]
100%|██████████| 200/200 [00:04<00:00, 46.25it/s]
100%|██████████| 200/200 [00:04<00:00, 46.53it/s]
100%|██████████| 200/200 [00:04<00:00, 46.54it/s]
100%|██████████| 200/200 [00:04<00:00, 46.93it/s]
100%|██████████| 200/200 [00:04<00:00, 46.83it/s]
100%|██████████| 200/200 [00:04<00:00, 47.14it/s]
100%|██████████| 200/200 [00:04<00:00, 46.74it/s]
100%|██████████| 200/200 [00:04<00:00, 47.21it/s]
100%|██████████| 200/200 [00:04<00:00, 46.27it/s]
100%|██████████| 200/200 [00:04<00:00, 47.74it/s]
100%|██████████| 200/200 [00:04<00:00, 49.95it/s]
100%|██████████| 200/200 [00:04<00:00, 49.71it/s]
100%|██████████| 200/200 [00:04<00:00, 49.44it/s]
100%|██████████| 200/200 [00:04<00:00, 49.92it/s]
100%|██████████| 200/200 [00:04<00:00, 49.42it/s]
100%|██████████| 200/200 [00:03<00:00, 50.99it/s]
```





As shown in figure, SGD + L2 achives the EXACTLY SAME effect as weight decay. They both have better training and validation accuracy than SGD + L1.

Adam [2pt]

The update rule of Adam is as shown below:

$$t = t + 1$$
$$g_t : \text{gradients at update step } t$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= m_t/(1-\beta_1^t) \\
\hat{v}_t &= v_t/(1-\beta_2^t) \\
\theta_{t+1} &= \theta_t - \frac{\eta\, \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

$Completethe\backslash Adam()\backslash functionin\backslash lib/optim.py\backslash ImportantNotes:$

1. t must be updated before everything else
2. β_1^t is β_1 exponentiated to the t'th power
3. You should also enable weight decay in Adam, similar to what you did in SGD

```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

# Test Adam implementation; you should see errors around 1e-7 or less
N, D = 4, 5
test_adam = sequential(fc(N, D, name="adam_fc"))

w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

test_adam.layers[0].params = {"adam_fc_w": w}
test_adam.layers[0].grads = {"adam_fc_w": dw}

opt_adam = Adam(test_adam, 1e-2, 0.9, 0.999, t=5)
opt_adam.mt = {"adam_fc_w": m}
opt_adam.vt = {"adam_fc_w": v}
opt_adam.step()

updated_w = test_adam.layers[0].params["adam_fc_w"]
mt = opt_adam.mt["adam_fc_w"]
vt = opt_adam.vt["adam_fc_w"]

expected_updated_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([

  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
```

```
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,    ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

print ('The following errors should be around or less than 1e-7')
print ('updated_w error: ', rel_error(expected_updated_w, updated_w))
print ('mt error: ', rel_error(expected_m, mt))
print ('vt error: ', rel_error(expected_v, vt))
```

```
The following errors should be around or less than 1e-7
updated_w error:  1.1395691798535431e-07
mt error:  4.214963193114416e-09
vt error:  4.208314038113071e-09
```

Comparing the Weight Decay v.s. L2 Regularization in Adam [5pt]

Run the following code block to compare the plotted results between effects of weight decay and L2 regularization on Adam. Are they still the same? (we can make them the same as in SGD, can we also do it in Adam?)

```
seed = 1234
reset_seed(seed)
model_adam_wd      = FullyConnectedNetwork()
loss_f_adam_wd     = cross_entropy()
optimizer_adam_wd  = Adam(model_adam_wd.net, lr=1e-4, weight_decay=1e-6)


print ("Training with AdamW...")
results_adam_wd = train_net(small_data_dict, model_adam_wd, loss_f_adam_wd,
optimizer_adam_wd, batch_size=100,
                        max_epochs=50, show_every=10000, verbose=False)

reset_seed(seed)
model_adam_l2      = FullyConnectedNetwork()
loss_f_adam_l2     = cross_entropy()
optimizer_adam_l2 = Adam(model_adam_l2.net, lr=1e-4)
reg_lambda_l2 = 1e-4

print ("\nTraining with Adam + L2...")
results_adam_l2 = train_net(small_data_dict, model_adam_l2, loss_f_adam_l2,
optimizer_adam_l2, batch_size=100,
```

```
optimizer_adam_l2, batch_size=100,
                         max_epochs=50, show_every=10000, verbose=False,
regularization='l2', reg_lambda=reg_lambda_l2)

opt_params_adam_wd, loss_hist_adam_wd, train_acc_hist_adam_wd, val_acc_hist_adam_wd  =
results_adam_wd
opt_params_adam_l2, loss_hist_adam_l2, train_acc_hist_adam_l2, val_acc_hist_adam_l2 =
results_adam_l2

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd, 'o', label="Vanilla SGD")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd, '-o', label="Vanilla SGD")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgd, '-o', label="Vanilla SGD")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgdw, 'o', label="SGD with Weight Decay")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgdw, '-o', label="SGD with Weight Decay")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgdw, '-o', label="SGD with Weight Decay")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_adam_wd, 'o', label="Adam with Weight Decay")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_adam_wd, '-o', label="Adam with Weight Decay")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_adam_wd, '-o', label="Adam with Weight Decay")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_adam_l2, 'o', label="Adam with L2")

plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_adam_l2, '-o', label="Adam with L2")
plt.subplot(3, 1, 3)
```

```
plt.plot(val_acc_hist_adam_l2, '-o', label="Adam with L2")

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
  2%|▏         | 3/200 [00:00<00:07, 26.34it/s]

Training with AdamW...
```

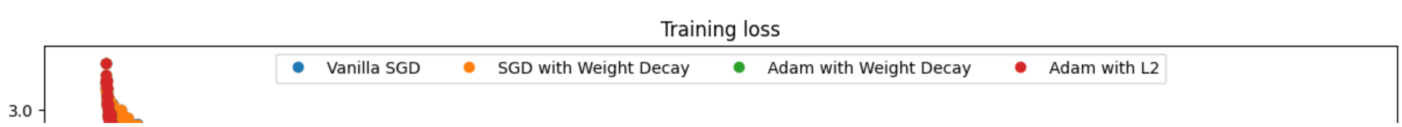
```
100%|██████████| 200/200 [00:06<00:00, 31.13it/s]
100%|██████████| 200/200 [00:06<00:00, 31.75it/s]
100%|██████████| 200/200 [00:06<00:00, 31.62it/s]
100%|██████████| 200/200 [00:06<00:00, 32.23it/s]
100%|██████████| 200/200 [00:06<00:00, 32.30it/s]
100%|██████████| 200/200 [00:06<00:00, 32.81it/s]
100%|██████████| 200/200 [00:06<00:00, 32.90it/s]
100%|██████████| 200/200 [00:06<00:00, 29.72it/s]
100%|██████████| 200/200 [00:06<00:00, 32.70it/s]
100%|██████████| 200/200 [00:06<00:00, 32.76it/s]
100%|██████████| 200/200 [00:06<00:00, 32.62it/s]
100%|██████████| 200/200 [00:06<00:00, 32.54it/s]
100%|██████████| 200/200 [00:06<00:00, 32.63it/s]
100%|██████████| 200/200 [00:06<00:00, 32.23it/s]
100%|██████████| 200/200 [00:06<00:00, 31.42it/s]
100%|██████████| 200/200 [00:06<00:00, 33.26it/s]
100%|██████████| 200/200 [00:06<00:00, 33.06it/s]
100%|██████████| 200/200 [00:06<00:00, 33.17it/s]
100%|██████████| 200/200 [00:06<00:00, 32.64it/s]
100%|██████████| 200/200 [00:06<00:00, 32.91it/s]
100%|██████████| 200/200 [00:06<00:00, 31.29it/s]
100%|██████████| 200/200 [00:05<00:00, 33.38it/s]
100%|██████████| 200/200 [00:05<00:00, 33.41it/s]
100%|██████████| 200/200 [00:06<00:00, 31.83it/s]
100%|██████████| 200/200 [00:06<00:00, 33.23it/s]
100%|██████████| 200/200 [00:06<00:00, 33.21it/s]
100%|██████████| 200/200 [00:05<00:00, 33.75it/s]
100%|██████████| 200/200 [00:06<00:00, 31.61it/s]
100%|██████████| 200/200 [00:06<00:00, 32.34it/s]
100%|██████████| 200/200 [00:06<00:00, 32.59it/s]
100%|██████████| 200/200 [00:06<00:00, 33.14it/s]
100%|██████████| 200/200 [00:05<00:00, 33.34it/s]
100%|██████████| 200/200 [00:06<00:00, 32.32it/s]
```

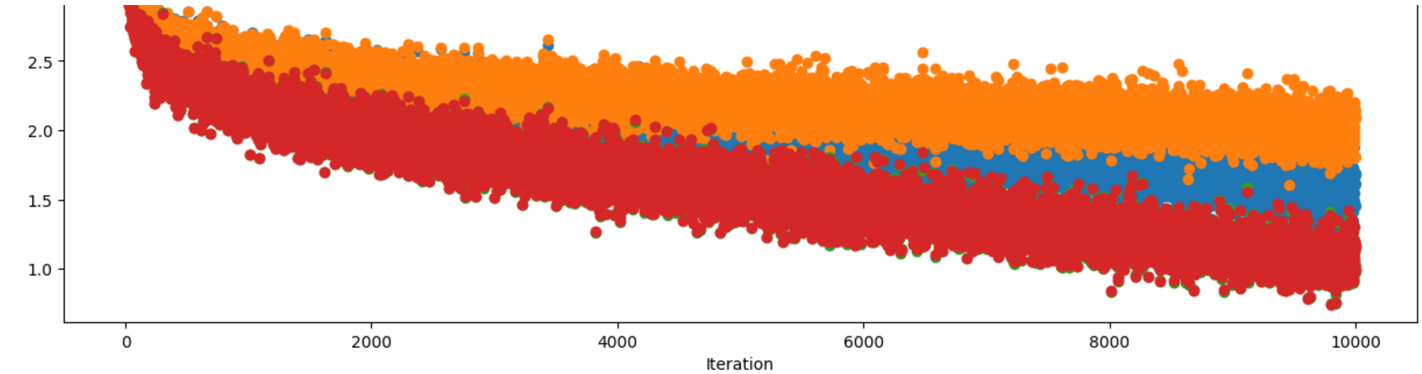
```
100%|██████████| 200/200 [00:06<00:00, 33.20it/s]
100%|██████████| 200/200 [00:06<00:00, 32.37it/s]
100%|██████████| 200/200 [00:05<00:00, 33.46it/s]
100%|██████████| 200/200 [00:05<00:00, 33.41it/s]
100%|██████████| 200/200 [00:06<00:00, 32.68it/s]
100%|██████████| 200/200 [00:05<00:00, 33.40it/s]
100%|██████████| 200/200 [00:05<00:00, 33.34it/s]
100%|██████████| 200/200 [00:06<00:00, 33.26it/s]
100%|██████████| 200/200 [00:06<00:00, 32.36it/s]
100%|██████████| 200/200 [00:05<00:00, 33.56it/s]
100%|██████████| 200/200 [00:05<00:00, 33.40it/s]
100%|██████████| 200/200 [00:06<00:00, 32.57it/s]
100%|██████████| 200/200 [00:06<00:00, 33.22it/s]
100%|██████████| 200/200 [00:05<00:00, 33.42it/s]
100%|██████████| 200/200 [00:06<00:00, 33.06it/s]
100%|██████████| 200/200 [00:06<00:00, 32.39it/s]
100%|██████████| 200/200 [00:05<00:00, 33.68it/s]
  2%|▏         | 3/200 [00:00<00:06, 28.60it/s]
```

```
Training with Adam + L2...
```

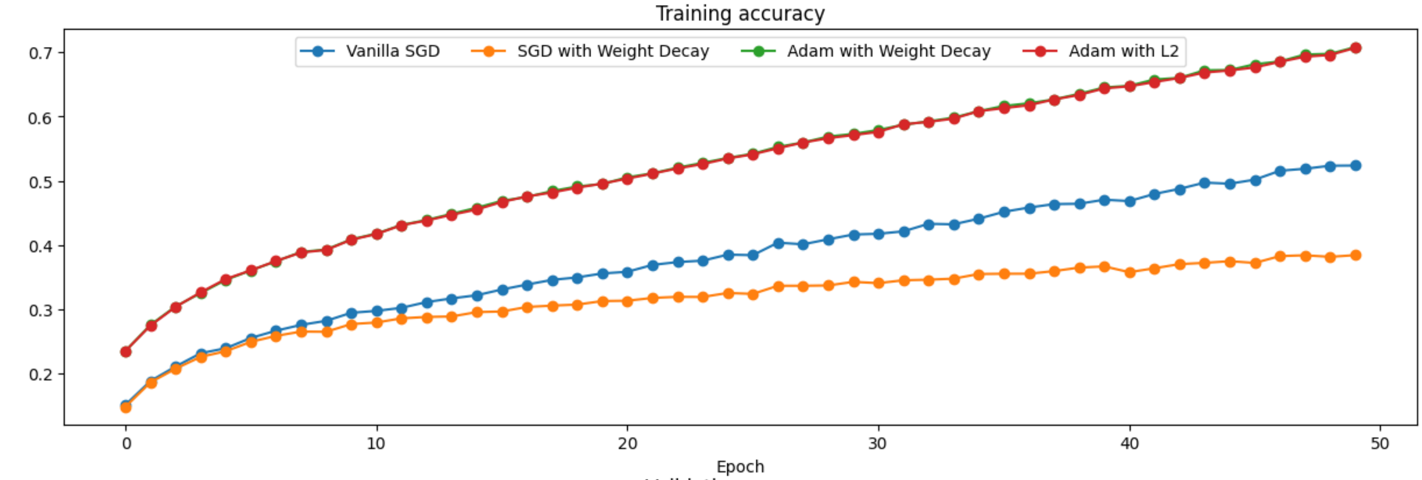
```
100%|██████████| 200/200 [00:06<00:00, 30.27it/s]
100%|██████████| 200/200 [00:06<00:00, 29.88it/s]
100%|██████████| 200/200 [00:06<00:00, 30.68it/s]
100%|██████████| 200/200 [00:06<00:00, 31.36it/s]
100%|██████████| 200/200 [00:06<00:00, 31.47it/s]
100%|██████████| 200/200 [00:07<00:00, 28.53it/s]
100%|██████████| 200/200 [00:06<00:00, 32.36it/s]
100%|██████████| 200/200 [00:06<00:00, 32.38it/s]
100%|██████████| 200/200 [00:06<00:00, 32.40it/s]
100%|██████████| 200/200 [00:06<00:00, 32.45it/s]
100%|██████████| 200/200 [00:06<00:00, 32.52it/s]
100%|██████████| 200/200 [00:06<00:00, 32.26it/s]
100%|██████████| 200/200 [00:06<00:00, 31.74it/s]
100%|██████████| 200/200 [00:06<00:00, 32.52it/s]
100%|██████████| 200/200 [00:06<00:00, 32.47it/s]
100%|██████████| 200/200 [00:06<00:00, 32.64it/s]
100%|██████████| 200/200 [00:06<00:00, 32.34it/s]
100%|██████████| 200/200 [00:06<00:00, 32.67it/s]
100%|██████████| 200/200 [00:06<00:00, 32.97it/s]
100%|██████████| 200/200 [00:06<00:00, 31.51it/s]
100%|██████████| 200/200 [00:06<00:00, 32.82it/s]
100%|██████████| 200/200 [00:06<00:00, 32.91it/s]
```

```
100%|██████████| 200/200 [00:06<00:00, 32.86it/s]
100%|██████████| 200/200 [00:06<00:00, 32.80it/s]
100%|██████████| 200/200 [00:06<00:00, 32.74it/s]
100%|██████████| 200/200 [00:06<00:00, 32.90it/s]
100%|██████████| 200/200 [00:06<00:00, 31.48it/s]
100%|██████████| 200/200 [00:06<00:00, 32.98it/s]
100%|██████████| 200/200 [00:06<00:00, 32.64it/s]
100%|██████████| 200/200 [00:06<00:00, 33.29it/s]
100%|██████████| 200/200 [00:06<00:00, 33.03it/s]
100%|██████████| 200/200 [00:06<00:00, 32.91it/s]
100%|██████████| 200/200 [00:06<00:00, 33.05it/s]
100%|██████████| 200/200 [00:06<00:00, 28.79it/s]
100%|██████████| 200/200 [00:06<00:00, 32.60it/s]
100%|██████████| 200/200 [00:06<00:00, 32.52it/s]
100%|██████████| 200/200 [00:06<00:00, 32.56it/s]
100%|██████████| 200/200 [00:06<00:00, 32.68it/s]
100%|██████████| 200/200 [00:06<00:00, 33.12it/s]
100%|██████████| 200/200 [00:06<00:00, 30.62it/s]
100%|██████████| 200/200 [00:06<00:00, 32.92it/s]
100%|██████████| 200/200 [00:06<00:00, 32.11it/s]
100%|██████████| 200/200 [00:06<00:00, 32.86it/s]
100%|██████████| 200/200 [00:06<00:00, 32.37it/s]
100%|██████████| 200/200 [00:06<00:00, 31.98it/s]
100%|██████████| 200/200 [00:06<00:00, 31.54it/s]
100%|██████████| 200/200 [00:06<00:00, 32.91it/s]
100%|██████████| 200/200 [00:06<00:00, 32.26it/s]
100%|██████████| 200/200 [00:06<00:00, 33.28it/s]
100%|██████████| 200/200 [00:06<00:00, 32.58it/s]
```

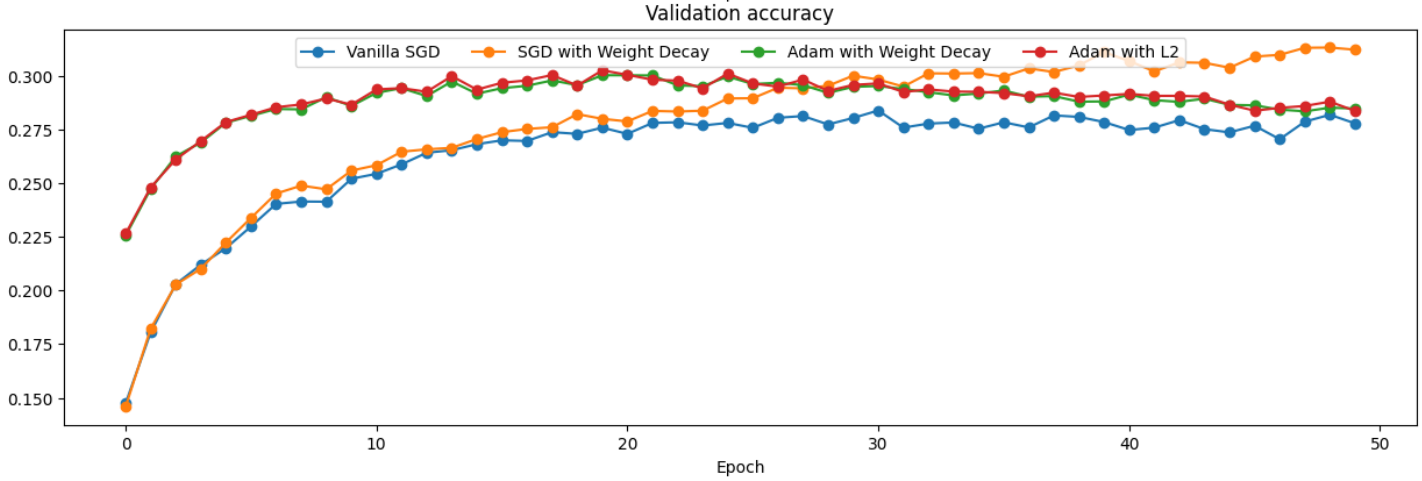




Training accuracy



Validation accuracy



AdamW use $weight\_decay = 1e-6$, Adam with L2 use $\lambda_{L_2} = 1e-4$ strength. There is not much difference between their effect. Both Adam have lower loss, and higher training accuracy than two kinds of SGD. But with epoch increasing, SGD with weight decay achieve highest validation loss.

Submission

Please prepare a PDF document `problem_1_solution.pdf` in the root directory of this repository with all plots and inline answers of your solution. Concretely, the document should contain the following items in strict order:

order:

1. Training loss / accuracy curves for the simple neural network training with > 30% validation accuracy
2. Plots for comparing vanilla SGD to SGD + Weight Decay, SGD + L1 and SGD + L2
3. "Comparing different Regularizations with Adam" plots

Note that you still need to submit the jupyter notebook with all generated solutions. We will randomly pick submissions and check that the plots in the PDF and in the notebook are equivalent.

Problem 2: Incorporating CNNs

- Learning Objective: In this problem, you will learn how to deeply understand how Convolutional Neural Networks work by implementing one.
- Provided Code: We provide the skeletons of classes you need to complete. Forward checking and gradient checkings are provided for verifying your implementation as well.
- TODOs: you will implement a Convolutional Layer and a MaxPooling Layer to improve on your classification results in part 1.

```
from lib.mlp.fully_conn import *
from lib.mlp.layer_utils import *
from lib.mlp.train import *
from lib.cnn.layer_utils import *
from lib.cnn.cnn_models import *
from lib.datasets import *
from lib.grad_check import *
from lib.optim import *
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

Loading the data (CIFAR-100 with 20 superclasses)

In this homework, we will be classifying images from the CIFAR-100 dataset into the 20 superclasses. More information about the CIFAR-100 dataset and the 20 superclasses can be found [here](#).

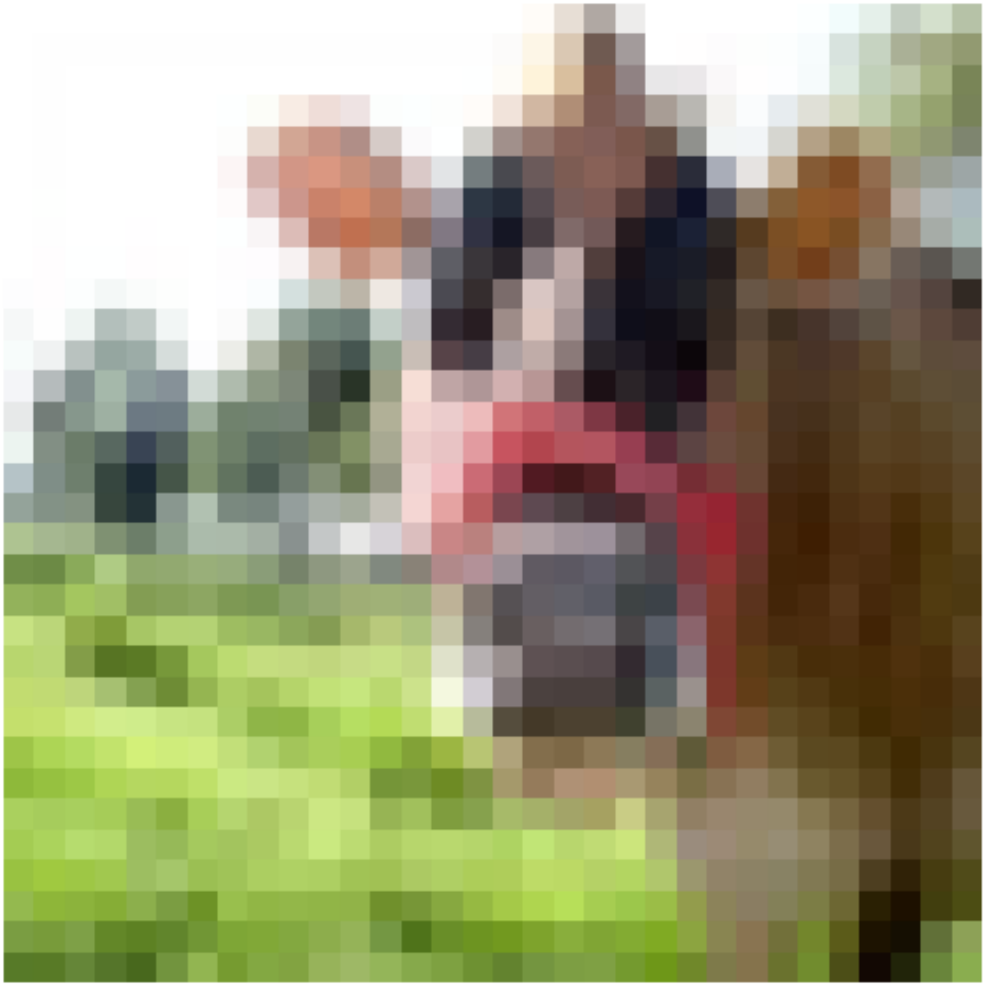
Download the CIFAR-100 data files [here](#), and save the `.mat` files to the `data/cifar100` directory.

```python
data = CIFAR100_data('data/cifar100/')
for k, v in data.items():
    if type(v) == np.ndarray:
        print ("Name: {} Shape: {}, {}".format(k, v.shape, type(v)))
    else:
        print("{}: {}".format(k, v))
label_names = data['label_names']
mean_image = data['mean_image'][0]
std_image = data['std_image'][0]
```

```
Name: data_train Shape: (40000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_train Shape: (40000,), <class 'numpy.ndarray'>
Name: data_val Shape: (10000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_val Shape: (10000,), <class 'numpy.ndarray'>
Name: data_test Shape: (10000, 32, 32, 3), <class 'numpy.ndarray'>
Name: labels_test Shape: (10000,), <class 'numpy.ndarray'>
label_names: ['aquatic_mammals', 'fish', 'flowers', 'food_containers',
'fruit_and_vegetables', 'household_electrical_devices', 'household_furniture', 'insects',
'large_carnivores', 'large_man-made_outdoor_things', 'large_natural_outdoor_scenes',
'large_omnivores_and_herbivores', 'medium_mammals', 'non-insect_invertebrates', 'people',
'reptiles', 'small_mammals', 'trees', 'vehicles_1', 'vehicles_2']
Name: mean_image Shape: (1, 1, 1, 3), <class 'numpy.ndarray'>
Name: std_image Shape: (1, 1, 1, 3), <class 'numpy.ndarray'>
```

```python
idx = 0
image_data = data['data_train'][idx]
image_data = ((image_data*std_image + mean_image) * 255).astype(np.int32)
plt.figure(figsize=(8, 6))
plt.imshow(image_data)
label = label_names[data['labels_train'][idx]]
plt.axis('off')
print("Label:", label)
```

```
Label: large_omnivores_and_herbivores
```



Convolutional Neural Networks

We will use convolutional neural networks to try to improve on the results from Problem 1. Convolutional layers make the assumption that local pixels are more important for prediction than far-away pixels. This allows us to form networks that are robust to small changes in positioning in images.

Convolutional Layer Output size calculation [2pts]

As you have learned, two important parameters of a convolutional layer are its stride and padding. To warm up, we will need to calculate the output size of a convolutional layer given its stride and padding. To do this, open the `lib/cnn/layer_utils.py` file and fill out the TODO section in the `get_output_size` function in the ConvLayer2D class.

Implement your function so that it returns the correct size as indicated by the block below.

```
%reload_ext autoreload

input_image = np.zeros([32, 28, 28, 3]) # a stack of 32 28 by 28 rgb images

in_channels = input_image.shape[-1] # must agree with the last dimension of the input
image
k_size = 4
n_filt = 16

conv_layer = ConvLayer2D(in_channels, k_size, n_filt, stride=2, padding=3)
output_size = conv_layer.get_output_size(input_image.shape)

print("Received {} and expected [32, 16, 16, 16]".format(output_size))
```

```
Received [32, 16, 16, 16] and expected [32, 16, 16, 16]
```

Convolutional Layer Forward Pass [5pts]

Now, we will implement the forward pass of a convolutional layer. Fill in the TODO block in the `forward` function of the ConvLayer2D class.

```
%reload_ext autoreload

# Test the convolutional forward function
input_image = np.linspace(-0.1, 0.4, num=1*8*8*1).reshape([1, 8, 8, 1]) # a single 8 by 8
grayscale image
in_channels, k_size, n_filt = 1, 5, 2

weight_size = k_size*k_size*in_channels*n_filt
bias_size = n_filt

single_conv = ConvLayer2D(in_channels, k_size, n_filt, stride=1, padding=0,
name="conv_test")

w = np.linspace(-0.2, 0.2, num=weight_size).reshape(k_size, k_size, in_channels, n_filt)
b = np.linspace(-0.3, 0.3, num=bias_size)

single_conv.params[single_conv.w_name] = w
single_conv.params[single_conv.b_name] = b

out = single_conv.forward(input_image)

print("Received output shape: {}, Expected output shape: (1, 4, 4, 2)".format(out.shape))
```

```
correct_out = np.array([[
   [[-0.03874312, 0.57000324],
   [-0.03955296, 0.57081309],
   [-0.04036281, 0.57162293],
   [-0.04117266, 0.57243278]],

  [[-0.0452219, 0.57648202],
   [-0.04603175, 0.57729187],
   [-0.04684159, 0.57810172],
   [-0.04765144, 0.57891156]],

  [[-0.05170068, 0.5829608 ],
   [-0.05251053, 0.58377065],
   [-0.05332038, 0.5845805 ],
   [-0.05413022, 0.58539035]],

  [[-0.05817946, 0.58943959],
   [-0.05898931, 0.59024943],
   [-0.05979916, 0.59105928],
   [-0.06060901, 0.59186913]]]])

# Compare your output with the above pre-computed ones.
# The difference should not be larger than 1e-7
print ("Difference: ", rel_error(out, correct_out))
```

```
Received output shape: (1, 4, 4, 2), Expected output shape: (1, 4, 4, 2)
Difference:  5.110565335399418e-08
```

Conv Layer Backward [5pts]

Now complete the backward pass of a convolutional layer. Fill in the TODO block in the `backward` function of the ConvLayer2D class. Check you results with this code and expect differences of less than 1e-6.

```
%reload_ext autoreload

# Test the conv backward function
img = np.random.randn(15, 8, 8, 3)
w = np.random.randn(4, 4, 3, 12)
b = np.random.randn(12)
dout = np.random.randn(15, 4, 4, 12)

single_conv = ConvLayer2D(input_channels=3, kernel_size=4, number_filters=12, stride=2,
padding=1, name="conv_test")
single_conv.params[single_conv.w_name] = w
single_conv.params[single_conv.b_name] = b
```

```
dimg_num = eval_numerical_gradient_array(lambda x: single_conv.forward(img), img, dout)
dw_num = eval_numerical_gradient_array(lambda w: single_conv.forward(img), w, dout)
db_num = eval_numerical_gradient_array(lambda b: single_conv.forward(img), b, dout)

out = single_conv.forward(img)

dimg = single_conv.backward(dout)
dw = single_conv.grads[single_conv.w_name]
db = single_conv.grads[single_conv.b_name]

# The error should be around 1e-6
print("dimg Error: ", rel_error(dimg_num, dimg))
# The errors should be around 1e-8
print("dw Error: ", rel_error(dw_num, dw))
print("db Error: ", rel_error(db_num, db))
# The shapes should be same
print("dimg Shape: ", dimg.shape, img.shape)
```

```
dimg Error:  1.730786582697031e-08
dw Error:  2.0463615563287753e-08
db Error:  1.6043784108481567e-10
dimg Shape:  (15, 8, 8, 3) (15, 8, 8, 3)
```

Max pooling Layer

Now we will implement maxpooling layers, which can help to reduce the image size while preserving the overall structure of the image.

Forward Pass max pooling [5pts]

Fill out the TODO block in the `forward` function of the MaxPoolingLayer class.

```
# Test the convolutional forward function
input_image = np.linspace(-0.1, 0.4, num=64).reshape([1, 8, 8, 1]) # a single 8 by 8
grayscale image

maxpool= MaxPoolingLayer(pool_size=4, stride=2, name="maxpool_test")
out = maxpool.forward(input_image)

print("Received output shape: {}, Expected output shape: (1, 3, 3, 1)".format(out.shape))

correct_out = np.array([[
   [[0.11428571],
   [0.13015873],
```

```
    [0.14603175]],

   [[0.24126984],
    [0.25714286],
    [0.27301587]],

   [[0.36825397],
    [0.38412698],
    [0.4       ]]]])

# Compare your output with the above pre-computed ones.
# The difference should not be larger than 1e-7
print ("Difference: ", rel_error(out, correct_out))
```

```
Received output shape: (1, 3, 3, 1), Expected output shape: (1, 3, 3, 1)
Difference:  1.8750000280978013e-08
```

Backward Pass Max pooling [5pts]

Fill out the `backward` function in the MaxPoolingLayer class.

```
img = np.random.randn(15, 8, 8, 3)

dout = np.random.randn(15, 3, 3, 3)

maxpool= MaxPoolingLayer(pool_size=4, stride=2, name="maxpool_test")

dimg_num = eval_numerical_gradient_array(lambda x: maxpool.forward(img), img, dout)

out = maxpool.forward(img)
dimg = maxpool.backward(dout)

# The error should be around 1e-8
print("dimg Error: ", rel_error(dimg_num, dimg))
# The shapes should be same
print("dimg Shape: ", dimg.shape, img.shape)
```

```
dimg Error:  1.8928989037561694e-11
dimg Shape:  (15, 8, 8, 3) (15, 8, 8, 3)
```

Test a Small Convolutional Neural Network [3pts]

Please find the `TestCNN` class in `lib/cnn/cnn_models.py`.
Again you only need to complete few lines of code in the TODO block.
Please design a Convolutional --> Maxpool --> flatten --> fc network where the shapes of parameters match the given shapes.
Please insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively.
Here you only modify the param_name part, the _w, and _b are automatically assigned during network setup.

```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

model = TestCNN()
loss_func = cross_entropy()

B, H, W, iC = 4, 8, 8, 3 #batch, height, width, in_channels
k = 3 #kernel size
oC, Hi, O = 3, 27, 5 # out channels, Hidden Layer input, Output size
std = 0.02
x = np.random.randn(B,H,W,iC)
y = np.random.randint(O, size=B)

print ("Testing initialization ... ")

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
w1_std = abs(model.net.get_params("conv1_w").std() - std)
b1 = model.net.get_params("conv1_b").std()
w2_std = abs(model.net.get_params("fc1_w").std() - std)
b2 = model.net.get_params("fc1_b").std()
###################################################
#                 END OF YOUR CODE                #
###################################################

assert w1_std < std / 10, "First layer weights do not seem right"
assert np.all(b1 == 0), "First layer biases do not seem right"
assert w2_std < std / 10, "Second layer weights do not seem right"
assert np.all(b2 == 0), "Second layer biases do not seem right"
print ("Passed!")

print ("Testing test-time forward pass ... ")

w1 = np.linspace(-0.7, 0.3, num=k*k*iC*oC).reshape(k,k,iC,oC)
```

```
w1 = np.linspace(-0.7, 0.3, num=k*k*iC*oC).reshape(k,k,iC,oC)
w2 = np.linspace(-0.2, 0.2, num=Hi*O).reshape(Hi, O)
b1 = np.linspace(-0.6, 0.2, num=oC)
b2 = np.linspace(-0.9, 0.1, num=O)

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
model.net.assign("conv1_w", w1)
model.net.assign("conv1_b", b1)
model.net.assign("fc1_w", w2)
model.net.assign("fc1_b", b2)
###################################################
#                 END OF YOUR CODE                 #
###################################################

feats = np.linspace(-5.5, 4.5, num=B*H*W*iC).reshape(B,H,W,iC)
scores = model.forward(feats)
correct_scores = np.asarray([[-13.85107294, -11.52845818,  -9.20584342,  -6.88322866,
 -4.5606139 ],
 [-11.44514171, -10.21200524 , -8.97886878 , -7.74573231 , -6.51259584],
 [ -9.03921048,  -8.89555231 , -8.75189413 , -8.60823596,  -8.46457778],
 [ -6.63327925 , -7.57909937 , -8.52491949 , -9.4707396 , -10.41655972]])
scores_diff = np.sum(np.abs(scores - correct_scores))
assert scores_diff < 1e-6, "Your implementation might be wrong!"
print ("Passed!")

print ("Testing the loss ...",)
y = np.asarray([0, 2, 1, 4])
loss = loss_func.forward(scores, y)
dLoss = loss_func.backward()
correct_loss = 4.56046848799693
assert abs(loss - correct_loss) < 1e-10, "Your implementation might be wrong!"
print ("Passed!")

print ("Testing the gradients (error should be no larger than 1e-6) ...")
din = model.backward(dLoss)
for layer in model.net.layers:
    if not layer.params:
        continue
    for name in sorted(layer.grads):
        f = lambda _: loss_func.forward(model.forward(feats), y)
        grad_num = eval_numerical_gradient(f, layer.params[name], verbose=False)
        print ('%s relative error: %.2e' % (name, rel_error(grad_num, layer.grads[name])))
```

```
Testing initialization ...
Passed!
Testing test-time forward pass ...
Passed!
Testing the loss ...
Passed!
Testing the gradients (error should be no larger than 1e-6) ...
conv1_b relative error: 2.97e-09
conv1_w relative error: 9.10e-10
fc1_b relative error: 9.76e-11
fc1_w relative error: 3.89e-07
```

Training the Network [25pts]

In this section, we defined a `SmallConvolutionalNetwork` class for you to fill in the TODO block in `lib/cnn/cnn_models.py`.

Here please design a network with at most two convolutions and two maxpooling layers (you may use less). You can adjust the parameters for any layer, and include layers other than those listed above that you have implemented (such as fully-connected layers and non-linearities).
You are also free to select any optimizer you have implemented (with any learning rate).

You will train your network on CIFAR-100 20-way superclass classification.
Try to find a combination that is able to achieve 40% validation accuracy.

Since the CNN takes significantly longer to train than the fully connected network, it is suggested to start off with fewer filters in your Conv layers and fewer intermediate fully-connected layers so as to get faster initial results.

```python
# Arrange the data
data_dict = {
    "data_train": (data["data_train"], data["labels_train"]),
    "data_val": (data["data_val"], data["labels_val"]),
    "data_test": (data["data_test"], data["labels_test"])
}
```

```python
print("Data shape:", data_dict["data_train"][0].shape)
print("Flattened data input size:", np.prod(data["data_train"].shape[1:]))
print("Number of data classes:", max(data['labels_train']) + 1)
```

```
Data shape: (40000, 32, 32, 3)
Flattened data input size: 3072
Number of data classes: 20
```

```
%reload_ext autoreload

seed = 123
np.random.seed(seed=seed)

# using dropout
model = SmallConvolutionalNetwork(seed=seed)
loss_f = cross_entropy()


results = None
#############################################################################
# TODO: Use the train_net function you completed to train a network         #
# You may only adjust the hyperparameters within this block                 #
#############################################################################
optimizer = Adam(model.net, 1e-3)

batch_size = 128
epochs = 10
lr_decay = 0.98
lr_decay_every = 100
regularization = "l2"
reg_lambda = 0.001
#############################################################################
#                             END OF YOUR CODE                              #
#############################################################################
results = train_net(data_dict, model, loss_f, optimizer, batch_size, epochs,
                    lr_decay, lr_decay_every, show_every=4000, verbose=True,
regularization=regularization, reg_lambda=reg_lambda)
opt_params, loss_hist, train_acc_hist, val_acc_hist = results
```

```
  0%|          | 1/312 [00:02<13:37,  2.63s/it]

(Iteration 1 / 3120) Average loss: 2.995974689730413
```

```
100%|██████████| 312/312 [14:18<00:00,  2.75s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 1 / 10) Training Accuracy: 0.333225, Validation Accuracy: 0.318
```

```
100%|██████████| 312/312 [14:37<00:00,  2.81s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 2 / 10) Training Accuracy: 0.409075, Validation Accuracy: 0.3955
```

```
100%|██████████| 312/312 [14:48<00:00,  2.85s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 3 / 10) Training Accuracy: 0.451575, Validation Accuracy: 0.4301
```

```
100%|██████████| 312/312 [14:51<00:00,  2.86s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 4 / 10) Training Accuracy: 0.483025, Validation Accuracy: 0.4531
```

```
100%|██████████| 312/312 [14:55<00:00,  2.87s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 5 / 10) Training Accuracy: 0.5027, Validation Accuracy: 0.4633
```

```
100%|██████████| 312/312 [14:58<00:00,  2.88s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 6 / 10) Training Accuracy: 0.5205, Validation Accuracy: 0.4806
```

```
100%|██████████| 312/312 [14:56<00:00,  2.87s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 7 / 10) Training Accuracy: 0.52185, Validation Accuracy: 0.4719
```

```
100%|██████████| 312/312 [14:57<00:00,  2.88s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 8 / 10) Training Accuracy: 0.545425, Validation Accuracy: 0.4911
```

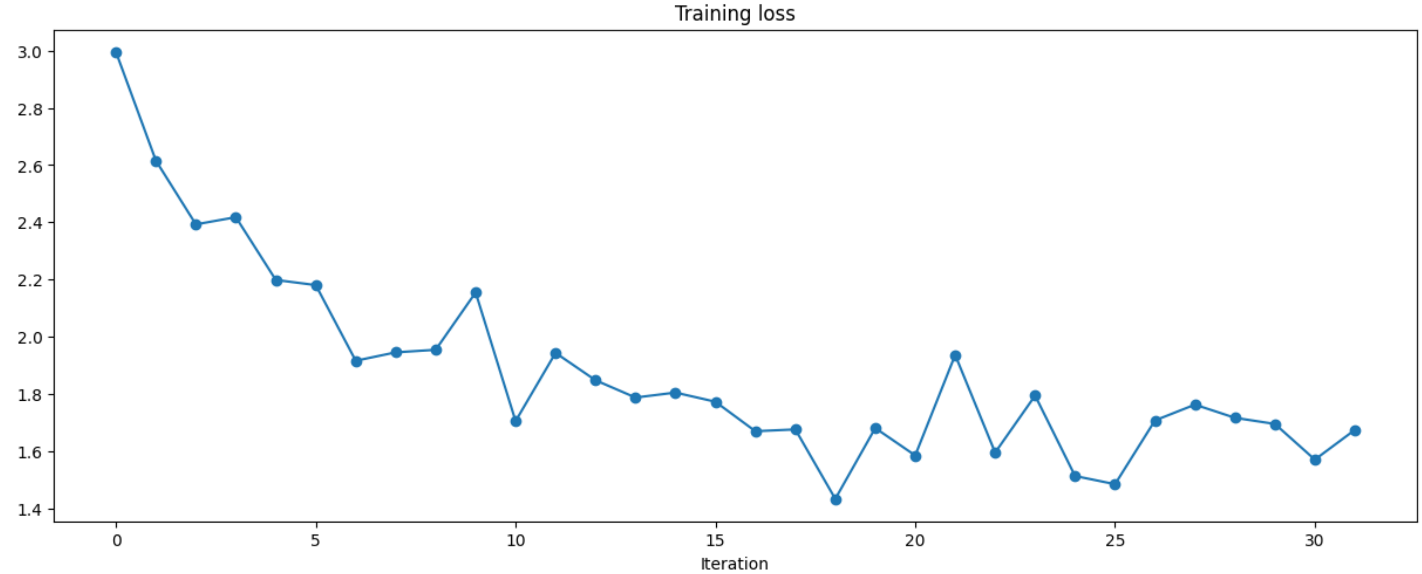
```
100%|██████████| 312/312 [14:56<00:00,  2.87s/it]
  0%|          | 0/312 [00:00<?, ?it/s]

(Epoch 9 / 10) Training Accuracy: 0.558475, Validation Accuracy: 0.503
```

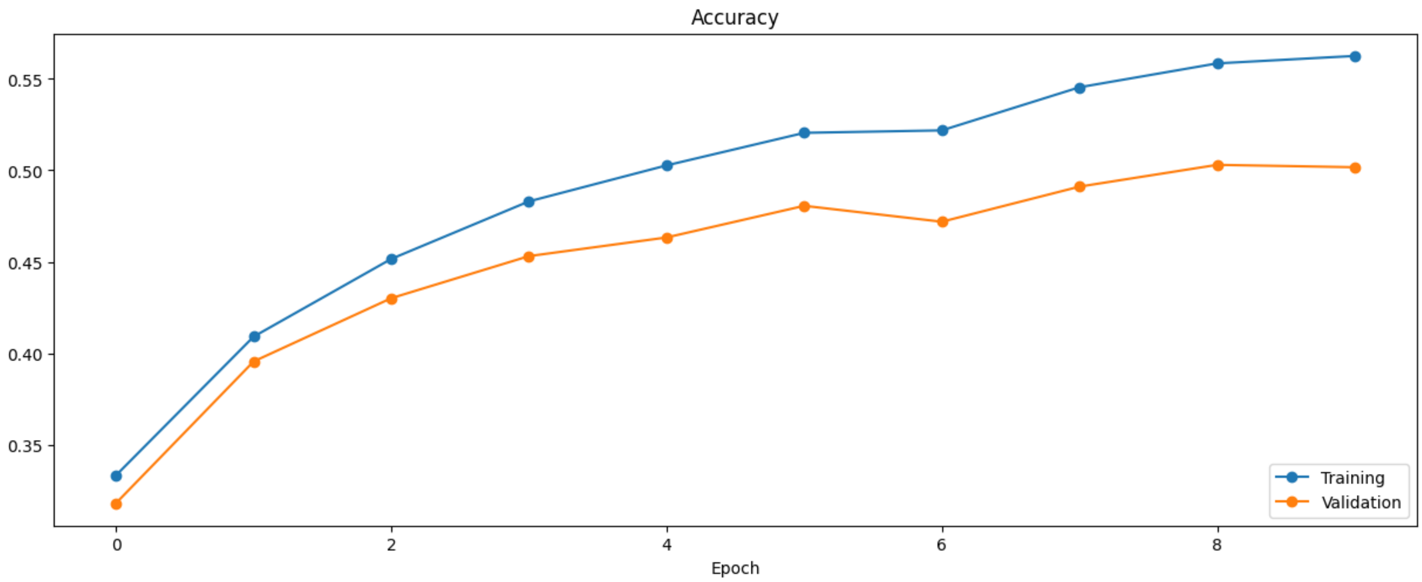
```
100%|██████████| 312/312 [14:56<00:00,  2.87s/it]
```

```
(Epoch 10 / 10) Training Accuracy: 0.56255, Validation Accuracy: 0.5017
```

Run the code below to generate the training plots.

```
%reload_ext autoreload

opt_params, loss_hist, train_acc_hist, val_acc_hist = results

# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss')
loss_hist_ = loss_hist[1::100]  # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(train_acc_hist, '-o', label='Training')
plt.plot(val_acc_hist, '-o', label='Validation')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)

plt.show()
```





The accuracy meets the requirements.

Visualizing Layers [5pts]

An interesting finding from early research in convolutional networks was that the learned convolutions resembled filters used for things like edge detection. Complete the code below to visualize the filters in the first convolutional layer of your best model.

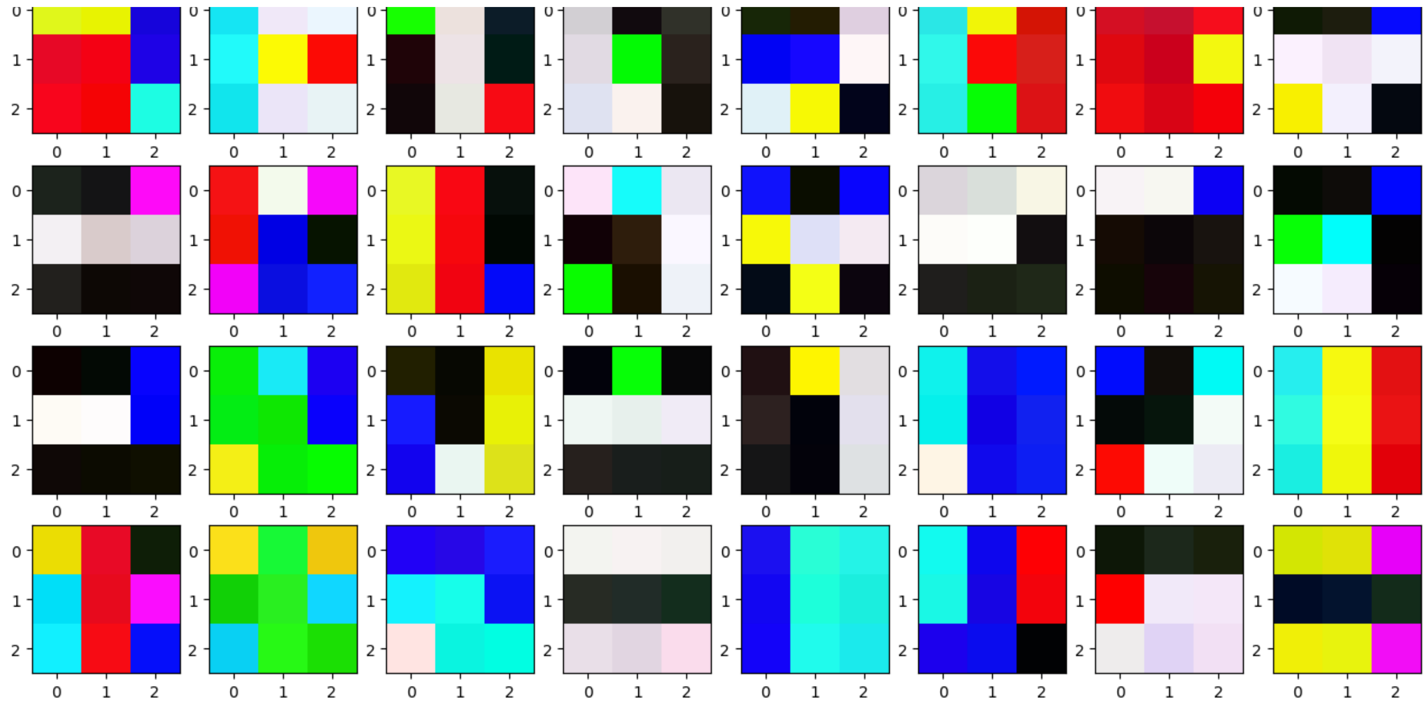
Visualize the first convolutional layer, which

$$size = (3, 3, 3, 32) = (hight, width, input\_channels, num\_filters(i.e. output\_channels))$$

```
im_array = None
nrows, ncols = None, None
```

```
#####################################################
# TODO: read the weights in the convolutional     #
# layer and reshape them to a grid of images to   #
# view with matplotlib.                           #
#####################################################
filters = model.net.get_params('conv1_w')
filters = filters[::, ::, ::, :] * 255 # 0-255, (h, w, c)
filters = filters.astype(np.uint8)
filters = np.transpose(filters, (3, 2, 0, 1)) # (h, w, c, n) -> (n, c, h, w)
# print(filters.shape)

nrows, ncols = filters.shape[0] // 8, 8
plt.figure(figsize=(16, 12), dpi=300)
plt.subplots(nrows, ncols, figsize = (16, 8))
# axs = axs.flatten()
for i, img in enumerate(filters):
    plt.subplot(4, 8, i+1)
    plt.imshow(img)
    # plt.axis('off')
plt.show()
#####################################################
#                 END OF YOUR CODE                #
#####################################################
```

```
<Figure size 4800x3600 with 0 Axes>
```



Inline Question: Comment below on what kinds of filters you see. Include your response in your submission [5pts]

Most of the filters are like edge detection. In other words, they are learning different pattern. They're like our human's eyes, each of them represent a learner dealing with different image feature. Some filters focus on vertical feature, like the (3, 8) (count from top left to bottom right), some focus on horizontal feature, like the (1, 7), while others focus on diagonal feature, like (3, 2). Some detects the edge feature, like (3, 4).

Convolution layer can better focus on local information, which has similarity with human-beings. They learn and memorize different pattern through the convolutional filters. Thus, they can deal with new data by these filters.

Extra-Credit: Analysis on Trained Model [5pts]

For extra credit, you can perform some additional analysis of your trained model. Some suggested analyses are:

1. Plot the confusion matrix of your model's predictions on the test set. Look for trends to see which classes are frequently misclassified as other classes (e.g. are the two vehicle superclasses frequently confused with each other?).
2. Implement BatchNorm and analyze how the models train with and without BatchNorm.
3. Introduce some small noise in the labels, and investigate how that affects training and validation accuracy.

You are free to choose any analysis question of interest to you. We will not be providing any starter code for the extra credit. Include your extra-credit analysis as the final section of your report pdf, titled "Extra Credit".

```
import pandas as pd
import seaborn as sns

# load the parameters to a newly defined network
model = SmallConvolutionalNetwork()
model.net.load(opt_params)
val_acc = compute_acc(model, data["data_val"], data["labels_val"])
print ("Validation Accuracy: {}%".format(val_acc*100))
test_acc = compute_acc(model, data["data_test"], data["labels_test"])
print ("Testing Accuracy: {}%".format(test_acc*100))
```

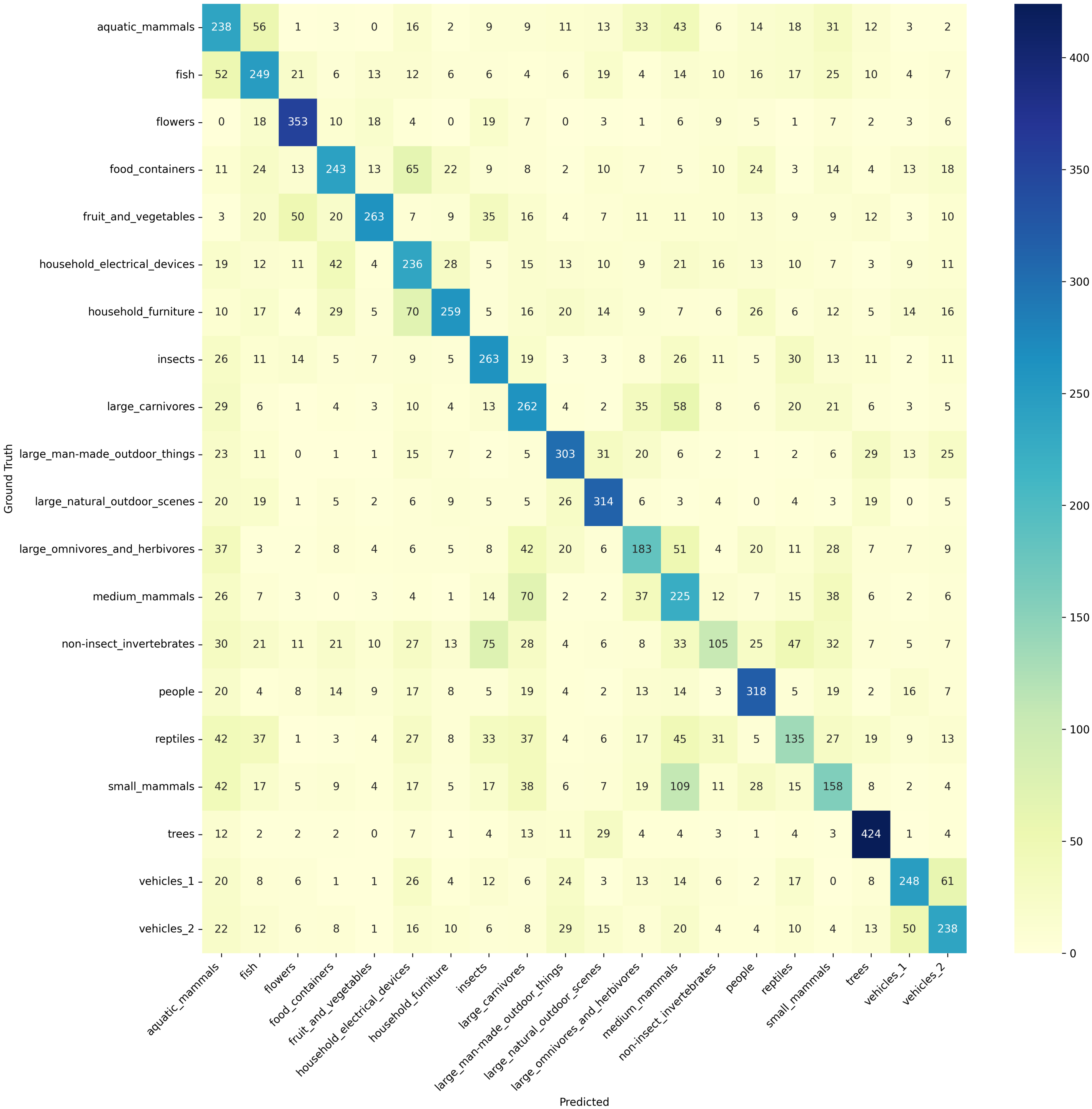
```
Loading Params: conv1_w Shape: (3, 3, 3, 32)
Loading Params: conv1_b Shape: (32,)
Loading Params: conv2_w Shape: (3, 3, 32, 32)
Loading Params: conv2_b Shape: (32,)
Loading Params: fc1_w Shape: (3200, 200)
Loading Params: fc1_b Shape: (200,)
Loading Params: fc2_w Shape: (200, 20)
Loading Params: fc2_b Shape: (20,)
Validation Accuracy: 50.17%
Testing Accuracy: 50.029999999999994%
```

```
# predict testset with model
labels = data["labels_val"] # labels
num_batches = data["data_val"].shape[0] // 100
preds = []
for i in range(num_batches):
    start = i * batch_size
    end = (i + 1) * batch_size
    output = model.forward(data["data_val"][start: end], False)
    scores = softmax(output)
    pred = np.argmax(scores, axis=1)
    preds.append(pred)
preds = np.hstack(preds) # predictions
```

```
def confusion_matrix(y_true, y_pred):
```

```python
def confusion_matrix(y_true, y_pred):
    # generate confusion matrix
    labels = np.unique(y_true)
    n_labels = len(labels)
    cm = np.zeros((n_labels, n_labels), dtype=int)
    for i in range(n_labels):
        for j in range(n_labels):
            cm[i, j] = np.sum((y_true == labels[i]) & (y_pred == labels[j]))
    return cm
```

```python
conf_mat = confusion_matrix(labels, preds)
df_cm = pd.DataFrame(conf_mat, index = label_names, columns = label_names)
plt.figure(figsize=(16, 16), dpi=300)
# generate heatmap
heatmap = sns.heatmap(df_cm, annot = True, fmt = 'd', cmap = 'YlGnBu')
heatmap.yaxis.set_ticklabels(heatmap.yaxis.get_ticklabels(), rotation = 0, ha = 'right')
heatmap.xaxis.set_ticklabels(heatmap.xaxis.get_ticklabels(), rotation = 45, ha = 'right')
plt.ylabel('Ground Truth')
plt.xlabel('Predicted')
plt.show()
```



As shown in the heat-map figure, small mammals and medium mammals has the largest number of misclassification. They are both often confused by CNN.

There are also some other class like vehicles 1 ⇔ vehicles 2, non-insect_invertebrate ⇔ insects, medium mammals ⇔ large carnivores, household furniture ⇔ household electrical devices, ... , are the easily confused pair.

To sum up, these easily confused categories are also really close to our daily life. And the CNN, as one of the bionic intelligent system as human eye, also have something in common with human.

Submission

Please prepare a PDF document `problem_2_solution.pdf` in the root directory of this repository with all plots and inline answers of your solution. Concretely, the document should contain the following items in strict order:

1. Training loss / accuracy curves for CNN training
2. Visualization of convolutional filters
3. Answers to inline questions about convolutional filters

Note that you still need to submit the jupyter notebook with all generated solutions. We will randomly pick submissions and check that the plots in the PDF and in the notebook are equivalent.