Convolutional Architectures

In this assignment we will explore convolution networks and the effects of their architecture on accuracy. We'll implement a common block-based deep CNN pattern and we'll perform various experiments on it while varying the architecture. Then we'll implement our own custom architecture to see whether we can get high classification results on CIFAR-10.

```
!pip install tqdm==4.17.1
```

```
Collecting tqdm==4.17.1
  Downloading tqdm-4.17.1-py2.py3-none-any.whl (47 kB)
     |████████████████████████████████| 47 kB 5.4 MB/s
Installing collected packages: tqdm
  Attempting uninstall: tqdm
    Found existing installation: tqdm 4.64.0
    Uninstalling tqdm-4.64.0:
      Successfully uninstalled tqdm-4.64.0
ERROR: pip's dependency resolver does not currently take into account all the
spacy 2.2.4 requires tqdm<5.0.0,>=4.38.0, but you have tqdm 4.17.1 which is i
panel 0.12.1 requires tqdm>=4.48.0, but you have tqdm 4.17.1 which is incompa
fbprophet 0.7.1 requires tqdm>=4.36.1, but you have tqdm 4.17.1 which is inco
Successfully installed tqdm-4.17.1
```

```
import os
import re
import sys
import glob
import numpy as np
import matplotlib.pyplot as plt
import unittest
import torch
import torchvision
import torchvision.transforms as tvtf

%matplotlib inline
%load_ext autoreload
%autoreload 2

seed = 42
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

plt.rcParams.update({'font.size': 12})
test = unittest.TestCase()
```

```
from google.colab import drive
drive.mount('/content/drive')
#change this
```

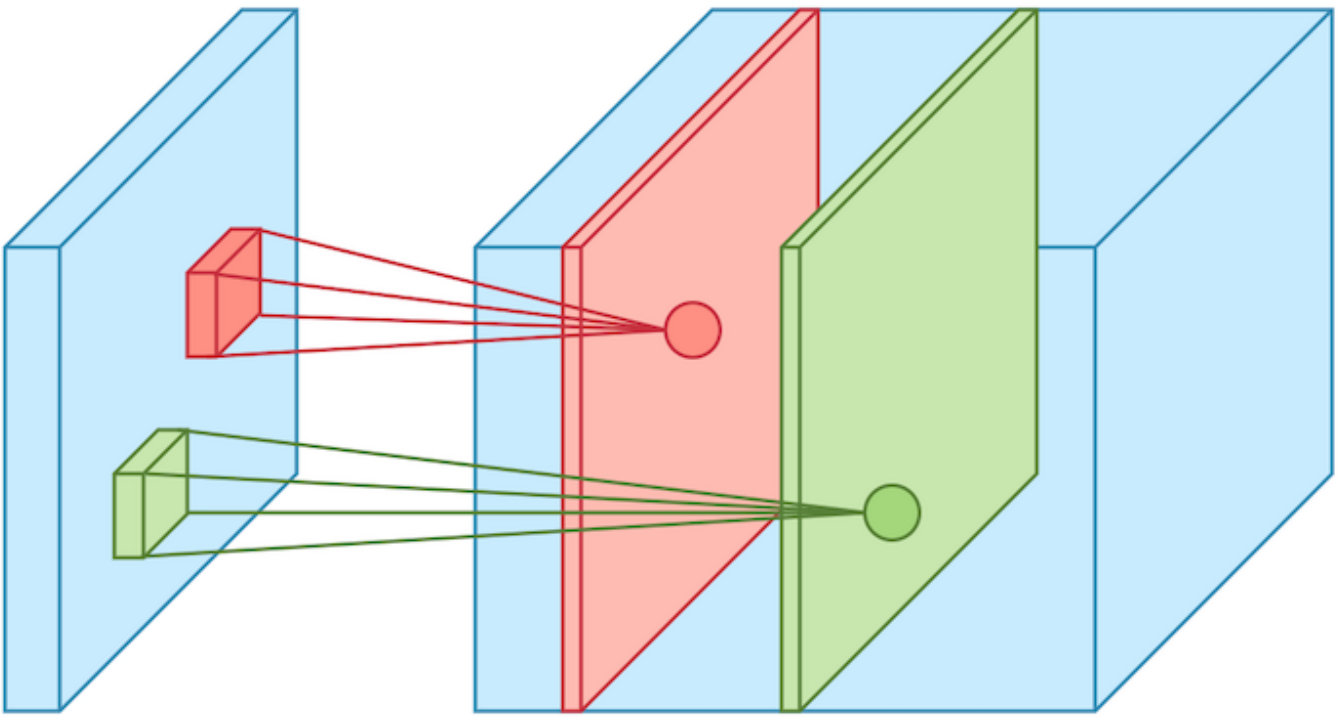
```
%cd /content/drive/MyDrive/Deep Learning/Assignment 2/CNN
from utils import *
```

```
Mounted at /content/drive
/content/drive/MyDrive/Deep Learning/Assignment 2/CNN
```

Convolutional layers and networks

Convolutional layers are the most essential building blocks of the state of the art deep learning image classification models and also play an important role in many other tasks. As we already saw, convolutional layers operate on and produce volumes (3D tensors) of activations.

One way to interpret convolutional layers is as a collection of 3D learnable filters, each of which operates on a small spatial region of the input volume. Each filter is convolved with the input volume ("slides over it"), and a dot product is computed at each location followed by a non-linearity which produces one activation. All these activations produce a 2D plane known as a **feature map**. Multiple feature maps (one for each filter) comprise the output volume.



A crucial property of convolutional layers is their translation invariance, i.e. their ability to detect features regardelss of their spatial location in the input.

Convolutional network architectures usually follow a pattern basic repeating blocks: one or more convolution layers, each followed by a non-linearity (generally ReLU) and then a pooling layer to reduce spatial dimensions. Usually, the number of convolutional filters increases the deeper they are in the network. These layers are meant to extract features from the input. Then, one or more fully-connected layers is used to combine the extracted features into the required number of output class scores.

Building convolutional networks with PyTorch

PyTorch provides all the basic building blocks needed for creating a convolutional arcitecture within the `torch.nn` package. Let's use them to create a basic convolutional network with the following architecture pattern:

```
[(CONV -> ReLU)*P -> MaxPool]*(N/P) -> (Linear -> ReLU)*M -> Linear
```

Here N is the total number of convolutional layers, P specifies how many convolutions to perform before each pooling layer and M specifies the number of hidden fully-connected layers before the final output layer.

TODO: Complete the implementaion of the `ConvClassifier` class in the `utils/models.py` module.

```
from  utils import models
torch.manual_seed(seed)

net = models.ConvClassifier((3,100,100), 10, filters=[32]*4, pool_every=2, hidden_c
print(net)

test_image = torch.randint(low=0, high=256, size=(3, 100, 100), dtype=torch.float).
test_out = net(test_image.unsqueeze(0))
print('out =', test_out)

expected_out = torch.load('tests/assets/expected_conv_out.pt').to(device)
test.assertLess(torch.norm(test_out - expected_out).item(), 1e-5)
```

```
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (5): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU()
    (7): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU()
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
  )
  (classifier): Sequential(
    (0): Linear(in_features=20000, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
```

```
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (5): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU()
    (7): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU()
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
  )
  (classifier): Sequential(
    (0): Linear(in_features=20000, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
out = tensor([[-0.0868, -0.3790, -0.4341, -0.1236, -0.2160,  0.1683,  0.4739,
          0.1151, -0.1606]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

Note about running on GPUs.

Notice how we called `.to(device)` on **both** the model and the input tensor. Here the `device` is a `torch.device` object that we created above. If an nvidia GPU is available on the machine you're running this on, the `device` will be `'cuda'`. When you run `.to(device)` on a model, it recursively goes over all the model parameter tensors and copies their memory to the GPU. Similarly, calling `.to(device)` on the input image also copies it.

In order to train on a GPU, you need to make sure to move **all** your tensors to it. You'll get errors if you try to mix CPU and GPU tensors in a computation.

```
print(f'This notebook is running with device={device}')
print(f'The model parameter tensors are therefore also on device={next(net.paramete
print(f'The test image is therefore also on device={test_image.device}')
```

```
This notebook is running with device=cuda
The model parameter tensors are therefore also on device=cuda:0
The test image is therefore also on device=cuda:0
```

Let's load CIFAR-10 again to use as our dataset.

```
data_dir = os.path.expanduser('~/.pytorch-datasets')
ds_train = torchvision.datasets.CIFAR10(root=data_dir, download=True, train=True, t
ds_test = torchvision.datasets.CIFAR10(root=data_dir, download=True, train=False, t

print(f'Train: {len(ds_train)} samples')
print(f'Test: {len(ds_test)} samples')
```

```
x0,_ = ds_train[0]
in_size = x0.shape
num_classes = 10
print('input image size =', in_size)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to /root/
170499072it [00:13, 13078117.82it/s]
Extracting /root/.pytorch-datasets/cifar-10-python.tar.gz to /root/.pytorch-d
Files already downloaded and verified
Train: 50000 samples
Test: 10000 samples
input image size = torch.Size([3, 32, 32])
```

Now as usual, as a sanity test let's make sure we can overfit a tiny dataset with our model. But first we need to adapt our `Trainer` for PyTorch models.

TODO: Complete the implementaion of the `TorchTrainer` class in the `utils/training.py` module. You should implement the funcitons:

- fit (...)
- train_batch(...)
- test_batch(...)

```
import utils.training as training
torch.manual_seed(seed)

# Define a tiny part of the CIFAR-10 dataset to overfit it
batch_size = 2
max_batches = 25
dl_train = torch.utils.data.DataLoader(ds_train, batch_size, shuffle=False)

# Create model, loss and optimizer instances
model = models.ConvClassifier(in_size, num_classes, filters=[32], pool_every=1, hi
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-2, momentum=0.9,)

# Use TorchTrainer to run only the training loop a few times.
trainer = training.TorchTrainer(model, loss_fn, optimizer, device)
best_acc = 0
for i in range(22):
    res = trainer.train_epoch(dl_train, max_batches=max_batches, verbose=(i%2==0))
    best_acc = res.accuracy if res.accuracy > best_acc else best_acc

# Test overfitting
test.assertGreaterEqual(best_acc, 95)
```

```
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
  )
```

```
  (classifier): Sequential(
    (0): Linear(in_features=8192, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=10, bias=True)
  )
)
train_batch (Avg. Loss 2.371, Accuracy 6.0): 100%|██████████| 25/25 [00:00<00
train_batch (Avg. Loss 2.238, Accuracy 16.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 2.131, Accuracy 22.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 1.829, Accuracy 34.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 1.050, Accuracy 64.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 1.424, Accuracy 52.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 1.125, Accuracy 66.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 0.737, Accuracy 80.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 0.107, Accuracy 96.0): 100%|██████████| 25/25 [00:00<0
train_batch (Avg. Loss 0.018, Accuracy 100.0): 100%|██████████| 25/25 [00:00<
train_batch (Avg. Loss 0.003, Accuracy 100.0): 100%|██████████| 25/25 [00:00<
```

Experimenting with model architectures

You will now perform a series of experiments that train various model configurations on a much larger part of the CIFAR-10 dataset.

To perform the experiments, you'll need to use a machine with a GPU since training time might be too long otherwise.

General notes for running experiments

- It's important to give each experiment run a name as specified by the notebook instructions later on. The each run has a `run_name` parameter that will also be the name of the results file which this notebook will expect to load.
- You will implement the code to run the experiments in the `utils/experiments.py` module.

Double-click (or enter) to edit

Experiment 1 - Network depth and number of filters

In this part we will test some different architecture configurations based on our `ConvClassifier`. Specifically, we want to try different depths and number of features to see the effects these parameters have on the model's performance.

To do this, we'll define two extra hyperparameters for our model, `K` (`filters_per_layer`) and `L` (`layers_per_block`).

- `K` is a list, containing the number of filters we want to have in our conv layers.

- `L` is the number of consecutive layers with the same number of filters to use.

For example, if `K=[32, 64]` and `L=2` it means we want two conv layers with 32 filters followed by two conv layers with 64 filters. The feature-extraction part of our model will therefore be:

```
Conv(X,32)->ReLu->Conv(32,32)->ReLU->MaxPool->Conv(32,64)->ReLU->Conv(64,64)->ReLU->MaxPo
```

We'll try various values of the `K` and `L` parameters in combination and see how each architecture trains.

First we need to write some code to run the experiment.

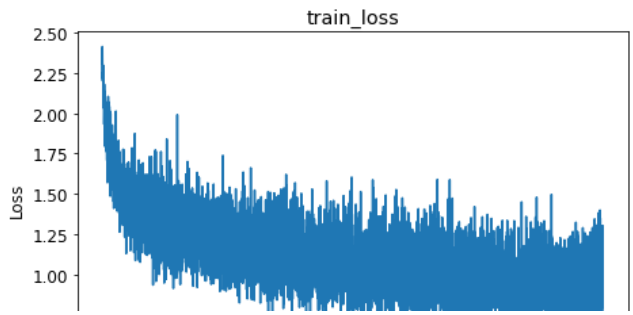
TODO:

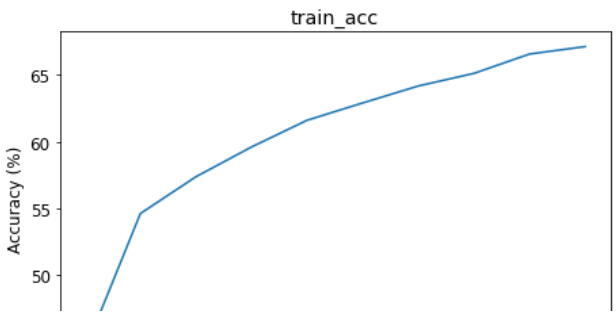
1. Implement the `run_experiment()` function in the `utils/experiments.py` module.
2. If you haven't done so already, it would be an excellent idea to implement the **early stopping** feature of the `Trainer` class.

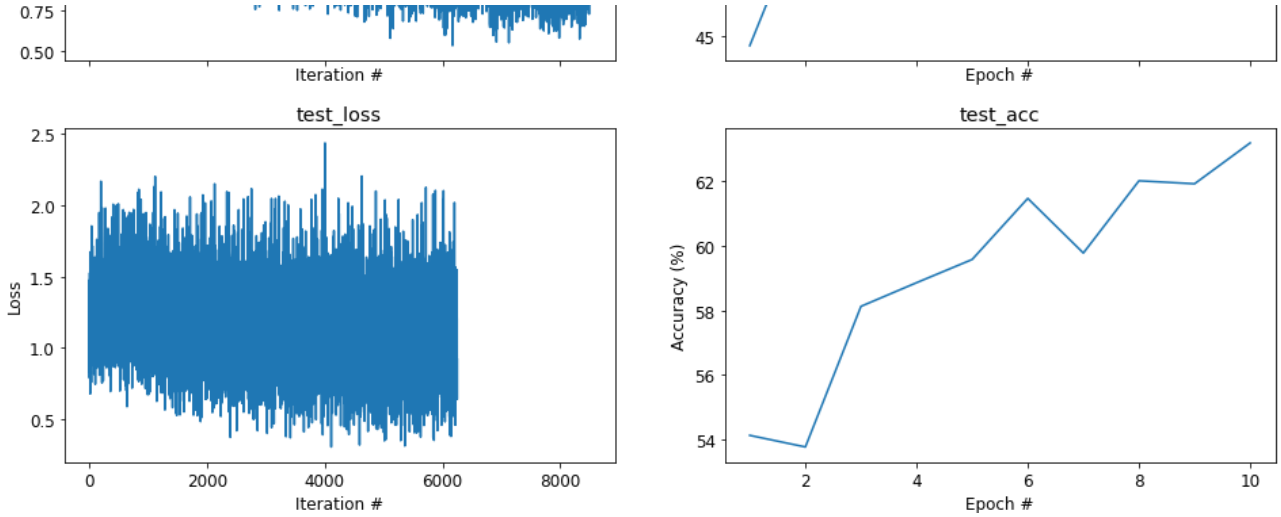
The following block tests that your implementation works. It's also meant to show you that each experiment run creates a result file containing the parameters to reproduce and the `FitResult` object for plotting.

```
import utils.experiments as experiments
from utils.experiments import load_experiment
from utils.plot import plot_fit

# Test experiment1 implementation on a few data samples and with a small model
experiments.run_experiment('test_run', seed=seed, bs_train=50, batches=10, epochs=1
                           filters_per_layer=[32],·layers_per_block=1, pool_every=1

# There should now be a file 'test_run.json' in your `results/` folder.
# We can use it to load the results of the experiment.
cfg, fit_res = load_experiment('results/test_run.json')
_, _ = plot_fit(fit_res)

# And `cfg` contains the exact parameters to reproduce it
print('experiment config: ', cfg)
```

```
Files already downloaded and verified
Files already downloaded and verified
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
  )
  (classifier): Sequential(
    (0): Linear(in_features=8192, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=10, bias=True)
  )
)
--- EPOCH 1/10 ---
train_batch (Avg. Loss 1.552, Accuracy 44.3): 100%|██████████| 850/850 [00:03
test_batch (Avg. Loss 1.313, Accuracy 54.1): 100%|██████████| 625/625 [00:01<
--- EPOCH 2/10 ---
train_batch (Avg. Loss 1.285, Accuracy 54.6): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.281, Accuracy 53.8): 100%|██████████| 625/625 [00:01<
--- EPOCH 3/10 ---
train_batch (Avg. Loss 1.204, Accuracy 57.4): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.169, Accuracy 58.1): 100%|██████████| 625/625 [00:01<
--- EPOCH 4/10 ---
train_batch (Avg. Loss 1.148, Accuracy 59.6): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.157, Accuracy 58.9): 100%|██████████| 625/625 [00:01<
--- EPOCH 5/10 ---
train_batch (Avg. Loss 1.098, Accuracy 61.6): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.127, Accuracy 59.6): 100%|██████████| 625/625 [00:01<
--- EPOCH 6/10 ---
train_batch (Avg. Loss 1.054, Accuracy 62.9): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.083, Accuracy 61.5): 100%|██████████| 625/625 [00:01<
--- EPOCH 7/10 ---
train_batch (Avg. Loss 1.022, Accuracy 64.2): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.113, Accuracy 59.8): 100%|██████████| 625/625 [00:01<
--- EPOCH 8/10 ---
train_batch (Avg. Loss 0.991, Accuracy 65.1): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.055, Accuracy 62.0): 100%|██████████| 625/625 [00:01<
--- EPOCH 9/10 ---
train_batch (Avg. Loss 0.961, Accuracy 66.6): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.050, Accuracy 61.9): 100%|██████████| 625/625 [00:01<
--- EPOCH 10/10 ---
train_batch (Avg. Loss 0.940, Accuracy 67.1): 100%|██████████| 850/850 [00:04
test_batch (Avg. Loss 1.048, Accuracy 63.2): 100%|██████████| 625/625 [00:01<
test_batch (Avg. Loss 1.062, Accuracy 63.1): 100%|██████████| 834/834 [00:03<
==================== Test Results ====================
Test Accuracy: 63.15
Test Accuracy: 1.0618921536371577
=====================================================
*** Output file ./results/test_run.json written
experiment config:  {'run_name': 'test_run', 'out_dir': './results', 'seed':
```







We'll use the following function to load multiple experiment results and plot them together.

```
def plot_exp_results(filename_pattern, results_dir='results'):
    fig = None
    result_files = glob.glob(os.path.join(results_dir, filename_pattern))
    result_files.sort()
    if len(result_files) == 0:
        print(f'No results found for pattern {filename_pattern}.', file=sys.stderr)
        return
    for filepath in result_files:
        m = re.match('exp\d_(\d_)?(.*)\.json', os.path.basename(filepath))
        cfg, fit_res = load_experiment(filepath)
        fig, axes = plot_fit(fit_res, fig, legend=m[2],log_loss=True)
    del cfg['filters_per_layer']
    del cfg['layers_per_block']
    print('common config: ', cfg)
```

Experiment 1.1: Varying the network depth (`L`)

First, we'll test the effect of the network depth on training.

Configuratons:

- `K=32` fixed, with `L=2,4,8,16` varying per run
- `K=64` fixed, with `L=2,4,8,16` varying per run

So 8 different runs in total.

Naming runs: Each run should be named `exp1_1_K{}_L{}` where the braces are placeholders for the values. For example, the first run should be named `exp1_1_K32_L2`.

TODO: Run the experiment on the above configuration. Make sure the result file names are as expected. Use the following blocks to display the results.

```
import utils.experiments as experiments
from utils.experiments import load_experiment
from utils.plot import plot_fit

for l in [2,4,8,16]:
    # Test experiment1 implementation on a few data samples and with a small model
    experiments.run_experiment(f'exp1_1_K32_L{l}',
                               seed=seed,
                               bs_train=50,
                               batches=10000,
                               epochs=30,
                               early_stopping=5,
                               reg=2e-3,
                               lr=1e-3,
                               filters_per_layer=[32],
                               layers_per_block=l,
                               pool_every=4,
                               hidden_dims=[100, 100])
```

```
    (13): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU()
    (15): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU()
    (17): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
    (18): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (19): ReLU()
    (20): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): ReLU()
    (22): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (23): ReLU()
    (24): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU()
    (26): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
    (27): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (28): ReLU()
    (29): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (30): ReLU()
    (31): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (32): ReLU()
    (33): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (34): ReLU()
    (35): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
  )
  (classifier): Sequential(
    (0): Linear(in_features=128, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
--- EPOCH 1/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 850/850 [00:14<
test_batch (Avg. Loss 2.303, Accuracy 9.9): 100%|██████████| 625/625 [00:03<0
--- EPOCH 2/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 850/850 [00:14<
test_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 625/625 [00:02<0
--- EPOCH 3/30 ---
```

```
train_batch (Avg. Loss 2.303, Accuracy 10.1): 100%|██████████| 850/850 [00:14
test_batch (Avg. Loss 2.303, Accuracy 9.9): 100%|██████████| 625/625 [00:02<0
--- EPOCH 4/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.6): 100%|██████████| 850/850 [00:14<
test_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 625/625 [00:02<0
--- EPOCH 5/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 850/850 [00:14<
test_batch (Avg. Loss 2.303, Accuracy 10.1): 100%|██████████| 625/625 [00:03<
--- EPOCH 6/30 ---
train_batch (Avg. Loss 2.303, Accuracy 10.0): 100%|██████████| 850/850 [00:15
test_batch (Avg. Loss 2.303, Accuracy 10.1): 100%|██████████| 625/625 [00:03<
--- EPOCH 7/30 ---
train_batch (Avg. Loss 2.303, Accuracy 10.0): 100%|██████████| 850/850 [00:15
test_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 625/625 [00:03<0
No improvement in 5 epochs, stopping early.
test_batch (Avg. Loss 2.303, Accuracy 10.0): 100%|██████████| 834/834 [00:05<
==================== Test Results ====================

Test Accuracy: 10.0
Test Accuracy: 2.3026433285477634
```

```
plot_exp_results('exp1_1_K32*.json')
```

```
common_config: {'run_name': 'exp1_1_K32_L8', 'out_dir': './results', 'seed':
```

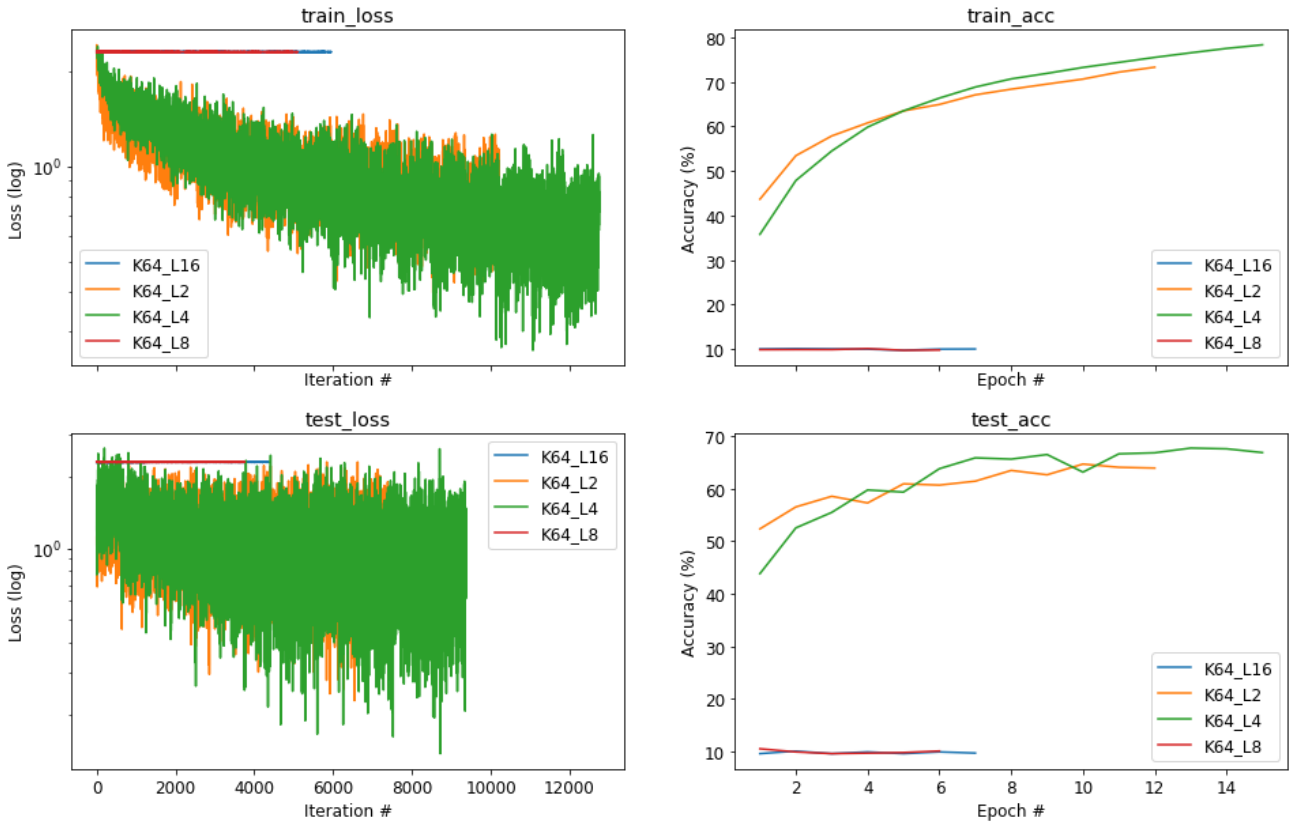
```
for l in [2,4,8,16]:
    # Test experiment1 implementation on a few data samples and with a small model
    experiments.run_experiment(f'exp1_1_K64_L{l}',
                               seed=seed,
                               bs_train=50,
                               batches=10000,
                               epochs=30,
                               early_stopping=5,
                               reg=2e-3,
                               lr=1e-3,
                               filters_per_layer=[64],
                               layers_per_block=l,
                               pool_every=4,
                               hidden_dims=[100, 100])
```

```
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (9): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU()
    (13): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU()
    (15): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU()
    (17): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
  )
  (classifier): Sequential(
    (0): Linear(in_features=4096, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
--- EPOCH 1/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 850/850 [00:17<
test_batch (Avg. Loss 2.303, Accuracy 10.5): 100%|██████████| 625/625 [00:02<
--- EPOCH 2/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 850/850 [00:16<
test_batch (Avg. Loss 2.303, Accuracy 9.9): 100%|██████████| 625/625 [00:02<0
--- EPOCH 3/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 850/850 [00:16<
test_batch (Avg. Loss 2.303, Accuracy 9.6): 100%|██████████| 625/625 [00:02<0
--- EPOCH 4/30 ---
train_batch (Avg. Loss 2.303, Accuracy 10.0): 100%|██████████| 850/850 [00:16
test_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 625/625 [00:02<0
--- EPOCH 5/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.6): 100%|██████████| 850/850 [00:16<
test_batch (Avg. Loss 2.303, Accuracy 9.8): 100%|██████████| 625/625 [00:02<0
--- EPOCH 6/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 850/850 [00:17<
test_batch (Avg. Loss 2.303, Accuracy 10.1): 100%|██████████| 625/625 [00:02<
No improvement in 5 epochs, stopping early.
test_batch (Avg. Loss 2.303, Accuracy 10.0): 100%|██████████| 834/834 [00:04<
==================== Test Results ====================
```

```
Test Accuracy: 10.0
Test Accuracy: 2.3026192173969258
=====================================================
*** Output file ./results/exp1_1_K64_L8.json written
Files already downloaded and verified
Files already downloaded and verified
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (9): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
plot_exp_results('exp1_1_K64*.json')
```

```
common config:  {'run_name': 'exp1_1_K64_L8', 'out_dir': './results', 'seed':
```



Question 1

Analyze your results from experiment 1.1. In particular,

1. Explain the effect of depth on the accuracy. What depth produces the best results and why do you think that's the case?
2. Were there values of `L` for which the network wasn't trainable? what causes this? Suggest two things which may be done to resolve it at least partially.

ANSWER:

1. The optimal depth is four layers. From the graph, we can infer that as we add layers, the accuracy increases, until we increase the depth too much. Thus, there is a lot of maxPooling layers, which results in inputs that are too small for learning.
2. The network wasn't trainable for L - 8,16. This is because with this depth, with the same *every_pooling* value, there are a lot of maximum pool layers, decreasing the input size too much.

Suggestions for partially resolve:

- **Limit the max pool layers** - Max pool layers can be limited. Limits will be calculated based on the input size - for example, allow pooling until the size is 1/8 (h,w) of the original, we'll do it by increase the gap between max pool layers based on **L**.
- **decrease the stride** to get larger output size

Extra Points:

Try solve the problem we saw, using your suggestions. implement it in utils/model.py/YourCodeNet and run again (one of the) experiments where the model wasn't trained to see if the problem have solved).

```
###### Extra Points ######
# run again (one of the) experiments where the model wasn't trained at all,
# but use YourCodeNet to see if your implemented suggestions fixed the problem.
# set ycn=True when you call experiments.run_experiment(...)
# ====== YOUR CODE: ======
# it is recommended to use loops in order to run all experiments.
# ========================
experiments.run_experiment('exp1_1_K32_L8_extra_points',
                           seed=seed,
                           bs_train=50,
                           batches=10000,
                           epochs=30,
                           early_stopping=5,
                           reg=2e-3,
                           lr=1e-3,
```

```
                                filters_per_layer=[32],
                                layers_per_block=8,
                                pool_every=4,
                                hidden_dims=[100, 100],
                                ycn = True)

plot_exp_results('exp1_1_K32_L8_extra_points.json')
```

```
Files already downloaded and verified
Files already downloaded and verified
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): ReLU()
    (6): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
    (9): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (12): ReLU()
    (13): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (14): ReLU()
    (15): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (16): ReLU()
    (17): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode
  )
  (classifier): Sequential(
    (0): Linear(in_features=2048, out_features=100, bias=True)
    (1): ReLU()
    (2): Linear(in_features=100, out_features=100, bias=True)
    (3): ReLU()
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
--- EPOCH 1/30 ---
train_batch (Avg. Loss 2.303, Accuracy 10.1): 100%|██████████| 850/850 [00:10
test_batch (Avg. Loss 2.303, Accuracy 10.3): 100%|██████████| 625/625 [00:02<
--- EPOCH 2/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.9): 100%|██████████| 850/850 [00:10<
test_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 625/625 [00:02<0
--- EPOCH 3/30 ---
train_batch (Avg. Loss 2.303, Accuracy 10.0): 100%|██████████| 850/850 [00:10
test_batch (Avg. Loss 2.303, Accuracy 9.7): 100%|██████████| 625/625 [00:02<0
--- EPOCH 4/30 ---
train_batch (Avg. Loss 2.303, Accuracy 9.9): 100%|██████████| 850/850 [00:10<
test_batch (Avg. Loss 2.303, Accuracy 9.6): 100%|██████████| 625/625 [00:02<0
--- EPOCH 5/30 ---
```

Experiment 1.2: Varying the number of filters per layer (`K`)

```
--- EPOCH 6/30 ---
```

Now we'll test the effect of the number of convolutional filters in each layer.

Configuratons:

- `L=2` fixed, with `K=[32],[64],[128],[256]` varying per run.
- `L=4` fixed, with `K=[32],[64],[128],[256]` varying per run.
- `L=8` fixed, with `K=[32],[64],[128],[256]` varying per run.

So 12 different runs in total. To clarify, each run `K` takes the value of a list with a single element.

Naming runs: Each run should be named `exp1_2_L{}_K{}` where the braces are placeholders for the values. For example, the first run should be named `exp1_2_L2_K32`.

TODO: Run the experiment on the above configuration. Make sure the result file names are as expected. Use the following blocks to display the results.

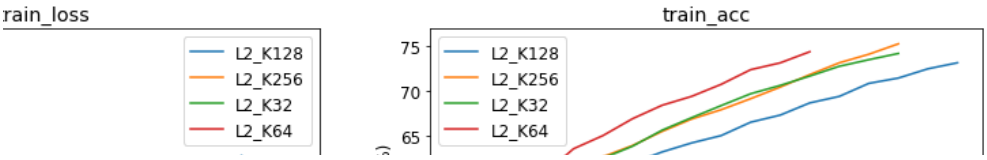
```
for l in [2,4,8]:
    for k in [32,64,128,256]:
        # Test experiment1 implementation on a few data samples and with a small mo
        experiments.run_experiment(f'exp1_2_L{l}_K{k}',
                                   seed=seed,
                                   bs_train=50,
                                   batches=10000,
                                   epochs=30,
                                   early_stopping=5,
                                   reg=2e-3,
                                   lr=1e-3,
                                   filters_per_layer=[k],
                                   layers_per_block=l,
                                   pool_every=4,
                                   hidden_dims=[100, 100])
```

```
--- EPOCH 2/30 ---
train_batch (Avg. Loss 1.308, Accuracy 53.2): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.237, Accuracy 56.1): 100%|██████████| 625/625 [00:02<
--- EPOCH 3/30 ---
train_batch (Avg. Loss 1.217, Accuracy 56.4): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.192, Accuracy 57.0): 100%|██████████| 625/625 [00:02<
--- EPOCH 4/30 ---
train_batch (Avg. Loss 1.154, Accuracy 58.7): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.153, Accuracy 58.2): 100%|██████████| 625/625 [00:02<
--- EPOCH 5/30 ---
train_batch (Avg. Loss 1.092, Accuracy 60.9): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.134, Accuracy 59.5): 100%|██████████| 625/625 [00:02<
--- EPOCH 6/30 ---
train_batch (Avg. Loss 1.047, Accuracy 62.7): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.109, Accuracy 59.7): 100%|██████████| 625/625 [00:02<
--- EPOCH 7/30 ---
train_batch (Avg. Loss 1.007, Accuracy 64.0): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.188, Accuracy 58.8): 100%|██████████| 625/625 [00:02<
--- EPOCH 8/30 ---
train_batch (Avg. Loss 0.966, Accuracy 65.5): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.088, Accuracy 60.7): 100%|██████████| 625/625 [00:02<
--- EPOCH 9/30 ---
train_batch (Avg. Loss 0.927, Accuracy 66.9): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.060, Accuracy 61.8): 100%|██████████| 625/625 [00:02<
--- EPOCH 10/30 ---
train_batch (Avg. Loss 0.897, Accuracy 67.9): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.111, Accuracy 61.1): 100%|██████████| 625/625 [00:02<
--- EPOCH 11/30 ---
train_batch (Avg. Loss 0.863, Accuracy 69.2): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.047, Accuracy 63.0): 100%|██████████| 625/625 [00:02<
--- EPOCH 12/30 ---
train_batch (Avg. Loss 0.830, Accuracy 70.4): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.018, Accuracy 64.0): 100%|██████████| 625/625 [00:02<
--- EPOCH 13/30 ---
train_batch (Avg. Loss 0.794, Accuracy 71.8): 100%|██████████| 850/850 [00:34
```

```
test_batch (Avg. Loss 1.046, Accuracy 62.5): 100%|██████████| 625/625 [00:02<
--- EPOCH 14/30 ---
train_batch (Avg. Loss 0.761, Accuracy 73.1): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.016, Accuracy 64.3): 100%|██████████| 625/625 [00:02<
--- EPOCH 15/30 ---
train_batch (Avg. Loss 0.731, Accuracy 74.1): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.121, Accuracy 61.1): 100%|██████████| 625/625 [00:02<
--- EPOCH 16/30 ---
train_batch (Avg. Loss 0.702, Accuracy 75.2): 100%|██████████| 850/850 [00:34
test_batch (Avg. Loss 1.064, Accuracy 63.2): 100%|██████████| 625/625 [00:02<
No improvement in 5 epochs, stopping early.
test_batch (Avg. Loss 1.073, Accuracy 62.8): 100%|██████████| 834/834 [00:05<
==================== Test Results ====================
Test Accuracy: 62.75
Test Accuracy: 1.0727084506329874
=====================================================
*** Output file ./results/exp1_2_L2_K256.json written
Files already downloaded and verified
Files already downloaded and verified
ConvClassifier(
  (feature_extractor): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

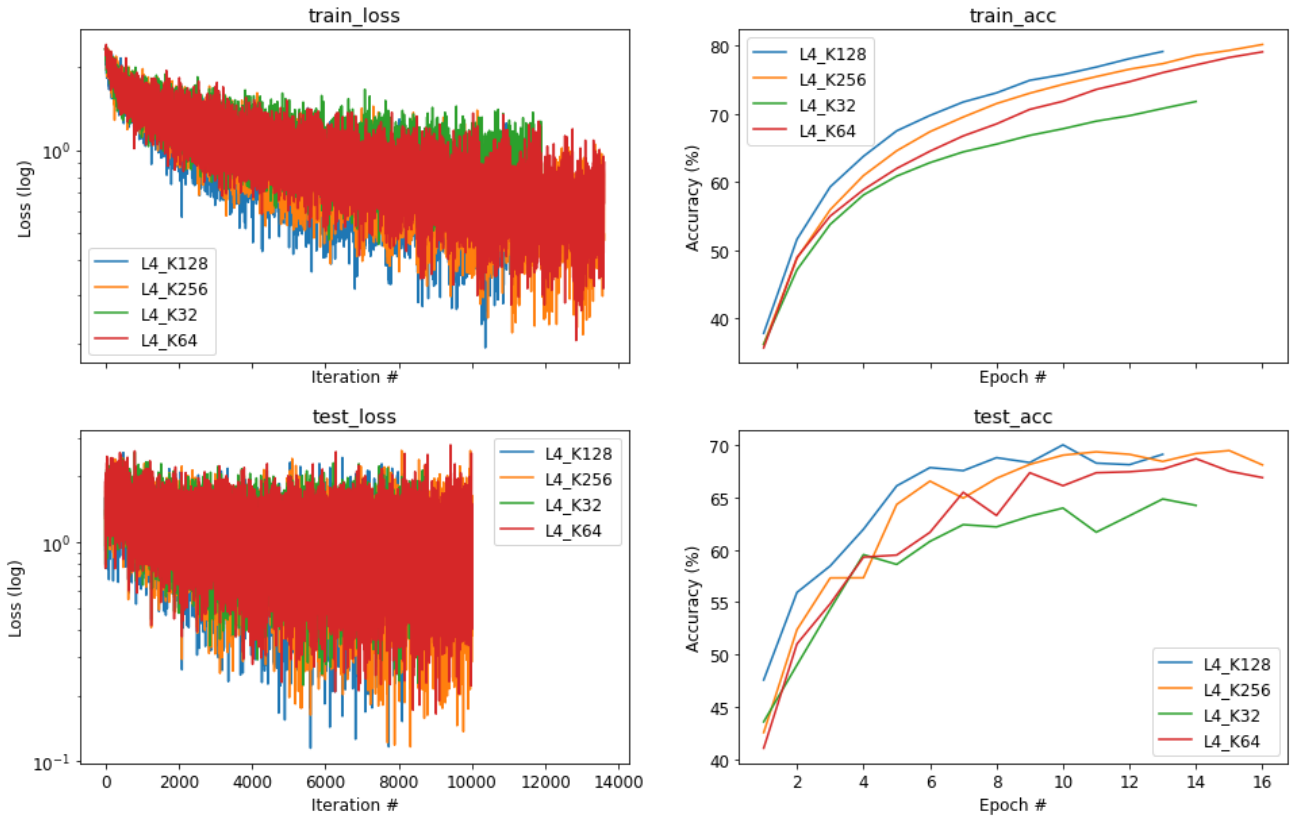
```
plot_exp_results('exp1_2_L2*.json')
```

```
run_name': 'exp1_2_L2_K64', 'out_dir': './results', 'seed': 42, 'bs_train': 50
```



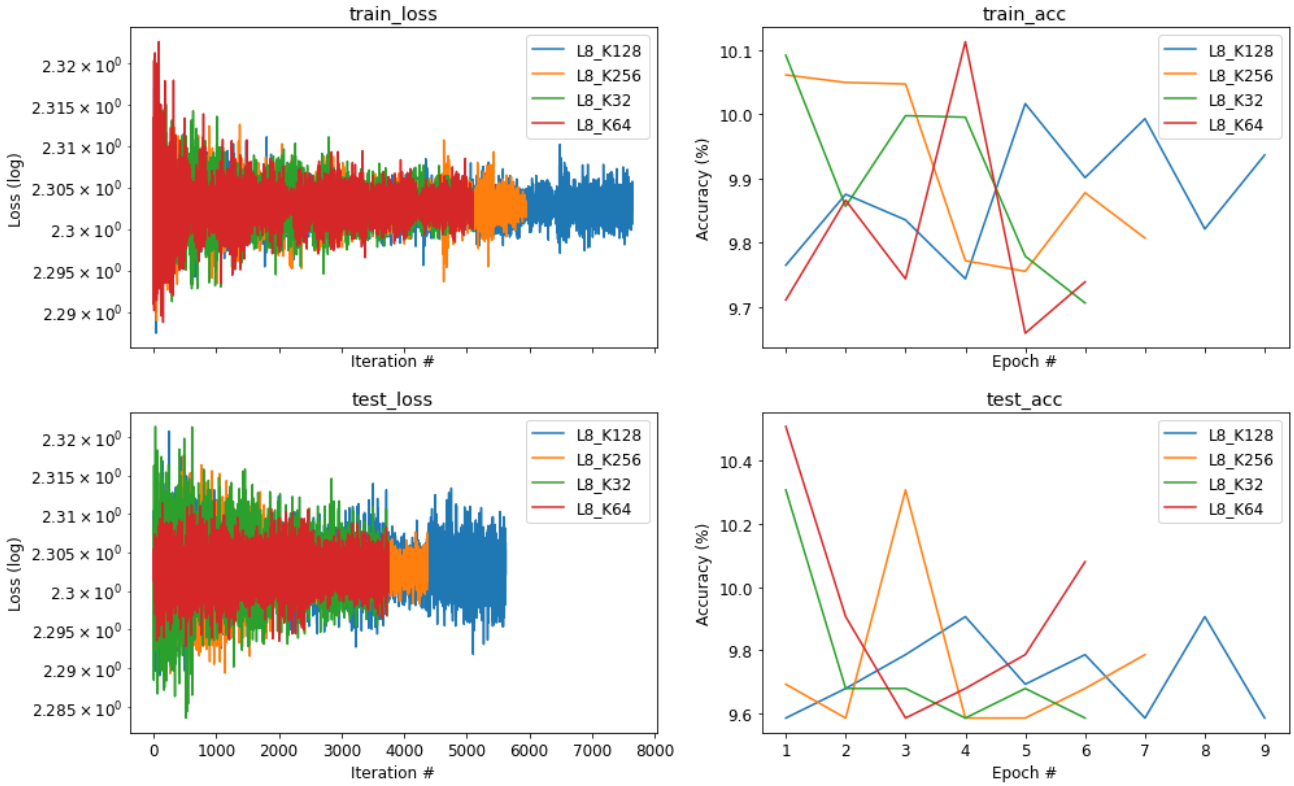

```
plot_exp_results('exp1_2_L4*.json')
```

```
common config:  {'run_name': 'exp1_2_L4_K64', 'out_dir': './results', 'seed':
```




```
plot_exp_results('exp1_2_L8*.json')
```

```
common config:  {'run_name': 'exp1_2_L8_K64', 'out_dir': './results', 'seed':
```



Question 2

Analyze your results from experiment 1.2. In particular, compare to the results of experiment 1.1. For a spesific value of L, how the preformance change with respect to K? Does we saw the same phenomena in 1.1 for a spresific value of K?

ANSWER:

As is observed in Experiment 1.1, we see the same phenomenon with L = 8. With a large number of max pooling layers, the network is unable to learn, and the K values have little significance.

Lower values of K (32,64) produced better results, while higher values of K (128,256) produced worse results for L = 2.

We see opposite results for L = 4, higher values of K (128,256) produce better results, while lower values (32,64) produce the worst results.

We can see that the effect of changing K values is less significant than changing L values