460cw2_final

February 13, 2019

1 Coursework2: Convolutional Neural Networks

1.1 instructions

Please submit a version of this notebook containing your answers **together with your trained model** on CATe as CW2.zip. Write your answers in the cells below each question.

A PDF version of this notebook is also provided in case the figures do not render correctly.

The deadline for submission is 19:00, Thu 14th February, 2019

1.1.1 Setting up working environment

For this coursework you will need to train a large network, therefore we recommend you work with Google Colaboratory, which provides free GPU time. You will need a Google account to do so.

Please log in to your account and go to the following page: https://colab.research.google.com. Then upload this notebook.

For GPU support, go to "Edit" -> "Notebook Settings", and select "Hardware accelerator" as "GPU".

You will need to install pytorch by running the following cell:

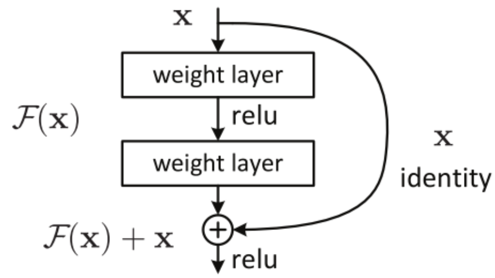
```
In [1]: !pip install torch torchvision
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.6/dist-packages (1.0.0)
Requirement already satisfied: torchvision in /usr/local/lib/python3.6/dist-packages (0.2.1)
Requirement already satisfied: pillow>=4.1.1 in /usr/local/lib/python3.6/dist-packages (from to
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from torchvisio
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from torchvision)
```

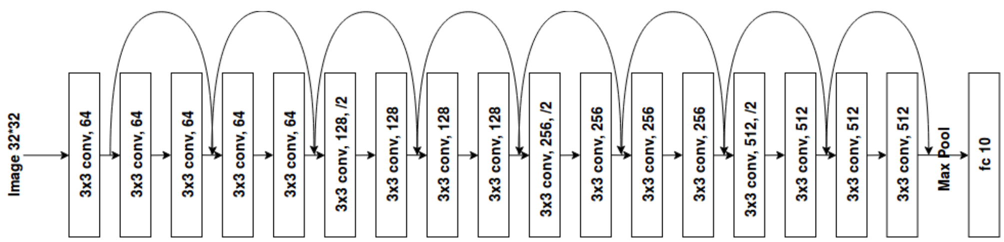
1.2 Introduction

For this coursework you will implement one of the most commonly used model for image recognition tasks, the Residual Network. The architecture is introduced in 2015 by Kaiming He, et al. in the paper "Deep residual learning for image recognition".

In a residual network, each block contains some convolutional layers, plus "skip" connections, which allow the activations to by pass a layer, and then be summed up with the activations of the skipped layer. The image below illustrates a building block in residual networks.



resnet-block



resnet

Depending on the number of building blocks, resnets can have different architectures, for example ResNet-50, ResNet-101 and etc. Here you are required to build ResNet-18 to perform classification on the CIFAR-10 dataset, therefore your network will have the following architecture:

1.3 Part 1 (40 points)

In this part, you will use basic pytorch operations to define the 2D convolution and max pooling operation.

1.3.1 YOUR TASK

- implement the forward pass for Conv2D and MaxPool2D
- You can only fill in the parts which are specified as "YOUR CODE HERE"
- You are **NOT** allowed to use the torch.nn module and the conv2d/maxpooling functions in torch.nn.functional

```
In [0]: import torch
        import torch.nn as nn
        import torch.nn.functional as F
```

```
In [0]: class Conv2D(nn.Module):

            def __init__(self, inchannel, outchannel, kernel_size, stride, padding, bias = Tru

                super(Conv2D, self).__init__()

                self.inchannel = inchannel
                self.outchannel = outchannel
```

```python
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding

        self.weights = nn.Parameter(torch.Tensor(outchannel, inchannel,
                                                 kernel_size, kernel_size))
        self.weights.data.normal_(-0.1, 0.1)

        if bias:
            self.bias = nn.Parameter(torch.Tensor(outchannel, ))
            self.bias.data.normal_(-0.1, 0.1)
        else:
            self.bias = None


    def forward(self, x):

        ##############################################################
        #                       YOUR CODE HERE                       #
        ##############################################################
        x_unfolded = F.unfold(x, kernel_size=self.kernel_size, stride=self.stride, padd
        x_h = (x.shape[2] - self.kernel_size + 2 * self.padding) // self.stride + 1
        x_w = (x.shape[3] - self.kernel_size + 2 * self.padding) // self.stride + 1

        output = x_unfolded.transpose(1, 2).matmul(self.weights.view(self.weights.shape
        output = output.view(x.shape[0], self.outchannel, x_h, x_w)

        if self.bias is not None:
            bias = self.bias.view(1, self.outchannel, 1, 1)
            output = torch.add(output, bias)


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


        return output
```

```python
In [0]: class MaxPool2D(nn.Module):

            def __init__(self, pooling_size):
                # assume pooling_size = kernel_size = stride

                super(MaxPool2D, self).__init__()

                self.pooling_size = pooling_size
```

```
    def forward(self, x):


        ###############################################################
        #                       YOUR CODE HERE                        #
        ###############################################################

        bs, channel, h, w = x.shape
        k = self.pooling_size
        s = self.pooling_size
        out_h = (h - k) // s + 1
        out_w = (w - k) // s + 1
        x_unfolded = F.unfold(x, kernel_size=k, stride=s)
        x_unfolded = x_unfolded.view(bs, channel, k*k, out_h*out_w)
        output = torch.max(x_unfolded, 2)[0].view(bs, channel, out_h, out_w)

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


        return output
```

```
In [0]: # define resnet building blocks

        class ResidualBlock(nn.Module):
            def __init__(self, inchannel, outchannel, stride=1):

                super(ResidualBlock, self).__init__()

                self.left = nn.Sequential(Conv2D(inchannel, outchannel, kernel_size=3,
                                                 stride=stride, padding=1, bias=False),
                                          nn.BatchNorm2d(outchannel),
                                          nn.ReLU(inplace=True),
                                          Conv2D(outchannel, outchannel, kernel_size=3,
                                                 stride=1, padding=1, bias=False),
                                          nn.BatchNorm2d(outchannel))

                self.shortcut = nn.Sequential()

                if stride != 1 or inchannel != outchannel:

                    self.shortcut = nn.Sequential(Conv2D(inchannel, outchannel,
                                                         kernel_size=1, stride=stride,
                                                         padding = 0, bias=False),
```

```
                                                    nn.BatchNorm2d(outchannel) )

        def forward(self, x):

            out = self.left(x)

            out += self.shortcut(x)

            out = F.relu(out)

            return out
```

In [0]:
```
# define resnet

class ResNet(nn.Module):

    def __init__(self, ResidualBlock, num_classes = 10):

        super(ResNet, self).__init__()

        self.inchannel = 64
        self.conv1 = nn.Sequential(Conv2D(3, 64, kernel_size = 3, stride = 1,
                                          padding = 1, bias = False),
                                   nn.BatchNorm2d(64),
                                   nn.ReLU())

        self.layer1 = self.make_layer(ResidualBlock, 64, 2, stride = 1)
        self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride = 2)
        self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride = 2)
        self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride = 2)
        self.maxpool = MaxPool2D(4)
        self.fc = nn.Linear(512, num_classes)


    def make_layer(self, block, channels, num_blocks, stride):

        strides = [stride] + [1] * (num_blocks - 1)

        layers = []

        for stride in strides:

            layers.append(block(self.inchannel, channels, stride))

            self.inchannel = channels

        return nn.Sequential(*layers)
```

```python
    def forward(self, x):

        x = self.conv1(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.maxpool(x)

        x = x.view(x.size(0), -1)

        x = self.fc(x)

        return x


def ResNet18():
    return ResNet(ResidualBlock)
```

1.4 Part 2 (40 points)

In this part, you will train the ResNet-18 defined in the previous part on the CIFAR-10 dataset. Code for loading the dataset, training and evaluation are provided.

1.4.1 Your Task

1. Train your network to achieve the best possible test set accuracy after a maximum of 10 epochs of training.
2. You can use techniques such as optimal hyper-parameter searching, data pre-processing
3. If necessary, you can also use another optimiser
4. **Answer the following question:** Given such a network with a large number of trainable parameters, and a training set of a large number of data, what do you think is the best strategy for hyperparameter searching?

YOUR ANSWER FOR 2.4 HERE

A: Acutually, the Grid Search we use here is not a good idea for such a large number of trainable parameters and a training set of a large number of data if we are trying to optimize more than one hyperparameters, as it has to try out all possible hyperparameter-combinations to find out a suitable ones, which is really time consuming.

A better way to invest our time is to use Random Search. The difference between Grid Search and Random Search is the strategy of picking hyperparameter-combinations – Random Search picks the point randomly from the configuration space.

As we pick hyperparameter-combinations randomly, it's extremely unlikely to select the same variables more than once. Thus it's more likely to try out different values of hyperparameters within fewer iterations if the search space is in higher dimentions, although it does not guarantee to find best hyperparameters.

In [7]:
```
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset

import numpy as np

import torchvision.transforms as T


transform = T.ToTensor()


# load data

NUM_TRAIN = 49000
print_every = 700


data_dir = './data'
cifar10_train = dset.CIFAR10(data_dir, train=True, download=True, transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10(data_dir, train=True, download=True, transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN, 50000)))

cifar10_test = dset.CIFAR10(data_dir, train=False, download=True, transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)


USE_GPU = True
dtype = torch.float32

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
    print('using CPU to train')
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.
Files already downloaded and verified
Files already downloaded and verified
```

In [0]:
```
from time import time

def check_accuracy(loader, model):
    # function for test accuracy on validation and test set

    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 * acc))
    return acc


def train_part(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train for

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
#         print(len(loader_train))
        t0 = time()
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)
```

```
                # Zero out all of the gradients for the variables which the optimizer
                # will update.
                optimizer.zero_grad()

                loss.backward()

                # Update the parameters of the model using the gradients
                optimizer.step()

                t1 = time()
                if t % print_every == 0:
#                 if t % print_every == 0 and t != 0:
                    print('Epoch: %d, Iteration %d, loss = %.4f, time= %.4fs' % (e, t, los
#                       check_accuracy(loader_val, model)
                    print()
```

In [9]:
```
# code for optimising your network performance

##############################################################
#                         YOUR CODE HERE                     #
##############################################################
lr_space = np.logspace(-3, -1, 6)
lr_res_l = list()
for lr in lr_space:
    model = ResNet18()
    print('training with learning rate: ' + str(lr))
    optimizer = optim.Adam(model.parameters(), lr=lr)
    train_part(model, optimizer, epochs = 5)
    print('validated accuracy:')
    lr_res_l.append(check_accuracy(loader_val, model))
    print()

print('all possible learning rate: ' + str(lr_space))
print('validation accuracy: ' + str(lr_res_l))
max_idx = lr_res_l.index(max(lr_res_l))
best_lr = lr_space[max_idx]
print('best learning rate: ' + str(best_lr))

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


# define and train the network
model = ResNet18()
optimizer = optim.Adam(model.parameters(), lr=best_lr)

train_part(model, optimizer, epochs = 10)
```

```python
        # report test set accuracy

        check_accuracy(loader_test, model)


        # save the model
        torch.save(model.state_dict(), 'model.pt')
```

```
training with learning rate: 0.001
Epoch: 0, Iteration 0, loss = 2.5858, time= 0.3367s

Epoch: 0, Iteration 700, loss = 1.9719, time= 238.0319s

Epoch: 1, Iteration 0, loss = 1.9323, time= 0.3196s

Epoch: 1, Iteration 700, loss = 1.5008, time= 238.4093s

Epoch: 2, Iteration 0, loss = 1.4411, time= 0.3192s

Epoch: 2, Iteration 700, loss = 1.3527, time= 238.6766s

Epoch: 3, Iteration 0, loss = 1.4217, time= 0.3189s

Epoch: 3, Iteration 700, loss = 1.1171, time= 238.7984s

Epoch: 4, Iteration 0, loss = 1.3013, time= 0.3204s

Epoch: 4, Iteration 700, loss = 1.2109, time= 238.7368s

validated accuracy:
Checking accuracy on validation set
Got 603 / 1000 correct (60.30)

training with learning rate: 0.0025118864315095794
Epoch: 0, Iteration 0, loss = 2.5391, time= 0.2242s

Epoch: 0, Iteration 700, loss = 1.7197, time= 238.4924s

Epoch: 1, Iteration 0, loss = 1.7362, time= 0.3181s

Epoch: 1, Iteration 700, loss = 1.3102, time= 238.4779s

Epoch: 2, Iteration 0, loss = 1.2252, time= 0.3192s

Epoch: 2, Iteration 700, loss = 0.9599, time= 238.5283s
```

```
Epoch: 3, Iteration 0, loss = 0.7846, time= 0.3186s

Epoch: 3, Iteration 700, loss = 0.9112, time= 238.5473s

Epoch: 4, Iteration 0, loss = 0.8565, time= 0.3186s

Epoch: 4, Iteration 700, loss = 0.6755, time= 238.5469s

validated accuracy:
Checking accuracy on validation set
Got 698 / 1000 correct (69.80)

training with learning rate: 0.00630957344480193
Epoch: 0, Iteration 0, loss = 2.6523, time= 0.2243s

Epoch: 0, Iteration 700, loss = 1.7009, time= 238.3580s

Epoch: 1, Iteration 0, loss = 1.4874, time= 0.3203s

Epoch: 1, Iteration 700, loss = 1.0109, time= 238.4730s

Epoch: 2, Iteration 0, loss = 1.1483, time= 0.3159s

Epoch: 2, Iteration 700, loss = 0.8117, time= 238.4568s

Epoch: 3, Iteration 0, loss = 0.8009, time= 0.3186s

Epoch: 3, Iteration 700, loss = 0.7222, time= 238.4502s

Epoch: 4, Iteration 0, loss = 0.5385, time= 0.3189s

Epoch: 4, Iteration 700, loss = 0.4198, time= 238.3417s

validated accuracy:
Checking accuracy on validation set
Got 742 / 1000 correct (74.20)

training with learning rate: 0.015848931924611134
Epoch: 0, Iteration 0, loss = 2.5601, time= 0.2226s

Epoch: 0, Iteration 700, loss = 1.6140, time= 238.3518s

Epoch: 1, Iteration 0, loss = 1.2816, time= 0.3180s

Epoch: 1, Iteration 700, loss = 0.9810, time= 238.3833s

Epoch: 2, Iteration 0, loss = 1.2582, time= 0.3201s
```

```
Epoch: 2, Iteration 700, loss = 0.7946, time= 238.5267s

Epoch: 3, Iteration 0, loss = 0.7970, time= 0.3193s

Epoch: 3, Iteration 700, loss = 0.8155, time= 238.5680s

Epoch: 4, Iteration 0, loss = 0.5917, time= 0.3189s

Epoch: 4, Iteration 700, loss = 0.5096, time= 238.3451s

validated accuracy:
Checking accuracy on validation set
Got 718 / 1000 correct (71.80)

training with learning rate: 0.039810717055349734
Epoch: 0, Iteration 0, loss = 2.4496, time= 0.2229s

Epoch: 0, Iteration 700, loss = 1.7018, time= 238.2853s

Epoch: 1, Iteration 0, loss = 1.3099, time= 0.3182s

Epoch: 1, Iteration 700, loss = 1.3237, time= 238.4245s

Epoch: 2, Iteration 0, loss = 1.2244, time= 0.3187s

Epoch: 2, Iteration 700, loss = 1.2020, time= 237.9042s

Epoch: 3, Iteration 0, loss = 0.9417, time= 0.3191s

Epoch: 3, Iteration 700, loss = 0.7641, time= 238.1949s

Epoch: 4, Iteration 0, loss = 0.5359, time= 0.3188s

Epoch: 4, Iteration 700, loss = 0.6986, time= 237.9509s

validated accuracy:
Checking accuracy on validation set
Got 726 / 1000 correct (72.60)

training with learning rate: 0.1
Epoch: 0, Iteration 0, loss = 2.3927, time= 0.2226s

Epoch: 0, Iteration 700, loss = 1.8770, time= 237.6680s

Epoch: 1, Iteration 0, loss = 1.7621, time= 0.3175s

Epoch: 1, Iteration 700, loss = 1.3975, time= 237.7558s
```

```
Epoch: 2, Iteration 0, loss = 1.4173, time= 0.3176s

Epoch: 2, Iteration 700, loss = 1.3308, time= 237.7140s

Epoch: 3, Iteration 0, loss = 1.2241, time= 0.3193s

Epoch: 3, Iteration 700, loss = 1.0543, time= 237.6698s

Epoch: 4, Iteration 0, loss = 0.8630, time= 0.3185s

Epoch: 4, Iteration 700, loss = 0.7519, time= 237.6900s

validated accuracy:
Checking accuracy on validation set
Got 681 / 1000 correct (68.10)

all possible learning rate: [0.001      0.00251189 0.00630957 0.01584893 0.03981072 0.1        ]
validation accuracy: [0.603, 0.698, 0.742, 0.718, 0.726, 0.681]
best learning rate: 0.00630957344480193
Epoch: 0, Iteration 0, loss = 2.5553, time= 0.2224s

Epoch: 0, Iteration 700, loss = 1.6286, time= 237.8244s

Epoch: 1, Iteration 0, loss = 1.5826, time= 0.3187s

Epoch: 1, Iteration 700, loss = 1.2909, time= 237.7698s

Epoch: 2, Iteration 0, loss = 1.0481, time= 0.3183s

Epoch: 2, Iteration 700, loss = 0.6985, time= 237.9184s

Epoch: 3, Iteration 0, loss = 0.7758, time= 0.3185s

Epoch: 3, Iteration 700, loss = 0.7248, time= 237.8557s

Epoch: 4, Iteration 0, loss = 0.6102, time= 0.3186s

Epoch: 4, Iteration 700, loss = 0.4394, time= 237.7732s

Epoch: 5, Iteration 0, loss = 0.4024, time= 0.3183s

Epoch: 5, Iteration 700, loss = 0.2623, time= 237.7283s

Epoch: 6, Iteration 0, loss = 0.2006, time= 0.3177s

Epoch: 6, Iteration 700, loss = 0.2443, time= 237.8605s

Epoch: 7, Iteration 0, loss = 0.1694, time= 0.3192s
```

```
Epoch: 7, Iteration 700, loss = 0.1627, time= 237.8938s

Epoch: 8, Iteration 0, loss = 0.1068, time= 0.3182s

Epoch: 8, Iteration 700, loss = 0.1221, time= 237.9039s

Epoch: 9, Iteration 0, loss = 0.1461, time= 0.3185s

Epoch: 9, Iteration 700, loss = 0.3010, time= 237.8311s

Checking accuracy on test set
Got 7730 / 10000 correct (77.30)
```

```
In [0]: ## Part 3 (20 points)
```

The code provided below will allow you to visualise the feature maps computed by different layers of your network. Run the code (install matplotlib if necessary) and **answer the following questions**:

1. Compare the feature maps from low-level layers to high-level layers, what do you observe?
2. Use the training log, reported test set accuracy and the feature maps, analyse the performance of your network. If you think the performance is sufficiently good, explain why; if not, what might be the problem and how can you improve the performance?
3. What are the other possible ways to analyse the performance of your network?

YOUR ANSWER FOR PART 3 HERE

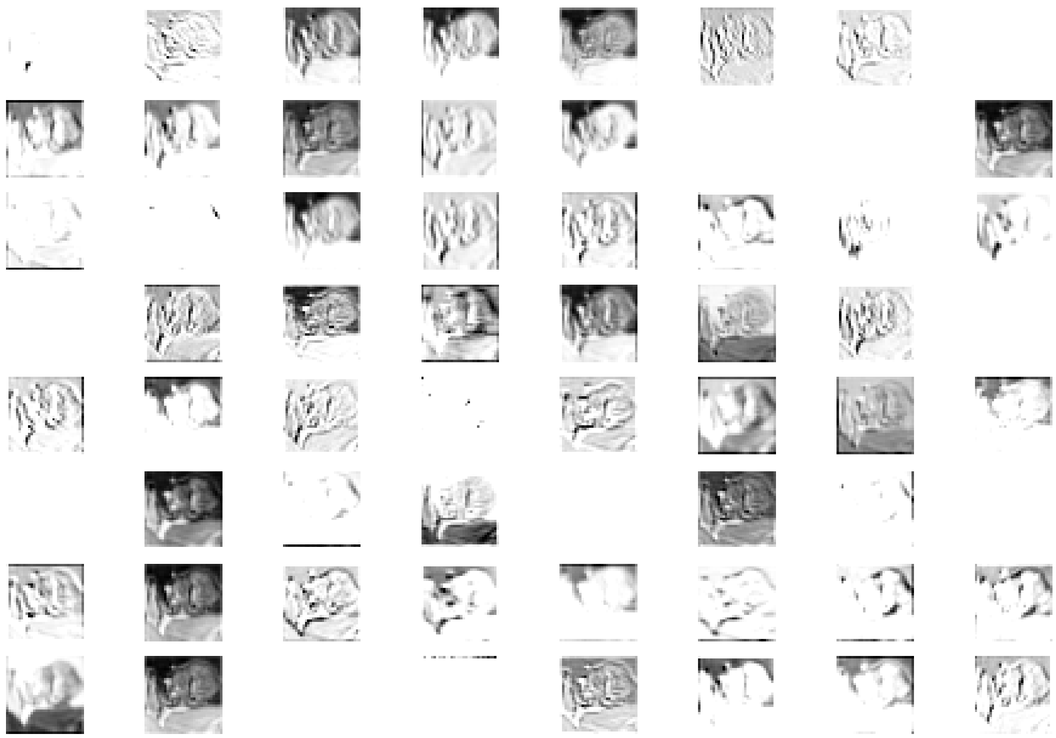
A: 1. The feature maps from low-level layers encode the simple structures, which looks like part of pictures. As we go deeper level, the layers build on top of each other and learn to encode more complex patterns, like the meaning of the pictures, which may be hard to visualize via feature map.

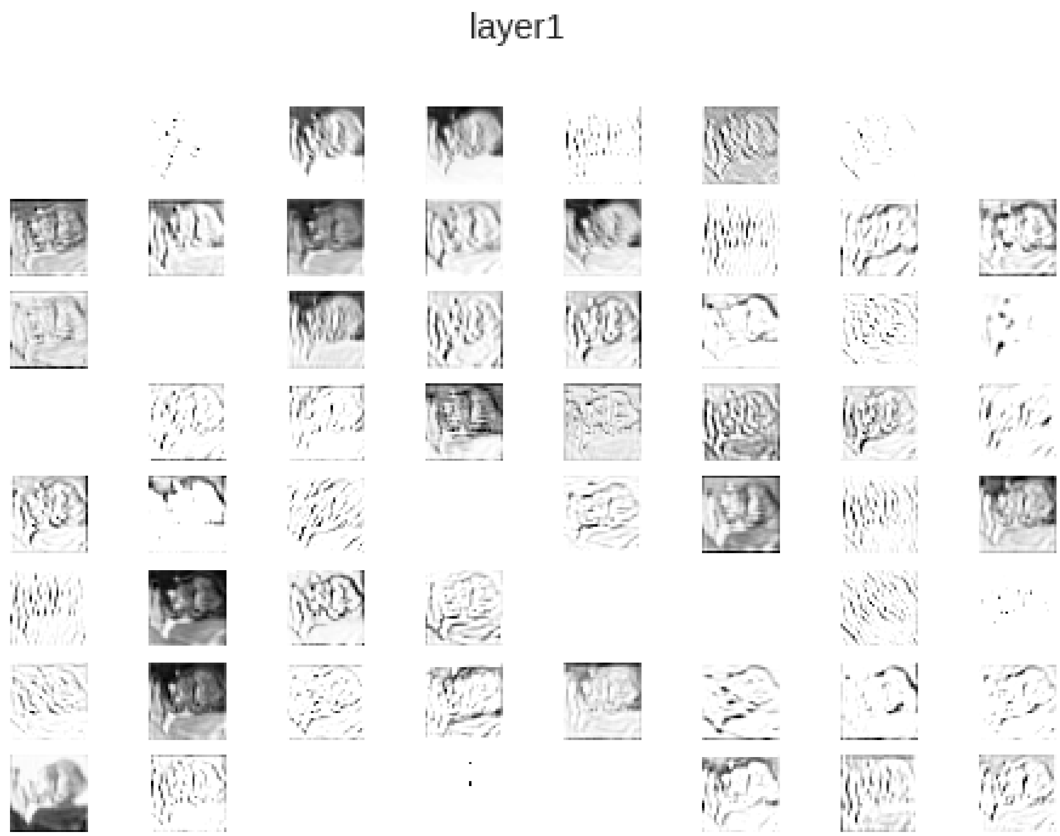
2. The performance of the built network is not sufficient good. There may be overfit of the model. At the final train of the network, a increase of the loss at epoch 9 can be found, which means that at the previous training process, the model may fit well for some part of the data, but not so well for others. And in previous attempt (not shown in this notebook), we found that the accuracy of the training set is about 10 ~ 20 percent higher than validation set. We can add some normalized term to the loss function or add some drop out layer within the network to deliver these kind of problem in our network.
3. Using the false positive, true positive, true negatives, false negatives to build confusion matrix, and then calculate the precision an recall of each class. By comparing the precision and recall of each class, we can analyse the performance of the network. Any further step may be take such as assigning weight to each class, then combine precision and recall of different classes into one single target.

In [11]:
```
#!pip install matplotlib

import matplotlib.pyplot as plt

plt.tight_layout()


activation = {}
def get_activation(name):
    def hook(model, input, output):
        activation[name] = output.detach()
    return hook

vis_labels = ['conv1', 'layer1', 'layer2', 'layer3', 'layer4']

for l in vis_labels:

    getattr(model, l).register_forward_hook(get_activation(l))


data, _ = cifar10_test[0]
data = data.unsqueeze_(0).to(device = device, dtype = dtype)

output = model(data)



for idx, l in enumerate(vis_labels):

    act = activation[l].squeeze()

    if idx < 2:
        ncols = 8
    else:
        ncols = 32

    nrows = act.size(0) // ncols

    fig, axarr = plt.subplots(nrows, ncols)
    fig.suptitle(l)


    for i in range(nrows):
        for j in range(ncols):
            axarr[i, j].imshow(act[i * nrows + j].cpu())
            axarr[i, j].axis('off')
```

```
<Figure size 576x396 with 0 Axes>
```

conv1

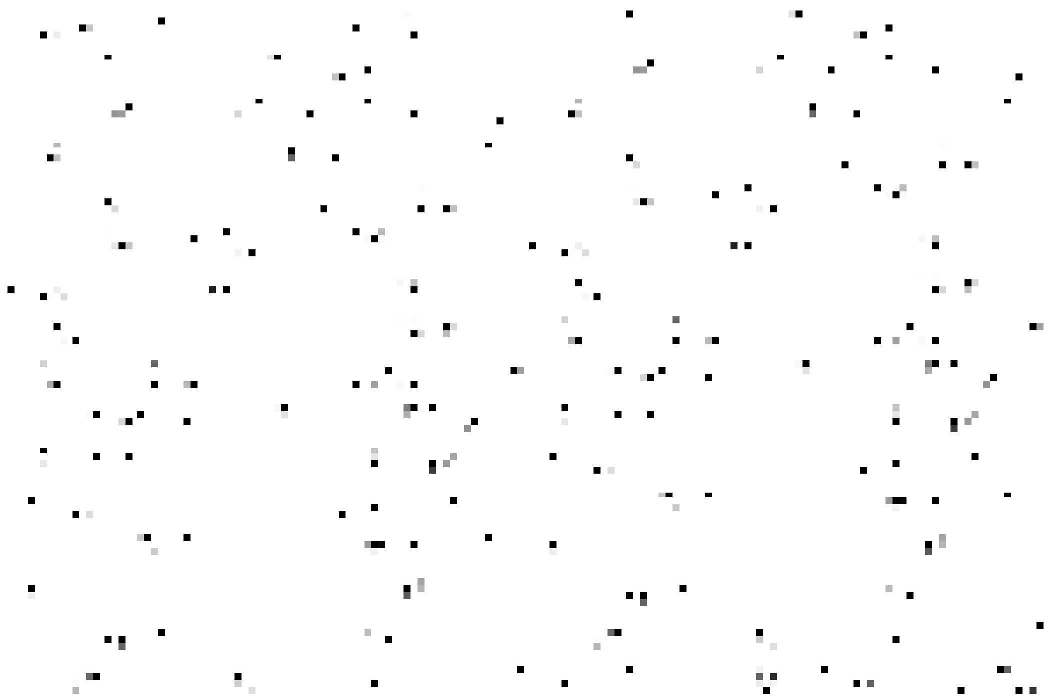



layer1

layer2

layer3

layer4



=============== END OF CW2 ===============