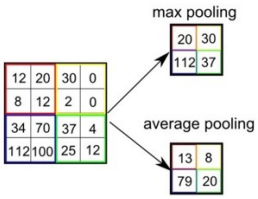
BS6207

Q1:

MaxPool2d and AvgPool2d:

The pooling layers are used to scale the data by using kernels. These 2 functions are similar, the only difference is max is to calculate the max value of a kernel scanned area, avg is to calculate the mean.



I check my results are the same as torch.

My MaxPool2D

```
class MyMaxPool2D(nn.Module):
  def __init__(self, kernel_size, stride):
    super(MyMaxPool2D, self).__init__()
    self.stride = stride
    self.kernel_size = kernel_size
    self.kernel_height = kernel_size[0]
    self.kernel_width = kernel_size[1]

  def forward(self, x):
    x_batch = x.size(0)
    x_channel = x.size(1)
    x_height = x.size(2)
    x_width = x.size(3)

    y_height = int((x_height - self.kernel_height) / self.stride) + 1
    y_width = int((x_width - self.kernel_width) / self.stride) + 1
    y = torch.zeros((x_batch, x_channel, y_height, y_width))

    for i in range(x_batch):
      for j in range(x_channel):
        for k in range(y_height):
          for l in range(y_width):
            start_k = k * self.stride
            start_l = l * self.stride
            end_k = start_k + self.kernel_height
            end_l = start_l + self.kernel_width
            y[i, j, k, l] = torch.max(x[i, j, start_k:end_k, start_l:end_l])

    return y
```

My AveragePool2D

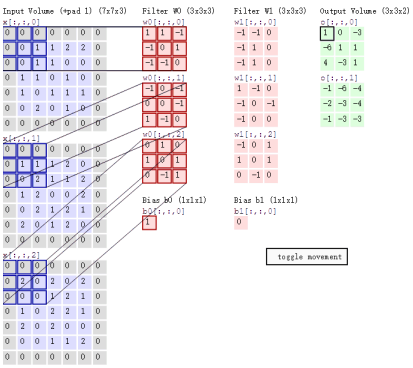
```
class MyAveragePool2D(nn.Module):
  def __init__(self, kernel_size, stride):
    super(MyAveragePool2D, self).__init__()
    self.stride = stride
    self.kernel_size = kernel_size
    self.kernel_height = kernel_size[0]
    self.kernel_width = kernel_size[1]

  def forward(self, x):
    x_batch = x.size(0)
    x_channel = x.size(1)
    x_height = x.size(2)
    x_width = x.size(3)

    y_height = int((x_height - self.kernel_height) / self.stride) + 1
    y_width = int((x_width - self.kernel_width) / self.stride) + 1
    y = torch.zeros((x_batch, x_channel, y_height, y_width))

    for i in range(x_batch):
      for j in range(x_channel):
        for k in range(y_height):
          for l in range(y_width):
            start_k = k * self.stride
            start_l = l * self.stride
            end_k = start_k + self.kernel_height
            end_l = start_l + self.kernel_width
            y[i, j, k, l] = x[i, j, start_k:end_k, start_l:end_l].mean()

    return y
```

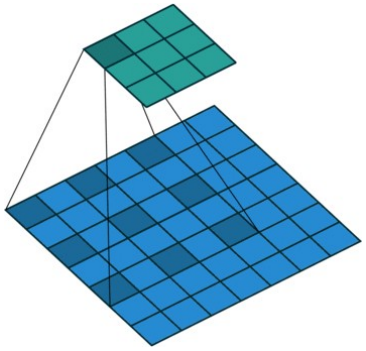
Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

Conv2d





My Conv2D

```
class Myconv2d(nn.Module):
  def __init__(self, in_channels, out_channels, kernel_size, stride, bias):
    super(Myconv2d, self).__init__()
    self.stride = stride
    self.kernel_size = kernel_size
    self.kernel_height = kernel_size[0]
    self.kernel_width = kernel_size[1]
    self.in_channels = in_channels
    self.out_channels = out_channels
    self.bias = bias

  def forward(self, x):
    x_batch = x.size(0)
    x_channel = x.size(1)
    x_height = x.size(2)
    x_width = x.size(3)

    y_height = int((x_height - self.kernel_height) / self.stride) + 1
    y_width = int((x_width - self.kernel_width) / self.stride) + 1
    y_channels = self.out_channels
    y = torch.zeros((x_batch, y_channels, y_height, y_width))

    for a in range(x_batch):
      for b in range(y_channels):
        for c in range(x_channel):
          for i in range(y_height):
            for j in range(y_width):
              start_i = i * self.stride
              start_j = j * self.stride
              end_i = start_i + self.kernel_height
              end_j = start_j + self.kernel_width
              y[a, b, i, j] += (x[a, c, start_i:end_i, start_j:end_j] * torch_kernel[b][c]).sum()

    for i in range(y_channels):
      y[0][i] += torch_bias[i]

    return y
```

My Conv2D

```
class Myconv2d(nn.Module):
  def __init__(self, in_channels, out_channels, kernel_size, stride, bias, dilation):
    super(Myconv2d, self).__init__()
    self.stride = stride
    self.kernel_size = kernel_size
    self.kernel_height = kernel_size[0]
    self.kernel_width = kernel_size[1]
    self.in_channels = in_channels
    self.out_channels = out_channels
    self.height = self.kernel_height * dilation - 1
    self.width = self.kernel_width * dilation - 1
    self.bias = bias
    self.dilation = dilation

  def forward(self, x):
    x_batch = x.size(0)
    x_channel = x.size(1)
    x_height = x.size(2)
    x_width = x.size(3)

    y_height = int((x_height - self.height) / self.stride) + 1
    y_width = int((x_width - self.width) / self.stride) + 1
    y_channels = self.out_channels
    y = torch.zeros((x_batch, y_channels, y_height, y_width))

    kernel = torch.zeros((y_channels, x_channel, self.height, self.width))

    for a in range(y_channels):
      for b in range(x_channel):
        for i in range(self.kernel_height):
          for j in range(self.kernel_width):
            kernel[a, b, i * self.dilation, j * self.dilation] = torch_kernel[a, b, i, j]

    for a in range(x_batch):
      for b in range(y_channels):
        for c in range(x_channel):
          for i in range(y_height):
            for j in range(y_width):
              start_i = i * self.stride
              start_j = j * self.stride
              end_i = start_i + self.height
              end_j = start_j + self.width
              y[a, b, i, j] += (x[a, c, start_i:end_i, start_j:end_j] * kernel[b][c]).sum()

    for i in range(y_channels):
      y[0][i] += torch_bias[i]

    return y
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

ConvTranspose2d

My ConvTranspose2D

```
class Myconvtrans2d(nn.Module):
  def __init__(self, in_channels, out_channels, kernel_size, stride, bias, dilation):
    super(Myconvtrans2d, self).__init__()
    self.stride = stride
    self.kernel_size = kernel_size
    self.kernel_height = kernel_size[0]
    self.kernel_width = kernel_size[1]
    self.in_channels = in_channels
    self.out_channels = out_channels
    self.bias = bias
    self.dilation = dilation

  def forward(self, x):
    x_batch = x.size(0)
    x_channel = x.size(1)
    x_height = x.size(2)
    x_width = x.size(3)

    y_height = (x_height + self.kernel_height) - 1
    y_width = (x_width + self.kernel_width) - 1
    y_channels = self.out_channels
    y = torch.zeros((x_batch, y_channels, y_height, y_width))

    for a in range(x_batch):
      for b in range(y_channels):
        for c in range(x_channel):
          for i in range(x_height):
            for j in range(x_width):
              y[a, b, i:i + self.kernel_height, j:j + self.kernel_width] += x[a, c, i, j] * torch_kernel[c][b]

    for i in range(y_channels):
      y[0][i] += torch_bias[i]

    return y
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

Flatten

My Flatten

```
def myflatten(X, start_dim, end_dim):
    shape = list(X.numpy().shape)
    if end_dim < 0:
        end_dim += len(shape)
    buffer = 1
    for i in range(start_dim, end_dim + 1):
        buffer *= shape[i]
    for j in range(end_dim, start_dim, -1):
        shape.pop(j)
    shape[start_dim] = buffer
    return X.reshape(shape)
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

Sigmoid

My Sigmoid

```
def mysigmoid(X):
    return 1 / (1 + torch.exp(-X))
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

ROI Pool

My Roi_pool

```
def my_roi_pool(X, boxes, output_size):

    roi_start_height = int(boxes[0][1])
    roi_start_width = int(boxes[0][2])
    roi_end_height = int(boxes[0][3])
    roi_end_width = int(boxes[0][4])

    roi_width = roi_end_width - roi_start_width + 1
    roi_height = roi_end_height - roi_start_height + 1

    bin_width = roi_width / output_size[0]
    bin_height = roi_height / output_size[1]

    result = []
    for i in range(X.shape[1]):
      tmp = []
      for j in range(output_size[0]):
        r = []
        for k in range(output_size[1]):
          temp = X[..., int(roi_start_height + bin_height * j):math.ceil(roi_start_height + bin_height * (j + 1)),
                   int(roi_start_width + bin_width * k):math.ceil(roi_start_width + bin_width * (k + 1))][0][i]
          r.append(temp.max())
        tmp.append(r)
      result.append(tmp)

    result = torch.Tensor(result)
    result = result.unsqueeze(0)
    return result
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

batch norm

My Batch_norm

```
def my_batch_norm(input, running_mean, running_var, momentum, eps):
    mean, var = running_mean[:, None, None], running_var[:, None, None]
    return (input - mean) / torch.sqrt(var + eps)
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

cross_entropy

My Cross_entropy

```
def softmax(X):
    result = torch.zeros(X.shape)
    for i in range(len(X)):
      e = torch.exp(X[i])
      s = torch.sum(e, dim = 0)
      soft = e / s
      result[i] = -torch.log(soft)
    return result

def loss(X, y):
    l = torch.zeros([len(X)])
    for i in range(len(X)):
      r = 0
      for j in range(y.shape[1]):
        for k in range(y.shape[2]):
          index = y[i, j, k]
          num = X[i, index, j ,k]
          r += num
        l[i] = r / y.shape[1] / y.shape[2]
    return l

def my_cross_entropy(X, y):
    log_softmax = softmax(X)
    l = loss(log_softmax, y)
    return l.mean()
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

mse_loss

My mse_loss

```
def my_mse_loss(X, y):
    sqr = (X - y) ** 2
    return sqr.mean()
```

Comparision

```
print(torch.all(torch.abs(my_out - torch_out) <= 1e-6))
```

```
tensor(True)
```

Q2A:

For this question, I first applied the standard pre-processing procedures to the CIFAR10 Dataset

```
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

num_classes = 10

# normalize the data
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255

# Convert class vectors to binary class matrices.
y_train = to_categorical(y_train, num_classes)
y_test = to_categorical(y_test, num_classes)

# partition training set into training and validation set
val_X = X_train[40000:,:]
train_X = X_train[:40000,:]
val_y = y_train[40000:,:]
train_y = y_train[:40000,:]
```

Which include normalization and covert data type. Rescale 1./255 is to transform every pixel value from range [0,255] -> [0,1]. And the benefits are: Treat all images in the same manner: some images are high pixel range, some are low pixel range. The images are all sharing the same model, weights and learning rate. The reason why 255 is because The RGB value is up to 255, and we want to standardize your colours between 0 and 1.

After that, I used Keras to build the CNN model that was specified.

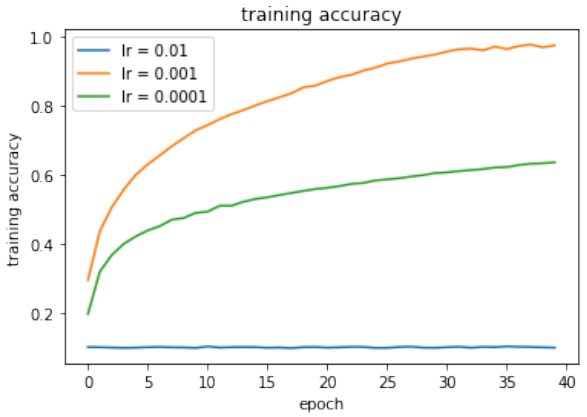
```
model1 = Sequential()
model1.add(Conv2D(16, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu", input_shape=train_X.shape[1:]))
model1.add(Conv2D(16, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model1.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(Conv2D(32, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model1.add(Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(Conv2D(64, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model1.add(Conv2D(128, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(Conv2D(128, kernel_size=(3, 3), strides=(1, 1), padding="same", activation="relu"))
model1.add(AveragePooling2D(strides=(4, 4), padding = "same"))
model1.add(Flatten())
model1.add(Dense(10, activation = "softmax"))
```

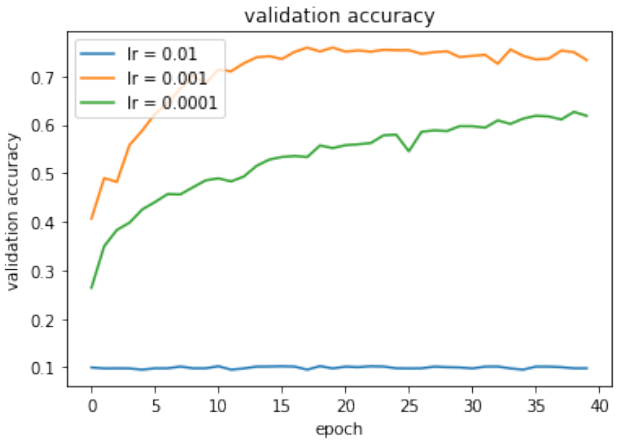
The only thing that needs to be specified is that since Keras doesn't have a build-in AdaptiveAveragePooling2D layer, I used the AveragePooling2D layer and calculated the parameters manually instead, by following the formula:

```
Stride = (input_size//output_size)
Kernel size = input_size - (output_size-1)*stride
Padding = 0
```

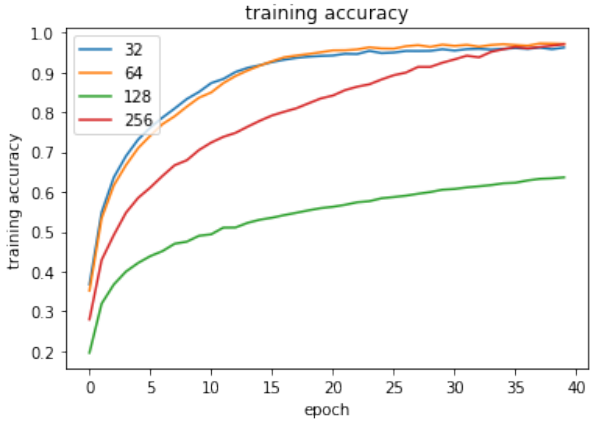
After that, I used GridSearch for parameter tuning to bump the model's accuracy to 80%+. Here are the result of different parameters. The parameter I considered includes the Learning Rate of the Adam Optimizer, Batch Size and Epochs.

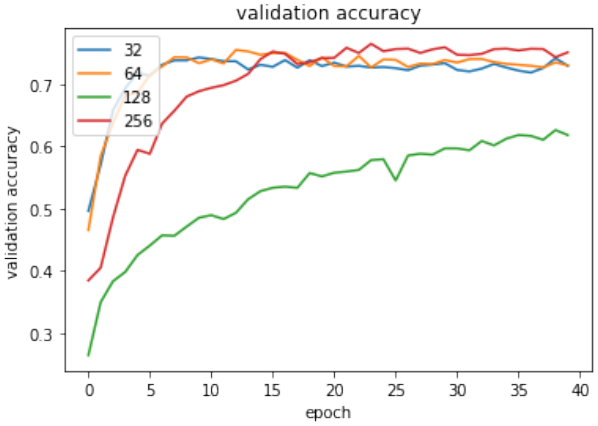
Learning Rate:



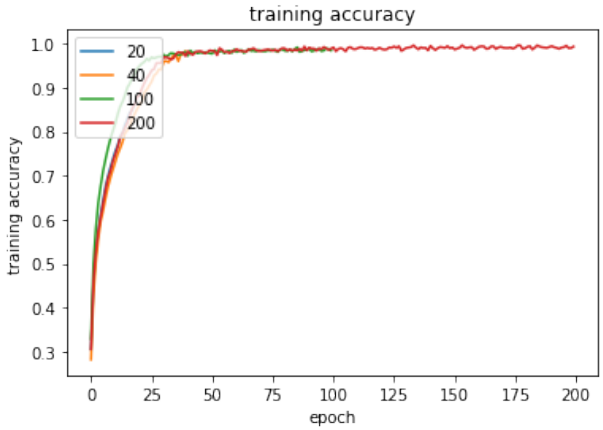


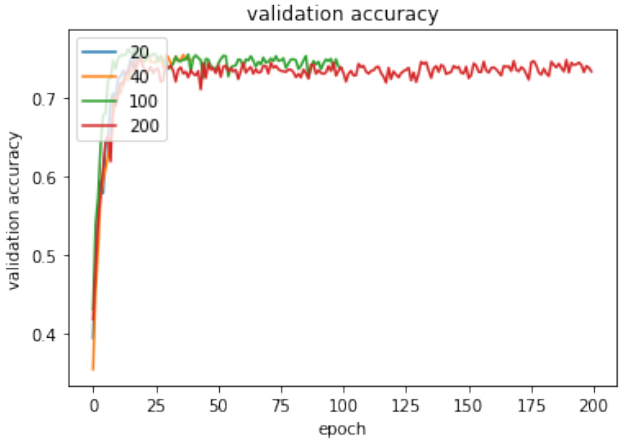
Batch Size:





Epochs:

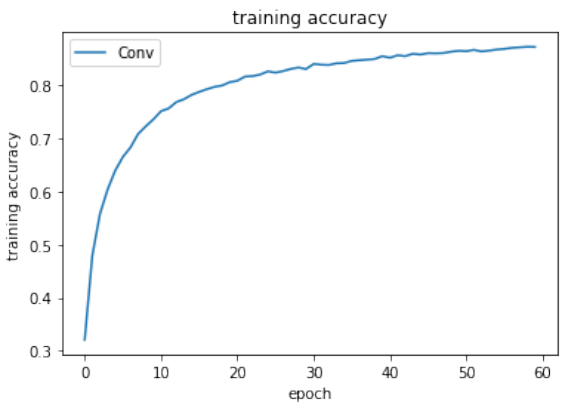


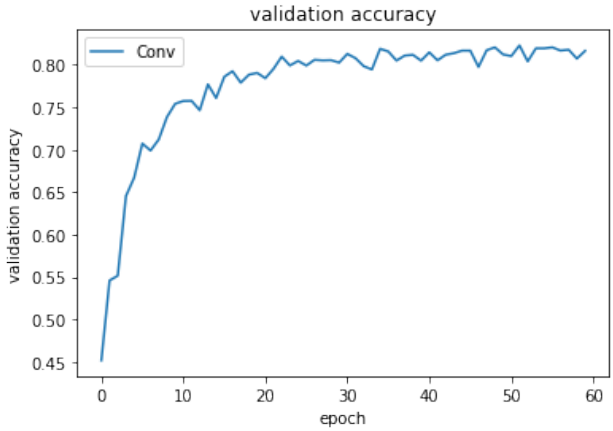


After analyzing all the results, I decided to go for learning rate = 0.01, batch size = 256 and epochs = 200. However, the maximum accuracy I can get is only 78%. Thus I added an image pre-processing step called data-augmentation to the CIFAR10 images. Image augmentation is a technique of altering the existing data to create some more data for the model training process. In other words, it is the process of artificially expanding the available dataset for training a deep learning model. In this picture, the image on the left is only the original image, and the rest of the images are gen erated from the original training image.

Since all these images are generated from training data, we don't have to collect them manually. Note that, the label for all the images will be the same and that is of the original image which is used to generate them. This increases the training sample without going out and collecting this data. I chose to augmentations include the horizontal and vertical shift of images and horizontal flip of images.

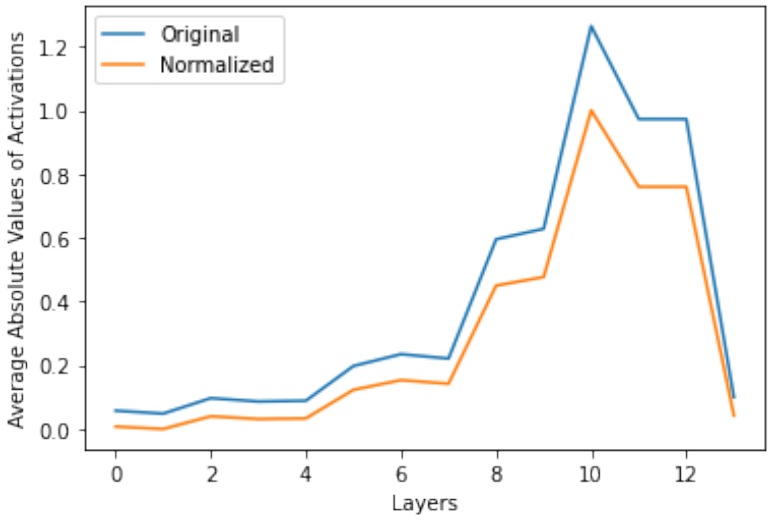
After applying this procedure, I was able to boost the model's accuracy to 80%+ and rising.





After I trained the model, I computed the average absolute value of each layers' activations/hidden layer values across all training sets and channels.

	Layers	Average Absolute Values of Weights	Average Absolute Values of Activations	Normalized Average Absolute Values of Activations
0	conv2d	9.737999	0.057458	0.007587
1	conv2d_1	41.646465	0.048232	0.000000
2	max_pooling2d	0.000000	0.096697	0.039854
3	conv2d_2	78.374420	0.086272	0.031281
4	conv2d_3	268.389618	0.088633	0.033223
5	max_pooling2d_1	0.000000	0.197698	0.122908
6	conv2d_4	484.296783	0.234770	0.153392
7	conv2d_5	1298.837646	0.221029	0.142093
8	max_pooling2d_2	0.000000	0.595410	0.449952
9	conv2d_6	2034.145996	0.628204	0.476919
10	conv2d_7	2736.633789	1.264313	1.000000
11	average_pooling2d	0.000000	0.972531	0.760064
12	flatten	0.000000	0.972531	0.760064
13	dense	49.231678	0.100000	0.042570



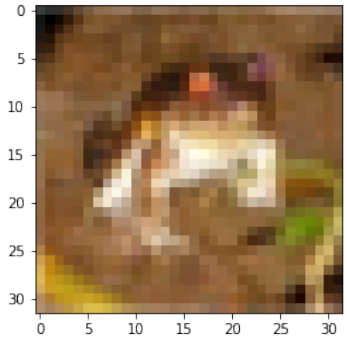
From the above results, we can see that the sum weights of each layer are increasing. This reason for this phenomenon is because Keras uses Xavier initialization sets a layer's weights to values chosen from a random uniform distribution that's bounded between

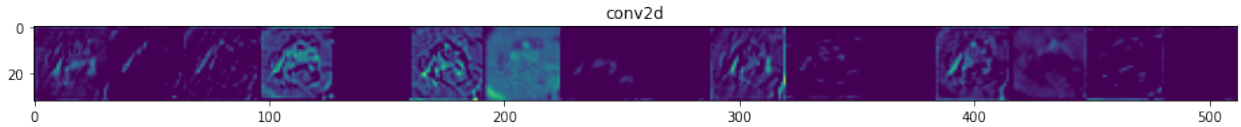
$$\pm \frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}}$$

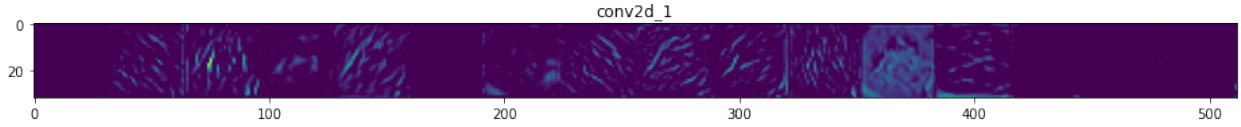
Thus, it is not guaranteed that weight will stay within the range [0, 1], resulting in the hidden values of each layer increasing until the flatten layer. This makes sense since

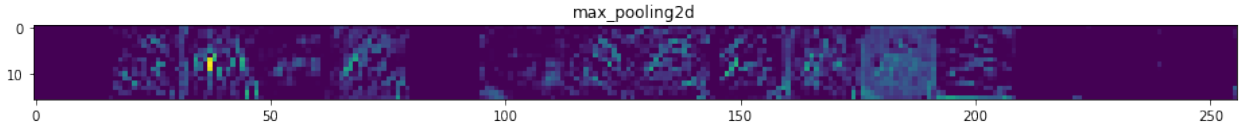
the purpose of stride and kernel is to make images brighter, and the larger the pixels, the brighter the images.

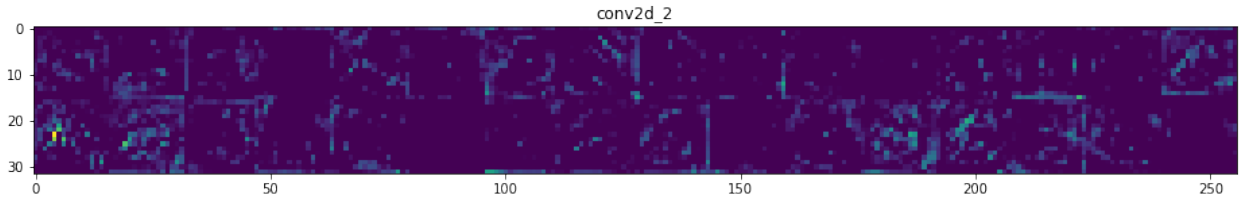
Lastly, I used an example image to illustrate the change of each layer.

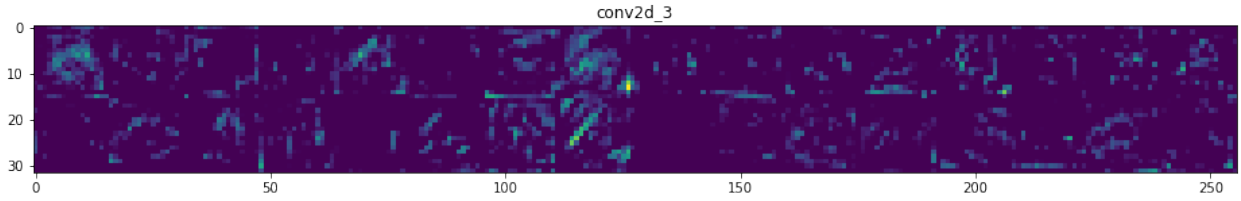


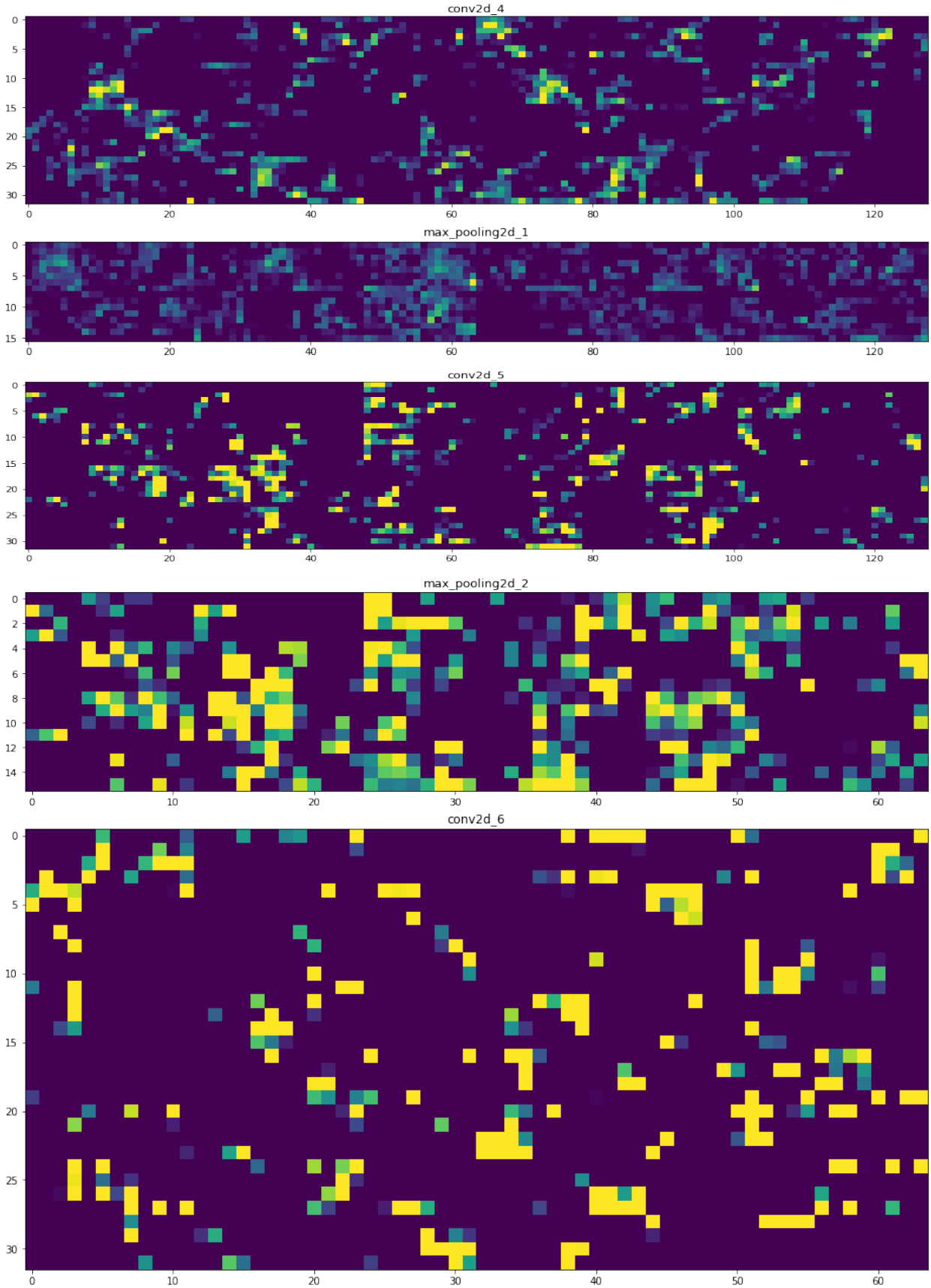


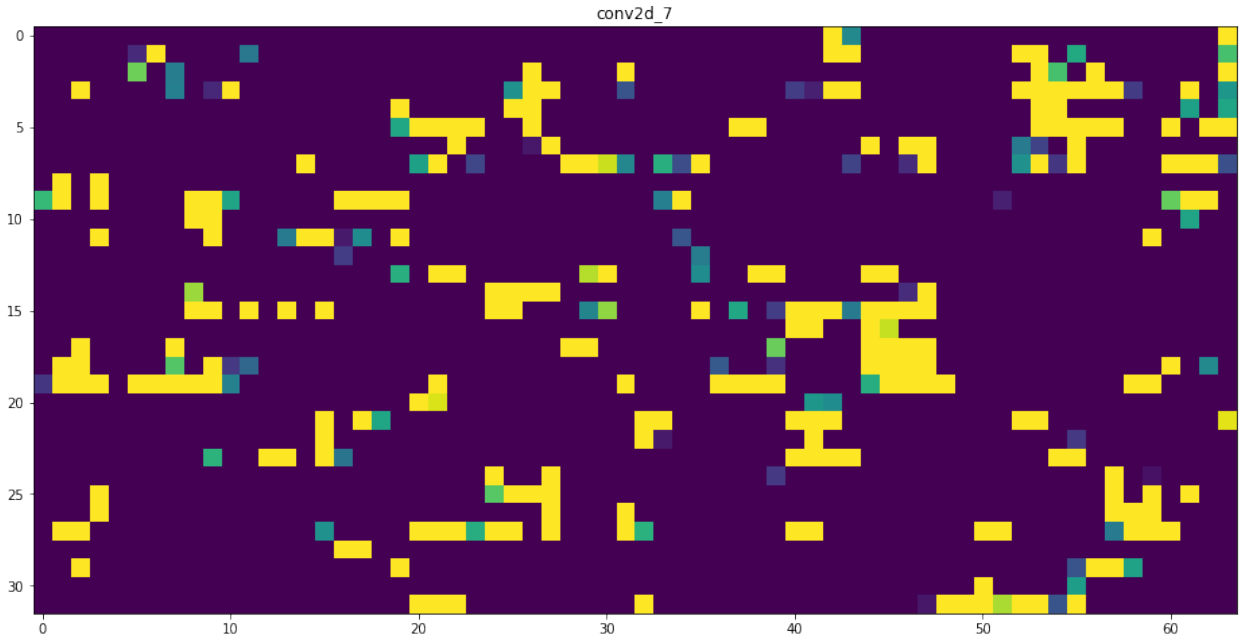


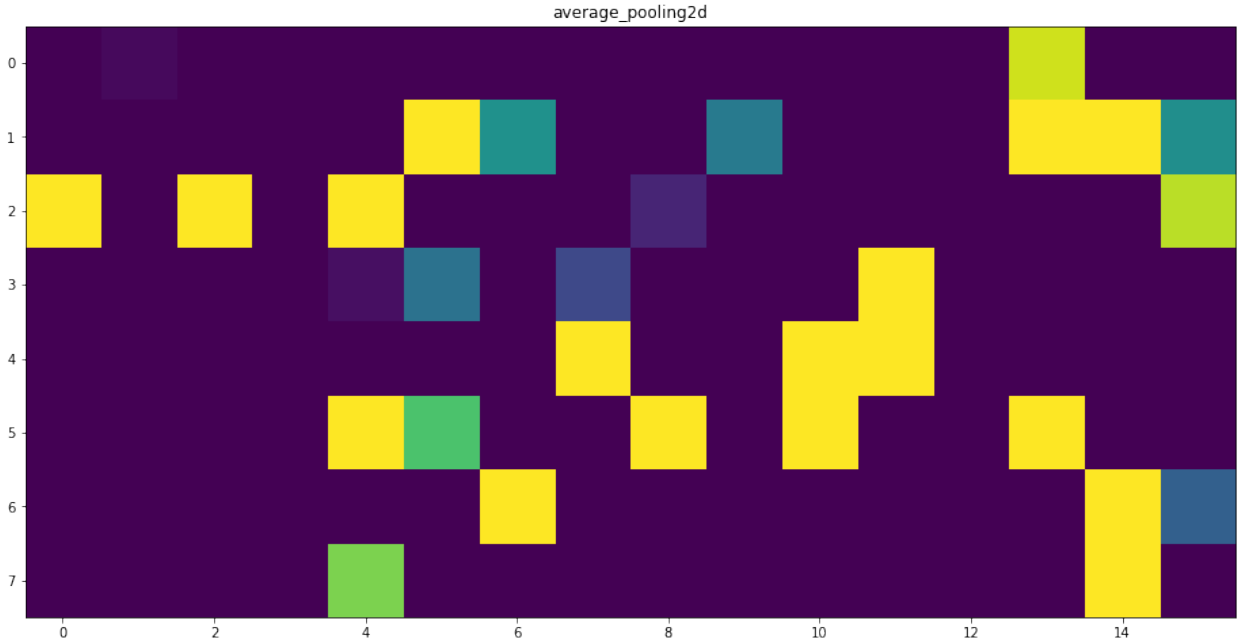







conv2d_4
max_pooling2d_1
conv2d_5
max_pooling2d_2
conv2d_6



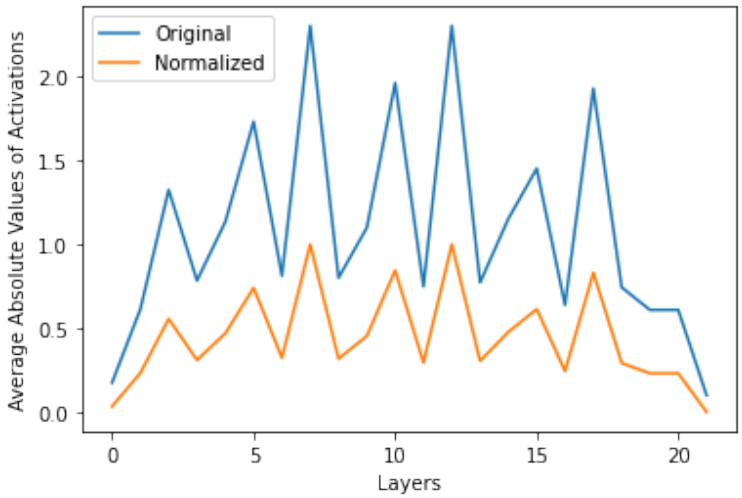


From the above images, we can see that The first layer arguably retains the full shape of the image, although several filters are not activated and are left blank. At that stage, the activations retain almost all of the information present in the initial picture. As we go deeper in the layers, the activations become increasingly abstract and less visually interpretable. They encode higher-level concepts such as single borders, corners and angles. Higher presentations carry less information about the visual contents of the image increasingly, and increasingly more information related to the class of the image.

Q2B:

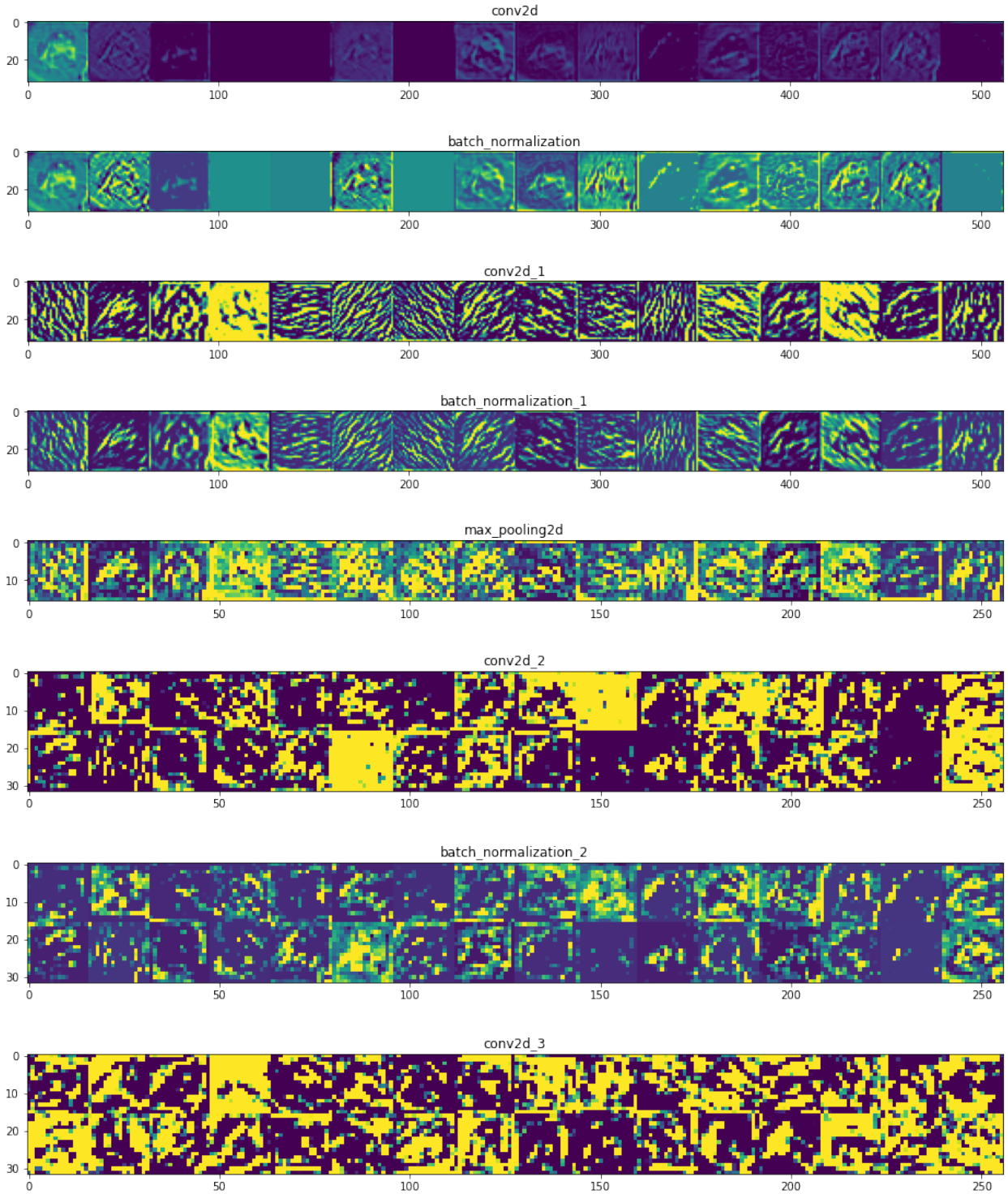
The training procedure of this question is quite similar to the previous question; the only major difference is the activation value of each layer.

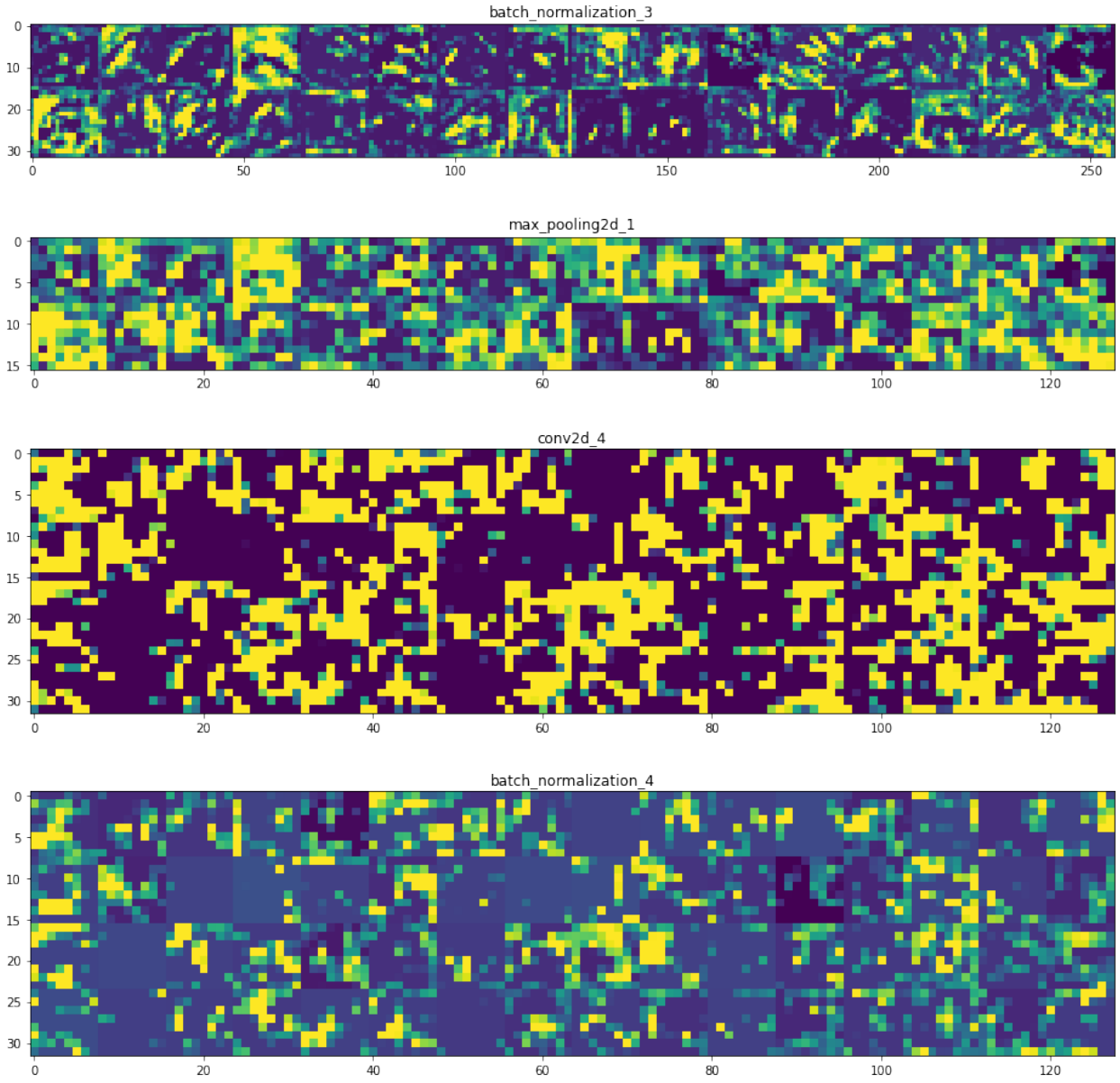
	Layers	Average Absolute Values of Activations	Normalized Average Absolute Values of Activations
0	conv2d	0.174759	0.033917
1	batch_normalization	0.613248	0.232854
2	conv2d_1	1.325504	0.555996
3	batch_normalization_1	0.784201	0.310413
4	max_pooling2d	1.133998	0.469112
5	conv2d_2	1.732257	0.740534
6	batch_normalization_2	0.812626	0.323309
7	conv2d_3	2.303298	0.999608
8	batch_normalization_3	0.800978	0.318025
9	max_pooling2d_1	1.099721	0.453561
10	conv2d_4	1.962624	0.845049
11	batch_normalization_4	0.749586	0.294709
12	conv2d_5	2.304161	1.000000
13	batch_normalization_5	0.772944	0.305306
14	max_pooling2d_2	1.153551	0.477982
15	conv2d_6	1.452601	0.613658
16	batch_normalization_6	0.638799	0.244446
17	conv2d_7	1.930164	0.830322
18	batch_normalization_7	0.744122	0.292230
19	average_pooling2d	0.608873	0.230869
20	flatten	0.608873	0.230869
21	dense	0.100000	0.000000



Batch normalization solves a major problem called internal covariate shift. It helps by making the data flowing between intermediate layers of the neural network look, this means you can use a higher learning rate. After adding the batch normalization, the value of each hidden layer begins to float between 0.6 and 0.8.

The printout images are the following:





Github: https://github.com/Ironarrow98/Zhang_Chenxi_BS6207_A3