Pretrain Model ای که در این پروژه استفاده شده ResNet50 است. Resnet50 یکی از انواع مدل‌های ResNet است که از ۵۰ تا لایه شامل ۴۸ لایه کانولوشنالی و یک لایه MaxPool و یک لایه Average Pool تشکیل شده است.

برای انجام این پروژه در ابتدا بعد از import کردن کتابخانه‌های لازم، مدل Pretrain شده ResNet50 را مطابق کد زیر بارگذاری میکنیم.

Load ResNet-50 Model

In [ ]:
```
# load the model
model = models.resnet50(pretrained=True)
print(model)
model_weights = [] # we will save the conv layer weights in this list
conv_layers = [] # we will save the 49 conv layers in this list
# get all the model children as list
model_children = list(model.children())
```

در اینجا دو تا لیست model_weights (در جهت ذخیره وزن‌های لایه‌های کانولوشنالی) و conv_layers (در جهت ذخیره لایه‌های کانولوشنالی) تعریف کرده‌ایم. با استفاده از model.children() توانستیم تمام لایه‌ها را مشاهده کنیم.

خروجی به دست آمده از کد بالا:

```
  0%|          | 0.00/97.8M [00:00<?, ?B/s]
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

در بخش بعدی کد تمام لایه ها و وزن‌های مربوط به لایه‌ها را در لیست‌های تعریف شده ذخیره میکنیم.

retrieve all the convolutional layers and their weights

```
# counter to keep count of the conv layers
counter = 0
# append all the conv layers and their respective weights to the list
for i in range(len(model_children)):
    if type(model_children[i]) == nn.Conv2d:
        counter += 1
        model_weights.append(model_children[i].weight)
        conv_layers.append(model_children[i])
    elif type(model_children[i]) == nn.Sequential:
        for j in range(len(model_children[i])):
            for child in model_children[i][j].children():
                if type(child) == nn.Conv2d:
                    counter += 1
                    model_weights.append(child.weight)
                    conv_layers.append(child)
```

در خروجی زیر می‌توان اطلاعات مربوط به لایه ها را مشاهده کرد.

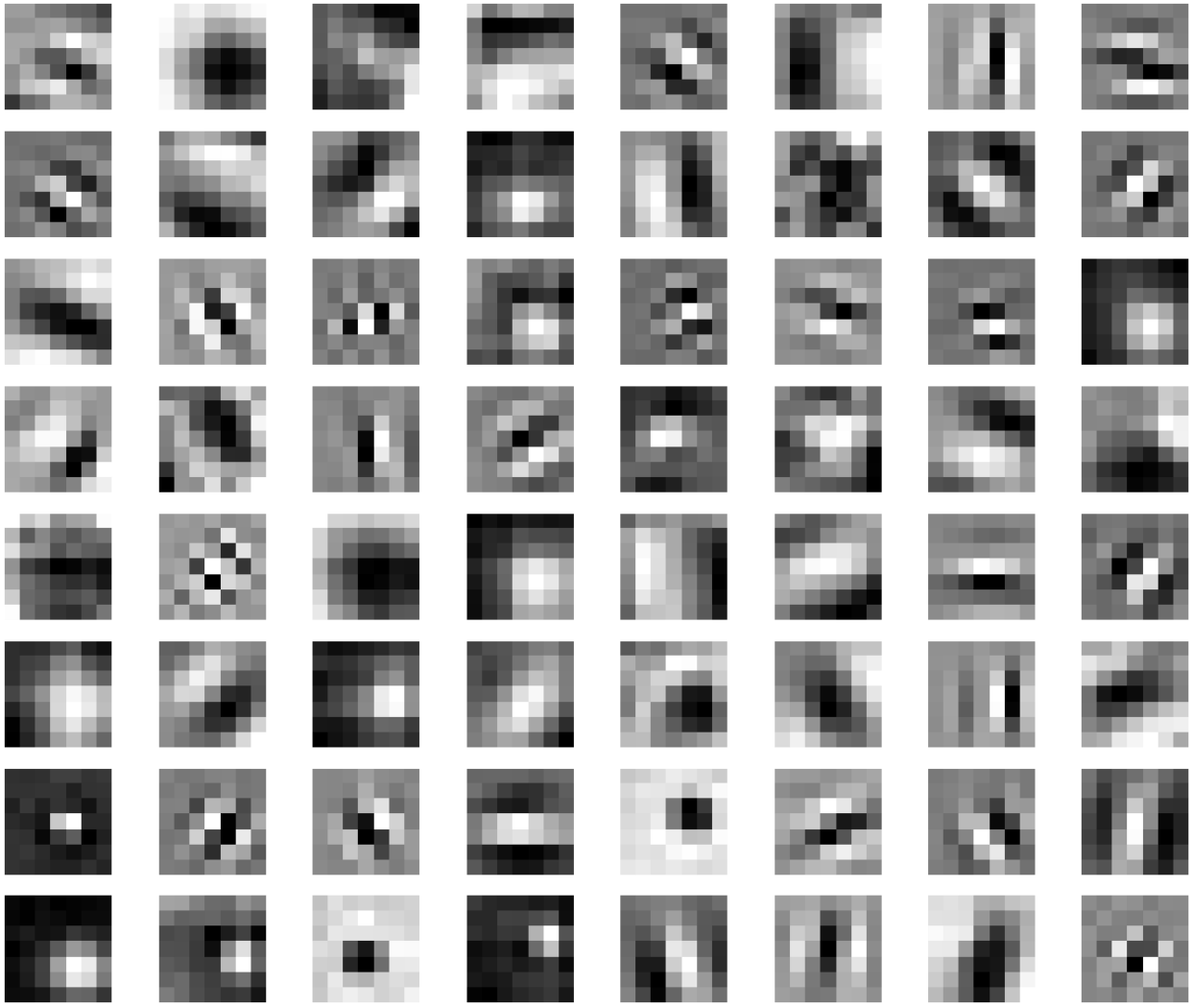
```
CONV: Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False) ====> SHAPE: torch.Size([64, 3, 7, 7])
CONV: Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([64, 64, 1, 1])
CONV: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([64, 64, 3, 3])
CONV: Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 64, 1, 1])
CONV: Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([64, 256, 1, 1])
CONV: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([64, 64, 3, 3])
CONV: Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 64, 1, 1])
CONV: Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([64, 256, 1, 1])
CONV: Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([64, 64, 3, 3])
CONV: Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 64, 1, 1])
CONV: Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 256, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 128, 3, 3])
CONV: Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 128, 1, 1])
CONV: Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 512, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 128, 3, 3])
CONV: Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 128, 1, 1])
CONV: Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 512, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 128, 3, 3])
CONV: Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 128, 1, 1])
CONV: Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 512, 1, 1])
CONV: Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([128, 128, 3, 3])
CONV: Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([512, 128, 1, 1])
CONV: Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 512, 1, 1])
CONV: Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False) ====> SHAPE: torch.Size([256, 256, 3, 3])
```

با استفاده از کد زیر، فیلتر یا کرنل ها را میتوانیم نمایش بدهیم. با استفاده از فیلتر یا کرنل میتوانیم ویژگی‌های خاصی را از تصویر ورودی استخراج کنیم.

Visualizing Convolutional Layer Filters

```
# visualize the first conv layer filters
plt.figure(figsize=(20, 17))
for i, filter in enumerate(model_weights[0]):
    plt.subplot(8, 8, i+1) # (8, 8) because in conv0 we have 7x7 filters and total of 64 (see printed shapes)
    plt.imshow(filter[0, :, :].detach(), cmap='gray')
    plt.axis('off')
    plt.savefig('/content/CNN/outputs/filter.png')
plt.show()
```

برای نمونه فقط فیلترهای لایه Conv0 را نشان داده‌ایم. سایز فیلترهای این لایه ۷در۷ میباشد و تعداد آنها ۶۴ تا است.



برای خواندن عکس ورودی باید یکسری تغییرات روی عکس صورت بگبرد.

Reading the Image and Defining the Transforms

```
# read and visualize an image
from google.colab.patches import cv2_imshow
img = cv.imread(f"/content/CNN/input/dog.jpg")

img = cv.cvtColor(img, cv.COLOR_BGR2RGB)
plt.imshow(img)
plt.show()
# define the transforms
transform = transforms.Compose([
    transforms.ToPILImage(),
    transforms.Resize((512, 512)),
    transforms.ToTensor(),
])
img = np.array(img)
# apply the transforms
img = transform(img)
print(img.size())
# unsqueeze to add a batch dimension
img = img.unsqueeze(0)
print(img.size())
```

عکس ورودی:

با استفاده از کد زیر عکس وارد شبکه میشود.

Passing the Input Image Through Each Convolutional Layer

```
# pass the image through all the layers
results = [conv_layers[0](img)]
for i in range(1, len(conv_layers)):
    # pass the result from the last layer to the next layer
    results.append(conv_layers[i](results[-1]))
# make a copy of the `results`
outputs = results
print(outputs)
```

بعد از اعمال فیلتر بر روی ورودی در هر لایه، Feature map به دست می‌آید. هر فیلتر یک ویژگی خاصی از تصویر را استخراج میکند که نتایج خروجی را در Feature map میتوان مشاهده کرد. در بعضی از Featuer mapها پس زمینه عکس و در بعضی خط های عکس و بسیاری از ویژگی‌های دیگر استخراج شده است.

با استفاده از کد زیر Feature map تمام لایه ها به دست می‌آید.

Visualizing the Feature Maps

```
# visualize 64 features from each layer
# (although there are more feature maps in the upper layers)
for num_layer in range(len(outputs)):
    plt.figure(figsize=(30, 30))
    layer_viz = outputs[num_layer][0, :, :, :]
    layer_viz = layer_viz.data
    print(layer_viz.size())
    for i, filter in enumerate(layer_viz):
        if i == 64: # we will visualize only 8x8 blocks from each layer
            break
        plt.subplot(8, 8, i + 1)
        plt.imshow(filter, cmap='gray')
        plt.axis("off")
    print(f"Saving layer {num_layer} feature maps...")
    plt.savefig(f"/content/CNN/outputs/layer_{num_layer}.png")
    # plt.show()
    plt.close()
```

در تصویر زیر خروجی کد بالا را مشاهده میکنید که تعداد و سایز Feature map های خروجی مربوط به لایه ها را نمایش میدهد.

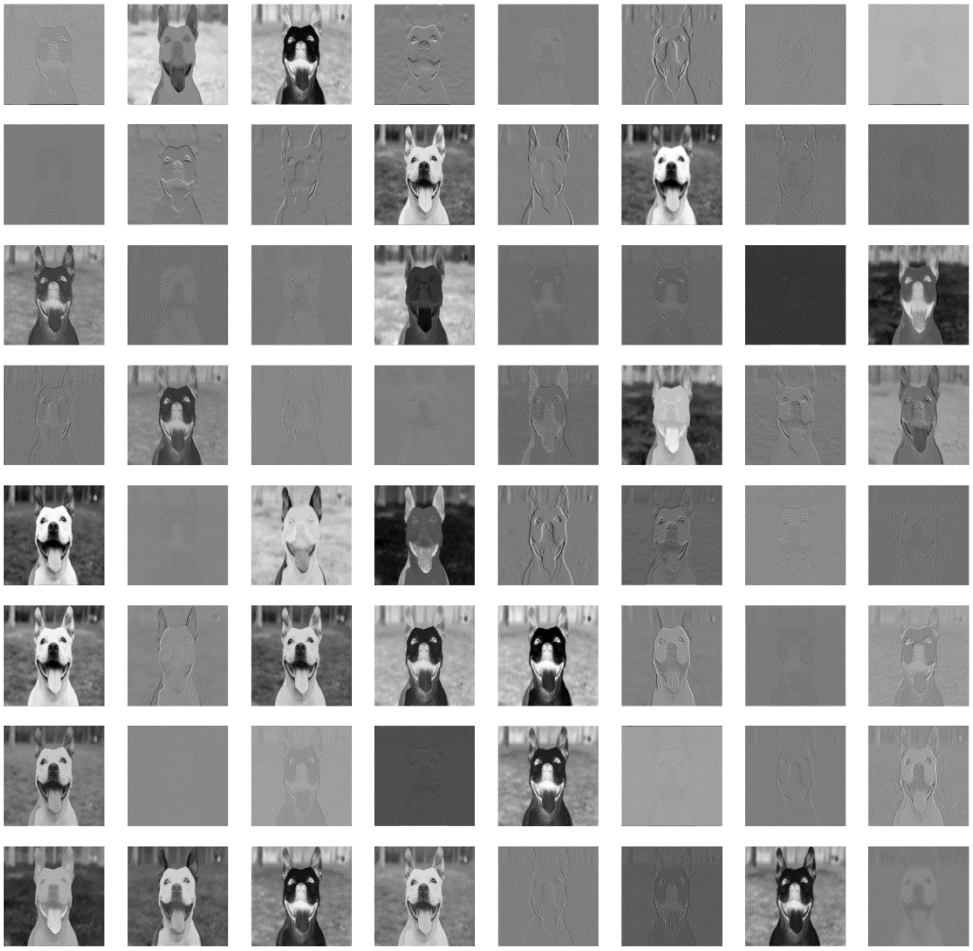
برای نمونه در لایه اول ۶۴ تا Feature map داریم. هرچه لایه‌ها عمیق‌تر شود تعداد Feature mapها زیاد خواهد شد.

```
torch.Size([64, 256, 256])
Saving layer 0 feature maps..
torch.Size([64, 256, 256])
Saving layer 1 feature maps..
torch.Size([64, 256, 256])
Saving layer 2 feature maps..
torch.Size([256, 256, 256])
Saving layer 3 feature maps..
torch.Size([64, 256, 256])
Saving layer 4 feature maps..
torch.Size([64, 256, 256])
Saving layer 5 feature maps..
torch.Size([256, 256, 256])
Saving layer 6 feature maps..
torch.Size([64, 256, 256])
Saving layer 7 feature maps..
torch.Size([64, 256, 256])
Saving layer 8 feature maps..
torch.Size([256, 256, 256])
Saving layer 9 feature maps..
torch.Size([128, 256, 256])
Saving layer 10 feature maps.
torch.Size([128, 128, 128])
Saving layer 11 feature maps.
torch.Size([512, 128, 128])
Saving layer 12 feature maps.
torch.Size([128, 128, 128])
Saving layer 13 feature maps.
torch.Size([128, 128, 128])
Saving layer 14 feature maps.
```

برای جلوگیری از شلوغی گزارش مربوط به پروژه فقط تعداد محدودی از تصاویر مربوط به Feature map ها در گزارش درج شده است.

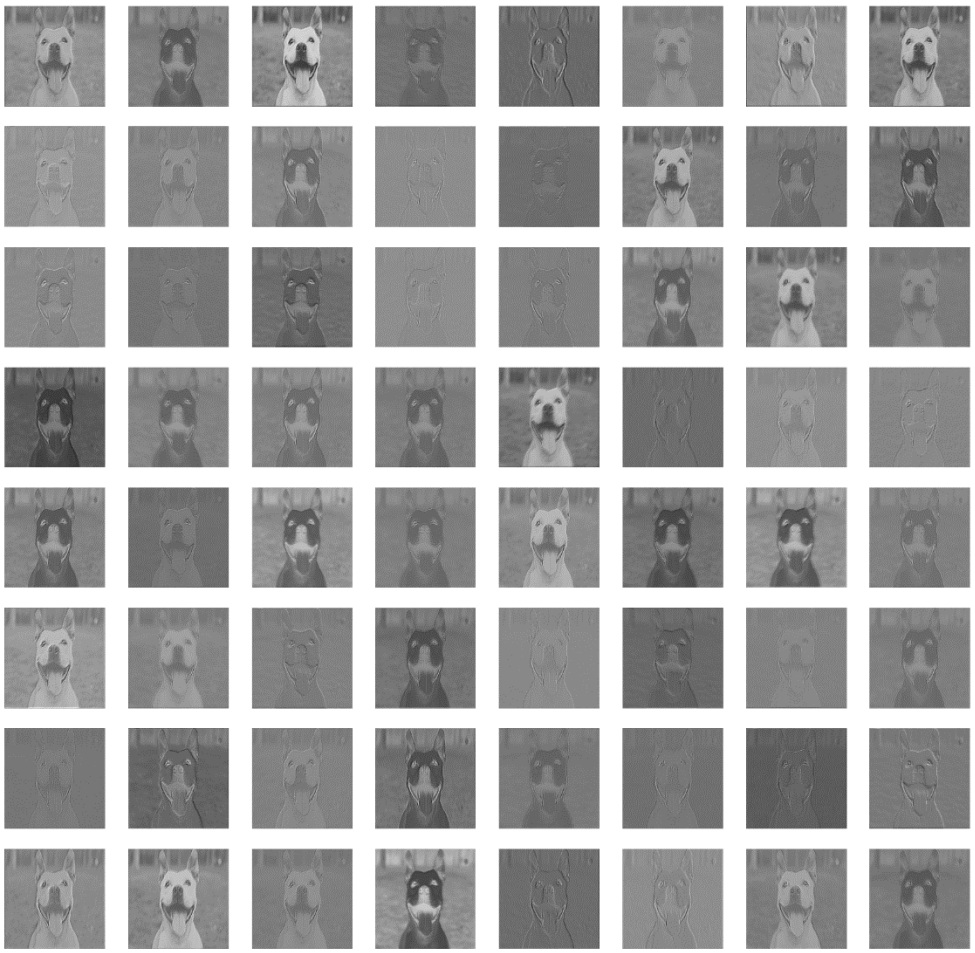
در Feature map های لایه‌های اولیه، رنگ و خطوط ساده مربوط به عکس ورودی استخراج شده است. در Feature mapهای لایه های میانی Texture و Patternهایی (که از ترکیب Feature های لایه‌های ابتدایی به دست آمده) استخراج شده است. وFeature map های لایه‌های آخر در برگیرنده قسمتی از Object ها (که از ترکیب Feature های لایه‌های قبلی به دست آمده است)میباشند.

Feature map خروجی مربوط به لایه اول کانولوشنال:

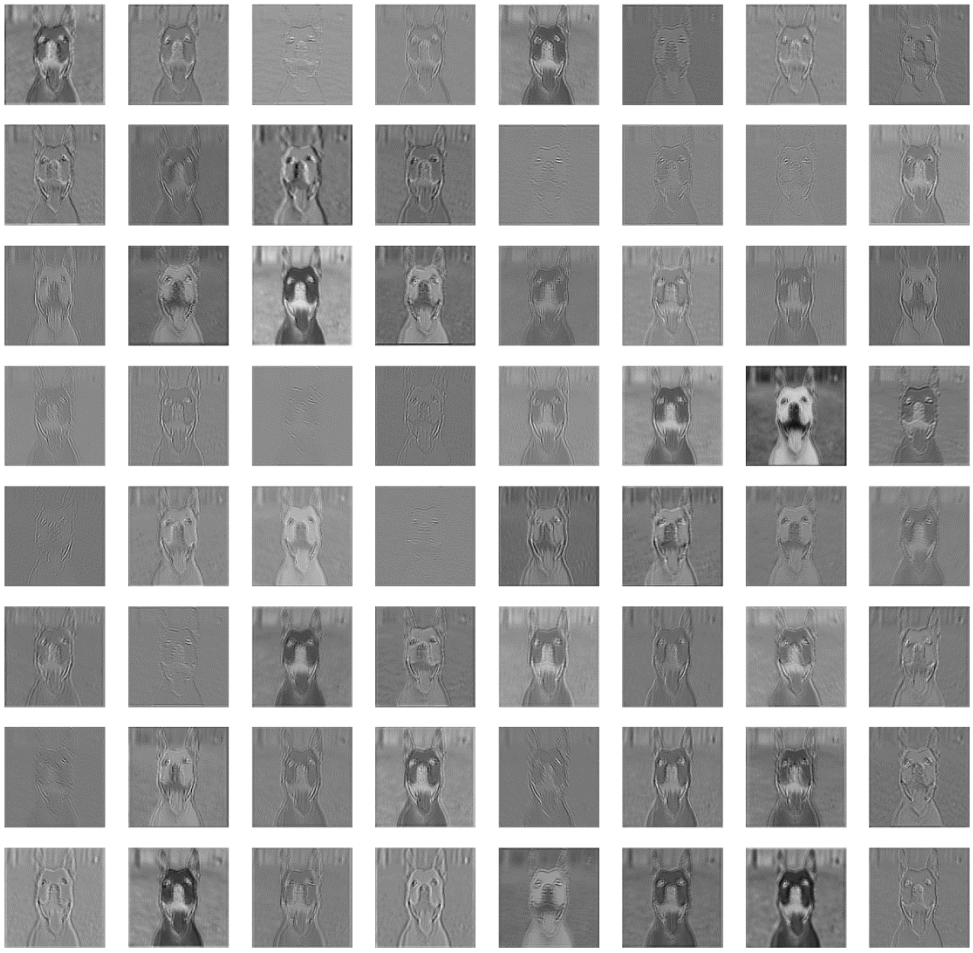


خطوط و رنگ‌های روشن نشان‌دهنده‌ی این است که فیلتر آن‌ها را تشخیص داده است.

Feature map خروجی مربوط به لایه دهم کانولوشنال:

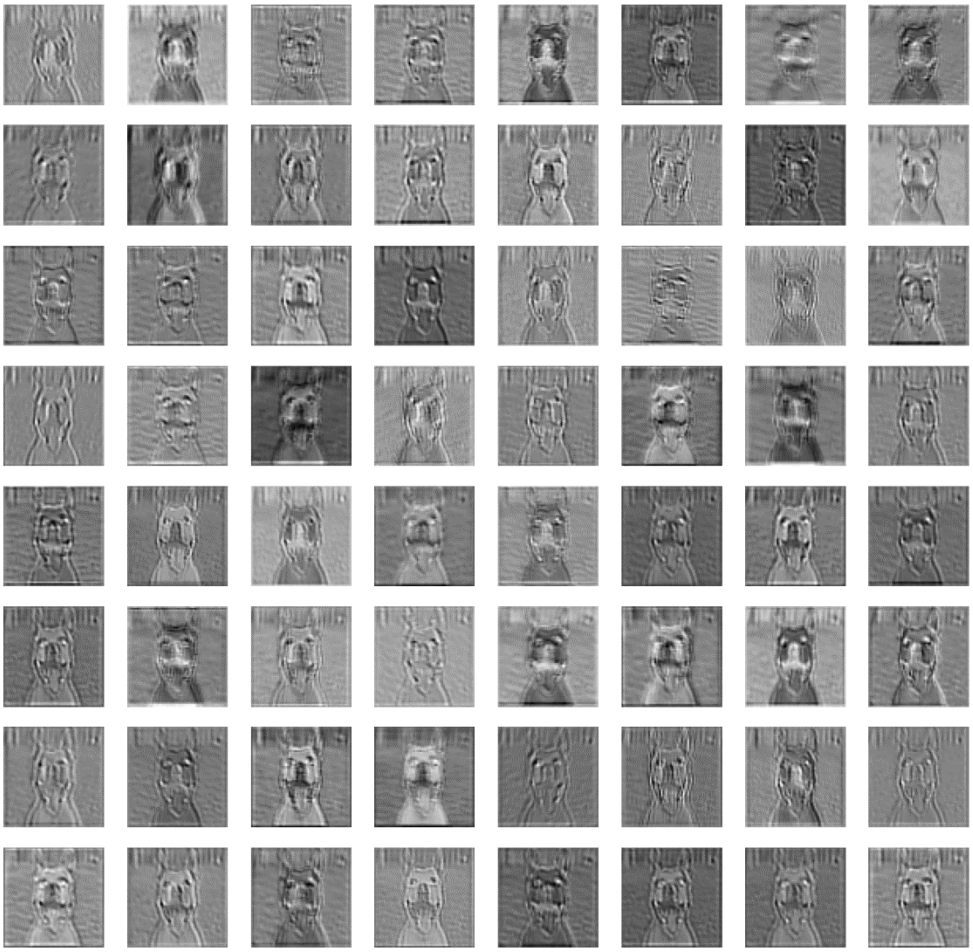


خروجی مربوط به لایه بیستم کانولوشنال:

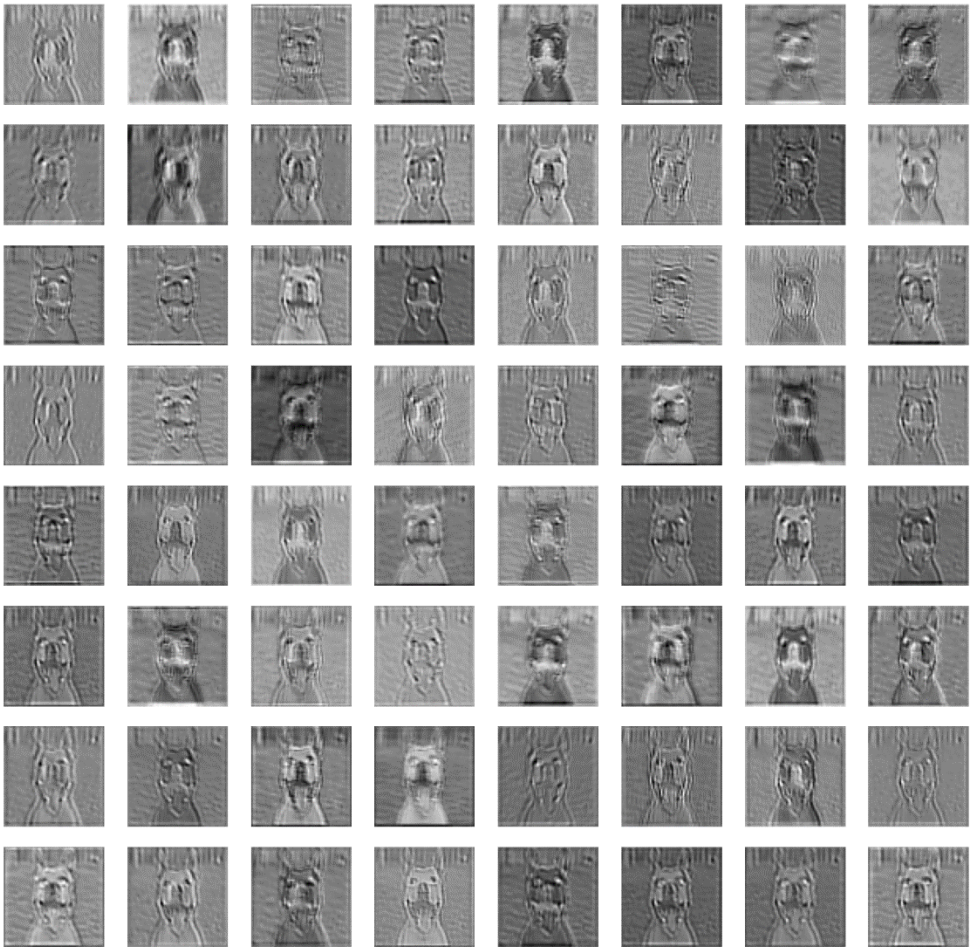


در اینجا با ترکیب ویژگی‌های مراحل قبل الگوهایی استخراج شده است.

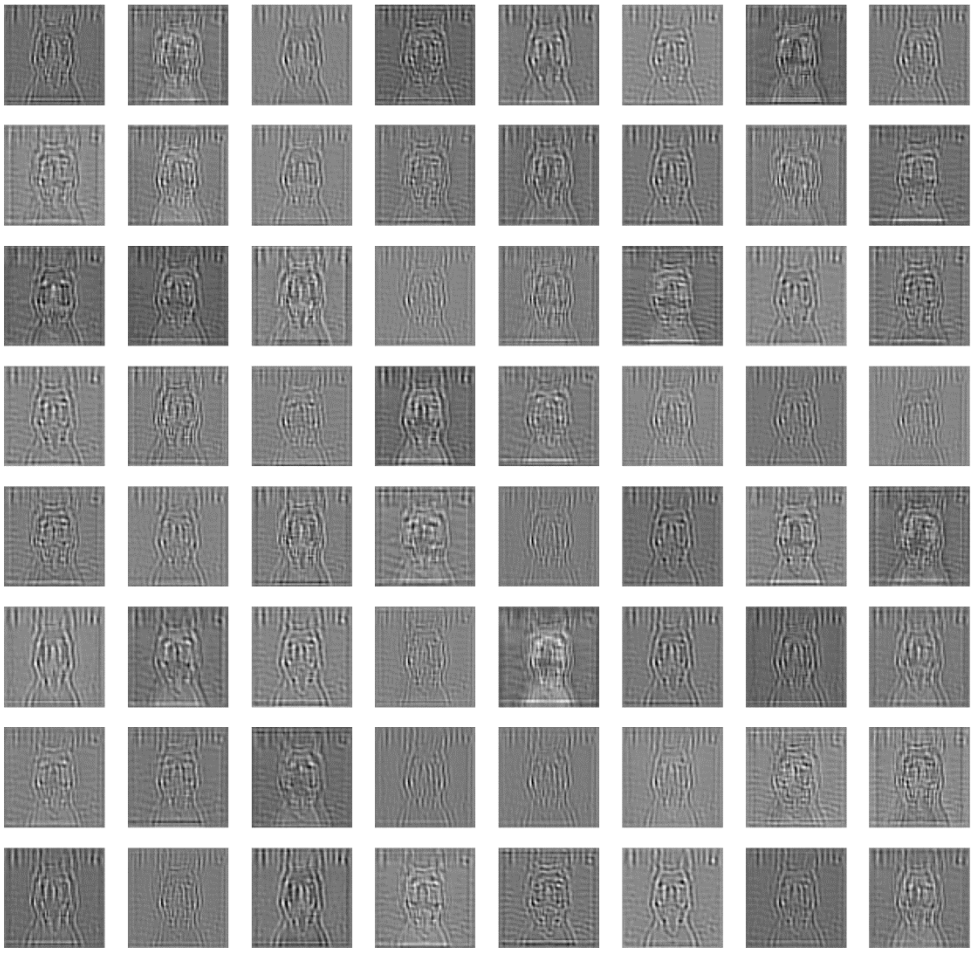
خروجی مربوط به لایه سی‌ام کانولوشنال:



خروجی مربوط به لایه چهلم کانولوشنال:



خروجی مربوط به لایه چهل و هشتم کانولوشنال:



همانطور که در تصاویر feature map مشاهده میکنید با عمیق‌تر شدن لایه‌ها به آرامی جزییات عکس‌ها ناپدید شده‌اند بطوریکه به نظر می‌رسد در Feature map مربوط به لایه‌های آخر noise وجود دارد اما الگوهایی که در این Feature mapها وجود دارد برای شبکه عصبی قابل تشخیص است.