MONASH University

FIT3181/5215 Deep Learning

Advanced Convolutional Neural Networks

Trung Le and Tutor team

Department of Data Science and AI
Faculty of Information Technology, Monash University
Email: **trunglm@monash.edu**

GROUP OF EIGHT AUSTRALIA

Summary

- Tutorial 6a: Implementation of ResNet (*****)
- Tutorial 6b: Adversarial Machine Learning (*****)
  - Questions 3.6 and 3.7 in Assignment 1.
- Tutorial 6c:
  - Visualization of Filters and Feature Maps of CNNs (***)
  - Gradcam and GuidedGradcam

ResNet

Residual block

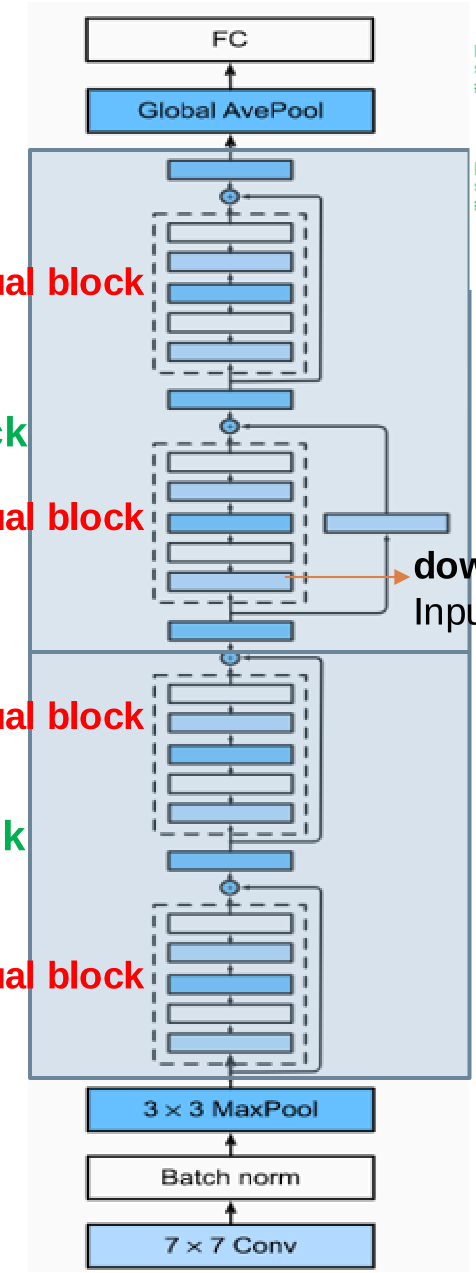
ResNet block

Residual block

Residual block

ResNet block

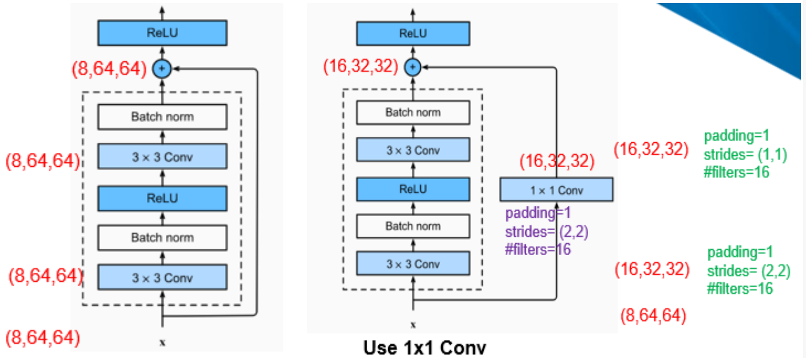
Residual block



padding=1
strides= (1,1)
#filters=8

padding=1
strides= (1,1)
#filters=8

down sample
Input size by **2**



Not use 1x1 Conv

Use 1x1 Conv

```
class Residual(nn.Module):
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, stride=strides, padding=1)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1)
        self.conv3 = None
        if use_1x1conv:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=strides)
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)
        self.relu = nn.ReLU()

    def forward(self, X):
        Y = self.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3 is not None:
            X = self.conv3(X)
        Y += X
        return self.relu(Y)
```

Residual block

```
class ResnetBlock(nn.Module):
    def __init__(self, num_channels, num_residuals, first_block=False, **kwargs):
        super(ResnetBlock, self).__init__(**kwargs)
        self.residual_layers = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                self.residual_layers.append(
                    Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                self.residual_layers.append(Residual(num_channels))
        self.residual_blk = nn.ModuleList(self.residual_layers)

    def forward(self, X):
        for layer in self.residual_blk:
            X = layer(X)
        return X
```

ResNet block

```
class create_ResNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.layers = nn.ModuleList([
                  nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
                  nn.LazyBatchNorm2d(),
                  nn.ReLU(),
                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
                  ResnetBlock(64, 2, first_block=True),
                  ResnetBlock(128, 2),
                  ResnetBlock(256, 2),
                  nn.AdaptiveAvgPool2d((1, 1)),
                  nn.Flatten(1),
                  nn.LazyLinear(10),
                  # nn.Softmax(dim=-1)
                  ])
  def forward(self, X):
    for _, layer in enumerate(self.layers):
      X = layer(X)
    return X
```

```
X = torch.rand(size=(1, 1, 28, 28))
for _, layer in enumerate(create_ResNet().layers):
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', list(X.shape))
```

```
Conv2d output shape:     [1, 64, 14, 14]
BatchNorm2d output shape:        [1, 64, 14, 14]
ReLU output shape:       [1, 64, 14, 14]
MaxPool2d output shape:  [1, 64, 7, 7]
ResnetBlock output shape:        [1, 64, 7, 7]
ResnetBlock output shape:        [1, 128, 4, 4]
ResnetBlock output shape:        [1, 256, 2, 2]
AdaptiveAvgPool2d output shape:  [1, 256, 1, 1]
Flatten output shape:    [1, 256]
Linear output shape:     [1, 10]
```

Our ResNet

```python
def pgd_attack(model, input_image, input_label=None,
               epsilon=0.3,
               num_steps=20,
               step_size=0.01,
               clip_value_min=0.,
               clip_value_max=1.0):

    if type(input_image) is np.ndarray:
        input_image = torch.tensor(input_image, requires_grad=True)

    if type(input_label) is np.ndarray:
        input_label = torch.tensor(input_label)

    # Ensure the model is in evaluation mode
    model.eval()

    # Create a copy of the input image and set it to require gradients
    adv_image = input_image.clone().detach().requires_grad_(True)  # Ensure requires_grad is True

    # Random initialization around input_image
    random_noise = torch.FloatTensor(input_image.shape).uniform_(-epsilon, epsilon)
    adv_image = adv_image + random_noise
    adv_image = torch.clamp(adv_image, clip_value_min, clip_value_max).detach().requires_grad_(True)

    # If no input label is provided, use the model's prediction
    if input_label is None:
        output = model(adv_image)
        input_label = torch.argmax(output, dim=1)

    # Perform PGD attack
    for _ in range(num_steps):
        adv_image.requires_grad_(True)  # Ensure requires_grad is True in each iteration
        output = model(adv_image)
        loss = nn.CrossEntropyLoss()(output, input_label)
        model.zero_grad()
        loss.backward()

        # Check if gradient is available before accessing 'data'
        if adv_image.grad is not None:
            gradient = adv_image.grad.data
            adv_image = adv_image + step_size * gradient.sign()
            adv_image = torch.clamp(adv_image, input_image - epsilon, input_image + epsilon)  # Clip to a valid boundary
            adv_image = torch.clamp(adv_image, clip_value_min, clip_value_max)  # Clip to a valid range
            adv_image = adv_image.detach()  # Detach to prevent gradient accumulation
        else:
            print("Warning: Gradient is None. Check for detach operations.")

    return adv_image.detach()
```

Our implementation of PGD attack

Attacks

FGSM, PGD, MIM, TRADES

$$\boldsymbol{x_{adv} = argmax_{x' \in B_\epsilon(x)} l(f(x'; \theta), y)}$$

- Projected Gradient Descent (Ascent) (PGD)
  - $x_0 = x + Uniform([-\epsilon, \epsilon])$
  - $\tilde{x}_{t+1} = x_t + \eta sign(\nabla_x l(f(x_t; \theta), y))$
  - $x_{t+1} = clip(\tilde{x}_{t+1}, min\_val, max\_val)$

Attack SOTA pretrained model

```python
# Example usage
# img_path = './imgs/cat.jpg'
# img_path = './imgs/spider.jpg'
# img_path = './imgs/chicken.jpeg'
img_path = './imgs/elephant.jpg'
img = Image.open(img_path)

# Preprocess the image
img_t = preprocess_input(img)
batch_t = torch.unsqueeze(img_t, 0)

# Get predictions
with torch.no_grad():
    out = vgg19(batch_t)

# Decode the predictions
preds = torch.nn.functional.softmax(out, dim=1)
# decoded_preds = decode_predictions(preds, top=3)
# print('Predicted:', decoded_preds)

print('Predicted:', decode_predictions(preds, top=3)[0])

plt.imshow(img)
plt.xlabel("True: elephant, predicted: {}".format(decode_predictions(preds, top=3)[0][0]), fontsize= 12)
plt.xticks([])
plt.yticks([])
plt.grid(False)
```

```
Predicted: ('tusker', 0.5307024121284485)
```

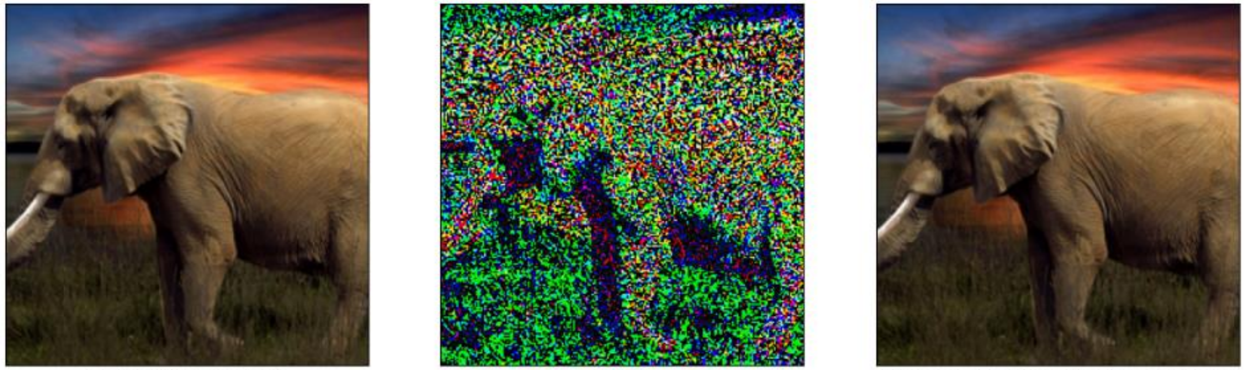
True: elephant, predicted: tusker

What is tusker?

A "tusker" typically refers to an animal, particularly an elephant, that has large, prominent tusks. The term is most commonly used to describe male elephants, especially those with exceptionally large or long tusks.

```python
attack_types = ['fgsm', 'trades', 'pgd']
attack_type = attack_types[2]

x_pgd = attack(attack_type, vgg19, batch_t, None, epsilon=0.01, num_steps=10, step_size=0.002, clip_value_min=-255.0, clip_value_max=255.0)


# Or directly use "pgd_attack" function
# x_pgd = pgd_attack(vgg19, batch_t, None, epsilon=0.01, num_steps=20, step_size=0.002, soft_label=True, clip_value_min=-25

pgd_pred = vgg19(x_pgd)
true_label = decode_predictions(preds, top=3)[0][0]
adv_label = decode_predictions(pgd_pred, top=3)[0][0]
print("True label: {}, adversarial label: {}".format(true_label, adv_label))

# Convert the adversarial image and original image from PyTorch tensors to PIL images
img_pgd_pil = revert_preprocess(x_pgd.squeeze(0))
img_pil = revert_preprocess(batch_t.squeeze(0))

# Convert PIL images to numpy arrays
img_pgd = np.array(img_pgd_pil)
img = np.array(img_pil)

# Calculate noise and clip values for visualization
noise_pgd = np.clip(np.abs(img_pgd - img) * 20, 0, 255).astype('int')  # Multiply the noise by 20 for visualization
```

```
True label: tusker, adversarial label: African bush elephant
<Figure size 640x480 with 0 Axes>
```



Original image: tusker | Noise | Adversarial image: African bush elephant

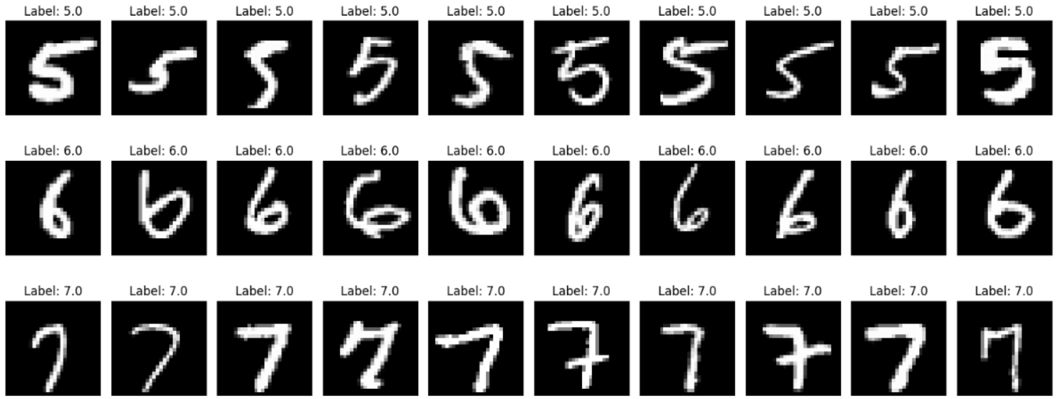
Attack Normal LeNet

```python
# Define LeNet in PyTorch
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(1, 6, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Conv2d(6, 16, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.AvgPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(16 * 7 * 7, 120),
            nn.ReLU(inplace=True),
            nn.Linear(120, 84),
            nn.ReLU(inplace=True),
            nn.Linear(84, 10)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Create LeNet

```python
# Initialize model, optimizer, and loss function
lenet = LeNet()
optimizer = optim.Adam(lenet.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()


# Training function
def train(model, train_loader, optimizer, criterion, epochs=5):
    model.train()

    for epoch in range(epochs):
        total_loss = 0.0
        y_pred = []
        y_true = []
        for batch_idx, (data, target) in enumerate(train_loader):
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            # Log
            total_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)
            y_pred.extend(pred.squeeze().cpu().numpy())
            y_true.extend(target.cpu().numpy())

        train_loss = total_loss / len(train_loader)
        train_acc = accuracy_score(y_true, y_pred)

        print(f"Epoch {epoch+1}, Training Loss: {train_loss:.4f}, Training Acc: {train_acc:.2f}%, ")

# Train the model
train(lenet, train_loader, optimizer, criterion, epochs=5)
```

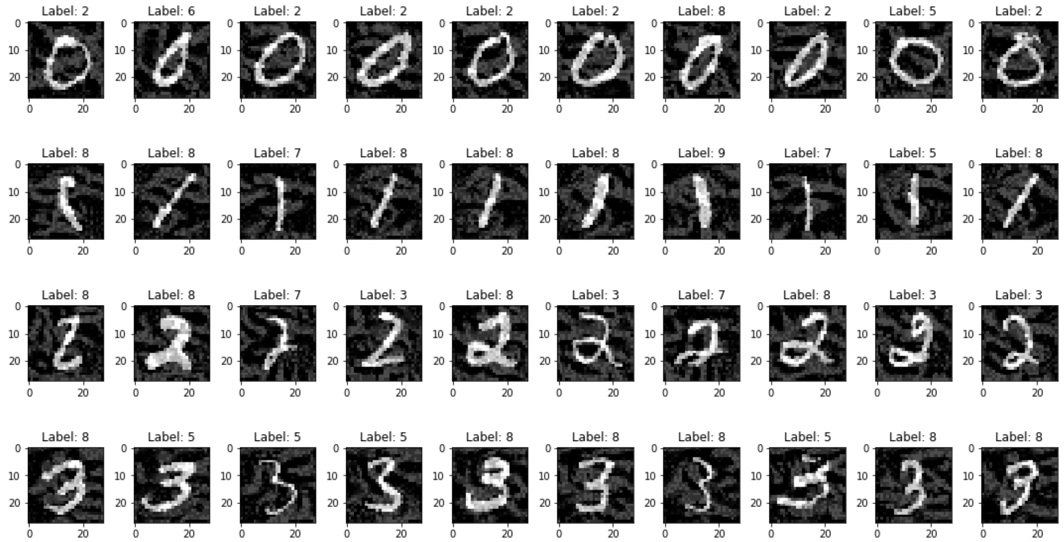
Train LeNet




```python
# Generate adversarial examples for the image samples
image_samples_tensor = torch.tensor(image_samples).permute(0, 3, 1, 2).float()
label_samples_tensor = torch.tensor(label_samples).long()

image_samples_adv = pgd_attack(lenet, image_samples_tensor, label_samples_tensor, epsilon=0.3,
                               num_steps=20, step_size=0.01, clip_value_min=0.0, clip_value_max=1.0)
image_samples_adv_np = image_samples_adv.permute(0, 2, 3, 1).detach().numpy()
label_sample_adv = np.argmax(lenet(image_samples_adv).detach().numpy(), axis=1)
```

```python
# Evaluate adversarial accuracy on the test set

y_adv = []
y_true = []

lenet.eval()
for data, target in test_loader:
    data, target = data.to(device), target.to(device)
    data_adv = pgd_attack(lenet, data, target, epsilon=0.3, num_steps=20, step_size=0.01,
                          clip_value_min=0.0, clip_value_max=1.0)
    output_adv = lenet(data_adv)
    pred_adv = output_adv.argmax(dim=1, keepdim=True)
    y_adv.extend(pred_adv.squeeze().cpu().numpy())
    y_true.extend(target.numpy())

test_adv_acc = accuracy_score(y_true, y_adv)
print("Test adversarial accuracy: {}%".format(test_adv_acc*100))
```

Very low robust accuracy



Adversarial Training to Improve LeNet

```
# Initialize model, optimizer, and loss function
lenet_defence = LeNet()
optimizer = optim.Adam(lenet_defence.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# Training function with adversarial examples
def train_step_adv(model, x, x_adv, y, optimizer, criterion):
    model.train()
    optimizer.zero_grad()
    logits = model(x)
    logits_adv = model(x_adv)
    loss = (criterion(logits, y) + criterion(logits_adv, y)) / 2
    loss.backward()
    optimizer.step()

    pred_adv = logits_adv.argmax(dim=1, keepdim=True)
    return loss.item(), pred_adv

# Metrics
train_loss = []
test_acc_clean = []
test_acc_pgd = []
```

Train one batch at an iteration

```
# Training loop
epochs = 5
for epoch in range(epochs):
    lenet_defence.train()
    total_loss = 0.0
    y_pred = []
    y_true = []
    for batch_idx, (x, y) in enumerate(train_loader):
        x, y = x.to(device), y.to(device)
        x_adv = pgd_attack(lenet_defence, x, y, epsilon=0.3, num_steps=10, step_size=0.01, clip_value_min=0.0, clip_value_max=1.0)
        loss, pred_adv= train_step_adv(lenet_defence, x, x_adv, y, optimizer, criterion)

        # Log
        total_loss += loss
        y_pred.extend(pred_adv.squeeze().cpu().numpy())
        y_true.extend(y.cpu().numpy())
    train_loss = total_loss / len(train_loader)
    train_acc = accuracy_score(y_true, y_pred)

    print(f"Epoch {epoch+1}, Training Loss: {train_loss:.4f}, Training Acc: {train_acc*100:.2f}%, ")
```

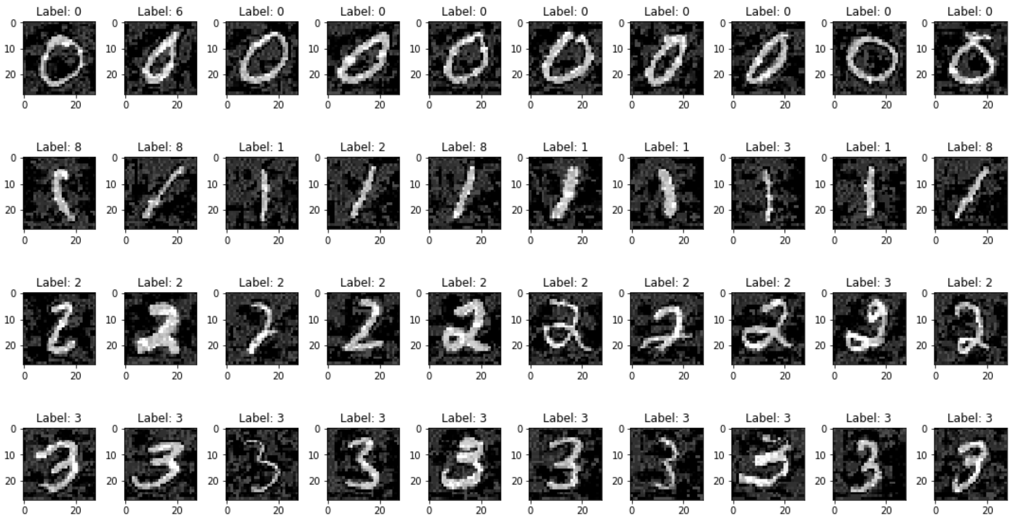
```
Epoch 1, Training Loss: 0.7438, Training Acc: 64.05%,
Epoch 2, Training Loss: 0.2900, Training Acc: 84.99%,
Epoch 3, Training Loss: 0.2081, Training Acc: 89.14%,
Epoch 4, Training Loss: 0.1683, Training Acc: 91.23%,
Epoch 5, Training Loss: 0.1443, Training Acc: 92.46%,
```

Train in several epochs using adversarial training

```
y_adv = []
y_true = []

lenet_defence.eval()
for data, target in test_loader:
    data, target = data.to(device), target.to(device)
    data_adv = pgd_attack(lenet_defence, data, target, epsilon=0.3, num_steps=20, step_size=0.01, clip_value_min=0.0, clip_value_max=1.0)
    output_adv = lenet_defence(data_adv)
    pred_adv = output_adv.argmax(dim=1, keepdim=True)
    y_adv.extend(pred_adv.squeeze().numpy())
    y_true.extend(target.numpy())

test_adv_acc = accuracy_score(y_true, y_adv)
print("Test adversarial accuracy: {}%".format(test_adv_acc*100))
```

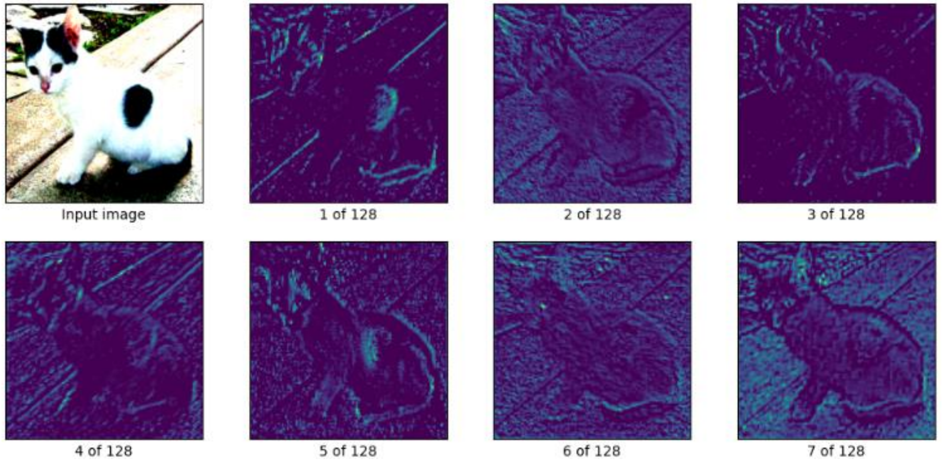
```
Test adversarial accuracy: 81.25%
```

Testing robust accuracy



CNN Visualization

- Visualize feature maps

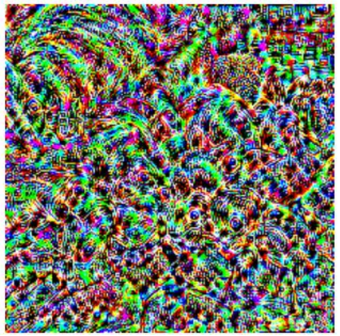


- Visualize **raw** filters

```
visualize_filter(vgg19, "block1_conv1", n_cols= 8)
```

- Visualize filters using activation map maximization

```
# visualize activation maximization
visualize_activation_maximization(model, "features.34", learning_rate=50, filter_index=400, iterations=200, alpha=50)
```



GradCam Model Explanability

- Which part of image that a CNN (e.g., ResNet 50) highly bases on to make prediction?

Original image Gradcam Guided Gradcam

Gradcam Model Explanability

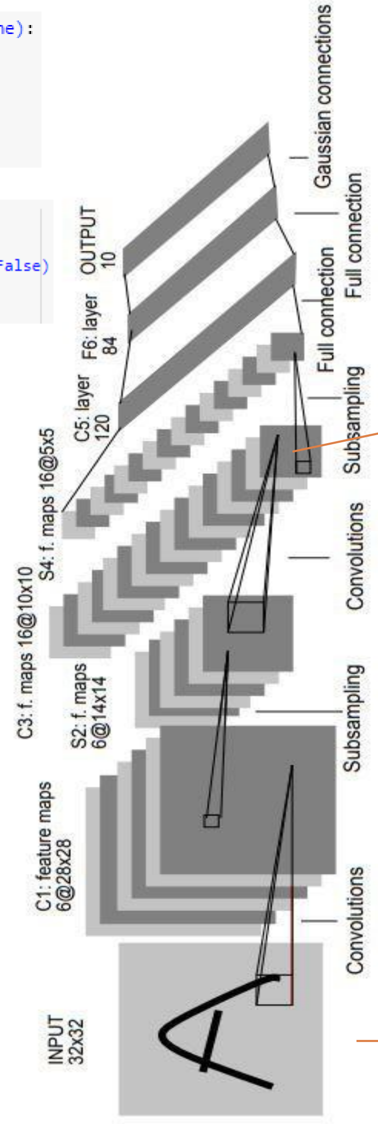
```
class GradCAM_Base(_BaseWrapper):
    """
    "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization"
    https://arxiv.org/pdf/1610.02391.pdf
    Look at Figure 2 on page 4
```

```
def generate(self, target_layer):
    fmaps = self._find(self.fmap_pool, target_layer)
    grads = self._find(self.grad_pool, target_layer)
    weights = F.adaptive_avg_pool2d(grads, 1)

    gcam = torch.mul(fmaps, weights).sum(dim=1, keepdim=True)
    gcam = F.relu(gcam)
    gcam = F.interpolate(
        gcam, self.image_shape, mode="bilinear", align_corners=False
    )

    B, C, H, W = gcam.shape
    gcam = gcam.view(B, -1)
    gcam -= gcam.min(dim=1, keepdim=True)[0]
    gcam /= gcam.max(dim=1, keepdim=True)[0]
    gcam = gcam.view(B, C, H, W)

    return gcam
```

```
class GuidedBackPropagation_Base(BackPropagation):
    """
    "Striving for Simplicity: the All Convolutional Net"
    https://arxiv.org/pdf/1412.6806.pdf
    Look at Figure 1 on page 8.
    """

    def __init__(self, model):
        super(GuidedBackPropagation_Base, self).__init__(model)

        def backward_hook(module, grad_in, grad_out):
            # Cut off negative gradients
            if isinstance(module, nn.ReLU):
                return (F.relu(grad_in[0]),)

        for module in self.model.named_modules():
            self.handlers.append(module[1].register_backward_hook(backward_hook))
```

```
def make_gradcam(input_tensor, model, target_layer, label_index = None):
    gcam = GradCAM_Base(model=model)
    _ = gcam.forward(input_tensor)
    gcam.backward(label_index)
    heatmap = gcam.generate(target_layer=target_layer)
    return heatmap[0,0]
```

```
heatmap = make_gradcam(input_tensor, model=model_resnet50,
                       target_layer="layer4", label_index = 285)
gradcam_img = save_gradcam(gcam=1 - heatmap, raw_image=raw_img, paper_cmap=False)
plt.rcParams['figure.figsize'] = [10, 5]
plt.imshow(gradcam_img,  alpha=0.5)
```



$p = [p_1, \dots, p_{\hat{y}}, \dots, p_{1000}]$

$\hat{y}$ is a **predicted label**

A layer with **3D tensor** feature maps

$A = [A_1, A_2, \dots, A_{2048}]$

with shape $[2048, 7, 7]$

A_i with shape $[7,7]$ is a feature map.

$heatmap = \sum_{i=1}^{2048} sum\left(\frac{\partial p_{\hat{y}}}{\partial A_i}\right) \times A_i$ has shape $[7,7]$

Normalize *heatmap* and **resize** to [224, 224]

[224, 224]

Thanks for your attention!