Attention (cont.)

Arush Iyer

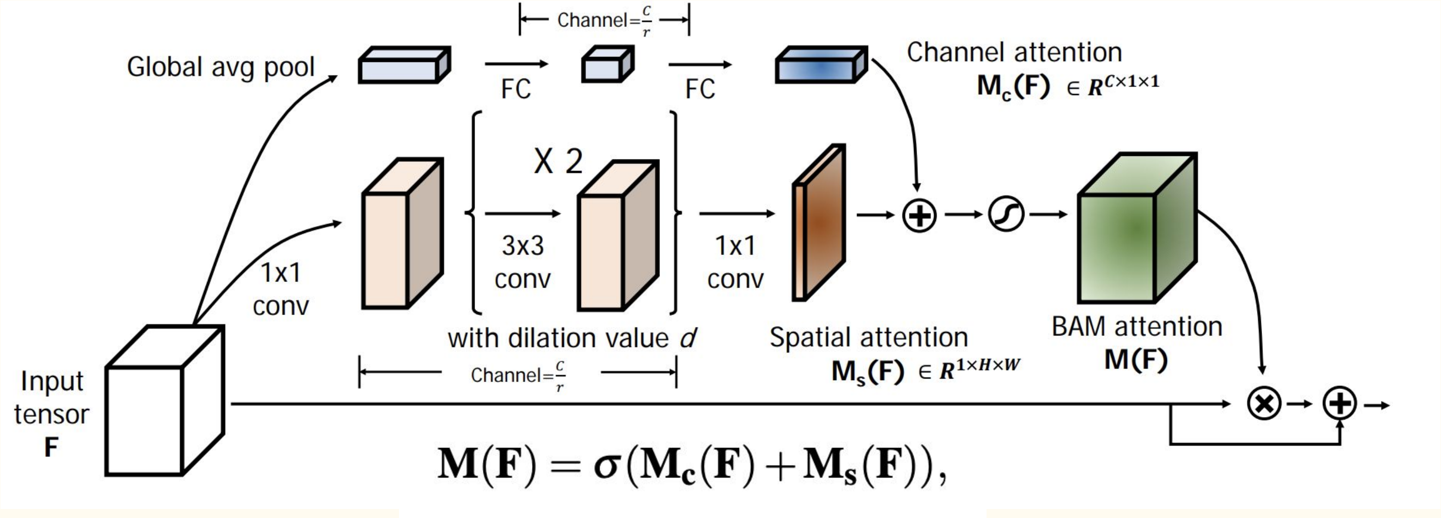
Review

- Attention lets models to selectively focus on the most important parts of input data
- Types
  - Self attention
  - Channel attention (what feature channels matter most)
  - Spatial attention (which regions are most important)
  - Post-hoc visualization methods
- Implementations in our code
  - CBAM (implements spatial and channel attention through two sequential sub-models)
  - Grad-CAM (post-hoc visualization method)

BAM

- High level:
  - Like CBAM, integrates spatial and channel attention
  - Unlike CBAM, applies attention streams in parallel (vs sequentially)
-
```
Input Feature Map
        |
   ┌────┴─────┐
   |          |
Channel    Spatial
Attention  Attention
   |          |
   └────┬─────┘
        | (Element-wise addition, then sigmoid)
        |
Output Refined Feature Map
```

BAM



- F: Input feature map (CxHxW)
- M(F): Output attention map,
- M_c(F): channel branch,
- M_s(F): spatial attention map

- Hyperparameters
  - d: dilation value
  - r: reduction factor

BAM

- Math
  - Channel Branch
    - Perform global pooling (implemented as average here but can also do max)
    - Pass it through an MLP
      - Reduction factor r reduces number of channels used to train MLP
      - Eventually restored to original number of channels
    - Apply Batch Normalization: normalize to align it with the scale of the Spatial Branch

$$\begin{aligned}\mathbf{M_c}(\mathbf{F}) &= BN(MLP(AvgPool(\mathbf{F}))) \\ &= BN(\mathbf{W_1}(\mathbf{W_0}AvgPool(\mathbf{F}) + \mathbf{b_0}) + \mathbf{b_1}),\end{aligned}$$

BAM

- Math
  - Spatial Branch
    - Reduce the number of channels by the reduction factor r
    - Apply two dilated convolutions
      - Basically: introduce holes between elements
        - d = 3 => skip every 3 pixels
      - Use dilation factor d to increase the receptive field
      - Each output sees a larger part of the input, allowing the model to capture longer-range spatial dependencies
    - Re-compress to a single attention map and apply batch normalization

$$\mathbf{M_s}(\mathbf{F}) = BN(f_3^{1\times1}(f_2^{3\times3}(f_1^{3\times3}(f_0^{1\times1}(\mathbf{F}))))),$$

BAM

- Math
  - Combining the two
    - Authors chose element wise summation
    - Take a sigmoid to get everything between 0 and 1

$$\mathbf{M}(\mathbf{F}) = \sigma(\mathbf{M_c}(\mathbf{F}) + \mathbf{M_s}(\mathbf{F})),$$

BAM

- Performance
  - Generally, subtle performance changes compared to CBAM
  - Possibly faster computation because spatial and channel attention are implemented in parallel
- Hyperparameter tuning
  - r: reduction ratio
    - Higher = stronger compression, using fewer parameters, potential risk of underfitting
    - Lower = more parameters but much slower
  - d: dilation value
    - Higher = potential to miss local features but capture more global context and vice versa
- Implementation
  - Github repos that have BAM modules

CBAM Implementation Explained

- Quick review
  - Channel submodule
    - Apply average and max pooling across each channel
    - Run those through an MLP
    - Apply a sigmoid activation function to get weights between 0 and 1

$$\mathbf{M_c}(\mathbf{F}) = \sigma(MLP(AvgPool(\mathbf{F})) + MLP(MaxPool(\mathbf{F})))$$

  - Spatial submodule
    - Apply average and max pooling
    - Stack average and max maps together
    - Apply a convolutional layer (usually a 7x7 filter size)
    - Apply the sigmoid function

$$\mathbf{M_s}(\mathbf{F}) = \sigma(f^{7\times 7}([AvgPool(\mathbf{F}); MaxPool(\mathbf{F})]))$$

CBAM Implementation Explained

Code from ResNet_CBAM _4_27_patches.ipynb

```
class ChannelAttention(nn.Module):
    def __init__(self, in_planes, ratio=16):
        super(ChannelAttention, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)
        self.fc1 = nn.Conv2d(in_planes, in_planes // ratio, 1, bias=False)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Conv2d(in_planes // ratio, in_planes, 1, bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = self.fc2(self.relu1(self.fc1(self.avg_pool(x))))
        max_out = self.fc2(self.relu1(self.fc1(self.max_pool(x))))
        return self.sigmoid(avg_out + max_out)
```

CBAM Implementation Explained

- Channel sub-module
  - self.avg_pool/self.max_pool: implementation of average and max pooling
  - self.fc1 = nn.Conv2d(in_planes, in_planes // ratio, 1, bias=False)
    - First layer of the neural network
    - Reduces the number of channels from in_planes to in_planes/ratio
    - Can change ratio number (smaller = less compression = more expressive)
  - self.fc2 = nn.Conv2d(in_planes // ratio, in_planes, 1, bias=False)
    - Second layer of the neural network
    - Returns the number of channels back to in_planes
    - Essentially, MLP compresses input, learns, and then reprojects back to its original size
  - Using ReLu to get rid of negative numbers and sigmoid so the weights are between 0 and 1

CBAM Implementation Explained

```python
class SpatialAttention(nn.Module):
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()
        padding = 3 if kernel_size == 7 else 1
        self.conv1 = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg_out = torch.mean(x, dim=1, keepdim=True)
        max_out, _ = torch.max(x, dim=1, keepdim=True)
        x = torch.cat([avg_out, max_out], dim=1)
        return self.sigmoid(self.conv1(x))
```

CBAM Implementation Explained

- Spatial sub-module
    - kernel_size: how big the convolutional filter is (higher means bigger)
        - Basically, how large the view is when we're looking over the image,
        - Can change this (smaller = see finer details)
    - padding: how much padding you need so the output size is the same
        - Not super important
    - self.conv1 = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
        - First number (2) is the input channel. Need 2 because you are feeding it average and max pooling maps
        - Second number (1) is the output channel. You are outputting 1 map
    - Again have sigmoid function

CBAM Implementation Explained

```
class ResNetCBAM(nn.Module):
    def __init__(self, num_classes=2):
        super(ResNetCBAM, self).__init__()
        base = models.resnet50(pretrained=True)
        self.features = nn.Sequential(
            base.conv1,
            base.bn1,
            base.relu,
            base.maxpool,
            base.layer1,
            CBAM(256),
            base.layer2,
            CBAM(512),
            base.layer3,
            CBAM(1024),
            base.layer4,
            CBAM(2048),
        )
        self.avgpool = base.avgpool
        self.fc = nn.Linear(2048, num_classes)
```

- Adding CBAM modules after each ResNet block
- Refining features at each layer
- Potential tweak: try just adding CBAM after layers 3 and 4
    - Logic: early layer capture basic features, attention might not be helpful

CBAM Implementation Explained

- Summary of what we can tune
    - Decrease the ratio in Channel submodule
    - Decrease kernel size in Spatial submodule
    - Add CBAM just after layers 3 and 4
    - Get rid of max pooling and just add average pooling in Channel submodule

Next steps

- End-to-end, contextualized example of attention
- Grad-CAM explanation
- Do more research on vision transformers
- Other?

Additional Sources

Park, Jongchan, et al. BAM: Bottleneck Attention Module. arXiv preprint arXiv:1807.06514, 2018. https://arxiv.org/abs/1807.06514.