Spring 2024

CMPE 258-01 Deep Learning

Dr. Kaikai Liu, Ph.D. Associate Professor

Department of Computer Engineering

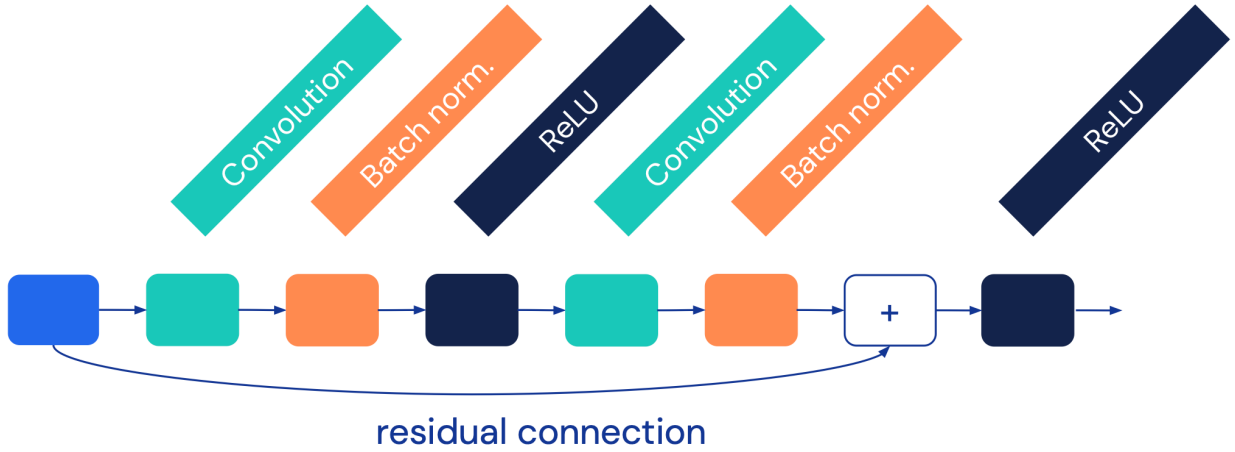
San Jose State University

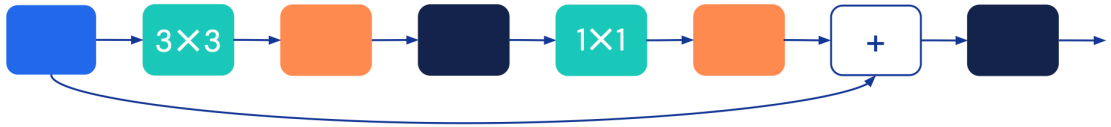
Email: kaikai.liu@sjsu.edu

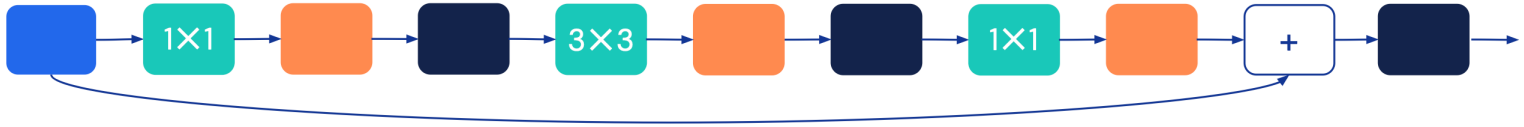
Website: https://www.sjsu.edu/cmpe/faculty/tenure-line/kaikai-liu.php

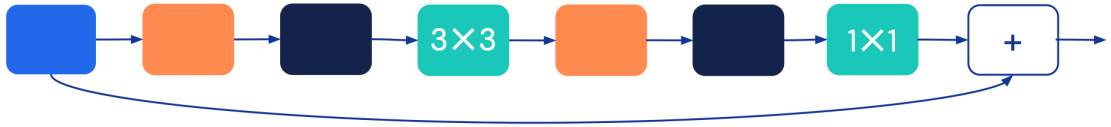
ResNet (2015)

- Residual connections facilitate training deeper networks
- Different flavours
  - ResNet V2 (bottom) avoids all nonlinearities in the residual pathway
  - Up to 152 layers









ResNet50

- Torchvision Resnet50
  - https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

```
model_resnet50 = get_model('resnet50', weights="DEFAULT")
summary(model=model_resnet50,
input_size=(32, 3, 224, 224), # make sure this is "input_size", not "input_shape"
# col_names=["input_size"], # uncomment for smaller output
col_names=["input_size", "output_size", "num_params", "trainable"],
col_width=20,
row_settings=["var_names"]
)
```

```
==========================================================================================
Layer (type (var_name))                  Input Shape          Output Shape         Param #
==========================================================================================
ResNet (ResNet)                          [32, 3, 224, 224]    [32, 1000]           --
├─Conv2d (conv1)                         [32, 3, 224, 224]    [32, 64, 112, 112]   9,408
├─BatchNorm2d (bn1)                      [32, 64, 112, 112]   [32, 64, 112, 112]   128
├─ReLU (relu)                            [32, 64, 112, 112]   [32, 64, 112, 112]   --
├─MaxPool2d (maxpool)                    [32, 64, 112, 112]   [32, 64, 56, 56]     --
├─Sequential (layer1)                    [32, 64, 56, 56]     [32, 256, 56, 56]    --
│    └─Bottleneck (0)                    [32, 64, 56, 56]     [32, 256, 56, 56]    --
│    │    └─Conv2d (conv1)               [32, 64, 56, 56]     [32, 64, 56, 56]     4,096
│    │    └─BatchNorm2d (bn1)            [32, 64, 56, 56]     [32, 64, 56, 56]     128
│    │    └─ReLU (relu)                  [32, 64, 56, 56]     [32, 64, 56, 56]     --
│    │    └─Conv2d (conv2)               [32, 64, 56, 56]     [32, 64, 56, 56]     36,864
│    │    └─BatchNorm2d (bn2)            [32, 64, 56, 56]     [32, 64, 56, 56]     128
│    │    └─ReLU (relu)                  [32, 64, 56, 56]     [32, 64, 56, 56]     --
│    │    └─Conv2d (conv3)               [32, 64, 56, 56]     [32, 256, 56, 56]    16,384
│    │    └─BatchNorm2d (bn3)            [32, 256, 56, 56]    [32, 256, 56, 56]    512
│    │    └─Sequential (downsample)      [32, 64, 56, 56]     [32, 256, 56, 56]    16,896
│    │    └─ReLU (relu)                  [32, 256, 56, 56]    [32, 256, 56, 56]    --
│    └─Bottleneck (1)                    [32, 256, 56, 56]    [32, 256, 56, 56]    --
│    │    └─Conv2d (conv1)               [32, 256, 56, 56]    [32, 64, 56, 56]     16,384
│    │    └─BatchNorm2d (bn1)            [32, 64, 56, 56]     [32, 64, 56, 56]     128
│    │    └─ReLU (relu)                  [32, 64, 56, 56]     [32, 64, 56, 56]     --
│    │    └─Conv2d (conv2)               [32, 64, 56, 56]     [32, 64, 56, 56]     36,864
│    │    └─BatchNorm2d (bn2)            [32, 64, 56, 56]     [32, 64, 56, 56]     128
│    │    └─ReLU (relu)                  [32, 64, 56, 56]     [32, 64, 56, 56]     --
│    │    └─Conv2d (conv3)               [32, 64, 56, 56]     [32, 256, 56, 56]    16,384
│    │    └─BatchNorm2d (bn3)            [32, 256, 56, 56]    [32, 256, 56, 56]    512
│    │    └─ReLU (relu)                  [32, 256, 56, 56]    [32, 256, 56, 56]    --
│    └─Bottleneck (2)                    [32, 256, 56, 56]    [32, 256, 56, 56]    --
│    │    └─Conv2d (conv1)               [32, 256, 56, 56]    [32, 64, 56, 56]     16,384
│    │    └─BatchNorm2d (bn1)            [32, 64, 56, 56]     [32, 64, 56, 56]     128
│    │    └─ReLU (relu)                  [32, 64, 56, 56]     [32, 64, 56, 56]     --
│    │    └─Conv2d (conv2)               [32, 64, 56, 56]     [32, 64, 56, 56]     36,864
│    │    └─BatchNorm2d (bn2)            [32, 64, 56, 56]     [32, 64, 56, 56]     128
│    │    └─ReLU (relu)                  [32, 64, 56, 56]     [32, 64, 56, 56]     --
│    │    └─Conv2d (conv3)               [32, 64, 56, 56]     [32, 256, 56, 56]    16,384
│    │    └─BatchNorm2d (bn3)            [32, 256, 56, 56]    [32, 256, 56, 56]    512
│    │    └─ReLU (relu)                  [32, 256, 56, 56]    [32, 256, 56, 56]    --
```

ResNet50

├─Sequential (layer2)	[32, 256, 56, 56]	[32, 512, 28, 28]	--
│ └─Bottleneck (0)	[32, 256, 56, 56]	[32, 512, 28, 28]	--
│ │ └─Conv2d (conv1)	[32, 256, 56, 56]	[32, 128, 56, 56]	32,768
│ │ └─BatchNorm2d (bn1)	[32, 128, 56, 56]	[32, 128, 56, 56]	256
│ │ └─ReLU (relu)	[32, 128, 56, 56]	[32, 128, 56, 56]	--
│ │ └─Conv2d (conv2)	[32, 128, 56, 56]	[32, 128, 28, 28]	147,456
│ │ └─BatchNorm2d (bn2)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv3)	[32, 128, 28, 28]	[32, 512, 28, 28]	65,536
│ │ └─BatchNorm2d (bn3)	[32, 512, 28, 28]	[32, 512, 28, 28]	1,024
│ │ └─Sequential (downsample)	[32, 256, 56, 56]	[32, 512, 28, 28]	132,096
│ │ └─ReLU (relu)	[32, 512, 28, 28]	[32, 512, 28, 28]	--
│ └─Bottleneck (1)	[32, 512, 28, 28]	[32, 512, 28, 28]	--
│ │ └─Conv2d (conv1)	[32, 512, 28, 28]	[32, 128, 28, 28]	65,536
│ │ └─BatchNorm2d (bn1)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv2)	[32, 128, 28, 28]	[32, 128, 28, 28]	147,456
│ │ └─BatchNorm2d (bn2)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv3)	[32, 128, 28, 28]	[32, 512, 28, 28]	65,536
│ │ └─BatchNorm2d (bn3)	[32, 512, 28, 28]	[32, 512, 28, 28]	1,024
│ │ └─ReLU (relu)	[32, 512, 28, 28]	[32, 512, 28, 28]	--
│ └─Bottleneck (2)	[32, 512, 28, 28]	[32, 512, 28, 28]	--
│ │ └─Conv2d (conv1)	[32, 512, 28, 28]	[32, 128, 28, 28]	65,536
│ │ └─BatchNorm2d (bn1)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv2)	[32, 128, 28, 28]	[32, 128, 28, 28]	147,456
│ │ └─BatchNorm2d (bn2)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv3)	[32, 128, 28, 28]	[32, 512, 28, 28]	65,536
│ │ └─BatchNorm2d (bn3)	[32, 512, 28, 28]	[32, 512, 28, 28]	1,024
│ │ └─ReLU (relu)	[32, 512, 28, 28]	[32, 512, 28, 28]	--
│ └─Bottleneck (3)	[32, 512, 28, 28]	[32, 512, 28, 28]	--
│ │ └─Conv2d (conv1)	[32, 512, 28, 28]	[32, 128, 28, 28]	65,536
│ │ └─BatchNorm2d (bn1)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv2)	[32, 128, 28, 28]	[32, 128, 28, 28]	147,456
│ │ └─BatchNorm2d (bn2)	[32, 128, 28, 28]	[32, 128, 28, 28]	256
│ │ └─ReLU (relu)	[32, 128, 28, 28]	[32, 128, 28, 28]	--
│ │ └─Conv2d (conv3)	[32, 128, 28, 28]	[32, 512, 28, 28]	65,536
│ │ └─BatchNorm2d (bn3)	[32, 512, 28, 28]	[32, 512, 28, 28]	1,024
│ │ └─ReLU (relu)	[32, 512, 28, 28]	[32, 512, 28, 28]	--

ResNet50

├─Sequential (layer3)	[32, 512, 28, 28]	[32, 1024, 14, 14]	--
│ └─Bottleneck (0)	[32, 512, 28, 28]	[32, 1024, 14, 14]	--
│ │ └─Conv2d (conv1)	[32, 512, 28, 28]	[32, 256, 28, 28]	131,072
│ │ └─BatchNorm2d (bn1)	[32, 256, 28, 28]	[32, 256, 28, 28]	512
│ │ └─ReLU (relu)	[32, 256, 28, 28]	[32, 256, 28, 28]	--
│ │ └─Conv2d (conv2)	[32, 256, 28, 28]	[32, 256, 14, 14]	589,824
│ │ └─BatchNorm2d (bn2)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv3)	[32, 256, 14, 14]	[32, 1024, 14, 14]	262,144
│ │ └─BatchNorm2d (bn3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	2,048
│ │ └─Sequential (downsample)	[32, 512, 28, 28]	[32, 1024, 14, 14]	526,336
│ │ └─ReLU (relu)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ └─Bottleneck (1)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ │ └─Conv2d (conv1)	[32, 1024, 14, 14]	[32, 256, 14, 14]	262,144
│ │ └─BatchNorm2d (bn1)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv2)	[32, 256, 14, 14]	[32, 256, 14, 14]	589,824
│ │ └─BatchNorm2d (bn2)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv3)	[32, 256, 14, 14]	[32, 1024, 14, 14]	262,144
│ │ └─BatchNorm2d (bn3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	2,048
│ │ └─ReLU (relu)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ └─Bottleneck (2)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ │ └─Conv2d (conv1)	[32, 1024, 14, 14]	[32, 256, 14, 14]	262,144
│ │ └─BatchNorm2d (bn1)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv2)	[32, 256, 14, 14]	[32, 256, 14, 14]	589,824
│ │ └─BatchNorm2d (bn2)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv3)	[32, 256, 14, 14]	[32, 1024, 14, 14]	262,144
│ │ └─BatchNorm2d (bn3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	2,048
│ │ └─ReLU (relu)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ └─Bottleneck (3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ │ └─Conv2d (conv1)	[32, 1024, 14, 14]	[32, 256, 14, 14]	262,144
│ │ └─BatchNorm2d (bn1)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv2)	[32, 256, 14, 14]	[32, 256, 14, 14]	589,824
│ │ └─BatchNorm2d (bn2)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv3)	[32, 256, 14, 14]	[32, 1024, 14, 14]	262,144
│ │ └─BatchNorm2d (bn3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	2,048
│ │ └─ReLU (relu)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ └─Bottleneck (4)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ │ └─Conv2d (conv1)	[32, 1024, 14, 14]	[32, 256, 14, 14]	262,144
│ │ └─BatchNorm2d (bn1)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv2)	[32, 256, 14, 14]	[32, 256, 14, 14]	589,824
│ │ └─BatchNorm2d (bn2)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv3)	[32, 256, 14, 14]	[32, 1024, 14, 14]	262,144
│ │ └─BatchNorm2d (bn3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	2,048
│ │ └─ReLU (relu)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ └─Bottleneck (5)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
│ │ └─Conv2d (conv1)	[32, 1024, 14, 14]	[32, 256, 14, 14]	262,144
│ │ └─BatchNorm2d (bn1)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv2)	[32, 256, 14, 14]	[32, 256, 14, 14]	589,824
│ │ └─BatchNorm2d (bn2)	[32, 256, 14, 14]	[32, 256, 14, 14]	512
│ │ └─ReLU (relu)	[32, 256, 14, 14]	[32, 256, 14, 14]	--
│ │ └─Conv2d (conv3)	[32, 256, 14, 14]	[32, 1024, 14, 14]	262,144
│ │ └─BatchNorm2d (bn3)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	2,048
│ │ └─ReLU (relu)	[32, 1024, 14, 14]	[32, 1024, 14, 14]	--
├─Sequential (layer4)	[32, 1024, 14, 14]	[32, 2048, 7, 7]	--

ResNet50

```
├─Sequential (layer4)                    [32, 1024, 14, 14]   [32, 2048, 7, 7]     --
│    └─Bottleneck (0)                     [32, 1024, 14, 14]   [32, 2048, 7, 7]     --
│    │    └─Conv2d (conv1)                [32, 1024, 14, 14]   [32, 512, 14, 14]    524,288
│    │    └─BatchNorm2d (bn1)             [32, 512, 14, 14]    [32, 512, 14, 14]    1,024
│    │    └─ReLU (relu)                   [32, 512, 14, 14]    [32, 512, 14, 14]    --
│    │    └─Conv2d (conv2)                [32, 512, 14, 14]    [32, 512, 7, 7]      2,359,296
│    │    └─BatchNorm2d (bn2)             [32, 512, 7, 7]      [32, 512, 7, 7]      1,024
│    │    └─ReLU (relu)                   [32, 512, 7, 7]      [32, 512, 7, 7]      --
│    │    └─Conv2d (conv3)                [32, 512, 7, 7]      [32, 2048, 7, 7]     1,048,576
│    │    └─BatchNorm2d (bn3)             [32, 2048, 7, 7]     [32, 2048, 7, 7]     4,096
│    │    └─Sequential (downsample)       [32, 1024, 14, 14]   [32, 2048, 7, 7]     2,101,248
│    │    └─ReLU (relu)                   [32, 2048, 7, 7]     [32, 2048, 7, 7]     --
│    └─Bottleneck (1)                     [32, 2048, 7, 7]     [32, 2048, 7, 7]     --
│    │    └─Conv2d (conv1)                [32, 2048, 7, 7]     [32, 512, 7, 7]      1,048,576
│    │    └─BatchNorm2d (bn1)             [32, 512, 7, 7]      [32, 512, 7, 7]      1,024
│    │    └─ReLU (relu)                   [32, 512, 7, 7]      [32, 512, 7, 7]      --
│    │    └─Conv2d (conv2)                [32, 512, 7, 7]      [32, 512, 7, 7]      2,359,296
│    │    └─BatchNorm2d (bn2)             [32, 512, 7, 7]      [32, 512, 7, 7]      1,024
│    │    └─ReLU (relu)                   [32, 512, 7, 7]      [32, 512, 7, 7]      --
│    │    └─Conv2d (conv3)                [32, 512, 7, 7]      [32, 2048, 7, 7]     1,048,576
│    │    └─BatchNorm2d (bn3)             [32, 2048, 7, 7]     [32, 2048, 7, 7]     4,096
│    │    └─ReLU (relu)                   [32, 2048, 7, 7]     [32, 2048, 7, 7]     --
│    └─Bottleneck (2)                     [32, 2048, 7, 7]     [32, 2048, 7, 7]     --
│    │    └─Conv2d (conv1)                [32, 2048, 7, 7]     [32, 512, 7, 7]      1,048,576
│    │    └─BatchNorm2d (bn1)             [32, 512, 7, 7]      [32, 512, 7, 7]      1,024
│    │    └─ReLU (relu)                   [32, 512, 7, 7]      [32, 512, 7, 7]      --
│    │    └─Conv2d (conv2)                [32, 512, 7, 7]      [32, 512, 7, 7]      2,359,296
│    │    └─BatchNorm2d (bn2)             [32, 512, 7, 7]      [32, 512, 7, 7]      1,024
│    │    └─ReLU (relu)                   [32, 512, 7, 7]      [32, 512, 7, 7]      --
│    │    └─Conv2d (conv3)                [32, 512, 7, 7]      [32, 2048, 7, 7]     1,048,576
│    │    └─BatchNorm2d (bn3)             [32, 2048, 7, 7]     [32, 2048, 7, 7]     4,096
│    │    └─ReLU (relu)                   [32, 2048, 7, 7]     [32, 2048, 7, 7]     --
├─AdaptiveAvgPool2d (avgpool)             [32, 2048, 7, 7]     [32, 2048, 1, 1]     --
├─Linear (fc)                             [32, 2048]           [32, 1000]           2,049,000
==========================================================================================
Total params: 25,557,032
Trainable params: 25,557,032
Non-trainable params: 0
Total mult-adds (G): 130.86
==========================================================================================
Input size (MB): 19.27
Forward/backward pass size (MB): 5690.62
Params size (MB): 102.23
Estimated Total Size (MB): 5812.11
```

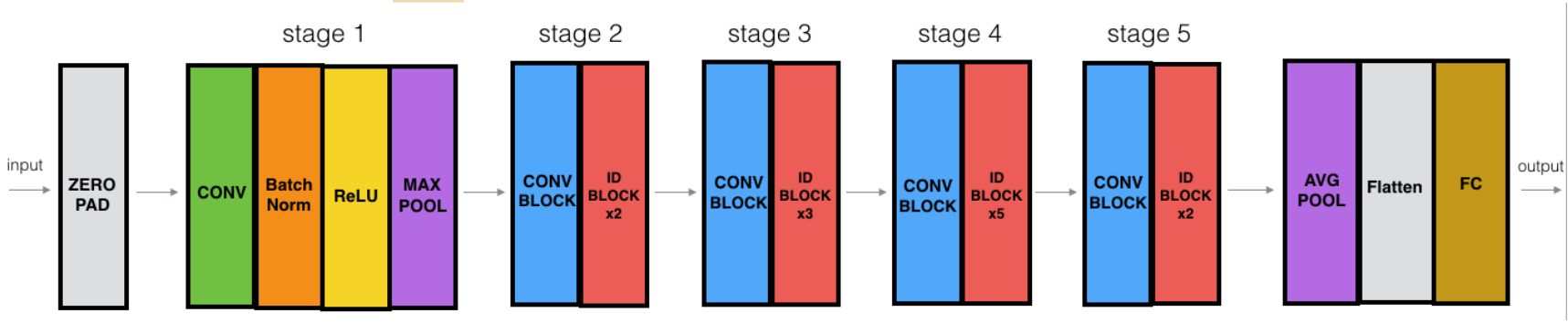
ResNet TorchVision

- Resnet50

```
@register_model()
@handle_legacy_interface(weights=("pretrained", ResNet50_Weights.IMAGENET1K_V1))
def resnet50(*, weights: Optional[ResNet50_Weights] = None, progress: bool = True, **kwargs: Any) -> ResNet:
    model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
    model.load_state_dict(weights.get_state_dict(progress=progress, check_hash=True))
```

```
x = self.conv1(x)
x = self.bn1(x)
x = self.relu(x)
x = self.maxpool(x)

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

x = self.avgpool(x)
x = torch.flatten(x, 1)
x = self.fc(x)

return x
```

```
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2, dilate=replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2, dilate=replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2, dilate=replace_stride_with_dilation[2])
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)
```



https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

ResNet TorchVision

- Resnet50

  model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
  model.load_state_dict(weights.get_state_dict(progress=progress, check_hash=True))

```
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2, dilate=replace_stride_with_dilation[0])
self.layer3 = self._make_layer(block, 256, layers[2], stride=2, dilate=replace_stride_with_dilation[1])
self.layer4 = self._make_layer(block, 512, layers[3], stride=2, dilate=replace_stride_with_dilation[2])
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion, num_classes)
```

def _make_layer

```
if stride != 1 or self.inplanes != planes * block.expansion:
    downsample = nn.Sequential(
        conv1x1(self.inplanes, planes * block.expansion, stride),
        norm_layer(planes * block.expansion),
    )
```

```
layers = []
layers.append(
    block(
        self.inplanes, planes, stride, downsample, self.groups
    )
)
self.inplanes = planes * block.expansion
for _ in range(1, blocks):
    layers.append(
        block(
            self.inplanes,
            planes,
            groups=self.groups,
            base_width=self.base_width,
            dilation=self.dilation,
            norm_layer=norm_layer,
        )
    )
```

https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

ResNet TorchVision

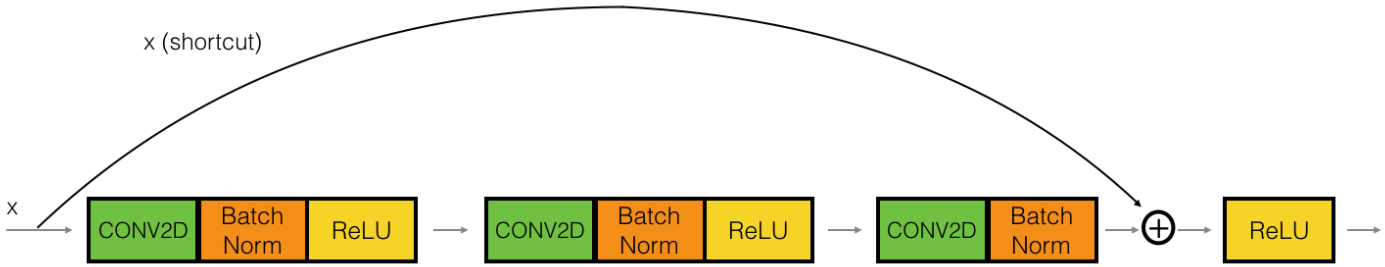
- Resnet50

```
model = ResNet(Bottleneck, [3, 4, 6, 3], **kwargs)
model.load_state_dict(weights.get_state_dict(progress=progress, check_hash=True))
```

Bottleneck

```
def forward(self, x: Tensor) -> Tensor:
    identity = x

    out = self.conv1(x)
    out = self.bn1(out)
    out = self.relu(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu(out)

    out = self.conv3(out)
    out = self.bn3(out)

    if self.downsample is not None:
        identity = self.downsample(x)

    out += identity
    out = self.relu(out)

    return out
```

```
if norm_layer is None:
    norm_layer = nn.BatchNorm2d
width = int(planes * (base_width / 64.0)) * groups
# Both self.conv2 and self.downsample layers downsample the :
self.conv1 = conv1x1(inplanes, width)
self.bn1 = norm_layer(width)
self.conv2 = conv3x3(width, width, stride, groups, dilation)
self.bn2 = norm_layer(width)
self.conv3 = conv1x1(width, planes * self.expansion)
self.bn3 = norm_layer(planes * self.expansion)
self.relu = nn.ReLU(inplace=True)
self.downsample = downsample
self.stride = stride
```



https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py

ResNet TorchVision

- ADAPTIVEAVGPOOL2D: Applies a 2D adaptive average pooling over an input signal composed of several input planes.
  - torch.nn.AdaptiveAvgPool2d(*output_size*)
    - The output is of size H x W, for any input size. The number of output features is equal to the number of input planes.

Shape:

- Input: (N, C, H_{in}, W_{in}) or (C, H_{in}, W_{in}).
- Output: (N, C, S_0, S_1) or (C, S_0, S_1), where $S = \text{output\_size}$.

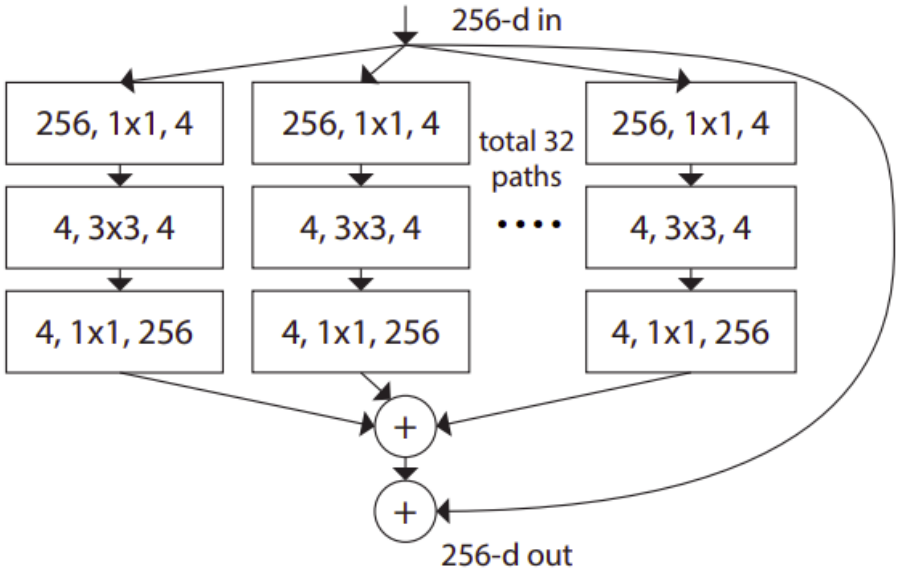
```
self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
self.fc = nn.Linear(512 * block.expansion,
num_classes)
```

- TORCH.FLATTEN
  - torch.flatten(input, start_dim=0, end_dim=-1)

```
x = self.avgpool(x)
x = torch.flatten(x, 1)
x = self.fc(x)
```

ResNeXt

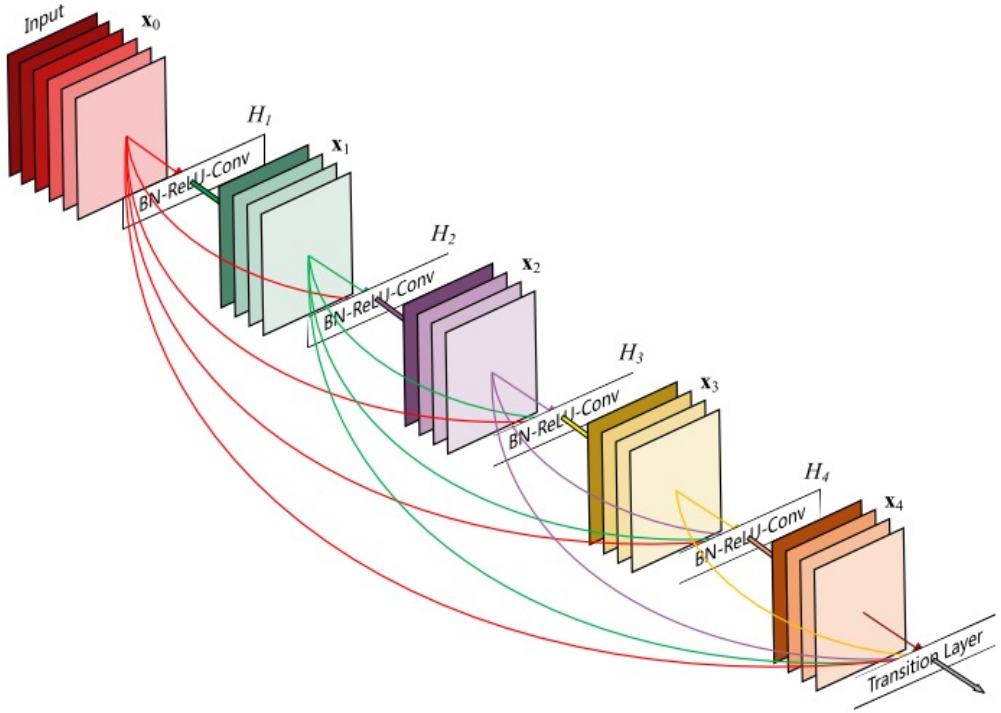
- ResNeXt
  - Paper: https://arxiv.org/abs/1611.05431
  - The authors of ResNeXt pursue an architecture that exploits Inception's split-transform-merge strategy whilst keeping in mind VGG's philosophy of repeating blocks and branches with identical topologies.
  - A module with many uniform branches (the number of branches is called cardinality)
  - Each branch would go like this: Shrink the data to 4 channels, feed them through 3 X 3 convolutions without changing the number of channels, and bring the width back to 256. These 256-dimensional tensors are summed together, and we utilize skip connections by adding the input to the result



it has a simple paradigm and only one hyper-parameter to be adjusted, while Inception has many hyper-parameters (like the kernel size of the convolutional layer of each path) to tune

DenseNet

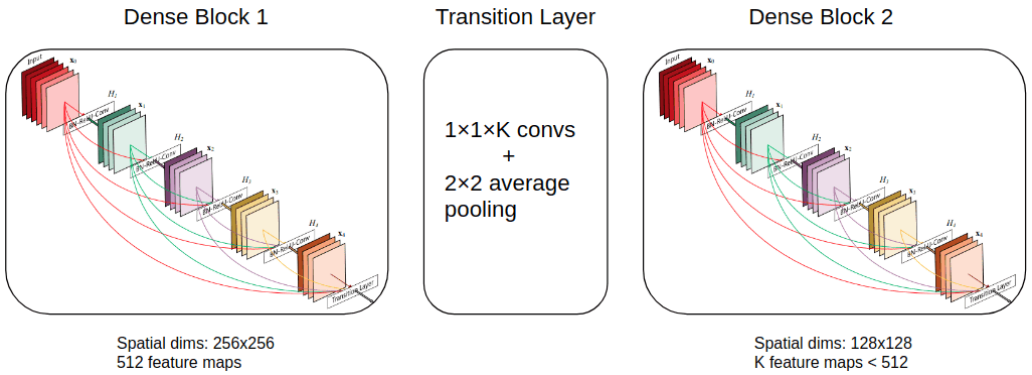
- DenseNet (2016): Densely Connected CNN
  - Paper: https://arxiv.org/abs/1608.06993
  - The input of each layer consists of the feature maps of all earlier layer, and its output is passed to each subsequent layer. The feature maps are aggregated with depth-concatenation.
  - Other than tackling the vanishing gradients problem, the authors argue that this architecture also encourages feature reuse, making the network highly parameter-efficient.



Since each layer receives feature maps from all preceding layers, network can be thinner and compact, i.e. number of channels can be fewer. The growth rate k is the additional number of channels for each layer.

DenseNet

- DenseNet (2016): Densely Connected CNN
  - Thus, the core idea behind it is feature reuse, which leads to very compact models. As a result it requires fewer parameters than other CNNs, as there are no repeated feature-maps.
  - There are two concerns here:
    - The feature maps have to be of the same size.
    - The concatenation with all the previous feature maps may result in memory explosion.
  - We have two solutions:
    - a) use conv layers with appropriate padding that maintain the spatial dims or
    - b) use dense skip connectivity only inside blocks called Dense Blocks.



The transition layer can down-sample the image dimensions with average pooling.

SqueezeNet

- SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size
  - Paper: https://arxiv.org/pdf/1602.07360.pdf
  - Architectural Design Strategies:
    - Strategy 1. Replace 3×3 filters with 1×1 filters
    - Strategy 2. Decrease the number of input channels to 3×3 filters
    - Strategy 3. Downsample late in the network so that convolution layers have large activation maps -> improve accuracy
  - Fire Module
    - comprised of: a squeeze convolution layer (which has only 1×1 filters), feeding into an expand layer that has a mix of 1×1 and 3×3 convolution filters.
    - There are three tunable dimensions (hyperparameters) in a Fire module: s1×1, e1×1, and e3×3.
    - s1×1: The number of 1×1 in squeeze layer.
    - e1×1 and e3×3: The number of 1×1 and 3×3 in expand layer.

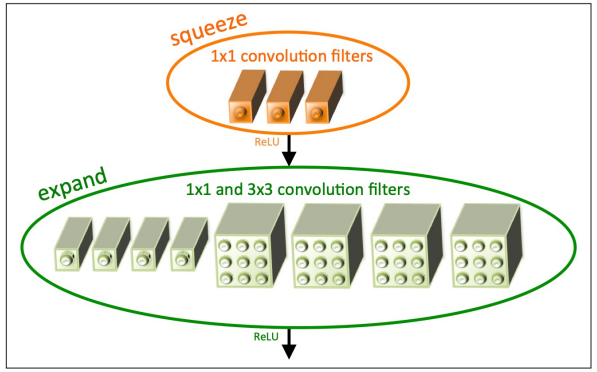


Figure 1: Microarchitectural view: Organization of convolution filters in the **Fire module**. In this example, $s_{1x1} = 3$, $e_{1x1} = 4$, and $e_{3x3} = 4$. We illustrate the convolution filters but not the activations.

SqueezeNet

- SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size
  - Paper: https://arxiv.org/pdf/1602.07360.pdf
  - SqueezeNet (Left): begins with a standalone convolution layer (conv1), followed by 8 Fire modules (fire2–9), ending with a final conv layer (conv10).
  - SqueezeNet with simple bypass (Middle) and SqueezeNet with complex bypass (Right): The use of bypass is inspired by ResNet.

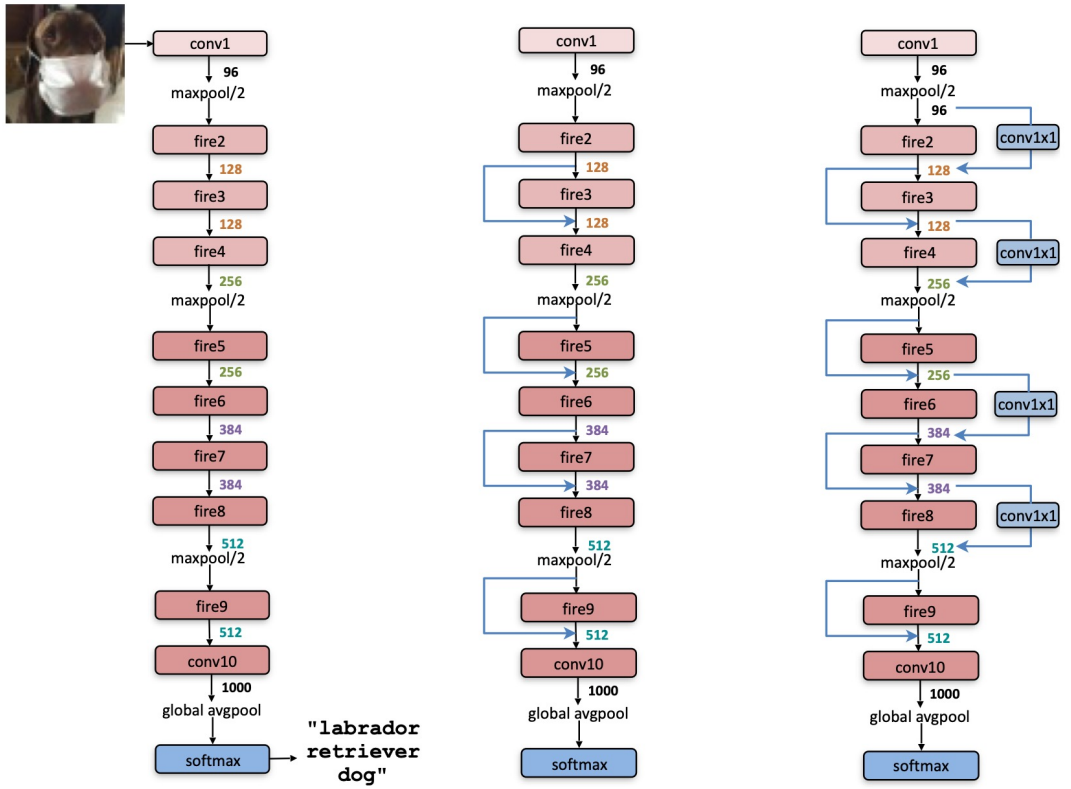


Figure 2: Macroarchitectural view of our SqueezeNet architecture. Left: SqueezeNet (Section 3.3); Middle: SqueezeNet with simple bypass (Section 6); Right: SqueezeNet with complex bypass (Section 6).

Squeeze-and-excitation

- Squeeze-and-excitation networks (2017)
  - Hu, J.; Shen, L.; Sun, G., ***Squeeze-and-excitation networks***, IEEE conference on computer vision and pattern recognition (2018)

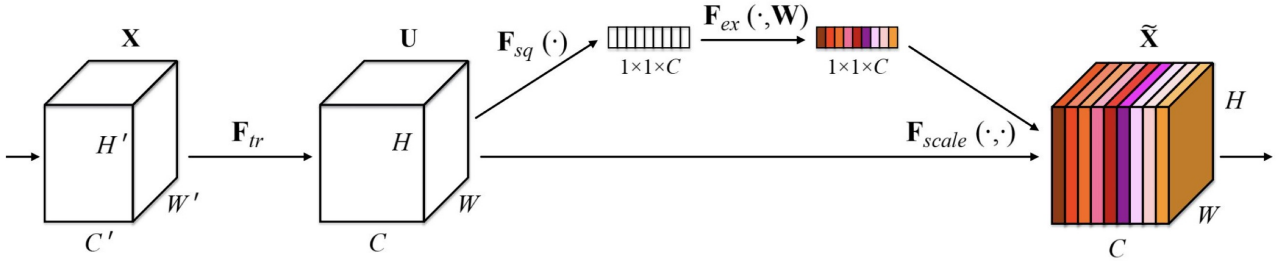
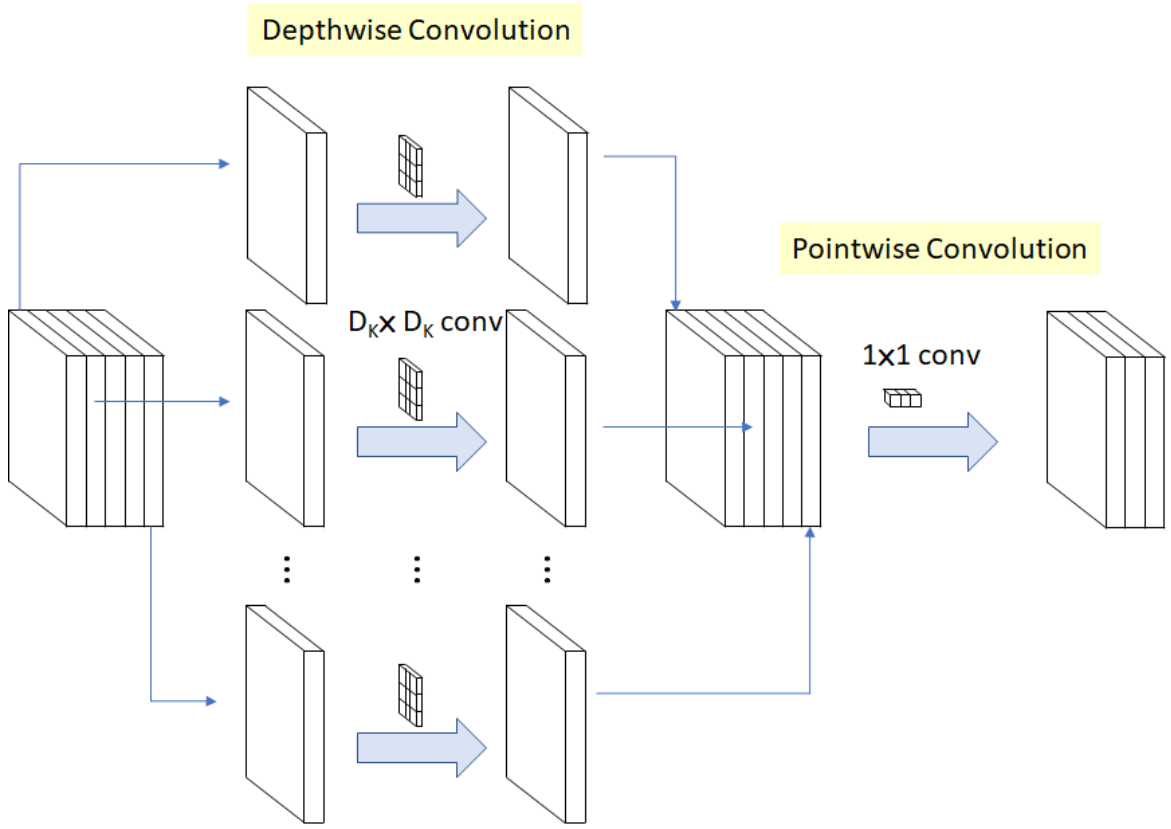


Figure from Hu et al. (2018)

  - Adding a content aware mechanism to weight each channel adaptively. In it's most basic form this could mean adding a single parameter to each channel and giving it a linear scalar how relevant each one is.
  - First, they get a global understanding of each channel by squeezing the feature maps to a single numeric value. This results in a vector of size n, where n is equal to the number of convolutional channels. Afterwards, it is fed through a two-layer neural network, which outputs a vector of the same size. These n values can now be used as weights on the original features maps, scaling each channel based on its importance.

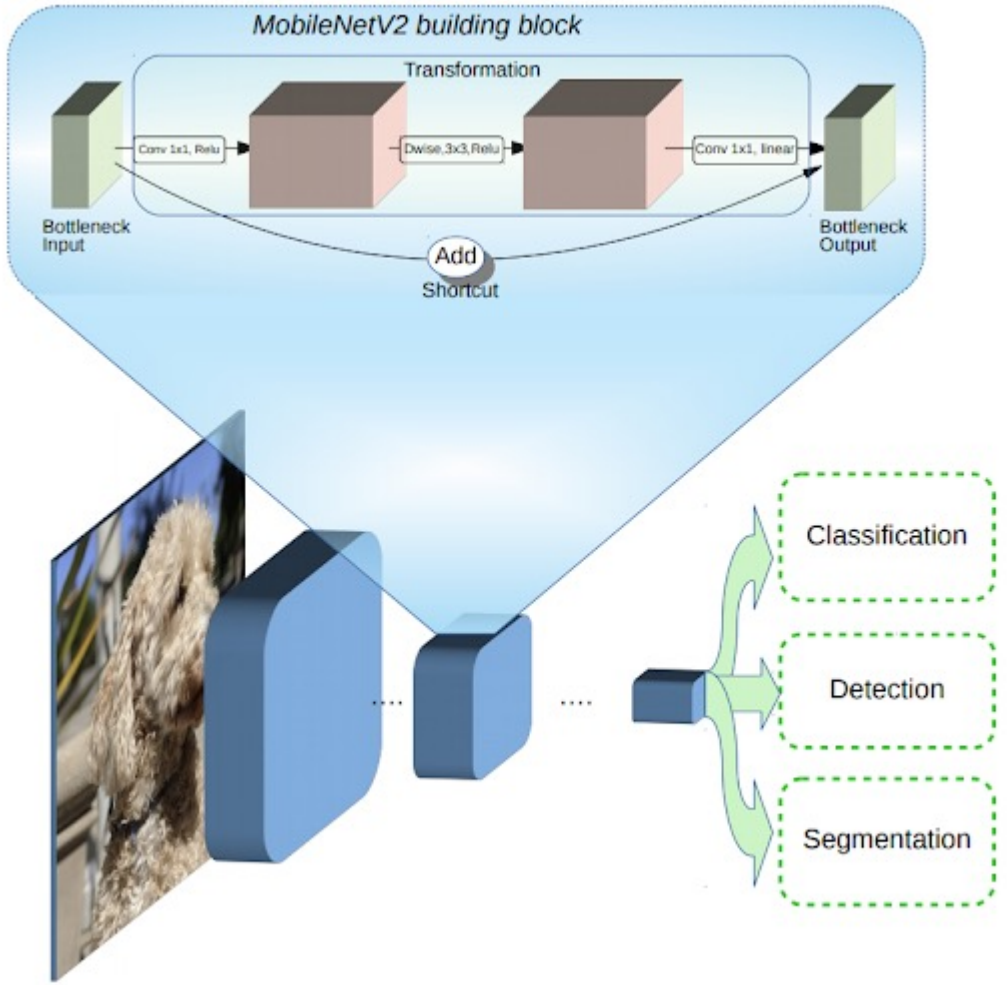
MobileNet

- MobileNetV1
  - Depthwise Separable Convolution is used to reduce the model size and complexity. It is particularly useful for mobile and embedded vision applications.
  - Smaller model size: Fewer number of parameters
  - Smaller complexity: Fewer Multiplications and Additions (Multi-Adds)
  - MobileNet, a smaller and efficient network architecture optimized for speed, has approximately 3.3M parameters, while ResNet-152 (yes, 152 layers), once the state of the art in the ImageNet classification competition, has around 60M.



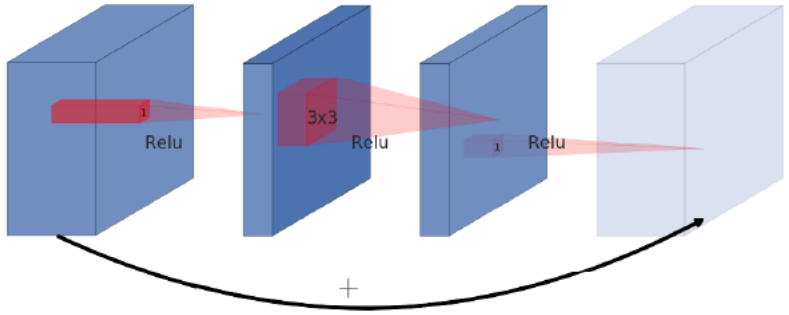
MobileNet V2

- MobileNetV2 builds upon the ideas from MobileNetV1, using depthwise separable convolution as efficient building blocks.
- V2 introduces two new features to the architecture: 1) linear bottlenecks between the layers, and 2) shortcut connections between the bottlenecks.



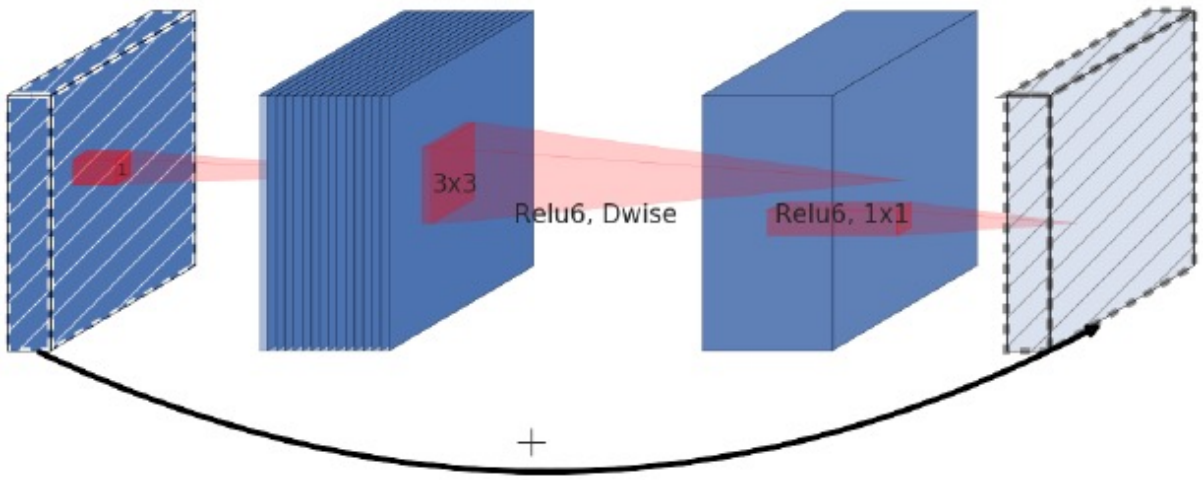
MobileNet V2

- Inverted Residuals
  - Residual blocks connect the beginning and end of a convolutional block with a skip connection. By adding these two states the network has the opportunity of accessing earlier activations that weren't modified in the convolutional block. This approach turned out to be essential in order to build networks of great depth.
  - An original residual block follows a **wide->narrow->wide** approach concerning the number of channels. The input has a high number of channels, which are compressed with an inexpensive 1x1 convolution. That way the following 3x3 convolution has **far fewer parameters**. In order to add input and output in the end the number of channels is increased again using another 1x1 convolution.



MobileNet V2

- Inverted Residuals
  - MobileNetV2 follows a **narrow->wide->narrow** approach. The first step widens the network using a 1x1 convolution because the following 3x3 depthwise convolution already greatly reduces the number of parameters. Afterwards another 1x1 convolution squeezes the network in order to match the initial number of channels.



MobileNet V2

- Linear Bottlenecks
  - ReLU discards values that are smaller than 0. This loss of information can be tackled by **increasing the number of channels** in order to increase the capacity of the network.
  - With inverted residual blocks, MobileNetV2 do the opposite and squeeze the layers where the skip connections are linked. This hurts the performance of the network
  - Linear bottleneck: the last convolution of a residual block has a linear output before it's added to the initial activations

```
def inverted_residual_block(x, expand=64, squeeze=16):
  m = Conv2D(expand, (1,1), activation='relu')(x)
  m = DepthwiseConv2D((3,3), activation='relu')(m)
  m = Conv2D(squeeze, (1,1), activation='relu')(m)
  return Add()([m, x])
```

```
def inverted_linear_residual_block(x, expand=64, squeeze=16):
  m = Conv2D(expand, (1,1), activation='relu')(x)
  m = DepthwiseConv2D((3,3),  activation='relu')(m)
  m = Conv2D(squeeze, (1,1))(m)
  return Add()([m, x])
```

MobileNet V2

- Other changes
  - adds Batch Normalization behind every convolutional layer
  - use ReLU6 instead of ReLU: limits the value of activations to a maximum of 6. The activation is linear as long as it's between 0 and 6.
    - This is helpful when you're dealing with fixed point inference. It limits the information left of the decimal point to 3 bits, meaning we have a guaranteed precision right of the decimal point.

```
def bottleneck_block(x, expand=64, squeeze=16):
  m = Conv2D(expand, (1,1))(x)
  m = BatchNormalization()(m)
  m = Activation('relu6')(m)
  m = DepthwiseConv2D((3,3))(m)
  m = BatchNormalization()(m)
  m = Activation('relu6')(m)
  m = Conv2D(squeeze, (1,1))(m)
  m = BatchNormalization()(m)
  return Add()([m, x])
```

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

The MobileNetV2 architecture

Thank You

Address:
ENG257, SJSU

Email Address:
Kaikai.liu@sjsu.edu