CHAPTER 8

MobileNet v1

There were some interesting attempts to get smaller models running on device post SqueezeNet. What was needed was a model designed specifically on mobile devices. What a group of researchers at Google produced was called MobileNet, which is an important family of networks for you to understand and where we will be spending a few chapters. At a high level, we will use depthwise separable convolutions to produce an even more accurate network than SqueezeNet that runs well on mobile phone hardware.

MobileNet (v1)

> MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

```
> https://arxiv.org/abs/1704.04861
```

Model designed specifically to run on mobile hardware, much better use of parameter + data space.

Spatial separable convolutions

Let's look again at our Sobel filter from our chapter where we introduced convolutions. There, we looked at it as being two 3x3 matrix operations. But if we are clever with our math, we can reduce this to a [3x1] and [1x3] multiplication.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_8

This gives us the same result, but has the additional property that it can be computed much more cheaply. Our [3x3] • [3x3] combination ends up requiring nine operations, whereas our [3x1] • [1x3] only requires six operations, a reduction of 33% percent! However, not all kernels can be broken up like this.

Depthwise convolutions

We can exploit one key property in our image data: color. We have three channels – red, green, blue – that we are running through the same sets of filter operations each time we evaluate our neural network.

We can create separate sets of convolutional filters for each area of the input image, combined together by color channel. In academic settings, channels are also referred to as depth, so these are called depthwise convolutions. A variant of this you need to know is increasing the number of filter outputs, which is called a channel multiplier.

Pointwise convolutions

This is only half the puzzle; we still need to combine our channel data back together. In our last chapter on SqueezeNet, we looked at how we can put a 1x1 convolution into our stack as a way to reduce data down significantly before applying our 3x3 convolution. Conceptually, this is called a pointwise convolution because all of the channel input data passes through it. By using these pointwise convolutions, we can map our reduced data space back to our desired final filter size. Then, we simply need to increase the number of pointwise operators to match our desired number of output filters.

Conceptually, we are taking our input image and running groups of depthwise convolutions and then using a stack of small pointwise convolutions to combine them back to our desired output shape. This combination of filters together is called a depthwise separable convolution

and is key to the performance of this network. We have gotten most of the benefits of SqueezeNet's compression approach, but with a less destructive approach than SqueezeNet. In addition, we are now using cheaper operations because depthwise separable convolutions can be accelerated in mobile hardware.

ReLU 6

We have used a ReLU activation function for our models so far, which looks like this:

```
relu(x) = max(features, 0)
```

When building models which we know we are going to quantize, it is valuable to instead limit the output of the ReLU layers and by extension force the network to work with smaller numbers from the start. So, we simply introduce a ceiling function for our ReLU activation like so:

```
relu6(x) = min(max(features, 0), 6)
```

Now, we can simplify our output logic to take advantage of this reduced space.

Example of the reduction in MACs with this approach

> Benchmark Analysis of Representative Deep Neural Network Architectures

```
> https://arxiv.org/abs/1810.00736
```

This paper has a nice graph on page 3 visualizing the differences between these networks. Conceptually, we have a slightly larger network than SqueezeNet, but we have a top 1 accuracy comparable to ResNet 18

(a smaller version of ResNet 34 from earlier). Look at VGG16 vs. MobileNet v2 if you want to know where we're going next.

Code

This network uses many more types of layers than our SqueezeNet approach, but produces significantly better results because they are cheaper computationally. This is something we will see repeatedly going forward.

```
```

```
import TensorFlow

public struct ConvBlock: Layer {
  public var zeroPad = ZeroPadding2D<Float>(padding:
  ((0, 1), (0, 1)))
  public var conv: Conv2D<Float>
  public var batchNorm: BatchNorm<Float>

  public init(filterCount: Int, strides: (Int, Int)) {
    conv = Conv2D<Float>(
      filterShape: (3, 3, 3, filterCount),
      strides: strides,
      padding: .valid)
    batchNorm = BatchNorm<Float>(featureCount: filterCount)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = input.sequenced(through: zeroPad, conv,
    batchNorm)
    return relu6(convolved)
  }
}
```

```
public struct DepthwiseConvBlock: Layer {
  @noDerivative let strides: (Int, Int)
  @noDerivative public let zeroPad =
  ZeroPadding2D<Float>(padding: ((0, 1), (0, 1)))

  public var dConv: DepthwiseConv2D<Float>
  public var batchNorm1: BatchNorm<Float>
  public var conv: Conv2D<Float>
  public var batchNorm2: BatchNorm<Float>

  public init(
    filterCount: Int, pointwiseFilterCount: Int,
    strides: (Int, Int)
  ) {
    self.strides = strides
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filterCount, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    batchNorm1 = BatchNorm<Float>(
      featureCount: filterCount)
    conv = Conv2D<Float>(
      filterShape: (
        1, 1, filterCount,
        pointwiseFilterCount
      ),
      strides: (1, 1),
      padding: .same)
    batchNorm2 = BatchNorm<Float>(featureCount:
    pointwiseFilterCount)
  }
```

```
  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    var convolved1: Tensor<Float>
    if self.strides == (1, 1) {
      convolved1 = input.sequenced(through: dConv, batchNorm1)
    } else {
      convolved1 = input.sequenced(through: zeroPad, dConv,
      batchNorm1)
    }
    let convolved2 = relu6(convolved1)
    let convolved3 = relu6(convolved2.sequenced(through: conv,
    batchNorm2))
    return convolved3
  }
}

public struct MobileNetV1: Layer {
  @noDerivative let classCount: Int
  @noDerivative let scaledFilterShape: Int

  public var convBlock1: ConvBlock
  public var dConvBlock1: DepthwiseConvBlock
  public var dConvBlock2: DepthwiseConvBlock
  public var dConvBlock3: DepthwiseConvBlock
  public var dConvBlock4: DepthwiseConvBlock
  public var dConvBlock5: DepthwiseConvBlock
  public var dConvBlock6: DepthwiseConvBlock
  public var dConvBlock7: DepthwiseConvBlock
  public var dConvBlock8: DepthwiseConvBlock
  public var dConvBlock9: DepthwiseConvBlock
  public var dConvBlock10: DepthwiseConvBlock
  public var dConvBlock11: DepthwiseConvBlock
  public var dConvBlock12: DepthwiseConvBlock
```

```
public var dConvBlock13: DepthwiseConvBlock
public var avgPool = GlobalAvgPool2D<Float>()
public var dropoutLayer: Dropout<Float>
public var outputConv: Conv2D<Float>

public init(
  classCount: Int = 10,
  dropout: Double = 0.001
) {
  self.classCount = classCount
  scaledFilterShape = Int(1024.0 * 1.0)

  convBlock1 = ConvBlock(filterCount: 32, strides: (2, 2))
  dConvBlock1 = DepthwiseConvBlock(
    filterCount: 32,
    pointwiseFilterCount: 64,
    strides: (1, 1))
  dConvBlock2 = DepthwiseConvBlock(
    filterCount: 64,
    pointwiseFilterCount: 128,
    strides: (2, 2))
  dConvBlock3 = DepthwiseConvBlock(
    filterCount: 128,
    pointwiseFilterCount: 128,
    strides: (1, 1))
  dConvBlock4 = DepthwiseConvBlock(
    filterCount: 128,
    pointwiseFilterCount: 256,
    strides: (2, 2))
  dConvBlock5 = DepthwiseConvBlock(
    filterCount: 256,
    pointwiseFilterCount: 256,
    strides: (1, 1))
```

```
dConvBlock6 = DepthwiseConvBlock(
  filterCount: 256,
  pointwiseFilterCount: 512,
  strides: (2, 2))
dConvBlock7 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock8 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock9 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock10 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock11 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 512,
  strides: (1, 1))
dConvBlock12 = DepthwiseConvBlock(
  filterCount: 512,
  pointwiseFilterCount: 1024,
  strides: (2, 2))
dConvBlock13 = DepthwiseConvBlock(
  filterCount: 1024,
  pointwiseFilterCount: 1024,
  strides: (1, 1))
```

```
    dropoutLayer = Dropout<Float>(probability: dropout)
    outputConv = Conv2D<Float>(
      filterShape: (1, 1, scaledFilterShape, classCount),
      strides: (1, 1),
      padding: .same)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = input.sequenced(
      through: convBlock1, dConvBlock1,
      dConvBlock2, dConvBlock3, dConvBlock4)
    let convolved2 = convolved.sequenced(
      through: dConvBlock5, dConvBlock6,
      dConvBlock7, dConvBlock8, dConvBlock9)
    let convolved3 = convolved2.sequenced(
      through: dConvBlock10, dConvBlock11, dConvBlock12,
      dConvBlock13, avgPool
    ).reshaped(to: [
      input.shape[0], 1, 1, scaledFilterShape,
    ])
    let convolved4 = convolved3.sequenced(through:
    dropoutLayer, outputConv)
    return convolved4.reshaped(to: [input.shape[0], classCount])
  }
}
```

Results

Our results are on par with our Resnet 50 network from before, but this network is smaller in general and can be evaluated much, much faster at runtime and so is a solid improvement for mobile devices.

```
Starting training...
[Epoch 1 ] Accuracy: 50/500  (0.1)   Loss: 2.5804458
[Epoch 2 ] Accuracy: 262/500 (0.524) Loss: 1.5034955
[Epoch 3 ] Accuracy: 224/500 (0.448) Loss: 1.928577
[Epoch 4 ] Accuracy: 286/500 (0.572) Loss: 1.4074985
[Epoch 5 ] Accuracy: 306/500 (0.612) Loss: 1.3206513
[Epoch 6 ] Accuracy: 334/500 (0.668) Loss: 1.0112444
[Epoch 7 ] Accuracy: 362/500 (0.724) Loss: 0.8360394
[Epoch 8 ] Accuracy: 343/500 (0.686) Loss: 1.0489439
[Epoch 9 ] Accuracy: 317/500 (0.634) Loss: 1.6159635
[Epoch 10] Accuracy: 338/500 (0.676) Loss: 1.0420185
[Epoch 11] Accuracy: 354/500 (0.708) Loss: 1.0034739
[Epoch 12] Accuracy: 358/500 (0.716) Loss: 0.9746185
[Epoch 13] Accuracy: 344/500 (0.688) Loss: 1.152486
[Epoch 14] Accuracy: 365/500 (0.73)  Loss: 0.96197647
[Epoch 15] Accuracy: 353/500 (0.706) Loss: 1.2438473
[Epoch 16] Accuracy: 367/500 (0.734) Loss: 1.044013
[Epoch 17] Accuracy: 365/500 (0.73)  Loss: 1.1098087
[Epoch 18] Accuracy: 352/500 (0.704) Loss: 1.3609929
[Epoch 19] Accuracy: 376/500 (0.752) Loss: 1.2861694
[Epoch 20] Accuracy: 376/500 (0.752) Loss: 1.0280938
[Epoch 21] Accuracy: 369/500 (0.738) Loss: 1.1655327
[Epoch 22] Accuracy: 369/500 (0.738) Loss: 1.1702954
[Epoch 23] Accuracy: 363/500 (0.726) Loss: 1.151112
[Epoch 24] Accuracy: 378/500 (0.756) Loss: 0.94088197
[Epoch 25] Accuracy: 386/500 (0.772) Loss: 1.03443
```

```
[Epoch 26] Accuracy: 379/500 (0.758) Loss: 1.1582794
[Epoch 27] Accuracy: 384/500 (0.768) Loss: 1.1210178
[Epoch 28] Accuracy: 377/500 (0.754) Loss: 1.136668
[Epoch 29] Accuracy: 382/500 (0.764) Loss: 1.2300915
[Epoch 30] Accuracy: 381/500 (0.762) Loss: 1.0231776
```

Recap

We've looked at MobileNet, an important computer vision network from 2017 that makes heavy use of depthwise separable convolutions in order to produce results on par with ResNet 18 (a smaller version of our ResNet 34 network) at a significantly reduced size and computational budget. We can run this on a phone at near real time (e.g., ~50ms/prediction speed) with hardware of the day. Next, let's look at how we can tweak our MobileNet network slightly to produce even better results.