CHAPTER 9

# MobileNet v2

In this chapter, we'll look at how we can modify our MobileNet v1 approach to produce MobileNet v2, which is slightly more accurate and computationally cheaper. This network came out in 2018 and delivered an improved version of the v1 architecture.

> MobileNetV2: Inverted Residuals and Linear Bottlenecks

```
> https://arxiv.org/abs/1801.04381
```

The key concepts the Google team introduced in this paper were inverted residual blocks and linear bottleneck layers, so let's look at how they work.

## Inverted residual blocks

In our ResNet 50 bottleneck blocks from before, we pass our input layer through a 1x1 convolution in our initial layer of each group, which reduces the data at this point. After passing the data through an expensive 3x3 convolution, we then use a 1x1 convolution to expand the number of filters.

In an inverted residual block, which is what MobileNet v2 uses, we instead use an initial 1x1 convolution to increase our network depth, then apply our depthwise convolution from MobileNet v1, and then use a 1x1 convolution to squeeze our network back down at the end.


B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_9

## Inverted skip connections

In our ResNet networks, we applied our skip connection (e.g., the add operation) to pass data from our input to our output layer. MobileNet v2 does this in a subtly different way by only performing this operation on blocks where the number of inputs and outputs are the same (e.g., not the first block of each stack but between the remaining ones). What this means is that this network is not as strongly connected as the original ResNet and less data passes through, but on the flip side, it is significantly cheaper to evaluate.

## Linear bottleneck layers

The next subtle tweak is tied to our inverted skip connections. In the original ResNet network, we apply a ReLU activation function to the combined output of our bottleneck layer and input. Interestingly enough, the MobileNet *v2* authors found that we can eliminate this activation function and improve the network's performance. This activation then simply becomes a linear function, so they call the result a linear bottleneck function.

## Code

For this network, we'll use our block operator to generate the sublayers (e.g., InvertedBottleneckBlockStack). Conceptually, the major difference from our MobileNet v1 architecture is the addition of a depthwise conv to our residual blocks and our inverted method of calculating our gradients each pass.

```
```

```
import TensorFlow

public struct InitialInvertedBottleneckBlock: Layer {
  public var dConv: DepthwiseConv2D<Float>
  public var batchNormDConv: BatchNorm<Float>
  public var conv2: Conv2D<Float>
  public var batchNormConv: BatchNorm<Float>

  public init(filters: (Int, Int)) {
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filters.0, 1),
      strides: (1, 1),
      padding: .same)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, filters.0, filters.1),
      strides: (1, 1),
      padding: .same)
    batchNormDConv = BatchNorm(featureCount: filters.0)
    batchNormConv = BatchNorm(featureCount: filters.1)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let depthwise = relu6(batchNormDConv(dConv(input)))
    return batchNormConv(conv2(depthwise))
  }
}

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

```
  public var conv1: Conv2D<Float>
  public var batchNormConv1: BatchNorm<Float>
  public var dConv: DepthwiseConv2D<Float>
  public var batchNormDConv: BatchNorm<Float>
  public var conv2: Conv2D<Float>
  public var batchNormConv2: BatchNorm<Float>

  public init(
    filters: (Int, Int),
    depthMultiplier: Int = 6,
    strides: (Int, Int) = (1, 1)
  ) {
    self.strides = strides
    self.addResLayer = filters.0 == filters.1 && strides == (1, 1)

    let hiddenDimension = filters.0 * depthMultiplier
    conv1 = Conv2D<Float>(
      filterShape: (1, 1, filters.0, hiddenDimension),
      strides: (1, 1),
      padding: .same)
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, hiddenDimension, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, hiddenDimension, filters.1),
      strides: (1, 1),
      padding: .same)
    batchNormConv1 = BatchNorm(featureCount: hiddenDimension)
    batchNormDConv = BatchNorm(featureCount: hiddenDimension)
    batchNormConv2 = BatchNorm(featureCount: filters.1)
  }
```

```
  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let pointwise = relu6(batchNormConv1(conv1(input)))
    var depthwise: Tensor<Float>
    if self.strides == (1, 1) {
      depthwise = relu6(batchNormDConv(dConv(pointwise)))
    } else {
     depthwise = relu6(batchNormDConv(dConv(zeroPad(pointwise))))
    }
    let pointwiseLinear = batchNormConv2(conv2(depthwise))

    if self.addResLayer {
      return input + pointwiseLinear
    } else {
      return pointwiseLinear
    }
  }
}

public struct InvertedBottleneckBlockStack: Layer {
  var blocks: [InvertedBottleneckBlock] = []

  public init(
    filters: (Int, Int),
    blockCount: Int,
    initialStrides: (Int, Int) = (2, 2)
  ) {
    self.blocks = [
      InvertedBottleneckBlock(
        filters: (filters.0, filters.1),
        strides: initialStrides)
    ]
```

```
    for _ in 1..<blockCount {
      self.blocks.append(
        InvertedBottleneckBlock(
          filters: (filters.1, filters.1))
      )
    }
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    return blocks.differentiableReduce(input) { $1($0) }
  }
}

public struct MobileNetV2: Layer {
  @noDerivative public let zeroPad = ZeroPadding2D<Float>
  (padding: ((0, 1), (0, 1)))
  public var inputConv: Conv2D<Float>
  public var inputConvBatchNorm: BatchNorm<Float>
  public var initialInvertedBottleneck: InitialInverted
  BottleneckBlock

  public var residualBlockStack1: InvertedBottleneckBlockStack
  public var residualBlockStack2: InvertedBottleneckBlockStack
  public var residualBlockStack3: InvertedBottleneckBlockStack
  public var residualBlockStack4: InvertedBottleneckBlockStack
  public var residualBlockStack5: InvertedBottleneckBlockStack

  public var invertedBottleneckBlock16: InvertedBottleneckBlock

  public var outputConv: Conv2D<Float>
  public var outputConvBatchNorm: BatchNorm<Float>
  public var avgPool = GlobalAvgPool2D<Float>()
  public var outputClassifier: Dense<Float>
```

```
public init(classCount: Int = 10) {
  inputConv = Conv2D<Float>(
    filterShape: (3, 3, 3, 32),
    strides: (2, 2),
    padding: .valid)
  inputConvBatchNorm = BatchNorm(
    featureCount: 32)

  initialInvertedBottleneck = InitialInvertedBottleneckBlock(
    filters: (32, 16))

  residualBlockStack1 = InvertedBottleneckBlockStack(filters:
  (16, 24), blockCount: 2)
  residualBlockStack2 = InvertedBottleneckBlockStack(filters:
  (24, 32), blockCount: 3)
  residualBlockStack3 = InvertedBottleneckBlockStack(filters:
  (32, 64), blockCount: 4)
  residualBlockStack4 = InvertedBottleneckBlockStack(
    filters: (64, 96), blockCount: 3,
    initialStrides: (1, 1))
  residualBlockStack5 = InvertedBottleneckBlockStack(filters:
  (96, 160), blockCount: 3)

  invertedBottleneckBlock16 = InvertedBottleneckBlock(filters:
  (160, 320))

  outputConv = Conv2D<Float>(
    filterShape: (1, 1, 320, 1280),
    strides: (1, 1),
    padding: .same)
  outputConvBatchNorm = BatchNorm(featureCount: 1280)

  outputClassifier = Dense(inputSize: 1280, outputSize:
  classCount)
}
```

```
  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = relu6(input.sequenced(through: zeroPad,
    inputConv, inputConvBatchNorm))
    let initialConv = initialInvertedBottleneck(convolved)
    let backbone = initialConv.sequenced(
      through: residualBlockStack1, residualBlockStack2,
      residualBlockStack3,
      residualBlockStack4, residualBlockStack5)
    let output = relu6(outputConvBatchNorm(outputConv(inverted
    BottleneckBlock16(backbone))))
    return output.sequenced(through: avgPool, outputClassifier)
  }
}
```

## Results

This network performs better than our MobileNet v1 architecture using the same training loop and basic setup.

```
Starting training...
[Epoch 1 ] Accuracy: 50/500  (0.1)   Loss: 3.0107288
[Epoch 2 ] Accuracy: 276/500 (0.552) Loss: 1.4318728
[Epoch 3 ] Accuracy: 324/500 (0.648) Loss: 1.2038971
[Epoch 4 ] Accuracy: 337/500 (0.674) Loss: 1.1165649
[Epoch 5 ] Accuracy: 347/500 (0.694) Loss: 0.9973701
[Epoch 6 ] Accuracy: 363/500 (0.726) Loss: 0.9118728
[Epoch 7 ] Accuracy: 310/500 (0.62)  Loss: 1.2533528
[Epoch 8 ] Accuracy: 372/500 (0.744) Loss: 0.797099
[Epoch 9 ] Accuracy: 368/500 (0.736) Loss: 0.8001915
[Epoch 10] Accuracy: 350/500 (0.7)   Loss: 1.1580966
```

```
[Epoch 11] Accuracy: 372/500 (0.744) Loss: 0.84680176
[Epoch 12] Accuracy: 358/500 (0.716) Loss: 1.1446275
[Epoch 13] Accuracy: 388/500 (0.776) Loss: 0.90346915
[Epoch 14] Accuracy: 394/500 (0.788) Loss: 0.82173353
[Epoch 15] Accuracy: 365/500 (0.73)  Loss: 0.9974839
[Epoch 16] Accuracy: 359/500 (0.718) Loss: 1.2463648
[Epoch 17] Accuracy: 333/500 (0.666) Loss: 1.5243211
[Epoch 18] Accuracy: 390/500 (0.78)  Loss: 0.8723967
[Epoch 19] Accuracy: 383/500 (0.766) Loss: 1.0088551
[Epoch 20] Accuracy: 372/500 (0.744) Loss: 1.1002765
[Epoch 21] Accuracy: 392/500 (0.784) Loss: 0.9233314
[Epoch 22] Accuracy: 395/500 (0.79)  Loss: 0.9421617
[Epoch 23] Accuracy: 367/500 (0.734) Loss: 1.1607682
[Epoch 24] Accuracy: 372/500 (0.744) Loss: 1.1685853
[Epoch 25] Accuracy: 375/500 (0.75)  Loss: 1.1443601
[Epoch 26] Accuracy: 389/500 (0.778) Loss: 1.0197723
[Epoch 27] Accuracy: 392/500 (0.784) Loss: 1.0215062
[Epoch 28] Accuracy: 387/500 (0.774) Loss: 1.1886547
[Epoch 29] Accuracy: 400/500 (0.8)   Loss: 0.9691738
[Epoch 30] Accuracy: 383/500 (0.766) Loss: 1.1193326
```

# Recap

We've looked at MobileNet v2, a state-of-the-art network from 2018 for performing image recognition on a device with limited computational capacity (e.g., a phone). Next, let's look at how with some reinforcement learning we can get even better results!