CHAPTER 10

EfficientNet

EfficientNet is the current state of the art for image recognition. I doubt this will remain the case forever, but I do not believe it is going to be replaced easily. It is the product of many years' worth of research in this field and combines multiple different techniques together. What is interesting to me in particular about this network is that we are seeing techniques developed for mobile devices having applications in the larger computer vision community. Or rather, research on building models for resource-constrained devices is driving progress in the cloud, while historically the reverse has been the case.

At a high level, EfficientNet was created using the inverted residual blocks of MobileNetV2 as an architecture type combined with the MnasNet search strategy. These smaller blocks weren't around when MnasNet was created, and by using them the researchers were able to find a significantly improved set of networks. In addition, they were able to find a reliably scalable set of heuristics for constructing larger networks given an initial starting point, which was the key limitation of the evolutionary strategies we looked at earlier in the chapter.

In addition, the researchers added two important concepts from other papers: the swish activation function and SE (Squeeze and Excitation) blocks.

B. Koonce, *Convolutional Neural Networks with Swift for Tensorflow*,
https://doi.org/10.1007/978-1-4842-6168-2_10

Swish

The ReLU function, which we introduced way back in Chapter 1, isn't the only activation function that's been tried. They're just extremely simple to implement and extremely performant, both at the mathematical and hardware levels, and so have stood the test of time, so to speak.

> Searching for Activation Functions

```
> https://arxiv.org/abs/1710.05941
```

This paper explores a variety of alternative activation functions and found that the swish function (discovered in this paper) produces even better results when used in networks.

Swish is defined mathematically as

```
```f(x)=x·sigmoid(βx)```
```sigmoid(y)=1/(1+e^(-y))```
```

Combining these two together has the interesting property of going slightly negative around zero, whereas most traditional activation functions are always >= zero. Conceptually, this produces a smoother gradient space and by extension makes it easier for the network to learn the underlying data distribution, which translates into improved accuracy. Swish has been shown to improve performance in other reinforcement learning problem scenarios, and so it is an important activation function for you to know in general.

There are some limitations to swish from an implementation standpoint, namely, that it uses more memory than a simple ReLU. We will come back to this in the next chapter.

SE (Squeeze + Excitation) block

This is an interesting paper from the Oxford Visual Geometry Group (e.g., the people who produced VGG) from 2017, which won the ImageNet competition that year.

> Squeeze-and-Excitation Networks

```
> https://arxiv.org/abs/1709.01507
```

Conceptually, we might think of what our neural networks are actually learning as a collection of features. Then, when the network sees a picture that matches a particular collection of features, we train it to fire a particular neuron. To take things to the next level and avoid random activations, ideally for each feature map, we could define a sort of master neuron that decides whether or not the feature should activate as a whole.

This is loosely the idea of Squeeze and Excitation blocks. By taking the feature input and reducing it dramatically down (to as small as a single pixel in some cases), we allow the network to sort of train each block to teach itself as to whether or not it should fire given a particular input, so to speak. This produces state-of-the-art results, but is also computationally expensive.

EfficientNet uses a simpler variant based around combining two convolutions to produce similar results at a much cheaper cost computationally.

Code

Pay attention to the squeeze and excite blocks and how they are used to boost the results in the convolutional blocks. With this addition, the rest of this backbone is extremely similar to MobileNet v2. Look also at the subtle differences in the parameters to the MBConvBlockStack generator, which we will see much more of in our next chapter.

```
import TensorFlow

struct InitialMBConvBlock: Layer {
  @noDerivative var hiddenDimension: Int
  var dConv: DepthwiseConv2D<Float>
  var batchNormDConv: BatchNorm<Float>
  var seAveragePool = GlobalAvgPool2D<Float>()
  var seReduceConv: Conv2D<Float>
  var seExpandConv: Conv2D<Float>
  var conv2: Conv2D<Float>
  var batchNormConv2: BatchNorm<Float>

  init(filters: (Int, Int), width: Float) {
    let filterMult = filters
    self.hiddenDimension = filterMult.0
    dConv = DepthwiseConv2D<Float>(
      filterShape: (3, 3, filterMult.0, 1),
      strides: (1, 1),
      padding: .same)
    seReduceConv = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, 8),
      strides: (1, 1),
      padding: .same)
    seExpandConv = Conv2D<Float>(
      filterShape: (1, 1, 8, filterMult.0),
      strides: (1, 1),
      padding: .same)
    conv2 = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, filterMult.1),
      strides: (1, 1),
      padding: .same)
```

```
    batchNormDConv = BatchNorm(featureCount: filterMult.0)
    batchNormConv2 = BatchNorm(featureCount: filterMult.1)
  }

  @differentiable
  func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let depthwise = swish(batchNormDConv(dConv(input)))
    let seAvgPoolReshaped = seAveragePool(depthwise).
    reshaped(to: [
      input.shape[0], 1, 1, self.hiddenDimension,
    ])
    let squeezeExcite =
      depthwise
      * sigmoid(seExpandConv(swish(seReduceConv(seAvgPool
      Reshaped))))
    return batchNormConv2(conv2(squeezeExcite))
  }
}

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

  var conv1: Conv2D<Float>
  var batchNormConv1: BatchNorm<Float>
  var dConv: DepthwiseConv2D<Float>
  var batchNormDConv: BatchNorm<Float>
  var seAveragePool = GlobalAvgPool2D<Float>()
  var seReduceConv: Conv2D<Float>
  var seExpandConv: Conv2D<Float>
```

```
  var conv2: Conv2D<Float>
  var batchNormConv2: BatchNorm<Float>

  init(
    filters: (Int, Int),
    width: Float,
    depthMultiplier: Int = 6,
    strides: (Int, Int) = (1, 1),
    kernel: (Int, Int) = (3, 3)
  ) {
    self.strides = strides
    self.addResLayer = filters.0 == filters.1 && strides == (1, 1)

    let filterMult = filters
    self.hiddenDimension = filterMult.0 * depthMultiplier
    let reducedDimension = max(1, Int(filterMult.0 / 4))
    conv1 = Conv2D<Float>(
      filterShape: (1, 1, filterMult.0, hiddenDimension),
      strides: (1, 1),
      padding: .same)
    dConv = DepthwiseConv2D<Float>(
      filterShape: (kernel.0, kernel.1, hiddenDimension, 1),
      strides: strides,
      padding: strides == (1, 1) ? .same : .valid)
    seReduceConv = Conv2D<Float>(
      filterShape: (1, 1, hiddenDimension, reducedDimension),
      strides: (1, 1),
      padding: .same)
    seExpandConv = Conv2D<Float>(
      filterShape: (1, 1, reducedDimension, hiddenDimension),
      strides: (1, 1),
      padding: .same)
```

```
  conv2 = Conv2D<Float>(
    filterShape: (1, 1, hiddenDimension, filterMult.1),
    strides: (1, 1),
    padding: .same)
  batchNormConv1 = BatchNorm(featureCount: hiddenDimension)
  batchNormDConv = BatchNorm(featureCount: hiddenDimension)
  batchNormConv2 = BatchNorm(featureCount: filterMult.1)
}

@differentiable
func forward(_ input: Tensor<Float>) -> Tensor<Float> {
  let piecewise = swish(batchNormConv1(conv1(input)))
  var depthwise: Tensor<Float>
  if self.strides == (1, 1) {
    depthwise = swish(batchNormDConv(dConv(piecewise)))
  } else {
   depthwise = swish(batchNormDConv(dConv(zeroPad(piecewise))))
  }
  let seAvgPoolReshaped = seAveragePool(depthwise).
  reshaped(to: [
    input.shape[0], 1, 1, self.hiddenDimension,
  ])
  let squeezeExcite =
    depthwise
    * sigmoid(seExpandConv(swish(seReduceConv(seAvgPool
    Reshaped))))
  let piecewiseLinear = batchNormConv2(conv2(squeezeExcite))

  if self.addResLayer {
    return input + piecewiseLinear
  } else {
    return piecewiseLinear
```

```
    }
  }
}

struct MBConvBlockStack: Layer {
  var blocks: [MBConvBlock] = []

  init(
    filters: (Int, Int),
    width: Float,
    initialStrides: (Int, Int) = (2, 2),
    kernel: (Int, Int) = (3, 3),
    blockCount: Int,
    depth: Float
  ) {
    let blockMult = blockCount
    self.blocks = [
      MBConvBlock(
        filters: (filters.0, filters.1), width: width,
        strides: initialStrides, kernel: kernel)
    ]
    for _ in 1..<blockMult {
      self.blocks.append(
        MBConvBlock(
          filters: (filters.1, filters.1),
          width: width, kernel: kernel))
    }
  }

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

```
public struct EfficientNet: Layer {
  @noDerivative let zeroPad = ZeroPadding2D<Float>(padding:
  ((0, 1), (0, 1)))
  var inputConv: Conv2D<Float>
  var inputConvBatchNorm: BatchNorm<Float>
  var initialMBConv: InitialMBConvBlock

  var residualBlockStack1: MBConvBlockStack
  var residualBlockStack2: MBConvBlockStack
  var residualBlockStack3: MBConvBlockStack
  var residualBlockStack4: MBConvBlockStack
  var residualBlockStack5: MBConvBlockStack
  var residualBlockStack6: MBConvBlockStack

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

  public init(
    classCount: Int = 1000,
    width: Float = 1.0,
    depth: Float = 1.0,
    resolution: Int = 224,
    dropout: Double = 0.2
  ) {
    inputConv = Conv2D<Float>(
      filterShape: (3, 3, 3, 32),
      strides: (2, 2),
      padding: .valid)
    inputConvBatchNorm = BatchNorm(featureCount: 32)
```

```
initialMBConv = InitialMBConvBlock(filters: (32, 16),
width: width)

residualBlockStack1 = MBConvBlockStack(
  filters: (16, 24), width: width,
  blockCount: 2, depth: depth)
residualBlockStack2 = MBConvBlockStack(
  filters: (24, 40), width: width,
  kernel: (5, 5), blockCount: 2, depth: depth)
residualBlockStack3 = MBConvBlockStack(
  filters: (40, 80), width: width,
  blockCount: 3, depth: depth)
residualBlockStack4 = MBConvBlockStack(
  filters: (80, 112), width: width,
  initialStrides: (1, 1), kernel: (5, 5), blockCount: 3,
  depth: depth)
residualBlockStack5 = MBConvBlockStack(
  filters: (112, 192), width: width,
  kernel: (5, 5), blockCount: 4, depth: depth)
residualBlockStack6 = MBConvBlockStack(
  filters: (192, 320), width: width,
  initialStrides: (1, 1), blockCount: 1, depth: depth)

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

```
    dropoutProb = Dropout<Float>(probability: dropout)
    outputClassifier = Dense(inputSize: 1280, outputSize:
    classCount)
  }

  @differentiable
  public func forward(_ input: Tensor<Float>) -> Tensor<Float> {
    let convolved = swish(input.sequenced(through: zeroPad,
    inputConv, inputConvBatchNorm))
    let initialBlock = initialMBConv(convolved)
    let backbone = initialBlock.sequenced(
      through: residualBlockStack1, residualBlockStack2,
      residualBlockStack3, residualBlockStack4,
      residualBlockStack5, residualBlockStack6)
    let output = swish(backbone.sequenced(through: outputConv,
    outputConvBatchNorm))
    return output.sequenced(through: avgPool, dropoutProb,
    outputClassifier)
  }
}
```

Results

This network trains extremely well, achieving higher accuracy than any of the networks we have seen so far without the addition of any data augmentation techniques.

```
Starting training...
[Epoch 1 ] Accuracy: 50/500  (0.1)   Loss: 3.919964
[Epoch 2 ] Accuracy: 315/500 (0.63)  Loss: 1.1730766
[Epoch 3 ] Accuracy: 340/500 (0.68)  Loss: 1.042603
```

```
[Epoch 4 ] Accuracy: 382/500 (0.764) Loss: 0.7738381
[Epoch 5 ] Accuracy: 358/500 (0.716) Loss: 0.8867168
[Epoch 6 ] Accuracy: 397/500 (0.794) Loss: 0.7941174
[Epoch 7 ] Accuracy: 384/500 (0.768) Loss: 0.7910826
[Epoch 8 ] Accuracy: 375/500 (0.75)  Loss: 0.9265955
[Epoch 9 ] Accuracy: 395/500 (0.79)  Loss: 0.7806258
[Epoch 10] Accuracy: 389/500 (0.778) Loss: 0.8921993
[Epoch 11] Accuracy: 393/500 (0.786) Loss: 0.913636
[Epoch 12] Accuracy: 395/500 (0.79)  Loss: 0.8772738
[Epoch 13] Accuracy: 396/500 (0.792) Loss: 0.819137
[Epoch 14] Accuracy: 393/500 (0.786) Loss: 0.7435807
[Epoch 15] Accuracy: 418/500 (0.836) Loss: 0.6915679
[Epoch 16] Accuracy: 404/500 (0.808) Loss: 0.79288286
[Epoch 17] Accuracy: 405/500 (0.81)  Loss: 0.8690043
[Epoch 18] Accuracy: 404/500 (0.808) Loss: 0.89440507
[Epoch 19] Accuracy: 409/500 (0.818) Loss: 0.85941887
[Epoch 20] Accuracy: 408/500 (0.816) Loss: 0.8633226
[Epoch 21] Accuracy: 404/500 (0.808) Loss: 0.7646436
[Epoch 22] Accuracy: 411/500 (0.822) Loss: 0.8865621
[Epoch 23] Accuracy: 424/500 (0.848) Loss: 0.6812671
[Epoch 24] Accuracy: 402/500 (0.804) Loss: 0.8662841
[Epoch 25] Accuracy: 425/500 (0.85)  Loss: 0.7081538
[Epoch 26] Accuracy: 423/500 (0.846) Loss: 0.7106852
[Epoch 27] Accuracy: 411/500 (0.822) Loss: 0.88567644
[Epoch 28] Accuracy: 410/500 (0.82)  Loss: 0.8509838
[Epoch 29] Accuracy: 409/500 (0.818) Loss: 0.85791296
[Epoch 30] Accuracy: 416/500 (0.832) Loss: 0.76689
```

EfficientNet variants

Once we have this base, we can then use our improved image recognition network to solve other related problems in different fields.

EfficientNet [B1-8]

To play off our exploration of network architecture search functions in the last chapter, the problem with these sort of approaches is that trying to make them larger is difficult because there's not a clear system for scaling them up.

What the authors introduce in this paper is a set of scaling heuristics for their base (B0) network that enables smooth scaling to produce larger and larger networks. Loosely speaking, we might say that each step of a larger network requires a squared amount of compute. Then, we can build large networks consistently given an extremely large amount of computational time to run on. So, here are EfficientNet variants that can be produced by simply scaling up our prior network compared to the various networks we've looked at so far in this book.

RandAugment

> RandAugment: Practical automated data augmentation with a reduced search space

```
> https://arxiv.org/abs/1909.13719
```

We discussed data augmentation briefly in a prior chapter, and I mentioned that it is an area of active research. This paper combines various augmentation techniques (e.g., flipping, rotating, zooming, etc.) with a reinforcement learning algorithm in order to find the optimal (largest effect on accuracy with the smallest set) combination of data augmentation filters when applied to a dataset. Then, they run this learned

algorithm against the ImageNet dataset and then train the EfficientNet variants on top to produce a significantly (~4–5%!) improved set of networks using nothing more than computational time.

Noisy Student

> Self-training with Noisy Student improves ImageNet classification

> https://arxiv.org/abs/1911.04252

Next, **network distillation** is an interesting area of research for building smaller networks. Loosely, we take a large network as a teacher and then train a smaller student network to give similar responses to the larger one given the same inputs and feedback on each answer from the teacher. This has interesting applications in building networks for devices with limited resources once a larger approach has proven itself on a GPU cluster, for example. The large area where this is of interest in natural language processing, where large networks (e.g., BERT) have achieved a state-of-the-art performance but are too large to be used for day-to-day problem solving.

Network distillation has been used to make networks smaller, but can it be used to make them larger? Loosely speaking, this paper takes data augmentation techniques and uses them to make the student's inputs much more noisy, but keeps on requiring the student network to give answers that match the teacher's answers. By iteratively training a larger student on a teacher and then replacing the teacher with the trained student, they were able to build a much larger network that was able to produce even more accurate ImageNet results than even Facebook's 2019 billion-picture Instagram corpus (see https://arxiv.org/abs/1905.00546).

EfficientDet

> EfficientDet: Scalable and Efficient Object Detection

> https://arxiv.org/abs/1911.09070

We've not talked about object detection networks in this book, but the basic idea of many approaches is to use a known good existing image recognition network (called a **backbone**), and then we can add an object detection output layer at the end (called a **head**). This approach enables a nice sort of mix and match style technique where we can use the same head with multiple different backbones or data augmentation strategies to find the best solution for a particular problem.

So, we take EfficientNet, add a custom object detection head, apply our scaling techniques, and voilà, we have an object detection (and with some other tweaks, semantic segmentation) network with state-of-the-art performance.

Recap

We've looked at EfficientNet, the current state of the art for image recognition. We've looked at how we can use the EfficientNet base to build state-of-the-art approaches in related fields. Next, let's look at how we can take these ideas back to the realm of mobile devices.