Implementing a ResNet model from scratch.

A basic description of how ResNet works and a hands-on approach to understanding the state-of-the-art network.

Gracelyn Shi
Jan 17 · 7 min read ★

Source: https://www.quantamagazine.org/brain-computer-interfaces-show-that-neural-networks-learn-by-recycling-20180327/

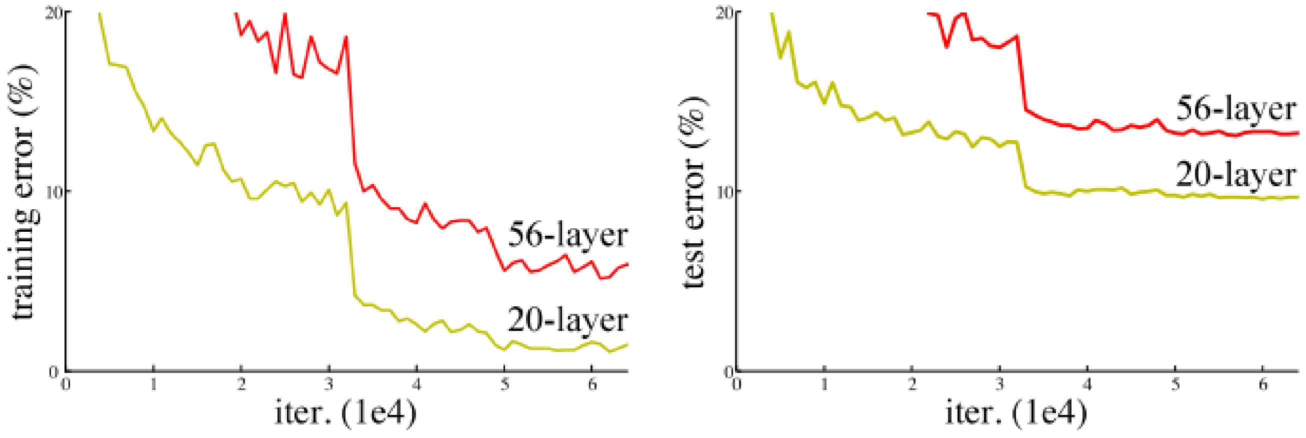
When implementing the **ResNet** architecture in a deep learning project I was working on, it was a huge leap from the basic, simple convolutional neural networks I was used to.

One prominent feature of ResNet is that it utilizes a *micro-architecture* within it's larger *macroarchitecture*: **residual blocks**!

I decided to look into the model myself to gain a better understanding of it, as well as look into why it was so successful at ILSVRC. I implemented the exact same ResNet model class in *Deep Learning for Computer Vision with Python* by Dr. Adrian Rosebrock [1], which followed the ResNet model from the 2015 ResNet academic publication, *Deep Residual Learning for Image Recognition* by He et al. [2].

ResNet

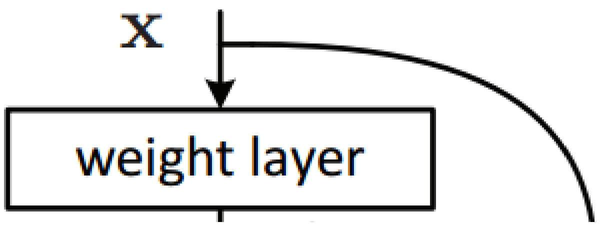
When ResNet was first introduced, it was revolutionary for proving a new solution to a huge problem for deep neural networks at the time: the **vanishing gradient problem**. Although neural networks are universal function approximators, at a certain threshold adding more layers makes training become slower and makes the accuracy saturate.

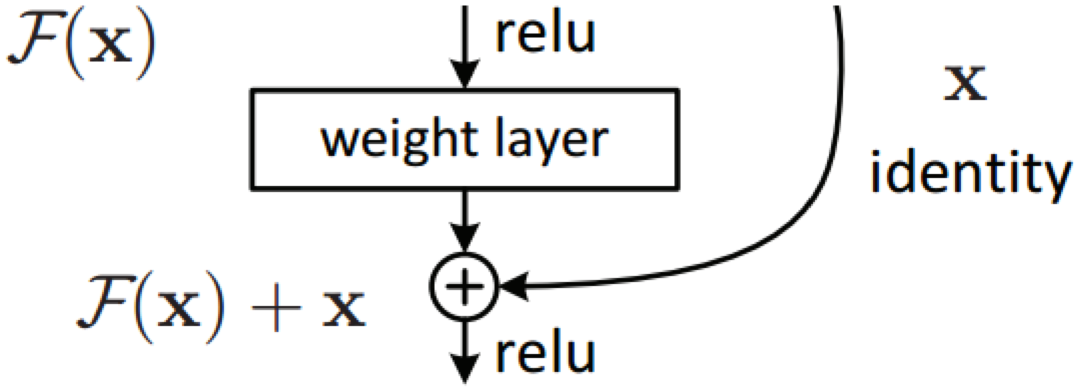


Source: https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035

This is due to the backpropagation of gradients as it goes from the final layers to the earliest ones — multiplying a number between 0 and 1 many times makes it increasingly smaller: thus the gradient begins to “disappear” when reaching the earlier layers. That means the earlier layers are not only slower to train but are also more prone to error. That’s a huge problem as the earliest layers are the building blocks of the whole network — they are responsible for identifying the basic, core features!

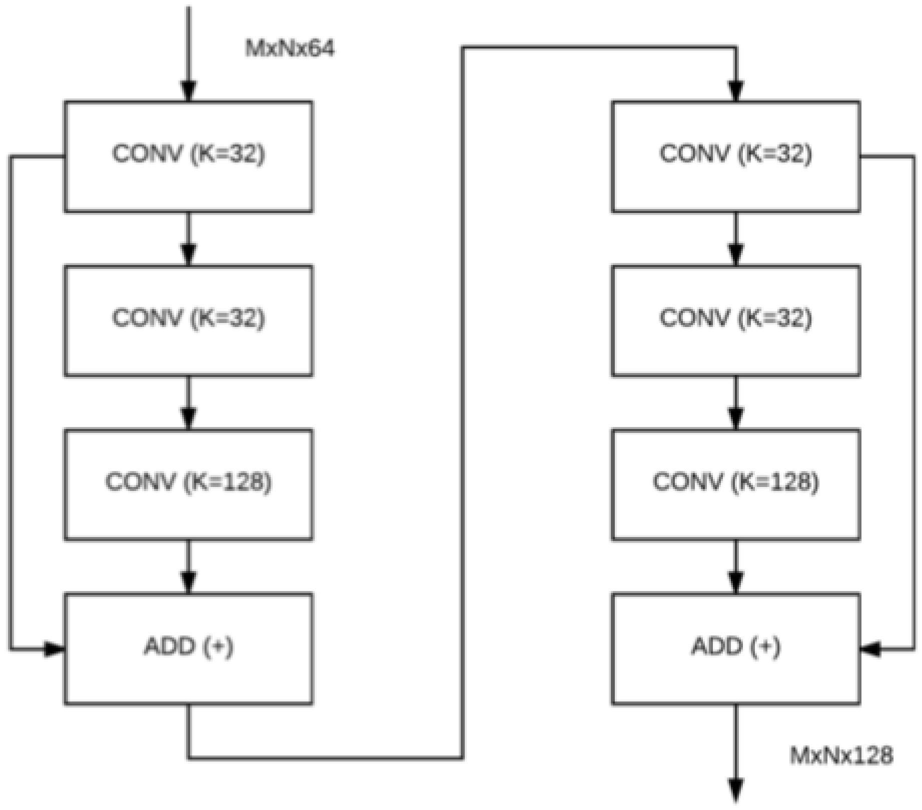
To mitigate this problem, ResNet incorporates **identity shortcut connections** which essentially skip the training of one or more layers — creating a **residual block**.




$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

A single residual block; the original one proposed by He et al. Source: [1]

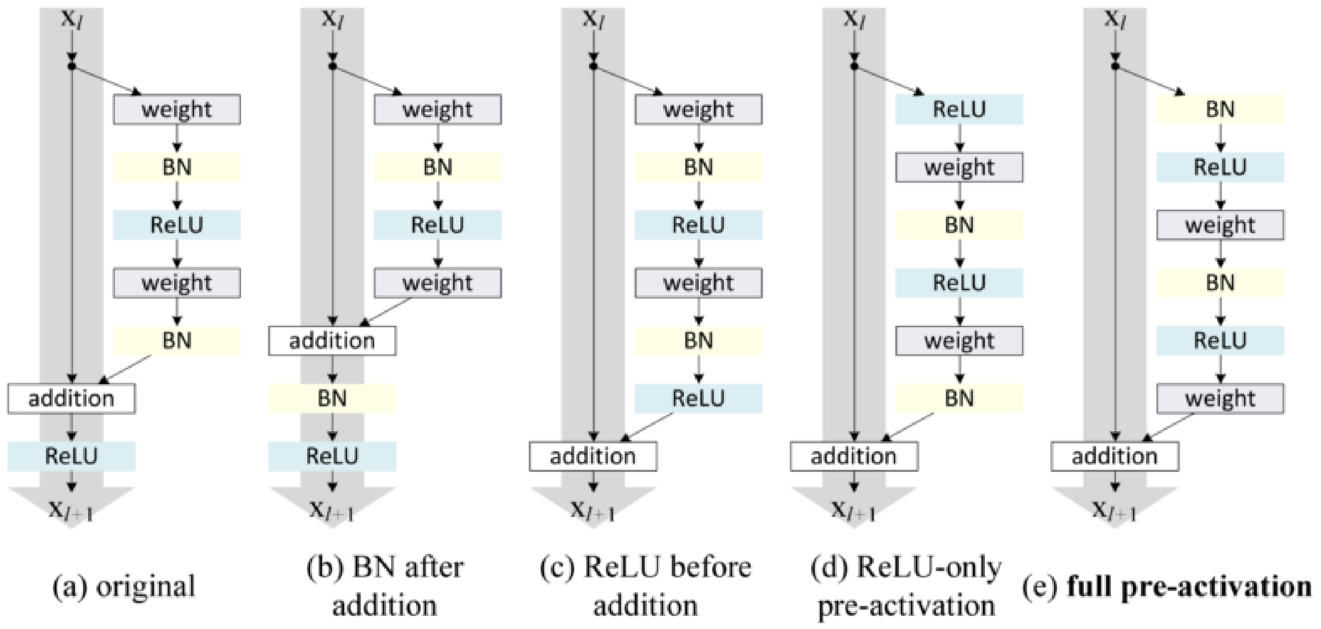
The authors then proposed an “optimized” residual block, adding an extension called a **bottleneck.** It would reduce the dimensionality in the first two CONV layers (1/4 of the filters learned in the final CONV layer) and then increase again during the final CONV layer. Here are two residual modules stacked on top of each other.



Source: Deep Learning for Computer Vision using Python: Practitioner Bundle [1]

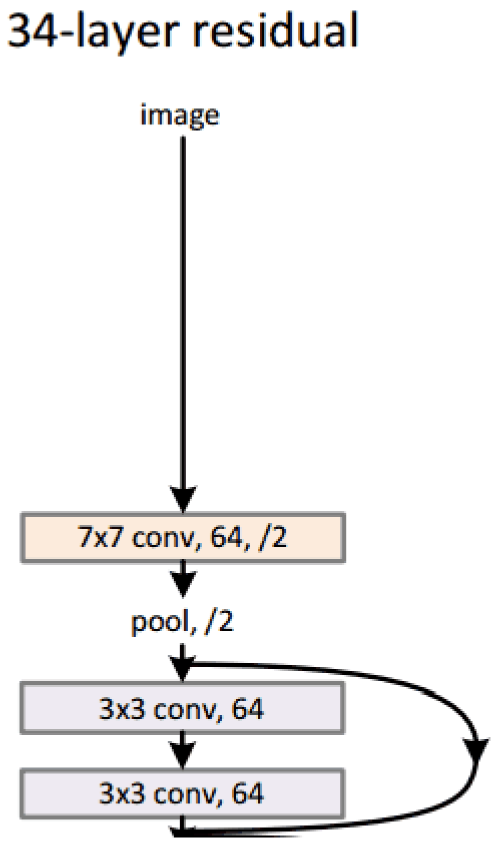
Finally, He et al. published a second paper on the residual module called *Identity Mappings in Deep Residual Networks* which provided an even better version of the residual block: the **pre-activation residual model**. This allows the gradients to propagate through the shortcut connections to any of the earlier layers without hindrance.

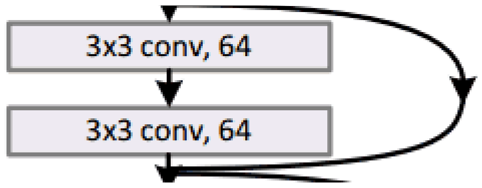
Instead of starting with a convolution (*weight*), we start with a series of (*BN => RELU => CONV*) * N layers (assuming bottleneck is being used). Then, the residual module outputs the *addition* operation that's fed into the next residual module in the network (since residual modules are stacked on top of each other).



(a) original bottleneck residual module. (e) full pre-activation residual module. Called pre-activation because BN and ReLU layers occur before the convolutions. Source: [2]

The overall network architecture looked like this, and our model will be similar to it.





Source: [2]

Let's start coding the actual network in Python. This specific implementation was inspired by both He et al. in their Caffe distribution and the mxnet implementation from Wei Wu.

We're going to write it as a class (*ResNet*) so we can call on it later while training a deep learning model.

```python
# import the necessary packages
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import AveragePooling2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.convolutional import ZeroPadding2D
from keras.layers.core import Activation
from keras.layers.core import Dense
from keras.layers import Flatten
from keras.layers import Input
from keras.models import Model
from keras.layers import add
from keras.regularizers import l2
from keras import backend as K

class ResNet:
        @staticmethod
        def residual_module(data, K, stride, chanDim, red=False,
                reg=0.0001, bnEps=2e-5, bnMom=0.9):
```

We begin with our standard CNN imports, and then start building our *residual_module* function. Take a look at the parameters:

- *data*: input to the residual module
- *K*: number of filters that will be learned by the final CONV layer (the first two CONV layers will learn K/4 filters)
- *stride*: controls the stride of the convolution (will help us reduce spatial dimensions without using max pooling)

- *chanDim*: defines the axis which will perform batch normalization
- *red* (i.e. reduce) will control whether we are reducing spatial dimensions (True) or not (False) as not all residual modules will reduce dimensions of our spatial volume
- *reg:* applies regularization strength for all CONV layers in the residual module
- *bnEps:* controls the Ɛ responsible for avoiding “division by zero” errors when normalizing inputs
- *bnMom:* controls the momentum for the moving average

Now let’s look at the rest of the function.

```
1           # the shortcut branch of the ResNet module should be
2           # initialize as the input (identity) data
3           shortcut = data
4
5           # the first block of the ResNet module are the 1x1 CONVs
6           bn1 = BatchNormalization(axis=chanDim, epsilon=bnEps, momentum=bnMom)(data)
7           act1 = Activation("relu")(bn1)
8           conv1 = Conv2D(int(K * 0.25), (1, 1), use_bias=False,
9                   kernel_regularizer=l2(reg))(act1)
```

First, we initialize the (identity) shortcut (connection), which is really just a reference to the input data. At the end of the residual module, we simply add the shortcut to the output of our pre-activation/bottleneck branch (Line 3).

On Lines 6–9, the first block of the ResNet module follows a BN ==> RELU ==> CONV ==> pattern. The CONV layer utilises 1x1 convolutions by *K/4* filters. Notice that the bias term is turned off for the CONV layer, as the biases are already in the following BN layers so there’s no need for a second bias term.

```
1           # the second block of the ResNet module are the 3x3 CONVs
2           bn2 = BatchNormalization(axis=chanDim, epsilon=bnEps, momentum=bnMom)(conv1)
3           act2 = Activation("relu")(bn2)
4           conv2 = Conv2D(int(K * 0.25), (3, 3), strides=stride, padding="same", use_bias=F
5                   kernel_regularizer=l2(reg)(act2)
```

resnet3 hosted with ❤ by **GitHub** view raw

As per the bottleneck, the second CONV layer learns *K/4* filters that are 3 x 3.

```
        # the third block of the ResNet module is another set of 1x1 CONVs
        bn3 = BatchNormalization(axis=chanDim, epsilon=bnEps,momentum=bnMom)(conv2)
        act3 = Activation("relu")(bn3)
        conv3 = Conv2D(K, (1, 1), use_bias=False, kernel_regularizer=l2(reg))(act3)
```

resnet4 hosted with ❤ by GitHub view raw

The final block will increase dimensionality once again, applying *K* filters with the dimensions 1 x 1.

To avoid applying max pooling, we need to check if reducing spatial dimensions is necessary.

```
1          # if we are to reduce the spatial size, apply a CONV layer to the shortcut
2          if red:
3             shortcut = Conv2D(K, (1, 1), strides=stride, use_bias=False,
4                             kernel_regularizer=l2(reg))(act1)
5
6          # add together the shortcut and the final CONV
7          x = add([conv3, shortcut])
8
9          # return the addition as the output of the ResNet module
10         return x
```

If we are commanded to reduce spatial dimensions, a convolutional layer with a stride > 1 will be applied to the shortcut (Lines 2–4).

Finally, we add together the shortcut and the final CONV layer creating the output to our ResNet module (Line 7). We finally have the “building block” to begin constructing our deep residual network.

Let’s start building the *build* method.

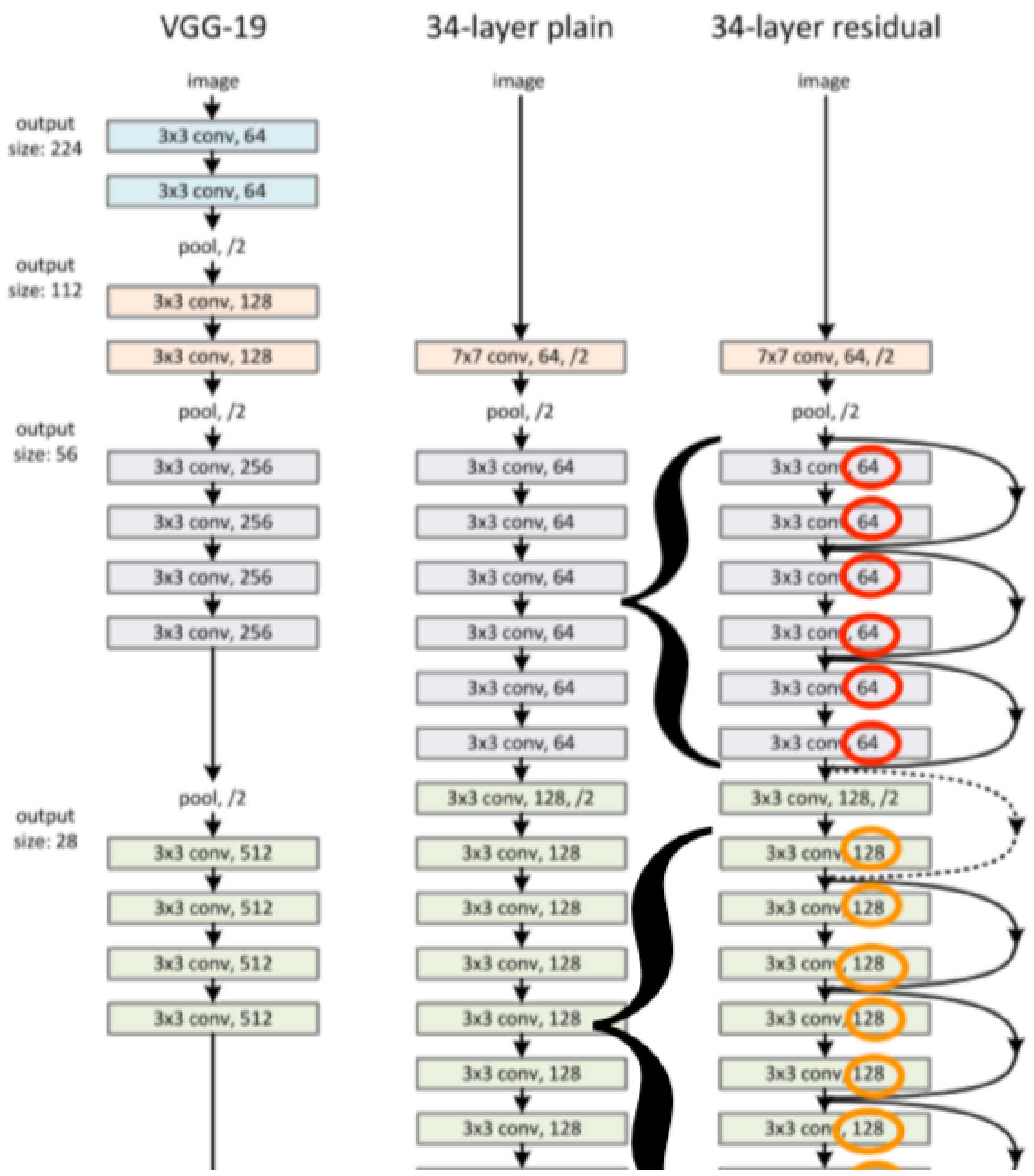
```
        @staticmethod
        def build(width, height, depth, classes, stages, filters,
            reg=0.0001, bnEps=2e-5, bnMom=0.9):
```

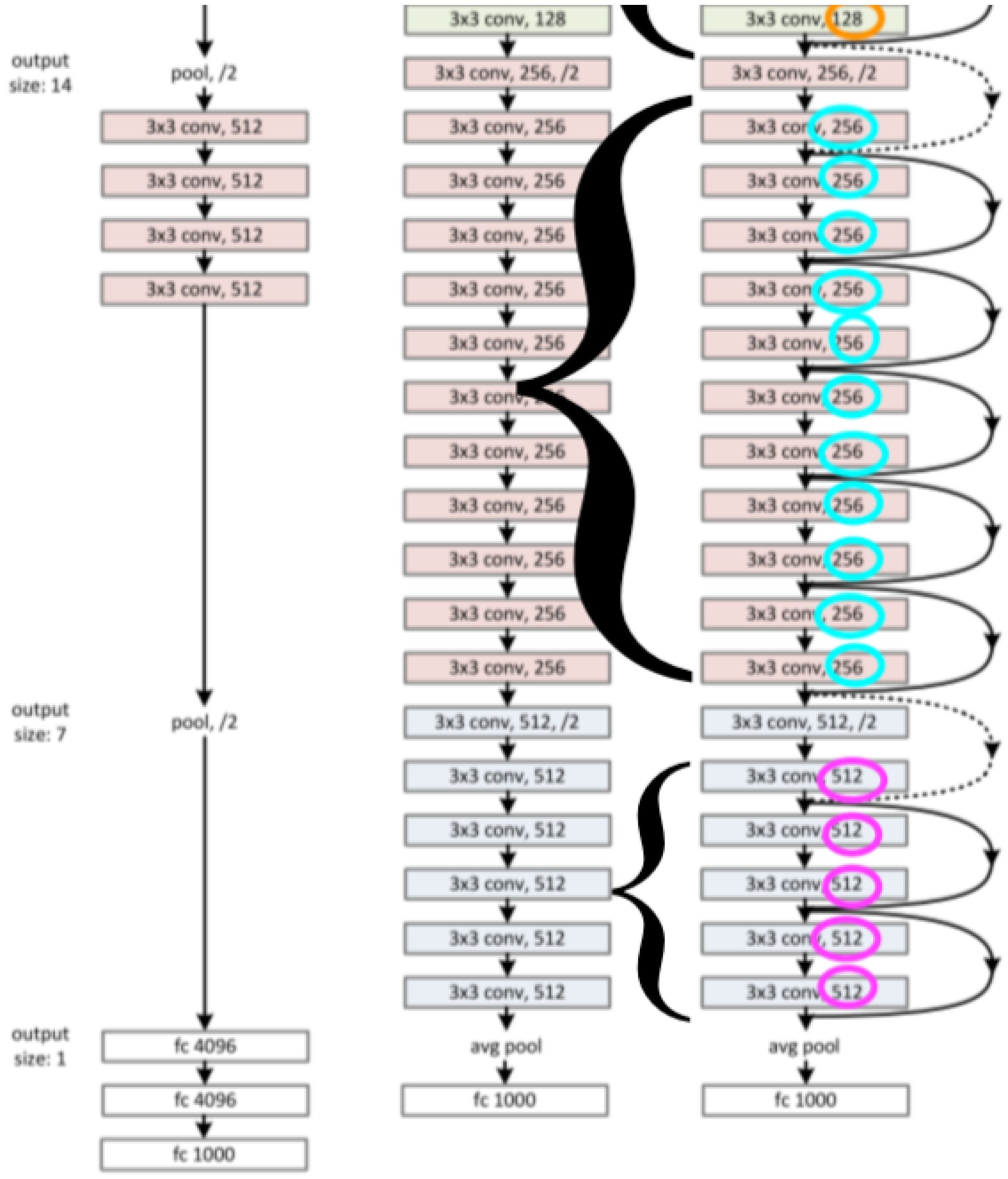
resnet6 hosted with ❤ by GitHub view raw

Take a look at the parameters *stages* and *filters* (which are both lists). In our architecture (shown above) we’re stacking N number of residual modules on top of each other (N = stage value). Each residual module in the same *stage* learns the same number of *filters*. After each stage learns its respective filters, it is followed by dimensionality reduction. We repeat this process until we are ready to apply the average pooling layer and softmax classifier.

Stages and Filters

For example, let's set stages=(3, 4, 6) and filters=(64, 128, 256, 512). The first filter (64) is applied to the only CONV layer not part of the residual module — the first CONV layer in the network. Then, *three* (stage = 3) residual modules are stacked on top of each other — each one will learn *128* filters. The spatial dimensions will be reduced, and then we stack *four* (stage = 4) residual modules on top of each other — each learning 256 filters. Finally, we reduce spatial dimensions again and move on to stacking *six* (stage = 6) residual modules on top of each other, each learning 512 filters.





ResNet architecture. Circled numbers are the filter values, while the brackets show the stacks. Notice how there is a dimensionality reduction after every stage. Unrelated to written example earlier.

Let's go back to building the *build* method.

```
        # initialize the input shape to be "channels last" and the
        # channels dimension itself
        inputShape = (height, width, depth)
        chanDim = -1

        # if we are using "channels first", update the input shape
        # and channels dimension
        if K.image_data_format() == "channels_first":
                inputShape = (depth, height, width)
                chanDim = 1
```

Initialize *inputShape* and *chanDim* based on whether we are using “channels last” or “channels first” ordering (Lines 3–4).

```
1                   # set the input and apply BN
2                   inputs = Input(shape=inputShape)
3                   x = BatchNormalization(axis=chanDim, epsilon=bnEps,
4                           momentum=bnMom)(inputs)
5
6                   # apply CONV => BN => ACT => POOL to reduce spatial size
7                   x = Conv2D(filters[0], (5, 5), use_bias=False,
8                           padding="same", kernel_regularizer=l2(reg))(x)
9                   x = BatchNormalization(axis=chanDim, epsilon=bnEps,
10                          momentum=bnMom)(x)
11                  x = Activation("relu")(x)
12                  x = ZeroPadding2D((1, 1))(x)
13                  x = MaxPooling2D((3, 3), strides=(2, 2))(x)
```

As mentioned above, ResNet uses a BN as the first layer as an added level of normalization to your input (Lines 2–4). Then, we apply a CONV =>, BN => ACT => POOL to reduce the spatial size (Lines 7–13). Now, let’s start stacking residual layers on top of each other.

```
1           # loop over the number of stages
2           for i in range(0, len(stages)):
3                   # initialize the stride, then apply a residual module
4                   # used to reduce the spatial size of the input volume
5                   stride = (1, 1) if i == 0 else (2, 2)
6                   x = ResNet.residual_module(x, filters[i + 1], stride,
7                                   chanDim, red=True, bnEps=bnEps, bnMom=bnMom)
8
9                   # loop over the number of layers in the stage
10                  for j in range(0, stages[i] - 1):
11                          # apply a ResNet module
12                          x = ResNet.residual_module(x, filters[i + 1],
13                                  (1, 1), chanDim, bnEps=bnEps, bnMom=bnMom)
```

To reduce volume size without using pooling layers, we can change the stride of the convolution. The first entry in the stage will have a stride of (1, 1) — signaling the absence of downsampling. Then, every stage after that we’ll apply a residual module with a stride of (2, 2) which will decrease the volume size. This is shown on **Line 5**.

Then, we loop over the number of layers in the current stage (number of residual modules that will be stacked on top of each other) on **Lines 10–13**. We use [i + 1] as the index into filters as the first filter was already used. Once we've stacked stages[i] residual modules on top of each other, we return to the **Lines 6–7** where we decrease the spatial dimensions of the volume and repeat the process.

To avoid dense fully-connected layers, we'll apply average pooling instead to reduce volume size to 1 x 1 x classes:

```
		# apply BN => ACT => POOL
		x = BatchNormalization(axis=chanDim, epsilon=bnEps,
			momentum=bnMom)(x)
		x = Activation("relu")(x)
		x = AveragePooling2D((8, 8))(x)
```

resnet10 hosted with ❤ by GitHub view raw

Finally, we'll create a dense layer for the total number of classes we are going to learn and then apply a softmax activation to generate our final output probabilities!

```
		# softmax classifier
		x = Flatten()(x)
		x = Dense(classes, kernel_regularizer=l2(reg))(x)
		x = Activation("softmax")(x)

		# create the model
		model = Model(inputs, x, name="resnet")

		# return the constructed network architecture
		return model
```

That concludes our build function, and now we have our fully constructed ResNet model! You can call on this class to implement the ResNet architecture in your deep learning projects.

· · ·

If you have any questions, feel free to comment down below or reach out!

- My Linkedin: https://www.linkedin.com/in/gracelyn-shi-963028aa/

- Email me at gracelyn.shi@gmail.com

. . .

References

[1] A. Rosebrock, *Deep Learning for Computer Vision with Python* (2017)

[2] K. He, X. Zhang, S. Ren, and J. Sun, *Deep Residual Learning for Image Recognition (2015),* https://arxiv.org/abs/1512.03385

Machine Learning Neural Networks Resnet Data Science Data