The Basics of ResNet50

In this article, we will explore the fundamentals of ResNet50, a powerful deep learning model, through practical examples using Keras and PyTorch libraries in Python, illustrating its versatile applications.

Mostafa Ibrahim

Created on January 11 | Last edited on January 16

Introduction

During the initial evolution of convolution neural networks (CNN), boosting accuracy meant creating deeper models by stacking layer upon layer. However, a pivotal moment emerged when the quest for deeper models hit a roadblock. As researchers delved into deeper models for better accuracy, the once-promising path led to a puzzling paradox: the deeper the model, the more elusive accuracy became!

Then came ResNet50, like a superhero. Instead of just piling on more layers, ResNet50 had this cool trick called "residual learning" that allowed the models to be deeper and yet, still be accurate.

In this piece, we'll look at this canonical model, why ResNet50 works, and create our own for an image classification task.

Here's what we'll be covering:

Table of Contents

What Is ResNet50?

ResNet50 is a deep learning model launched in 2015 by Microsoft Research for the purpose of visual recognition. ResNet is short for Residual Networks while the ‘50’ just means that the model is 50 layers deep.

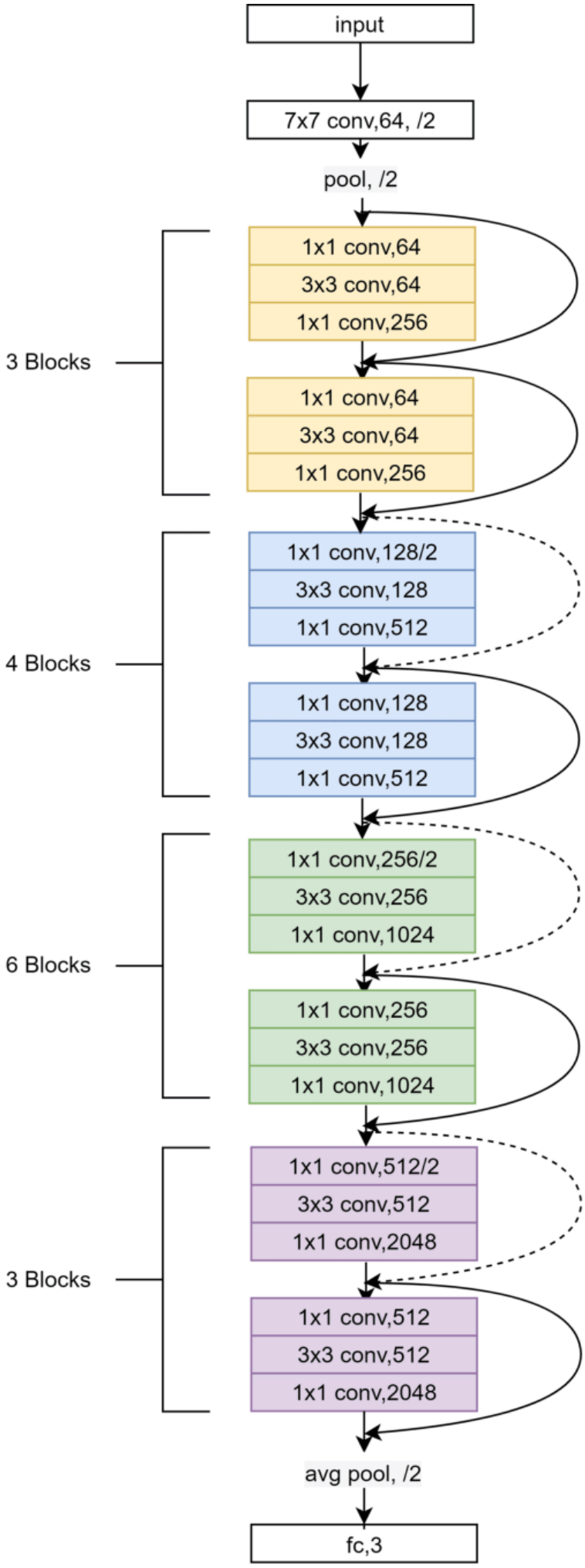
The complete architecture of ResNet50 is composed of four parts:

1. **Convolution layers**: These layers play a fundamental role in feature extraction. Convolution involves applying filters to input images, allowing the model to detect various patterns, edges, and textures within the data.
2. **Convolution blocks**: These blocks are composed of multiple convolution layers, often followed by normalization and activation functions. They facilitate the extraction of high-level features from the input data.
3. **Residual blocks**: Residual blocks serve as shortcuts or skip connections that allow the model to skip one or more layers. This helps in mitigating the vanishing gradient problem during training and aids in the smooth flow of information.
4. **Fully connected layers**: These layers are responsible for making predictions based on the extracted features. In the context of ResNet, the fully connected layers map the learned features to the final output classes.

The groundbreaking contribution of ResNet is the introduction of the **residual block**. These residual blocks allow connecting the activations of previous layers with the next while 'skipping' some layers in between, allowing the gradient to flow without being altered by a large magnitude.

Imagine you're trying to get from Point A to Point B, but there are some confusing twists and turns in the middle. Instead of going through every single twisty path, ResNet takes a clever shortcut, letting it skip over a few of the tricky parts. So, this magical shortcut helps ResNet learn faster and better. It's a little like learning to ride a bike: once you've mastered balancing, you don't need to relearn it every time you pedal.

The complete architecture of the ResNet50 model with four of its major parts is shown below:


input
7x7 conv,64, /2
pool, /2
1x1 conv,64
3x3 conv,64
1x1 conv,256
3 Blocks
1x1 conv,64
3x3 conv,64
1x1 conv,256
1x1 conv,128/2
3x3 conv,128
1x1 conv,512
4 Blocks
1x1 conv,128
3x3 conv,128
1x1 conv,512
1x1 conv,256/2
3x3 conv,256
1x1 conv,1024
6 Blocks
1x1 conv,256
3x3 conv,256
1x1 conv,1024
1x1 conv,512/2
3x3 conv,512
1x1 conv,2048
3 Blocks
1x1 conv,512
3x3 conv,512
1x1 conv,2048
avg pool, /2
fc,3

Source

Is ResNet50 a CNN?

CNNs are particularly adept at handling image-related tasks by leveraging convolutional layers that recognize patterns and hierarchies within the data. Therefore, since ResNet was trained on ImageNet (a large-scale dataset containing millions of labeled images across thousands of categories) and also incorporates convolutional layers, it is a CNN, allowing us to handle various visual recognition tasks.

The complete flow of ResNet involves:

- **Input Layer**: The journey begins with the input layer, where the raw image data is fed into the network.
- **Convolutional Layers**: Convolutional layers process the input image to detect features and generate feature maps.
- **Max Pooling**: Max pooling layers follow these convolutional layers. They downsample the spatial dimensions of the feature maps, retaining important information and reducing computational load.
- **Convolution and Residual Blocks**: Next are the convolutional blocks made up of multiple convolutional layers followed by residual blocks.
- **More Convolutional Blocks**: More convolutional blocks follow the residual blocks, refining features and learning intricate patterns.
- **Global Average Pooling**: Instead of fully connected layers, ResNet typically employs global average pooling. Global average pooling reduces the spatial dimensions of the feature maps to a single value per feature, simplifying the architecture.
- **Output Layer**: The reduced feature maps undergo a final classification step in the output layer, producing the model's predictions.

Is ResNet50 Supervised or Unsupervised?

Supervised learning involves training a model on a labeled dataset, where the input data is paired with corresponding output labels. In the case of ResNet, it is a type of **supervised** learning algorithm, where the model is trained to predict specific labels or outputs based on input images.

During the training process, ResNet is presented with input images along with their corresponding ground truth labels. The model learns to make predictions by adjusting its parameters (a.k.a. weights and biases) based on the discrepancies between its predictions and the true labels. This process is supervised because the training data includes explicit supervision in the form of labeled examples.

What Is the Purpose of ResNet50?

Before the introduction of ResNet, it was observed that as networks became deeper, they encountered difficulties in training and suffered from the vanishing gradient problem. The vanishing gradient problem arises when gradients during backpropagation become extremely small as they are propagated through many layers, making it challenging for the model to learn effectively. The introduction of residual learning and skip connections in ResNet addressed the vanishing gradient problem, enabling the training of very deep networks (e.g. ResNet50 with 50 layers) while maintaining or improving accuracy.

What Is ResNet50 Used For?

Resnet50 is a powerful model that captures complex and abstract features allowing it to be employed for tasks including but not limited to: image classification, object detection, and image segmentation. It can further be used as a backbone in Region-based Convolutional Neural Networks (RCNN) given its deep and powerful feature extraction capabilities. ResNet-50 is often used in transfer learning scenarios. Pre-trained ResNet-50 models, trained on large datasets like ImageNet, can be fine-tuned on smaller datasets for specific tasks. This helps leverage the knowledge gained from the larger dataset.

Is ResNet50 Better Than VGG?

VGG is also a CNN that was released before ResNet50 with the idea of having smaller filters with increased depth. A stack of multiple smaller filters allowed the model to have the same receptive field as a single large filter hence, inducing more non-linearity with increased depth.

Let's compare the two models under four criteria:

1. **Deeper Architecture**: ResNet50 is significantly deeper than VGG which is crucial for capturing more complex hierarchical features in images.
2. **Addressing Vanishing Gradient Problem**: ResNet50's skip connections effectively mitigate the vanishing gradient problem, which can be a challenge in training very deep networks. This allows ResNet50 to maintain high accuracy even as the number of layers increases, whereas VGG models may suffer from performance degradation in deeper architectures given the fact that they have a straightforward stack of convolutional layers.
3. **Improved Training and Convergence**: The skip connections in ResNet50 facilitate smoother training and faster convergence. Thus making it easier for the model to learn and update weights during training. This contributes to more efficient training dynamics compared to VGG.
4. **Parameter Efficiency**: ResNet50 achieves better parameter efficiency due to its bottleneck architecture, where 1x1 convolutions are used to reduce and then restore the dimensionality of the data. This design choice helps reduce the computational cost while maintaining expressive power, making ResNet50 more efficient in terms of parameters compared to VGG.

Therefore, ResNet50 is generally considered better than VGG, particularly in tasks that benefit from deeper architectures and efficient training dynamics.

How to Use ResNet50 for Image Classification

Now, let's delve into an illustrative example where we construct and train a ResNet50 model for image classification. This demonstration

will be conducted using two prominent libraries: Keras and PyTorch. Our chosen dataset for this exercise is CIFAR10, a widely accessible collection comprising 60,000 32x32 color images distributed across 10 distinct classes.

In this section, we also utilize the Weights & Biases (wandb) library. Wandb is a powerful yet user-friendly platform that seamlessly integrates with deep learning workflows. It provides real-time insights into the model's training process, enabling clear visualization of metrics like loss and accuracy, and providing a comprehensive overview of the model's performance.

First, let's carry out the implementation with Keras.

Keras

Step 1: Importing the Necessary Libraries

```
!pip install wandb
!wandb login
import wandb
from wandb.keras import WandbCallback
import numpy as np
import tensorflow
from tensorflow import keras
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
import matplotlib.pyplot as plt
import time
```

Step 2: Initialize WandB

```
wandb.init(project='resnet50_project', name='resnet50')
```

Step 3: Load and Preprocess the Dataset

Subsequently, we will retrieve the CIFAR-10 dataset from the Keras library utilizing the load_data() function and preprocess the images.

```
(x_train, y_train), (x_test, y_test) = tensorflow.keras.datasets.ci
```

```
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
```

Let's visualize some of the images within the data:

```
fig, axs = plt.subplots(2, 4)
axs[0, 0].imshow(x_train[0])
axs[0, 1].imshow(x_train[1])
axs[0, 2].imshow(x_train[2])
axs[0, 3].imshow(x_train[3])
axs[1, 0].imshow(x_test[0])
axs[1, 1].imshow(x_test[1])
axs[1, 2].imshow(x_test[2])
axs[1, 3].imshow(x_test[3])
plt.show()
```

Step 4: Building ResNet-50 Model

Transitioning to the core process of model construction, we leverage the power of transfer learning by importing a pre-trained ResNet50 model from 'keras.applications'. This model has been trained on the ImageNet dataset, encompassing a vast repository of 14,197,122 images spanning 1000 distinct classes. Given that the CIFAR-10 dataset comprises only 10 output classes, and the imported model includes 1000 classes, we mitigate this mismatch by setting

include_top=False. This configuration excludes the final layers of the imported model, aligning it with our specific classification requirements.

```
def feature_extractor(inputs):
        feature_extractor = keras.applications.resnet.ResNet50(input
        return feature_extractor
```

To extend the capabilities of the pretrained ResNet50 model, we introduce additional layers in the 'classifier_layers' function. Specifically, we incorporate a Global Average Pooling layer, which helps in spatially summarizing the features, followed by Fully Connected layers to increase the model's learning capacity. The last dense layer contains 10 units to account for 10 classes present in our classification problem.

```
def classifier_layers(inputs):
        x = layers.GlobalAveragePooling2D()(inputs)
        x = layers.Flatten()(x)
        x = layers.Dense(1024, activation="relu")(x)
        x = layers.Dense(512, activation="relu")(x)
        x = layers.Dense(10, activation="softmax", name="classificat
        return x
```

Before setting up the complete model, we upsample the input image size of Cifar10 from 32x32 to match with the input dimensions of resnet50 which was trained on imagenet data of size 224x224.

```
def modified_resnet(inputs):
        resize = layers.UpSampling2D(size=(7,7))(inputs)
        resnet_feature_extractor = feature_extractor(resize)
        classification_output = classifier_layers(resnet_feature_ext
        return classification_output
```

Next, we compile the model.

```
def compile_model():
        inputs = layers.Input(shape=(32,32,3))
```

```
        classification_output = modified_resnet(inputs)
        model = keras.Model(inputs=inputs, outputs = classification_
        model.compile(optimizer='SGD', loss='sparse_categorical_cros
        return model
model = compile_model()
```

Let's print the model summary:

```
model.summary()
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_3 (InputLayer)        [(None, 32, 32, 3)]       0

 up_sampling2d_1 (UpSamplin  (None, 224, 224, 3)       0
 g2D)

 resnet50 (Functional)       (None, 7, 7, 2048)        23587712

 global_average_pooling2d_1  (None, 2048)              0
  (GlobalAveragePooling2D)

 flatten_1 (Flatten)         (None, 2048)              0

 dense_2 (Dense)             (None, 1024)              2098176

 dense_3 (Dense)             (None, 512)               524800

 classification (Dense)      (None, 10)                5130

=================================================================
Total params: 26215818 (100.01 MB)
Trainable params: 26162698 (99.80 MB)
Non-trainable params: 53120 (207.50 KB)
_________________________________________________________________
```

▾ Step 5: Training the Model

```
EPOCHS = 3
history = model.fit(x_train, y_train, epochs=EPOCHS, batch_size=64)
```

▾ Step 6: Evaluating the Model

```
EPOCHS = 3
history = model.fit(x_train, y_train, epochs=EPOCHS, batch_size=64,
                    callbacks=[WandbCallback()])
```

```
test_loss, test_acc = model.evaluate(x_test, y_test)
percentage = round(test_acc * 100, 2)
print(f'Test accuracy: {percentage}%')
```

The resulting accuracy on the test set is 86%

The image below displays the complete logs, ranging from data loading to model training and evaluation, as tracked using wandb.
<add image here>

Now let's carry out the implementation with PyTorch.

PyTorch

Step 1: Importing the Necessary Libraries

```
import wandb
from wandb.keras import WandbCallback
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms, models
from torch.utils.data import DataLoader
import time
from tqdm import tqdm
import wandb
```

Step 2: Initialize WandB and Setup CUDA

```
wandb.init(project='resnet50_project', name='resnet_pytorch')
device = torch.device("cuda" if torch.cuda.is_available() else "cpu"
```

Step 3: Load and Preprocess the Dataset

```
transform = transforms.Compose([
        transforms.Resize((224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
])

train_dataset = datasets.CIFAR10(root='./data', train=True, downloa
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True
test_dataset = datasets.CIFAR10(root='./data', train=False, downloa
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False,
```

Step 4: Building ResNet-50 Model

```
class CustomResNet(nn.Module):
        def __init__(self):
                super(CustomResNet, self).__init__()
                self.resnet = models.resnet50(pretrained=True)
                self.resnet.fc = nn.Identity()
                self.classifier = nn.Sequential(
                        nn.Flatten(),
                        nn.Linear(2048, 1024),
                        nn.ReLU(),
                        nn.Linear(1024, 512),
                        nn.ReLU(),
                        nn.Linear(512, 10),
                        nn.Softmax(dim=1)
                 )
         def forward(self, x):
                x = self.resnet(x)
                x = self.classifier(x)
                return x
model = CustomResNet().to(device)
```

The pretrained ResNet50 model is imported from torchvision models and additional layers are added. It must be noted that in the PyTorch code provided, the global average pooling layer is not explicitly added after the pre-trained ResNet-50 backbone. The reason for this is that the architecture of ResNet-50 in Torchvision's models already includes a global average pooling layer as part of its final layer.

Step 5: Training the Model

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

EPOCHS = 3

for epoch in range(EPOCHS):
        running_loss = 0.0
        data_bar = tqdm(train_loader)
        i = 0
        correct = 0
        total = 0
        start_time = time.time()

        for data in data_bar:
                inputs, labels = data
                inputs = inputs.to(device)
                labels = labels.to(device)
                # Zero the parameter gradients
                optimizer.zero_grad()
                # Forward + backward + optimize
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                running_loss += loss.item()
                data_bar.set_description("Processing epoch {:d} mini
                i += 1

        # Log metrics to wandb
        wandb.log({'epoch': epoch + 1, 'train_loss': running_loss /
```

```
print('Finished Training')
```

Step 6: Evaluating the Model

```
correct = 0
total = 0
i = 0
with torch.no_grad():
        pbar = tqdm(test_loader)
        for data in data_bar:
                images, labels = data
                images = images.to(device)
                labels = labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                pbar.set_description("minibatch {:d} test accuracy {
                i += 1

print('Accuracy of the network on the 10000 test images: %4.2f %%' %
```

The resulting accuracy on the test set is 82.86%

The image below displays both epoch and train_loss graphs for the fine-tuning process, as tracked using wandb:

Conclusion

In summary, the ResNet50 architecture demonstrates remarkable versatility across a broad spectrum of tasks, including image classification and object detection. It particularly excels in scenarios where deeper models might compromise accuracy. The availability of a pre-trained ResNet50 model in both Keras and PyTorch libraries enhances its accessibility and ease of integration, making it an excellent choice for achieving high-quality results in various deep-learning applications.

Tags: Articles, Computer Vision, Tutorial, Intermediate

Created with ❤️ on Weights & Biases.

https://wandb.ai/mostafaibrahim17/ml-articles/reports/The-Basics-of-ResNet50---Vmlldzo2NDkwNDE2

Weights & Biases

Get weekly updates with the latest ML news.

Subscribe

PRODUCTS

Dashboard **Sweeps** **Artifacts** **Reports** **Tables**

QUICKSTART

Documentation

RESOURCES

Courses **Forum** **Tutorials** **Benchmarks**

W&B

About Us **Authors** **Contact** **Terms of Service** **Privacy Policy**
