50.039 Theory and Practice of Deep Learning

W4-S3 Introduction to Computer Vision and Convolutional Neural Networks

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

About this week (Week 4)

1. What are **images** and how is this datatype represented?
2. What is a **pixel** and how can **its information be interpreted**?
3. What is the **spatial dependence property** of pixels?
4. What is the **homophily property** of pixels?
5. Why is the **linear processing operation failing** on images?
6. What is the **convolution** operation?
7. How can we perform **image processing using convolutions**?
8. What is **padding** in convolutions?

About this week (Week 4)

9. What is **stride** in convolution?
10. What is **dilation** in convolution?
11. How does convolution apply to **higher dimensional images**?
12. How to write **our own convolutional processing layer** in PyTorch?
13. What is **Conv2d** in PyTorch?
14. What is a **Convolutional Neural Network** (CNN)?
15. What is the **intuition** behind using Convolutional layers in a Convolutional Neural Network?

About this week (Week 4)

16. How to **train** our first Convolutional Neural Network on MNIST?
17. What are **other typical image processing layers** in Computer Vision and how are they implemented?
18. What is the **pooling layer**?
19. What is the **batchnorm layer**?
20. What is the **dropout layer**?
21. What is **data augmentation** in Computer Vision?
22. What are some **milestone Computer Vision models** and their **contributions** to the field of Deep Learning?

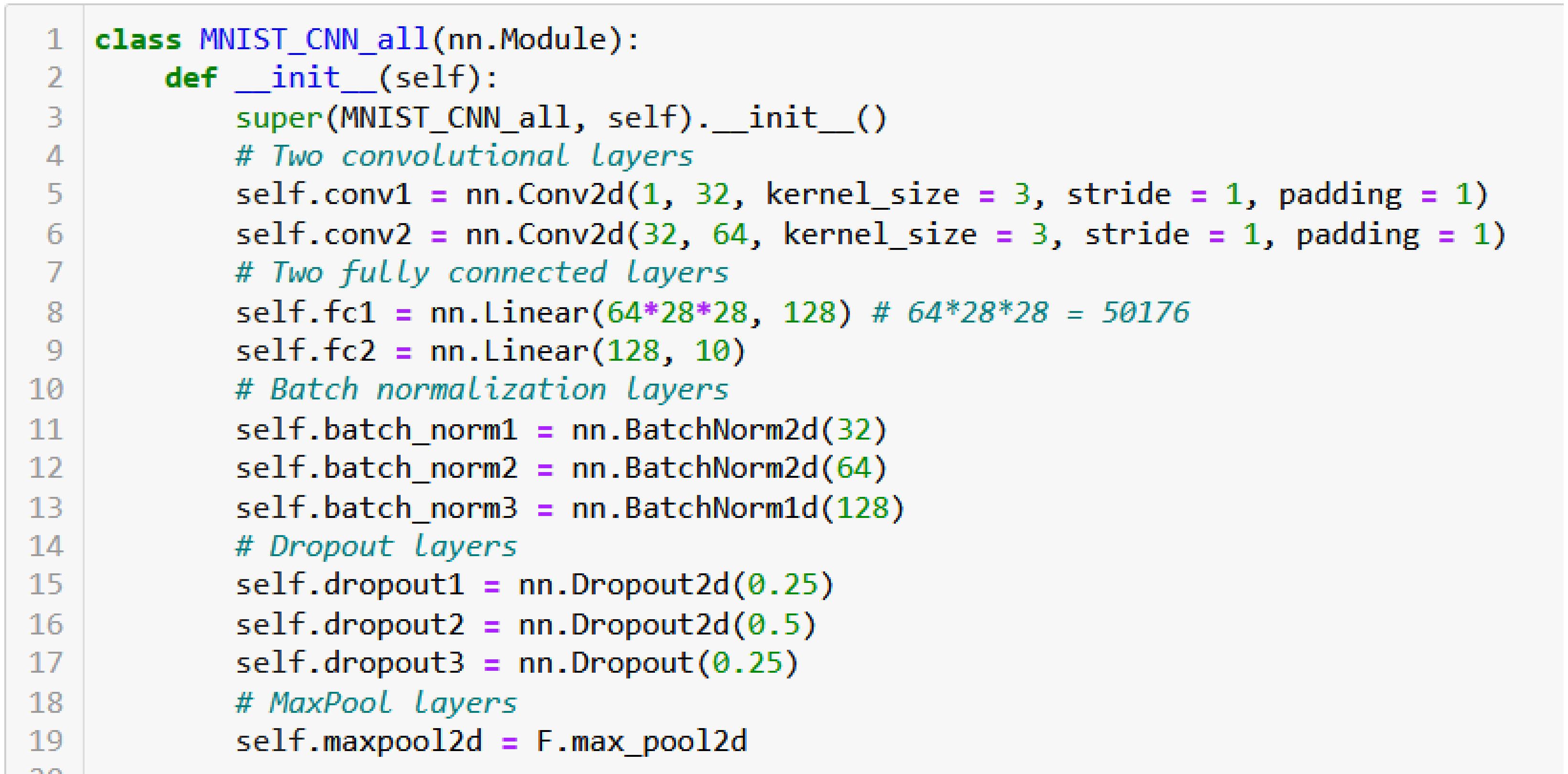
About this week (Week 4)

23. What are the **AlexNet** and **VGG models**?
24. What is a **skip connection/residual**? What is its effect on a Neural Network and **how does it help with vanishing gradient problems**?
25. What are **ResNet** and **DenseNet models**?
26. What is the **Inception model**?
27. What is the **EfficientNet model**?
28. What is **transfer learning** and its uses?
29. How to **freeze** and **fine-tune layers** in a Neural Network?

Implementing all layers in our CNN

Let us first define Batchnorm, dropout and maxpool layers in __init__.

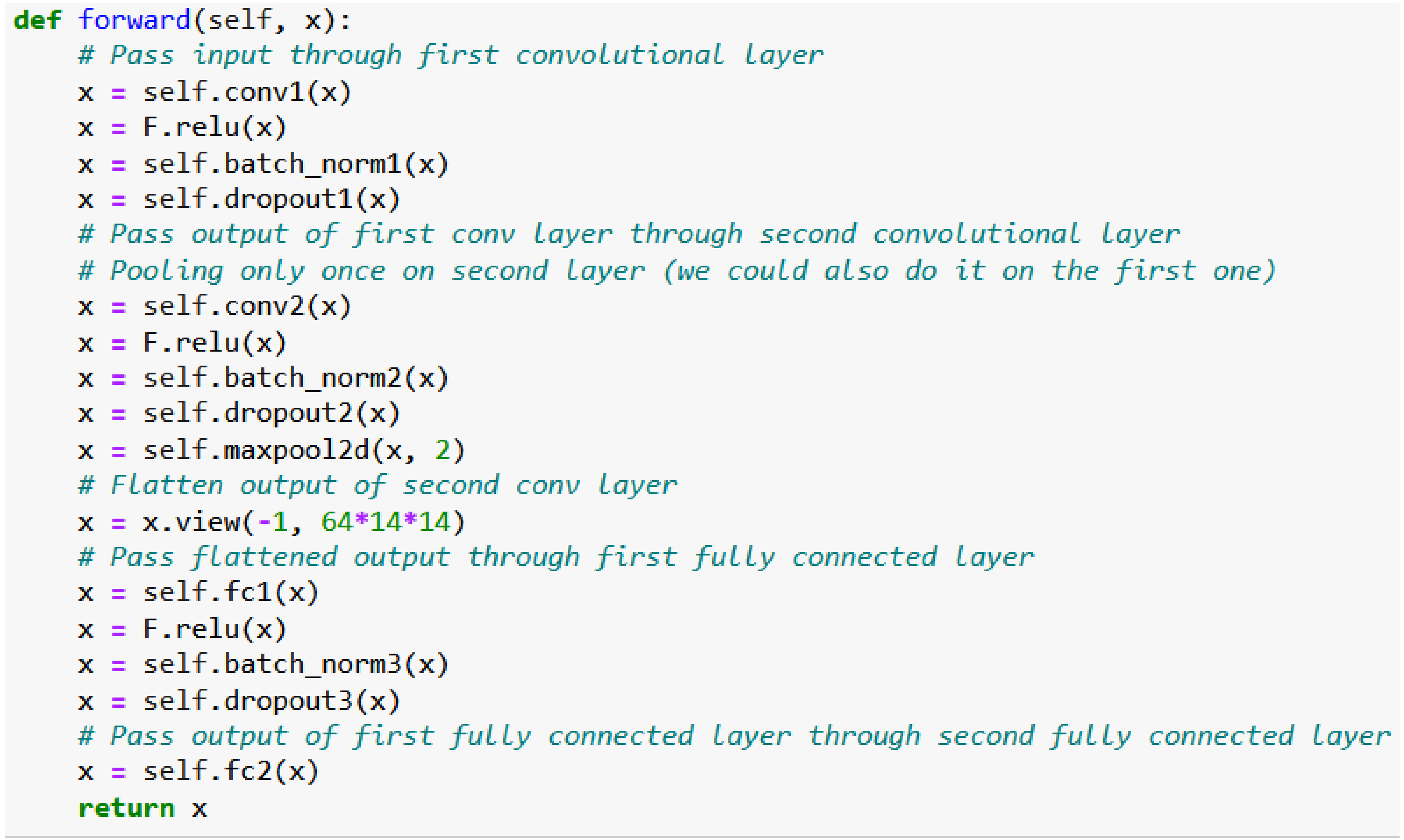
Arbitrarily fixing the dropout probabilities (should be hyperparameter?).



```python
class MNIST_CNN_all(nn.Module):
    def __init__(self):
        super(MNIST_CNN_all, self).__init__()
        # Two convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size = 3, stride = 1, padding = 1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size = 3, stride = 1, padding = 1)
        # Two fully connected layers
        self.fc1 = nn.Linear(64*28*28, 128) # 64*28*28 = 50176
        self.fc2 = nn.Linear(128, 10)
        # Batch normalization layers
        self.batch_norm1 = nn.BatchNorm2d(32)
        self.batch_norm2 = nn.BatchNorm2d(64)
        self.batch_norm3 = nn.BatchNorm1d(128)
        # Dropout layers
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.dropout3 = nn.Dropout(0.25)
        # MaxPool layers
        self.maxpool2d = F.max_pool2d
```

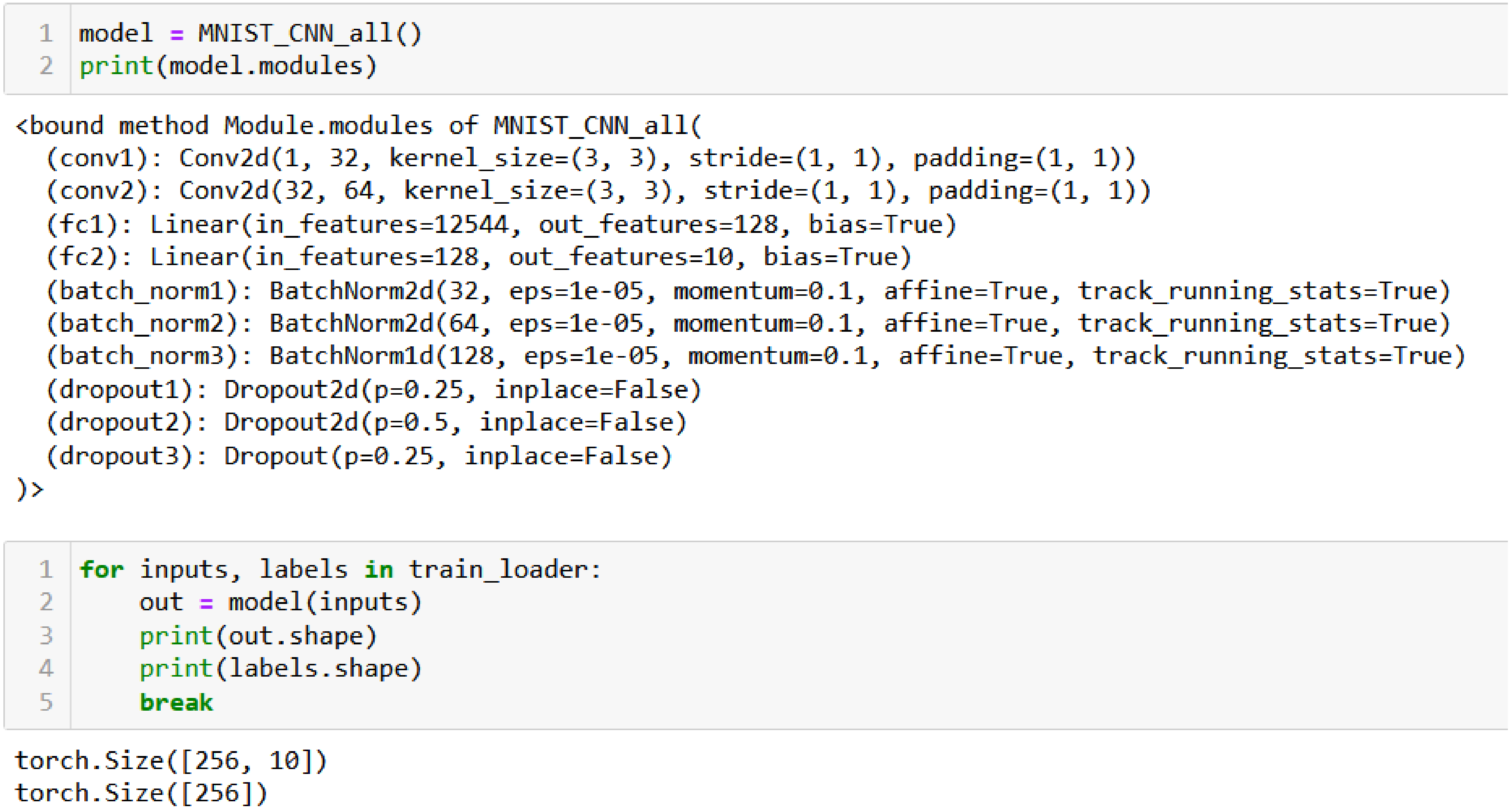
Implementing all layers in our CNN

- Using each layer in correct sequence of the forward() method.
- Also, one MaxPool layer after last Conv2d.
- 64*14*14 instead of 64*28*28 in Flatten due to MaxPool



```python
def forward(self, x):
    # Pass input through first convolutional layer
    x = self.conv1(x)
    x = F.relu(x)
    x = self.batch_norm1(x)
    x = self.dropout1(x)
    # Pass output of first conv layer through second convolutional layer
    # Pooling only once on second layer (we could also do it on the first one)
    x = self.conv2(x)
    x = F.relu(x)
    x = self.batch_norm2(x)
    x = self.dropout2(x)
    x = self.maxpool2d(x, 2)
    # Flatten output of second conv layer
    x = x.view(-1, 64*14*14)
    # Pass flattened output through first fully connected layer
    x = self.fc1(x)
    x = F.relu(x)
    x = self.batch_norm3(x)
    x = self.dropout3(x)
    # Pass output of first fully connected layer through second fully connected layer
    x = self.fc2(x)
    return x
```

Implementing all layers in our CNN



```python
model = MNIST_CNN_all()
print(model.modules)
```

```
<bound method Module.modules of MNIST_CNN_all(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=12544, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
  (batch_norm1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_norm2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batch_norm3): BatchNorm1d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (dropout1): Dropout2d(p=0.25, inplace=False)
  (dropout2): Dropout2d(p=0.5, inplace=False)
  (dropout3): Dropout(p=0.25, inplace=False)
)>
```

```python
for inputs, labels in train_loader:
    out = model(inputs)
    print(out.shape)
    print(labels.shape)
    break
```

```
torch.Size([256, 10])
torch.Size([256])
```

A final note

The Dropout, Batchnorm and Pooling layers have been introduced in the Computer Vision field.

However, these days, every other field of Deep Learning (e.g. Natural Language Processing) is reusing them in their own models.

Important lesson: you might not necessarily be interested in Computer Vision, but you should still consider learning about it, as many intuitions and tools could be reused in other problems.

A quick word about Data Augmentation

Definition (Data Augmentation):

Data augmentation is a technique used to **artificially increase the size of the training dataset** by applying various transformations to the existing data.

This helps to **expose the model to more variations of the data**, making it **more robust and less prone to overfitting** (according to [Krizhevsky2012]).

More specifically, the model can then learn to **recognize patterns and features that are supposed to be invariant to certain changes**, such as translations, rotations, and scaling.

Data augmentation can also be used to **correct imbalance** in the distribution of classes by artificially increasing the number of examples of the under-represented classes (e.g. if you have many pictures of 2s and few 6s), or **when the amount of available training data is limited**.

Some examples of data augmentation/transform

Some possible ways to perform data augmentation include:

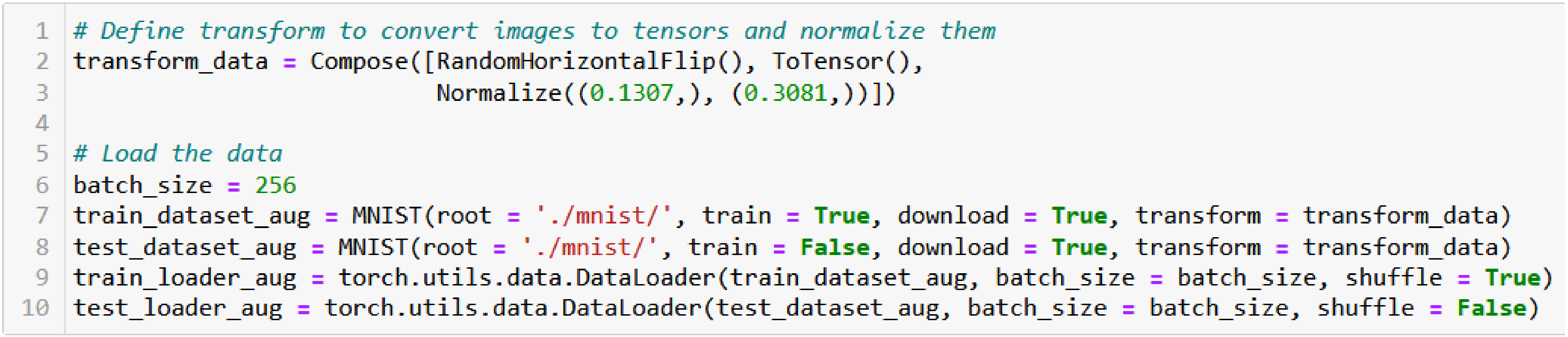
- Randomly **translating** images by a certain number of pixels,
- Randomly **rotating** the images by a small angle,
- Randomly **scaling** the images by a small factor,
- Randomly **cropping** the images,
- Randomly **flipping** the images horizontally/vertically,
- **Adding random noise** to the images,
- Randomly **changing the brightness or contrast** of the images,
- Etc.

Some examples of data augmentation/transform

- These can be accomplished by using pre-built data augmentation transforms such as **RandomRotation**, **RandomAffine**, **RandomCrop**, **RandomHorizontalFlip**, **RandomGrayscale** from the torchvision.transforms library or by using other libraries like imgaug, albumentations, etc.
- Feel free to have a look at: https://pytorch.org/vision/stable/transforms.html
- Additionally, you can also chain multiple transforms together using the Compose class from the torchvision.transforms library to apply multiple data augmentations at once.

Some examples of data augmentation/transform

- For instance, if we want to flip images horizontally, we will simply amend the code below, adding **RandomHorizontalFlip()** to the list of transforms.
- **Quick question:** Does it makes sense to perform this data augmentation transformation on all pictures?
  Can all digits be flipped horizontally to generate new valid images?



```python
# Define transform to convert images to tensors and normalize them
transform_data = Compose([RandomHorizontalFlip(), ToTensor(),
                          Normalize((0.1307,), (0.3081,))])

# Load the data
batch_size = 256
train_dataset_aug = MNIST(root = './mnist/', train = True, download = True, transform = transform_data)
test_dataset_aug = MNIST(root = './mnist/', train = False, download = True, transform = transform_data)
train_loader_aug = torch.utils.data.DataLoader(train_dataset_aug, batch_size = batch_size, shuffle = True)
test_loader_aug = torch.utils.data.DataLoader(test_dataset_aug, batch_size = batch_size, shuffle = False)
```

Studying remarkable CV models

There are several reasons why it is important to learn about milestone computer vision models like AlexNet, Inception, ResNet, etc.

1. **Understanding the history of computer vision and preparing for future advancements:** gives appreciation for the progress that has been made in the field and how far it has come.

   See how each new model was an improvement upon the previous one and how each model paved the way for new advancements.

2. **Understanding the key concepts and techniques:** These milestone models introduced key concepts and techniques that are still widely used today, such as convolutional neural networks, residual connections, and transfer learning.

   By understanding these concepts and techniques, you will have a better understanding of how modern computer vision models work.

Studying remarkable CV models

3. **Improving your skills:** Studying these models can help you improve your skills as a practitioner of computer vision.

   You can learn how to implement these models and use them to solve real-world problems, as well as understand how to adapt and extend these models to fit your own needs (e.g. using a them on a slightly different dataset than the one they were initially trained on).

Overall, learning about milestone computer vision models is an important part of advancing your knowledge and skills in the field of computer vision.

We will now discuss, **very briefly**, some of these important milestone problems.

We strongly advise to pick the Term 7 Computer Vision course, to continue your learning!

The AlexNet model

Definition (The AlexNet model):

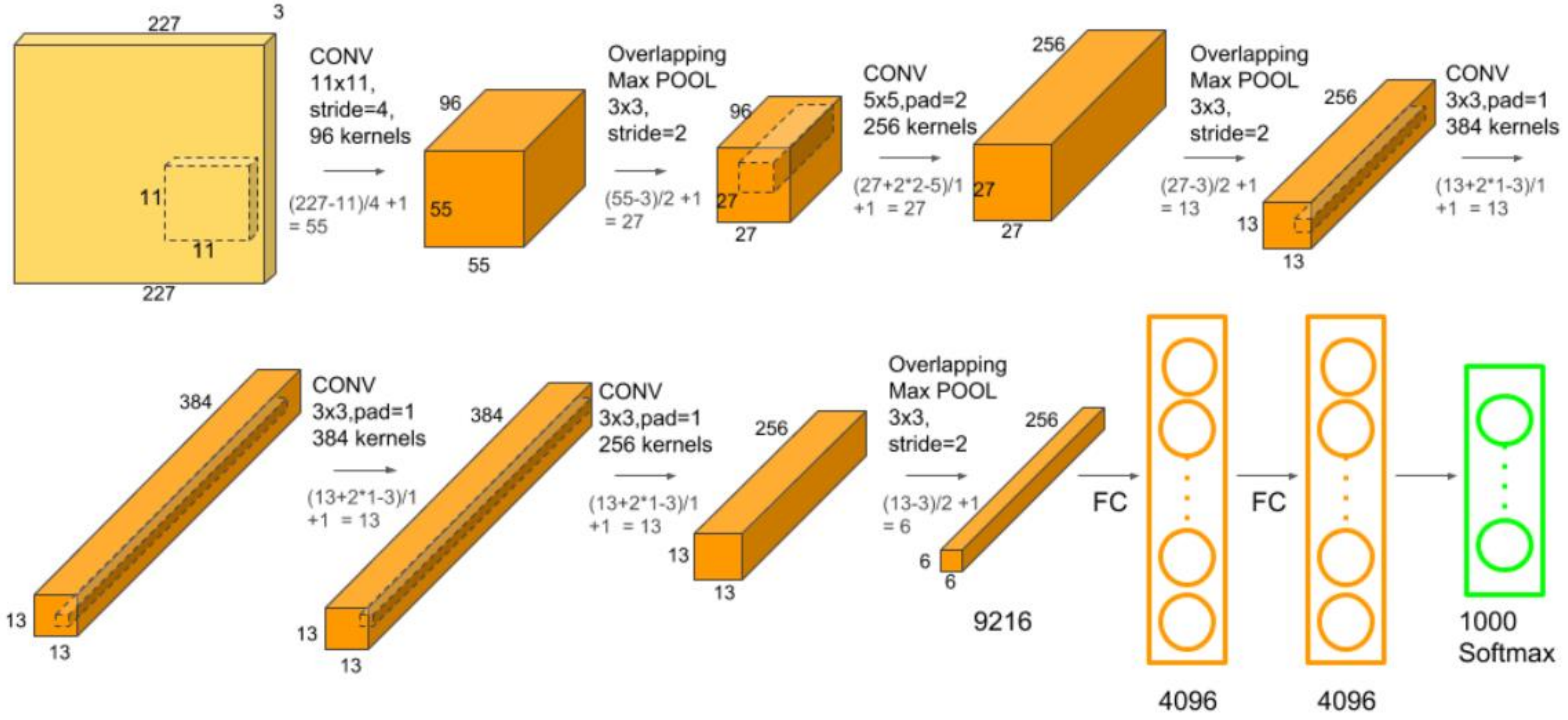
AlexNet is a model that competed in the "**ImageNet Large Scale Visual Recognition Challenge**" on September 30, 2012 ([Krizhevsky2012]).

It was trained on the ImageNet dataset, an image classification dataset consisting of 256×256 RGB images (used as inputs), with 1000 classes (used as outputs).

AlexNet consists of a CNN model, similar to the one in Week4, Notebook 5. It was one of the first architectures to combine Conv2d layers with Dropout, Pooling and ReLU layers.

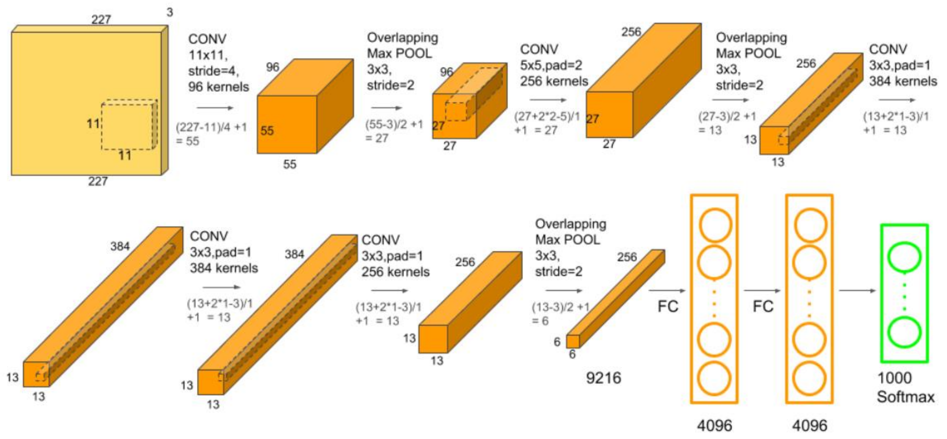
However, its main innovation at the time was the sheer number of layers and trainable parameters, and **how the model had been trained on GPUs instead of CPU to improve the computational performance of the training**.

The pre-trained model, with parameters defined as in 2012, can be loaded directly from PyTorch.


227
3
227
11
11
CONV
11x11,
stride=4,
96 kernels
(227-11)/4 +1
= 55
96
55
55
Overlapping
Max POOL
3x3,
stride=2
(55-3)/2 +1
= 27
96
27
27
CONV
5x5,pad=2
256 kernels
(27+2*2-5)/1
+1 = 27
256
27
27
Overlapping
Max POOL
3x3,
stride=2
(27-3)/2 +1
= 13
256
13
13
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
384
13
13
CONV
3x3,pad=1
384 kernels
(13+2*1-3)/1
+1 = 13
384
13
13
CONV
3x3,pad=1
256 kernels
(13+2*1-3)/1
+1 = 13
256
13
13
Overlapping
Max POOL
3x3,
stride=2
(13-3)/2 +1
= 6
256
6
6
9216
FC
4096
FC
4096
1000
Softmax

```python
model = torchvision.models.alexnet(weights = 'DEFAULT', progress = True)
model.eval()
print(model)
```

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

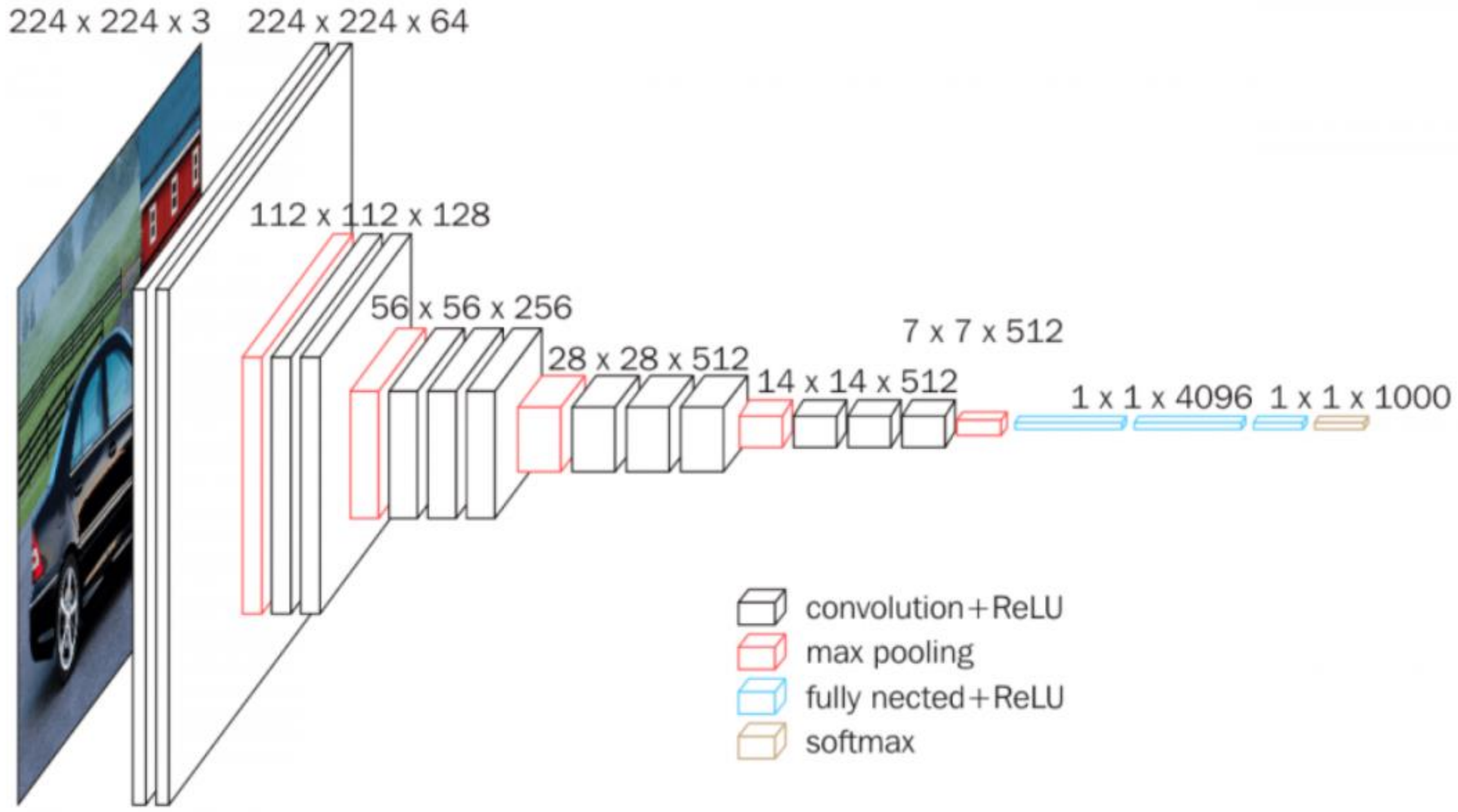

227
3
11
11
227
CONV 11x11, stride=4, 96 kernels
(227-11)/4 +1 = 55
96
55
55
Overlapping Max POOL 3x3, stride=2
(55-3)/2 +1 = 27
96
27
27
CONV 5x5,pad=2 256 kernels
(27+2*2-5)/1 +1 = 27
256
27
27
Overlapping Max POOL 3x3, stride=2
(27-3)/2 +1 = 13
256
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 384 kernels
(13+2*1-3)/1 +1 = 13
384
13
13
CONV 3x3,pad=1 256 kernels
(13+2*1-3)/1 +1 = 13
256
13
13
Overlapping Max POOL 3x3, stride=2
(13-3)/2 +1 = 6
256
6
6
9216
FC
4096
FC
4096
1000 Softmax

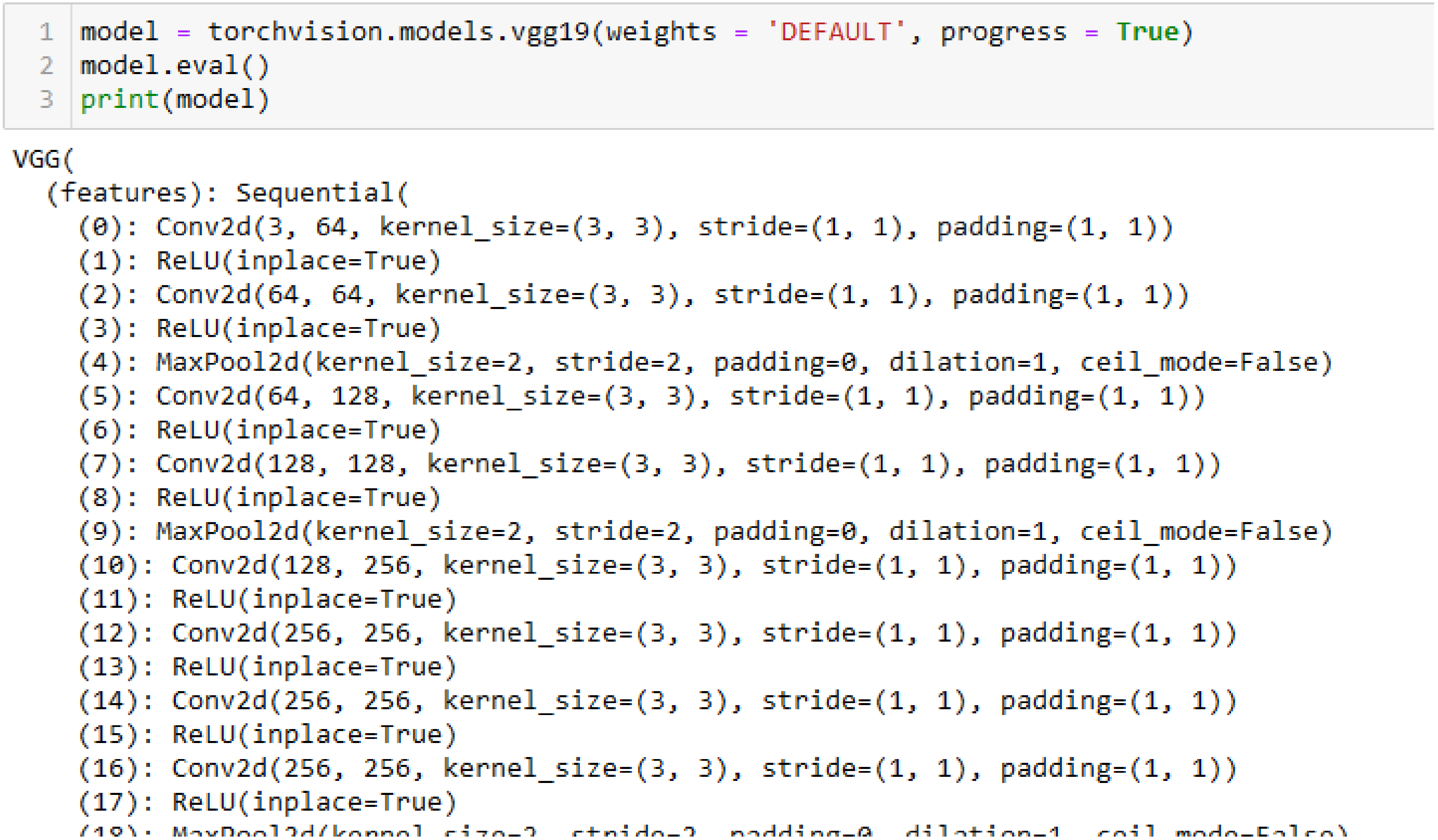
The VGG model

Definition (the VGG model):

VGG was based off of AlexNet, but it has several differences that separated it from other models ([]):

- Instead of using 11x11 kernels with a stride of 4, VGG uses smaller 3x3 kernels with a stride of 1. It also uses three ReLU units instead of just one. **For this reason, the decision function is more discriminative**, and it has **fewer parameters** (27 times the number of channels, instead of AlexNet's 49 times the number of channels).
- VGG also incorporates 1x1 convolutional layers to make the decision function more non-linear without changing the receptive fields.
- Having less parameters allows to have a **larger number of layers**, leading to improved performance.


224 x 224 x 3
224 x 224 x 64
112 x 112 x 128
56 x 56 x 256
28 x 28 x 512
14 x 14 x 512
7 x 7 x 512
1 x 1 x 4096
1 x 1 x 1000
convolution+ReLU
max pooling
fully nected+ReLU
softmax



```python
model = torchvision.models.vgg19(weights = 'DEFAULT', progress = True)
model.eval()
print(model)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace=True)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace=True)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace=True)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace=True)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace=True)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace=True)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

The problem with very deep models

- Very deep models, however, start to exhibit a **problem**…
- **Training the parameters on the first layers appears to be difficult**…
- For some reason, the gradients for the parameters of the first layer are close to zero (that is another instance of a **vanishing gradient**).

The problem with very deep models

- Very deep models, however, start to exhibit a **problem**…
- **Training the parameters on the first layers appears to be difficult**…
- For some reason, the gradients for the parameters of the first layer are close to zero (that is another instance of a **vanishing gradient**).
- **Reason:** using the chain rule many times in a row, multiplying partial derivatives with small values eventually leads to a very small, close to zero value for those partial derivatives.

$$\frac{\partial L}{\partial W_1} = \underbrace{\frac{\partial L}{\partial A_{25}}} \underbrace{\frac{\partial A_{25}}{\partial A_{24}}} \cdots \underbrace{\frac{\partial A_1}{\partial W_1}}$$

All have values in [-1,1]

The problem with very deep models

- Very deep models, however, start to exhibit a **problem**…
- **Training the parameters on the first layers appears to be difficult**…
- For some reason, the gradients for the parameters of the first layer are close to zero (that is another instance of a **vanishing gradient**).
- **Reason:** using the chain rule many times in a row, multiplying partial derivatives with small values eventually leads to a very small, close to zero value for those partial derivatives.

$$\underbrace{\frac{\partial L}{\partial W_1}}_{\text{Multiplication of all these values gives something close to 0}} = \underbrace{\frac{\partial L}{\partial A_{25}}} \underbrace{\frac{\partial A_{25}}{\partial A_{24}}} \cdots \underbrace{\frac{\partial A_1}{\partial W_1}}$$

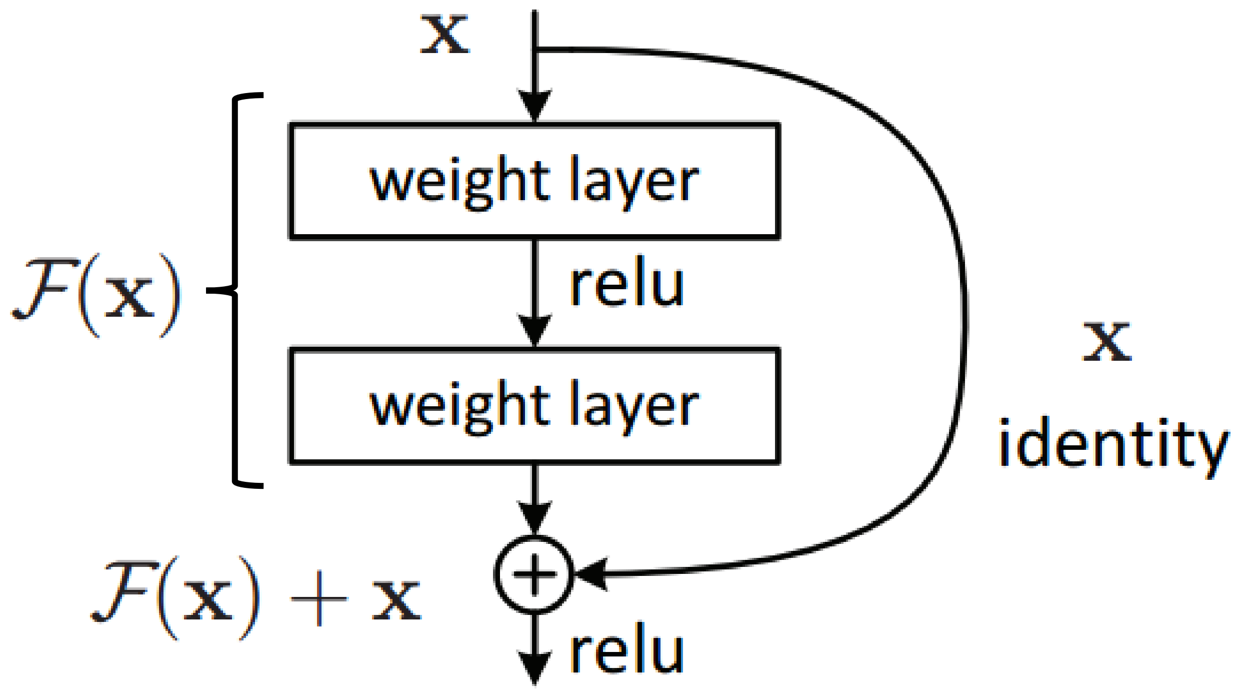
All have values in [-1,1]

Skip connections

Definition (skip connections):

Skip connections (or shortcut connections) refer to the direct connections between non-adjacent layers in a neural network architecture, bypassing one or more intermediate layers.

The purpose is to allow information to flow rapidly from the input of the network to its output, without being transformed or lost in the intermediate layers.



Skip connections

Skip connections allow information to flow rapidly from the input to the output, without being transformed or lost in the intermediate layers.

- First, it enables the network to **retain information from the input layer**, which can be particularly useful in the case of deep neural networks, where **the information can get transformed multiple times, hence losing its original information, as it passes through the hidden layers**.
- Second, skip connections are also mechanisms that **facilitate information flow across many layers** and help to **mitigate the degradation/vanishing gradient** problem that appears on large models with many layers and that we have discussed earlier.

The ResNet model

Definition (the ResNet model):

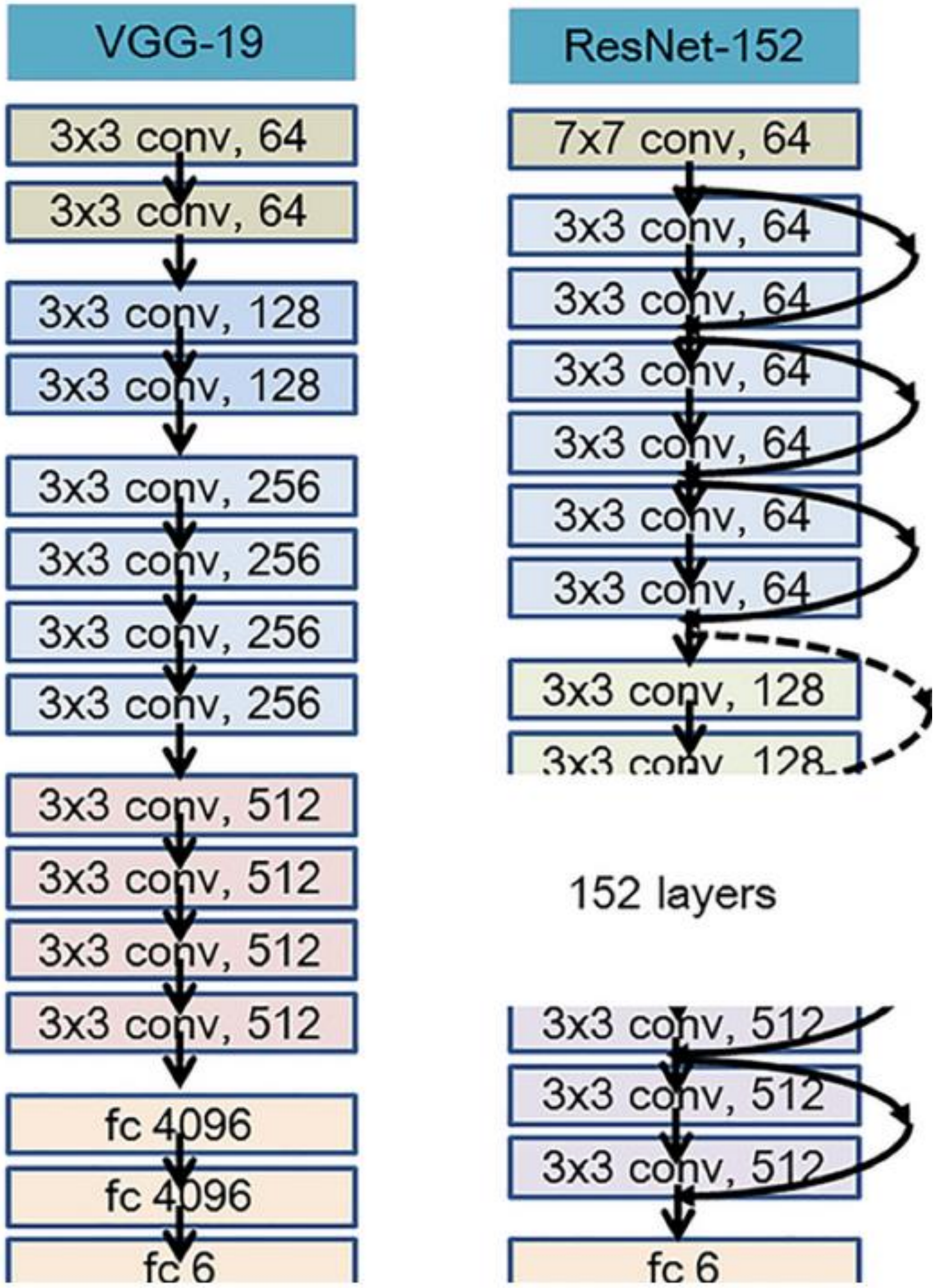
Skip connections effectively simplifies the network, helping the information flow in the network.

This speeds learning by **reducing the impact of vanishing gradients**.

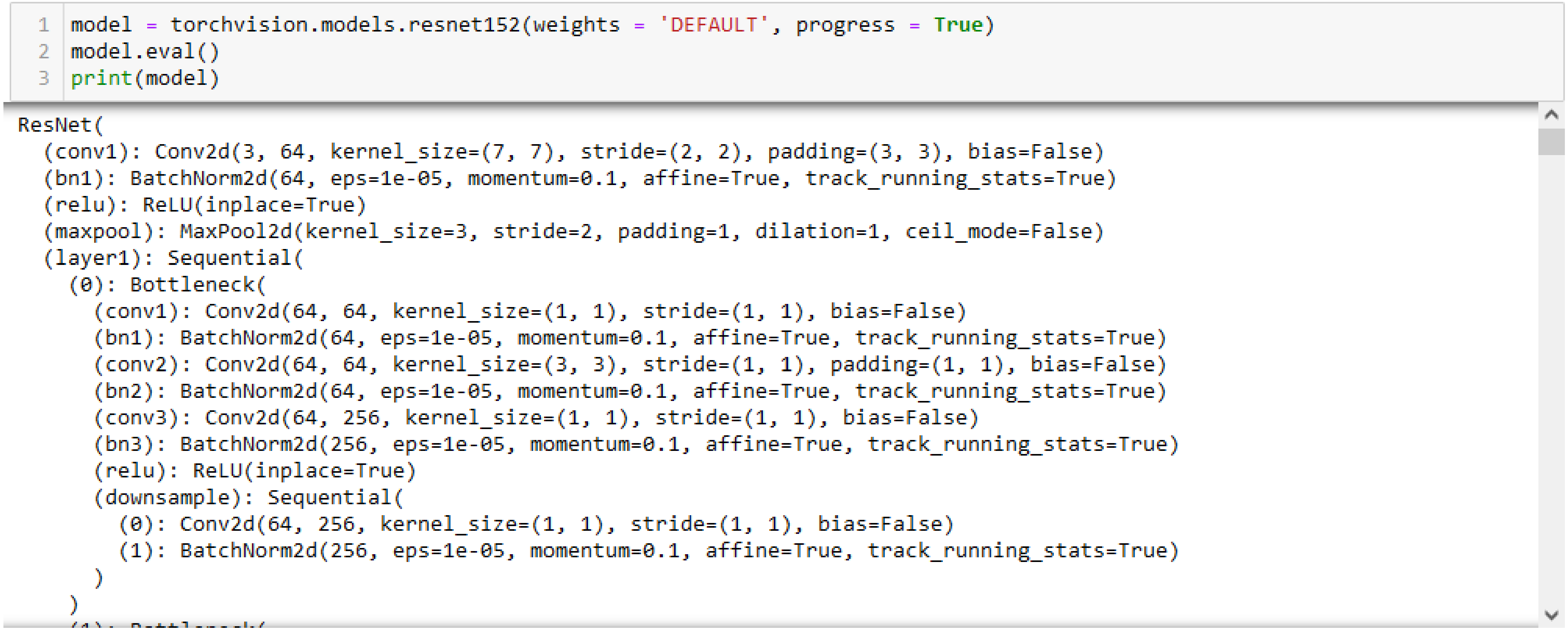
The network also gradually restores the skipped layers as it learns the feature space, reusing previous inputs in deeper layers.

Eventually, **this allows to train VERY large models,** called **Residual Networks or ResNets: ResNet** can have up to **152 layers** (!), compared to the previous VGG models, which only had up to 19 layers.

ResNet (introduced in [He2015]) has become the most cited neural network of the 21st century.


VGG-19
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128
3x3 conv, 128
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
fc 4096
fc 4096
fc 6
ResNet-152
7x7 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 64
3x3 conv, 128
3x3 conv, 128
152 layers
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
fc 6

The ResNet model



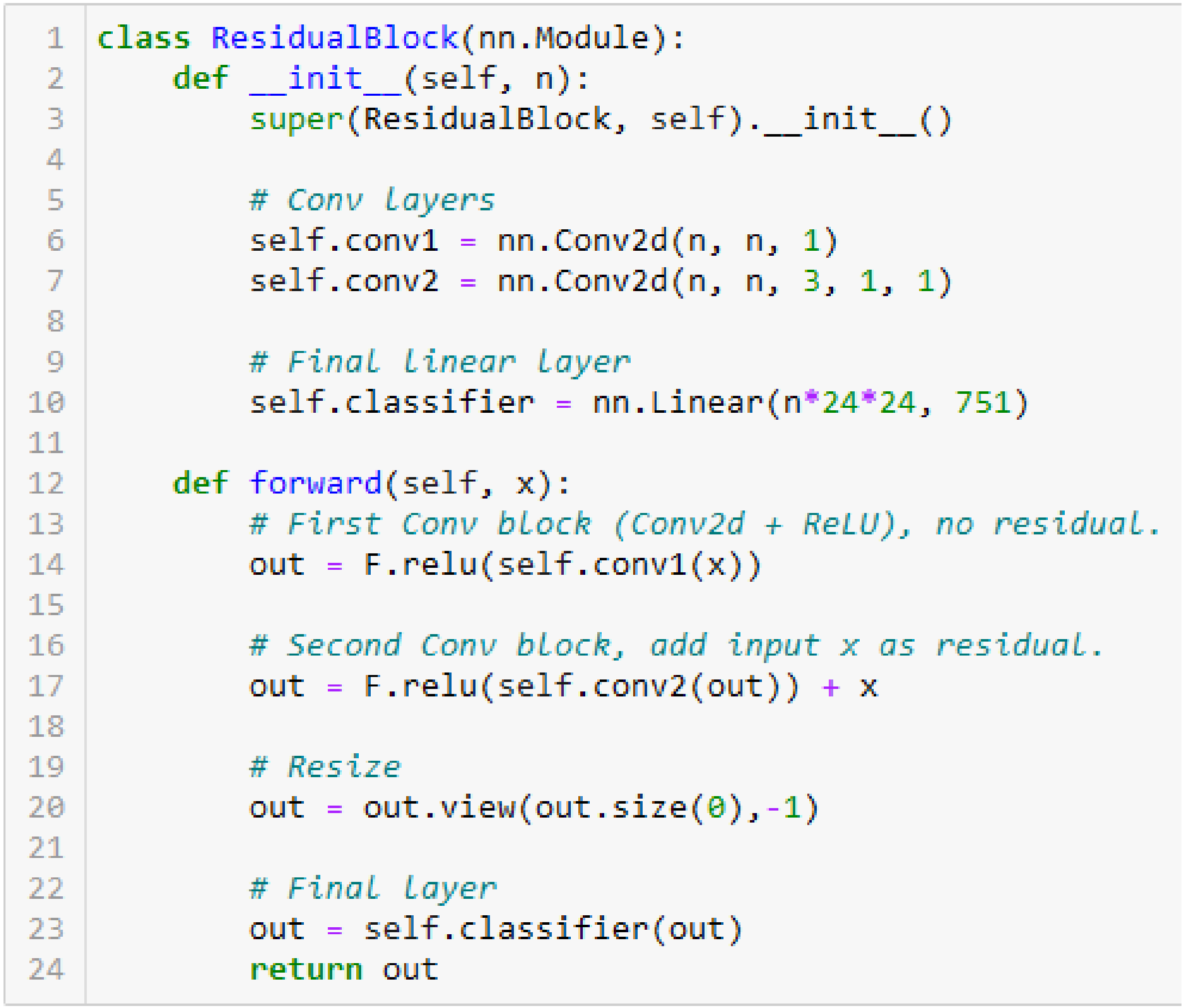
```
model = torchvision.models.resnet152(weights = 'DEFAULT', progress = True)
model.eval()
print(model)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
```

Implementing a skip connection/residual

- A residual is simply implemented in the forward method of a Residual block.
- Pay attention to the line below, and notice how we add the input x to the output of the second Conv block (but not the first one).

out = F.relu(self.conv2(out)) + x



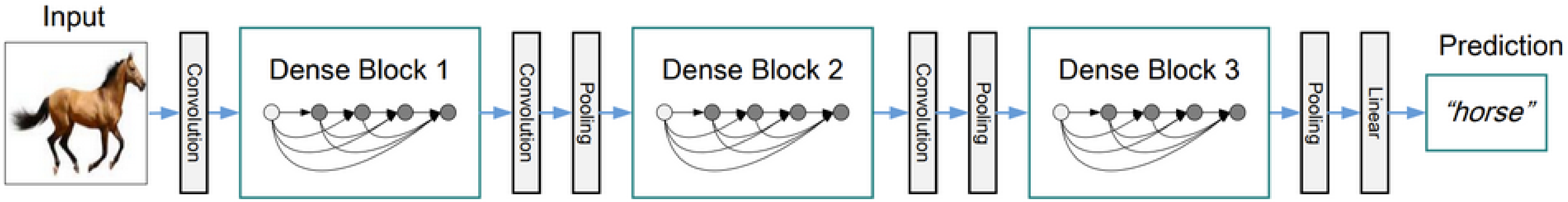
```python
class ResidualBlock(nn.Module):
    def __init__(self, n):
        super(ResidualBlock, self).__init__()

        # Conv layers
        self.conv1 = nn.Conv2d(n, n, 1)
        self.conv2 = nn.Conv2d(n, n, 3, 1, 1)

        # Final linear layer
        self.classifier = nn.Linear(n*24*24, 751)

    def forward(self, x):
        # First Conv block (Conv2d + ReLU), no residual.
        out = F.relu(self.conv1(x))

        # Second Conv block, add input x as residual.
        out = F.relu(self.conv2(out)) + x

        # Resize
        out = out.view(out.size(0),-1)

        # Final layer
        out = self.classifier(out)
        return out
```

The DenseNet model

Definition (the DenseNet model):

A **DenseNet** is a type of ResNet that utilises dense connections between layers, through DenseBlocks, where **all layers** (with matching feature-map sizes) **are connected directly with each other, as in ResNet**.

In Layman terms, a **DenseNet** is a **Resnet**, where **all possible skip connections have been drawn in every ResidualBlock**, boosting the benefits of ResNets even more ([Huang2016]).



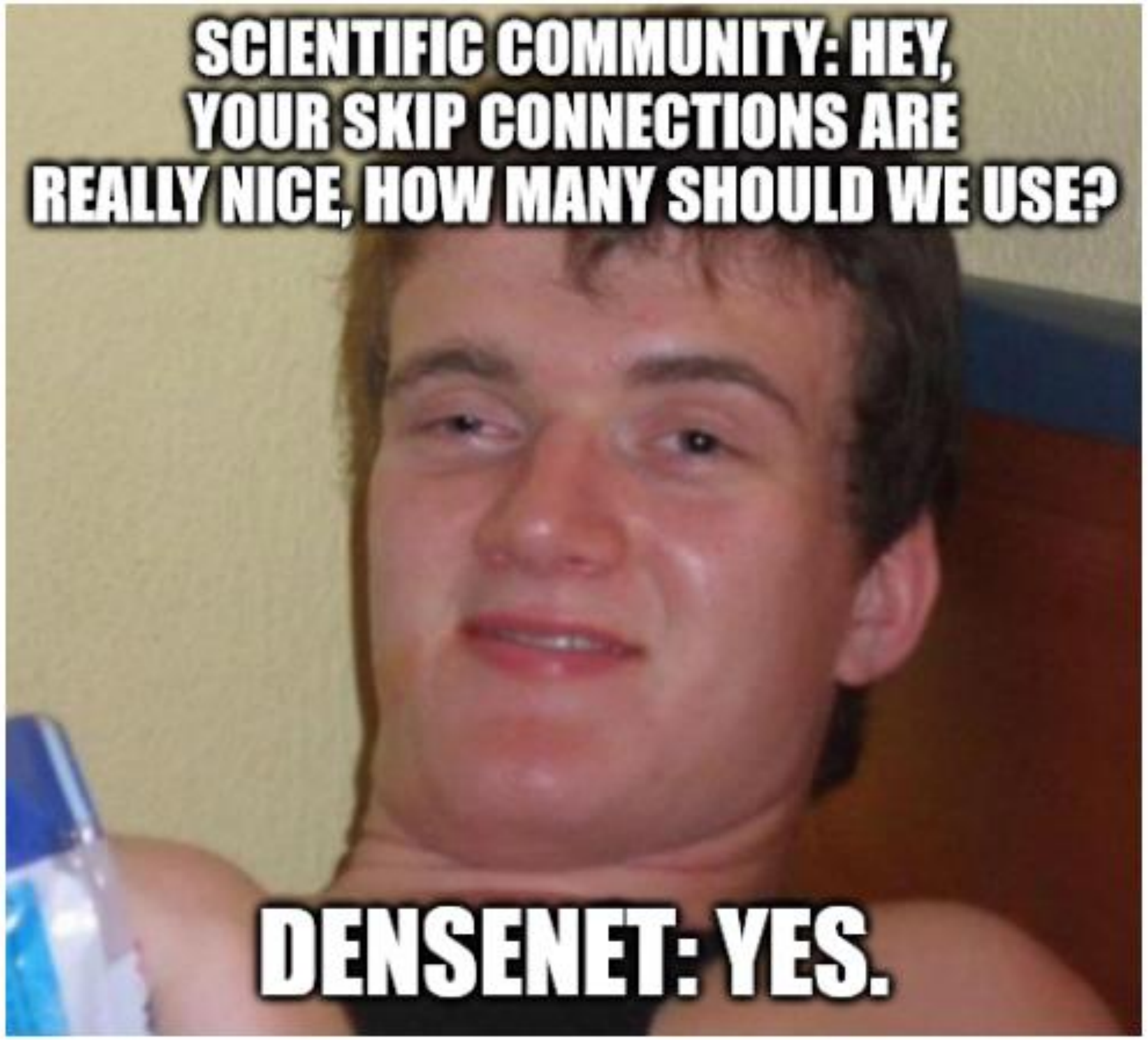
The DenseNet model

Definition (the

A **DenseNet** is a ctions between layers, through hing feature-map sizes) **are** **ResNet**.

In Layman term **ible skip** **connections ha** oosting the benefits of Res

Input
Convolution
Dense E
ock 3
Pooling
Linear
Prediction
"horse"

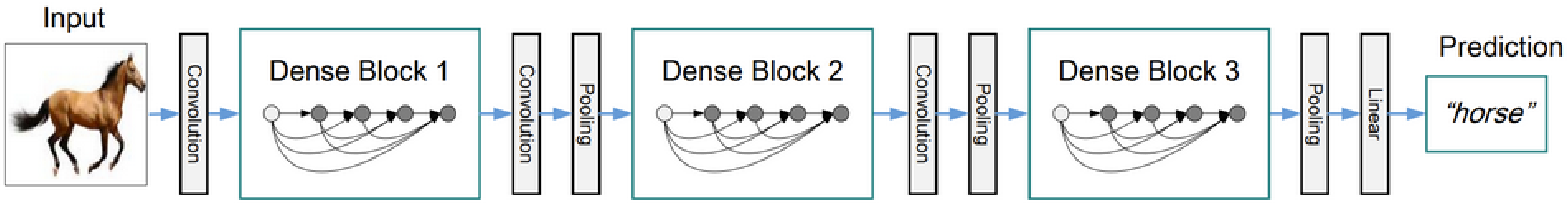

SCIENTIFIC COMMUNITY: HEY, YOUR SKIP CONNECTIONS ARE REALLY NICE, HOW MANY SHOULD WE USE?
DENSENET: YES.

The DenseNet model

Definition (the DenseNet model):

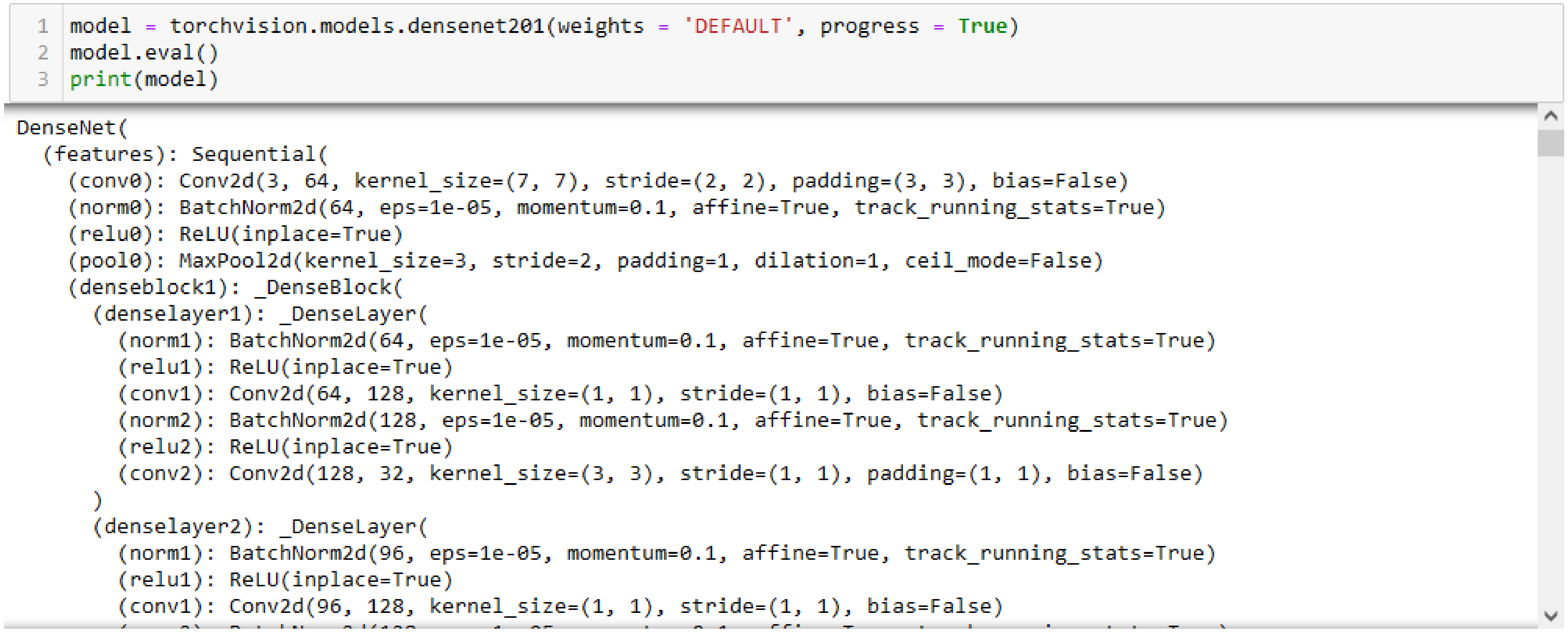
A **DenseNet** is a type of ResNet that utilises dense connections between layers, through DenseBlocks, where **all layers** (with matching feature-map sizes) **are connected directly with each other, as in ResNet**.

In Layman terms, a **DenseNet** is a **Resnet**, where **all possible skip connections have been drawn in every ResidualBlock**, boosting the benefits of ResNets even more ([Huang2016]).



The DenseNet model

Best part? It works, you can use this to train even bigger models…!
For instance, **DenseNet**, below, has **201 (!) layers**.



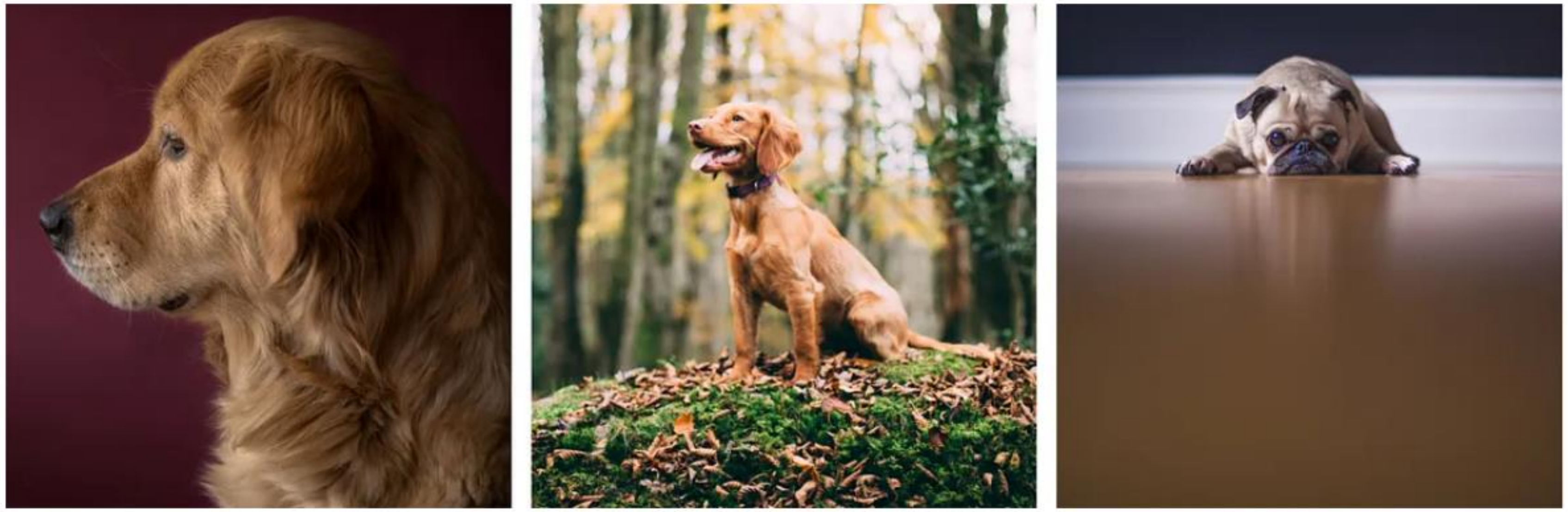
```python
model = torchvision.models.densenet201(weights = 'DEFAULT', progress = True)
model.eval()
print(model)
```

```
DenseNet(
  (features): Sequential(
    (conv0): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (norm0): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu0): ReLU(inplace=True)
    (pool0): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (denseblock1): _DenseBlock(
      (denselayer1): _DenseLayer(
        (norm1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (norm2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu2): ReLU(inplace=True)
        (conv2): Conv2d(128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      )
      (denselayer2): _DenseLayer(
        (norm1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu1): ReLU(inplace=True)
        (conv1): Conv2d(96, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

Starting with a problem

Inception models were created to address the following **problem**: **salient parts in the image sometimes have large variations in size**.

For instance, an image with a dog can be either of the ones shown below. **The area occupied by the dog is different in each image, which might make learning difficult for networks**.



The reason for Inception models

Because of this, choosing the right kernel size for the convolution operation becomes tough:

- A larger kernel is preferred for information that is distributed more globally (e.g. a zoomed-in picture),
- and a smaller kernel is preferred for information that is distributed more locally (e.g. a zoomed-out picture).

Proposed solution: to address this problem, Inception proposes to use **multiple Convolution filters, with different sizes, in parallel**.

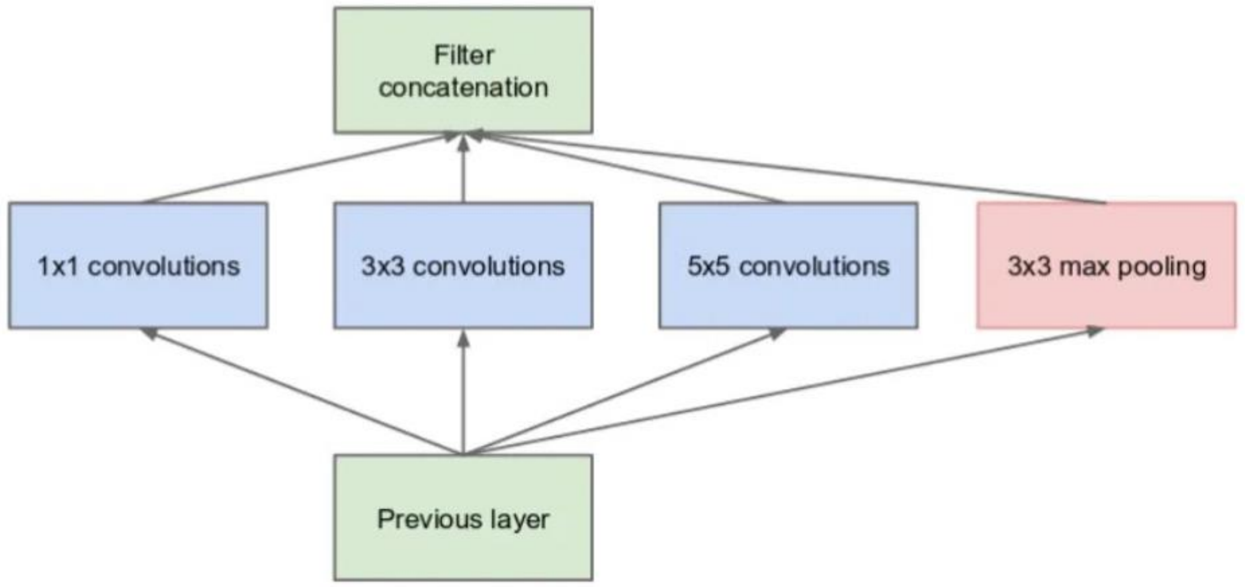
The Inception model

Definition (the Inception block):

Introduced in [Szegedy2014], the **Inception model** proposes to use **multiple Convolution filters, with different sizes, in parallel**.

Having different kernel sizes in parallel allows to process images with different salient sizes with the same network.

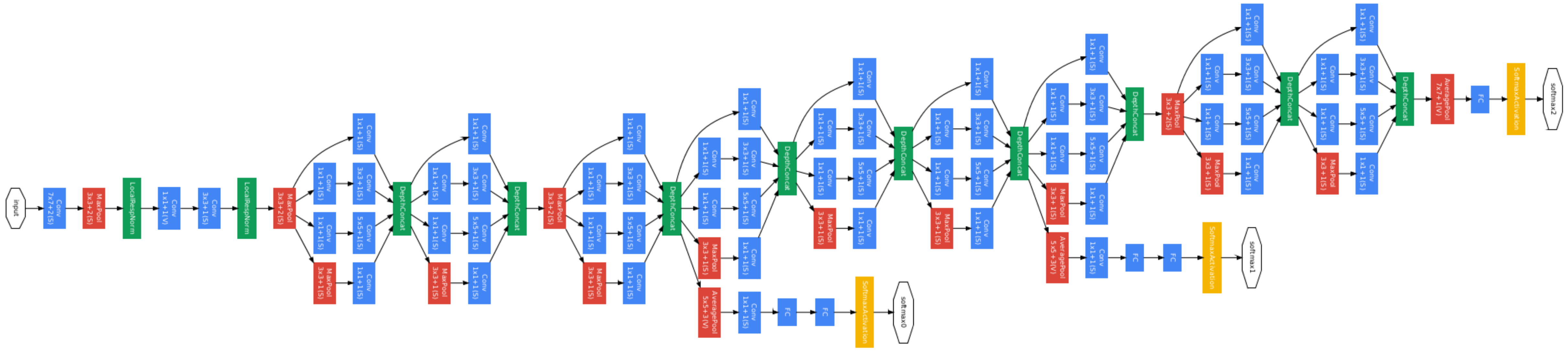
A typical Inception block is shown on the right.



The Inception model

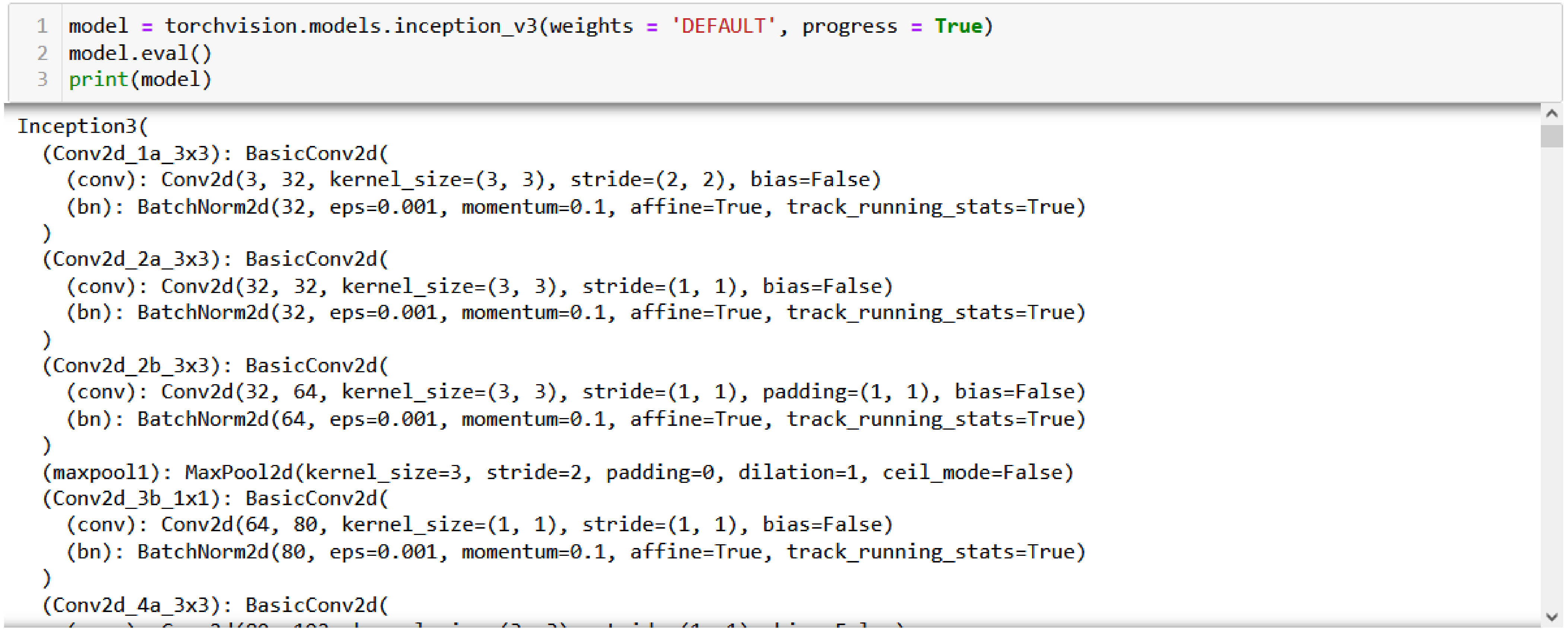
Definition (the Inception model):

The **Inception model** then **assembles multiple Inception blocks**, **in** sequence, also reusing the Pooling, BatchNorm, Dropout and Skip Connection concepts from the previous models.



The Inception model

Several version of Inception models were released, with various number of layers and operations.



```
model = torchvision.models.inception_v3(weights = 'DEFAULT', progress = True)
model.eval()
print(model)
```

```
Inception3(
  (Conv2d_1a_3x3): BasicConv2d(
    (conv): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_2a_3x3): BasicConv2d(
    (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(32, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_2b_3x3): BasicConv2d(
    (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn): BatchNorm2d(64, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (maxpool1): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  (Conv2d_3b_1x1): BasicConv2d(
    (conv): Conv2d(64, 80, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn): BatchNorm2d(80, eps=0.001, momentum=0.1, affine=True, track_running_stats=True)
  )
  (Conv2d_4a_3x3): BasicConv2d(
```

The EfficientNet model

EfficientNet followed up on the salient problem by proposing **a scaling method that uniformly scales all dimensions of depth, width and resolution** using a compound coefficient.

The compound scaling method is justified by the intuition that **if the input image is bigger, then the network needs more layers to increase the receptive field and more channels are needed to capture more fine-grained patterns on the bigger image**.

Different sizes for the pre-trained model were released, ranging from b0 (smaller) to b7 (larger).

The EfficientNet model

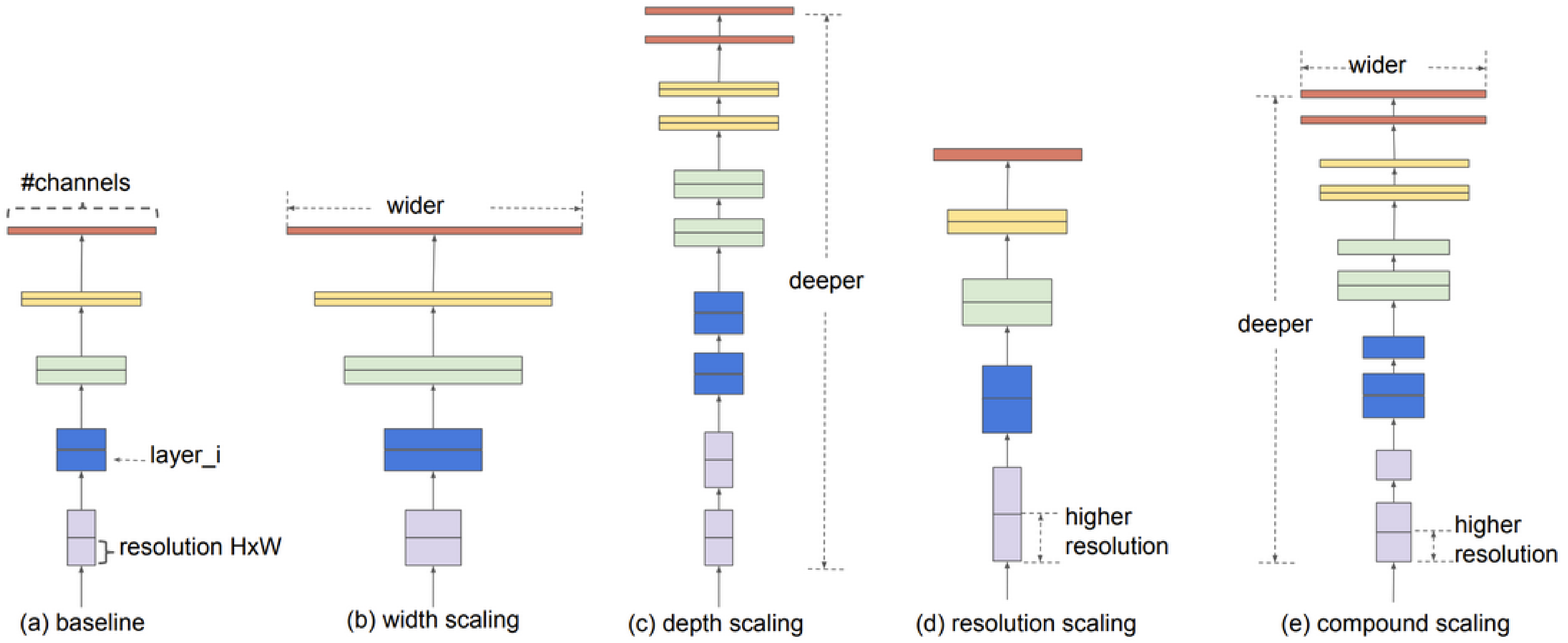
Definition (The EfficientNet model):

Introduced in [Tan2019], the EfficientNet model proposed a compound scaling method to adjust the size of the layers and the number of trainable parameters of the Neural Network, dynamically.

This is justified by the intuition that if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image.

The implementation of this dynamic scaling is out-of-scope.

The EfficientNet model



More on Computer Vision

- The previous milestone models have a very large number of layers and parameters: **training them on a laptop is out of the question**.
- Instead, we gladly **rely on the pre-trained versions** of these models that have been released on GitHub, and sometimes added to PyTorch.
- In fact, there are many more models that have marked the history of computer vision, but we have only covered the most famous ones. These days, however, these models are no longer considered best performance state-of-the-art. Some of these models, e.g. Inception, have even gone through many variations (Inception v1 was released in 2014, Inception v4 in 2016).

More on Computer Vision

- If you are curious about the different versions of these milestone models and their more recent implementations, we invite the reader to have a look at the following list/github repo.

https://github.com/gmalivenko/awesome-computer-vision-models

- We will NOT discuss the practical implementation of previous models, and gladly leave it to the Term 7 Computer Vision course at SUTD!
- Strongly advise you to continue your training on Term 7!

https://istd.sutd.edu.sg/undergraduate/courses/50035-computer-vision

Transfer Learning

Definition (Transfer Learning):

Many milestone models discussed earlier have been trained for long periods of time and their training could not be easily replicated, from scratch, without a super-calculator.

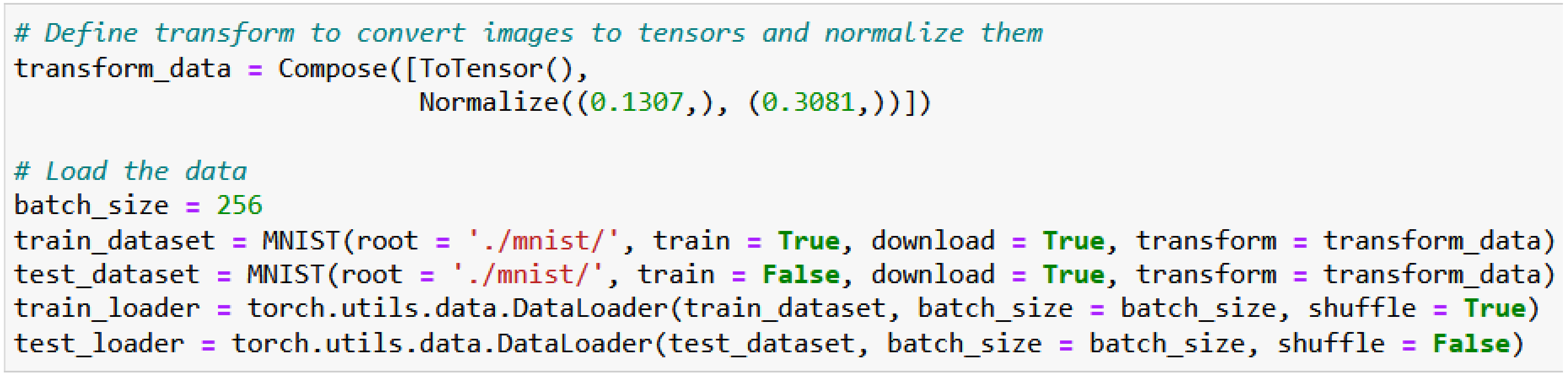
Transfer learning suggests to download and reuse the trained versions of said models: **with "minor adjustments"**, **they can potentially be reused on any dataset, even ones that are not even close to the original dataset they were trained on**.

For instance, ResNet models were trained on the ImageNet dataset, but with minor adjustments, we could use them on the MNIST dataset.

Transfer Learning of ResNet to MNIST

Let us

1. Prepare the MNIST dataset,

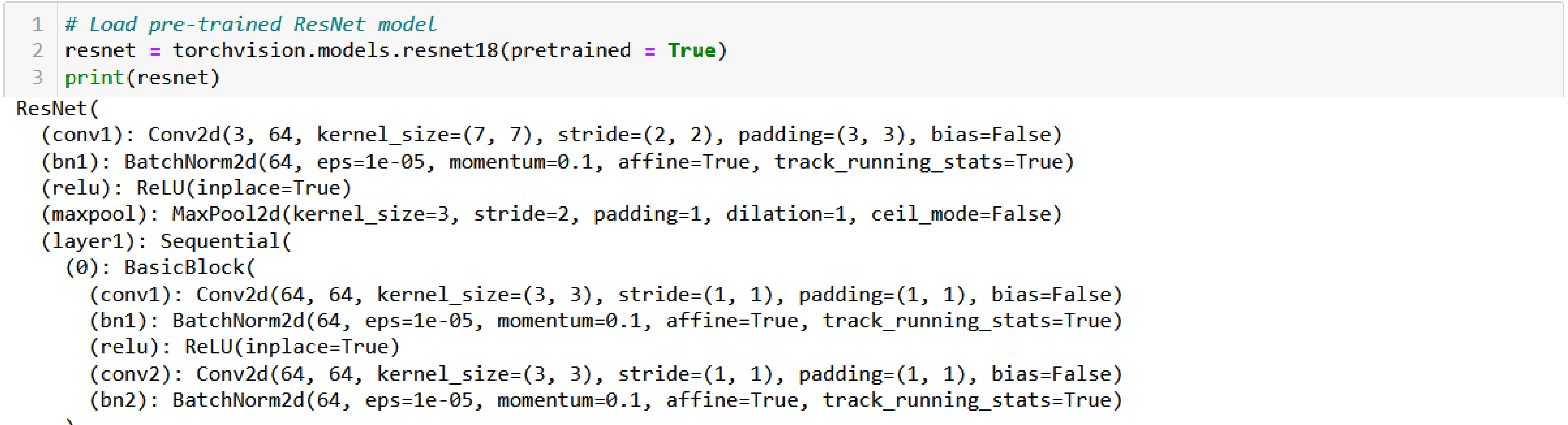


```
# Define transform to convert images to tensors and normalize them
transform_data = Compose([ToTensor(),
                          Normalize((0.1307,), (0.3081,))])

# Load the data
batch_size = 256
train_dataset = MNIST(root = './mnist/', train = True, download = True, transform = transform_data)
test_dataset = MNIST(root = './mnist/', train = False, download = True, transform = transform_data)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size = batch_size, shuffle = True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size = batch_size, shuffle = False)
```

Transfer Learning of ResNet to MNIST

Let us

1. Prepare the MNIST dataset,
2. Download the pre-trained version of a ResNet,



```python
# Load pre-trained ResNet model
resnet = torchvision.models.resnet18(pretrained = True)
print(resnet)
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

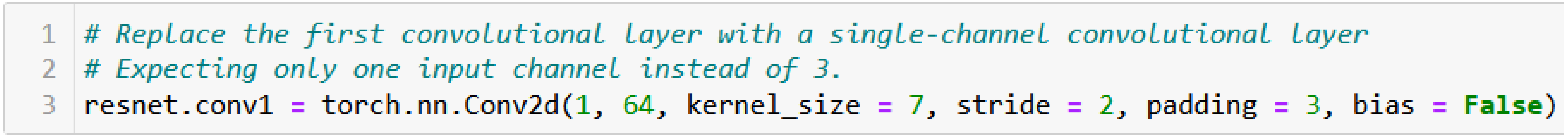
Transfer Learning of ResNet to MNIST

Let us

1. Prepare the MNIST dataset,
2. Download the pre-trained version of a ResNet,
3. Replace the initial layer so network uses MNIST images,

ImageNet is a RGB dataset, but MNIST is a greyscale one.

We simply remove the initial layer and replace it with an untrained layer that takes images with 1 channel instead of 3.



```python
# Replace the first convolutional layer with a single-channel convolutional layer
# Expecting only one input channel instead of 3.
resnet.conv1 = torch.nn.Conv2d(1, 64, kernel_size = 7, stride = 2, padding = 3, bias = False)
```

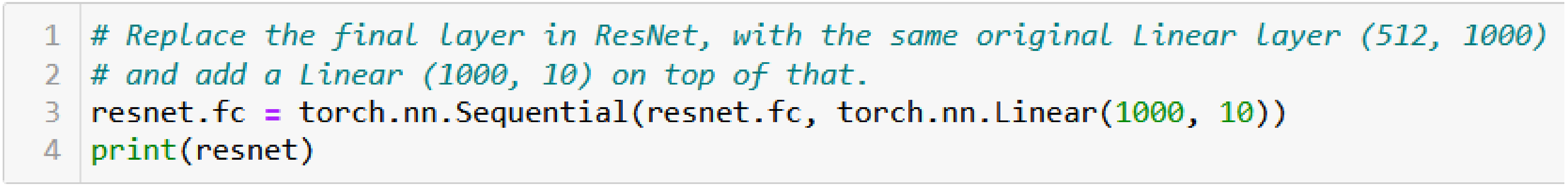
Transfer Learning of ResNet to MNIST

Let us

1. Prepare the MNIST dataset,
2. Download the pre-trained version of a ResNet,
3. Replace the initial layer so network uses MNIST images,
4. Replace final layer to match number of MNIST classes,

ImageNet had 1000 different classes, but MNIST only has 10.

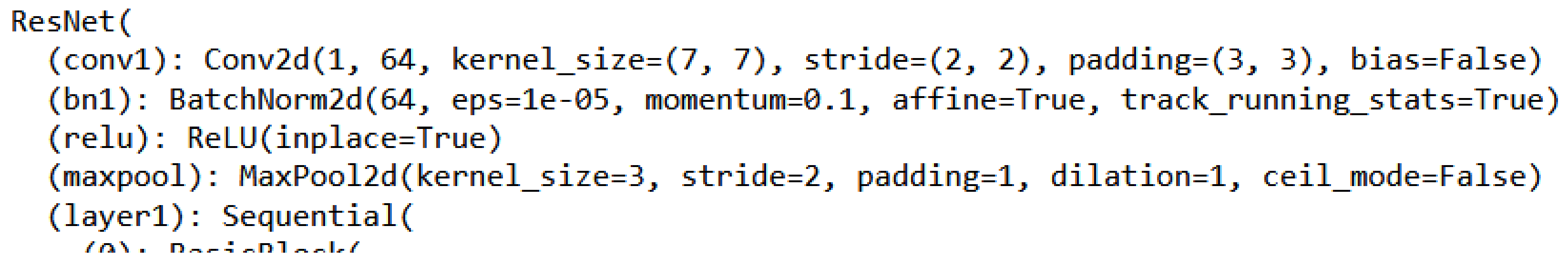
We then add an extra Linear layer, which will reduce the size of the output vector from a 1D vector with 1000 elements to a 1D vector with only 10 elements.



```python
# Replace the final layer in ResNet, with the same original Linear layer (512, 1000)
# and add a Linear (1000, 10) on top of that.
resnet.fc = torch.nn.Sequential(resnet.fc, torch.nn.Linear(1000, 10))
print(resnet)
```

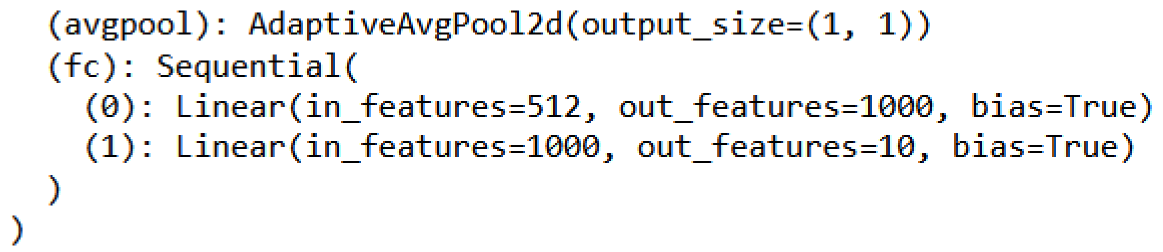
Transfer Learning of ResNet to MNIST

Our new model can now take greyscale images,



```
ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
```

Will output a 1D vector of 10 probabilities,



```
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Sequential(
    (0): Linear(in_features=512, out_features=1000, bias=True)
    (1): Linear(in_features=1000, out_features=10, bias=True)
  )
)
```

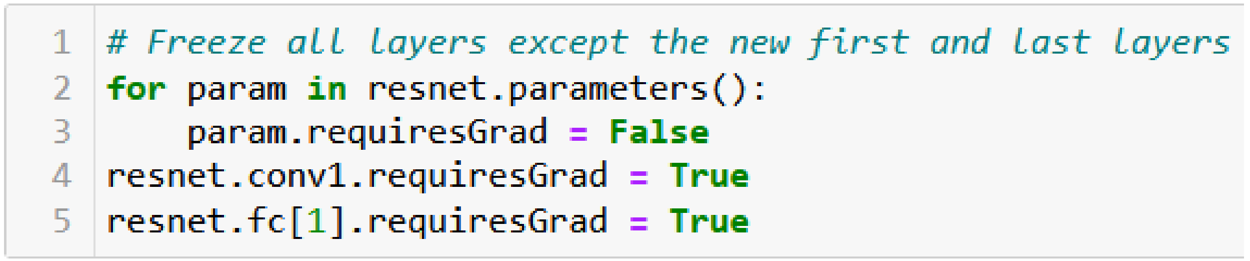
And – in the middle – will reuse most trained layers from the original ResNet model!

Transfer Learning of ResNet to MNIST

Let us

1. Prepare the MNIST dataset,
2. Download the pre-trained version of a ResNet,
3. Replace the initial layer so network uses MNIST images,
4. Replace final layer to match number of MNIST classes,
5. **Freeze** all layers except the new ones!

- **Freezing** prevents parameters from being changed during training.
- Let us only train the new layers (very few parameters then, so should not take too long to train these layers only!).

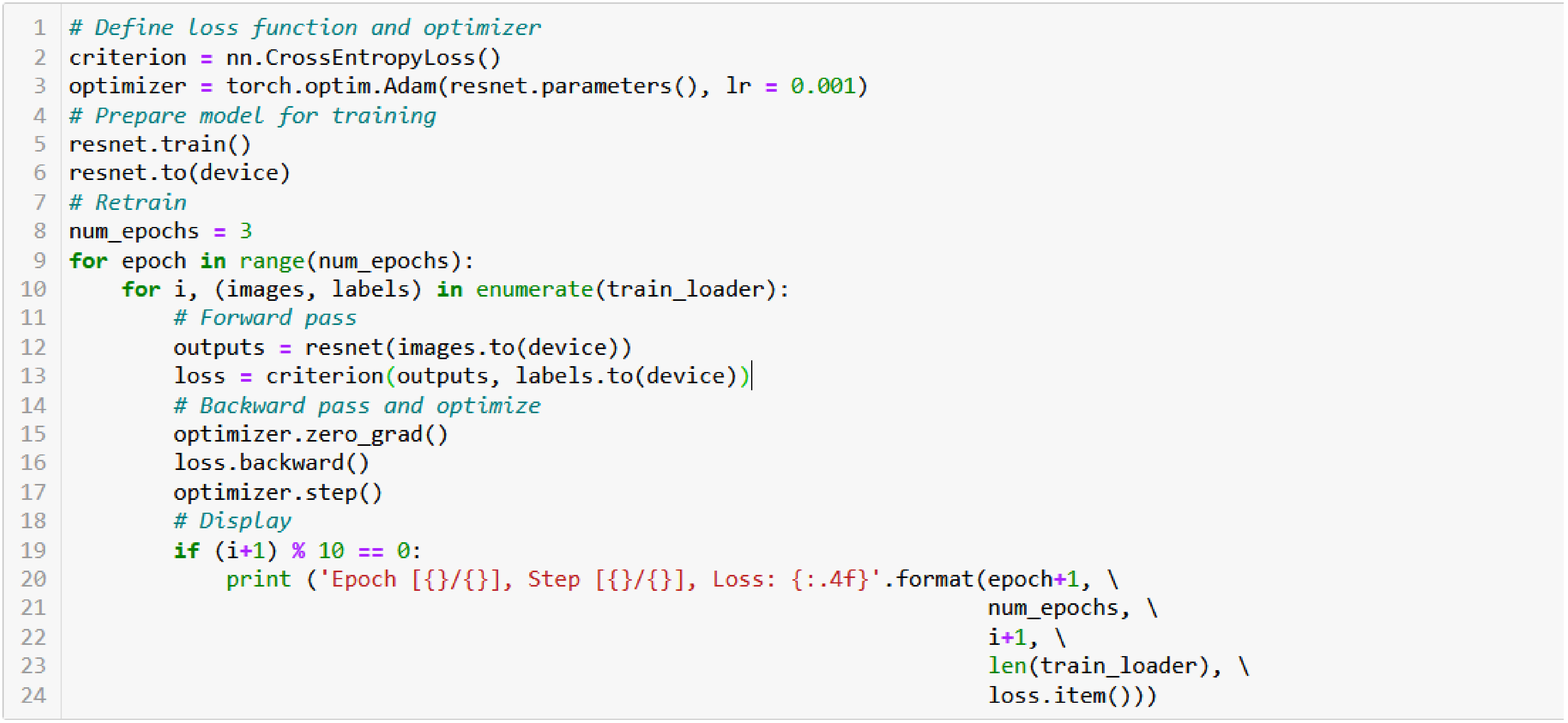


```python
# Freeze all layers except the new first and last layers
for param in resnet.parameters():
    param.requiresGrad = False
resnet.conv1.requiresGrad = True
resnet.fc[1].requiresGrad = True
```

Transfer Learning of ResNet to MNIST

Let us

1. Prepare the MNIST dataset,
2. Download the pre-trained version of a ResNet,
3. Replace the initial layer so network uses MNIST images,
4. Replace final layer to match number of MNIST classes,
5. **Freeze** all layers except the new ones!
6. **And finally, train these new layers that currently have random values in their trainable parameters, using the MNIST dataset!**



```python
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(resnet.parameters(), lr = 0.001)
# Prepare model for training
resnet.train()
resnet.to(device)
# Retrain
num_epochs = 3
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Forward pass
        outputs = resnet(images.to(device))
        loss = criterion(outputs, labels.to(device))
        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        # Display
        if (i+1) % 10 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(epoch+1, \
                                                               num_epochs, \
                                                               i+1, \
                                                               len(train_loader), \
                                                               loss.item()))
```

```
Epoch [1/3], Step [10/235], Loss: 0.5666
Epoch [1/3], Step [20/235], Loss: 0.3886
Epoch [1/3], Step [30/235], Loss: 0.2676
Epoch [1/3], Step [40/235], Loss: 0.1870
Epoch [1/3], Step [50/235], Loss: 0.1345
Epoch [1/3], Step [60/235], Loss: 0.0834
Epoch [1/3], Step [70/235], Loss: 0.0867
Epoch [1/3], Step [80/235], Loss: 0.1642
Epoch [1/3], Step [90/235], Loss: 0.1918
Epoch [1/3], Step [100/235], Loss: 0.0850
Epoch [1/3], Step [110/235], Loss: 0.1126
Epoch [1/3], Step [120/235], Loss: 0.1066
Epoch [1/3], Step [130/235], Loss: 0.0555
```

Wait that works?!

That feels borderline illegal!

```
Epoch [3/3], Step [170/235], Loss: 0.0607
Epoch [3/3], Step [180/235], Loss: 0.0486
Epoch [3/3], Step [190/235], Loss: 0.0548
Epoch [3/3], Step [200/235], Loss: 0.0373
Epoch [3/3], Step [210/235], Loss: 0.0676
Epoch [3/3], Step [220/235], Loss: 0.0216
Epoch [3/3], Step [230/235], Loss: 0.0173
```

Finetuning and unfreezing layers

Definition (fine-tuning and unfreezing layers):

Introduced in [Yosinski2014], **fine-tuning**, suggests to use a pre-trained network as a starting point, as before.

However, the network is further trained by progressively **unfreezing some of its layers**, therefore allowing the weights in those layers to be updated during re-training.

Usually, we will start by freezing all layers except the last few ones, and will progressively retrain layers, unfreezing more and more of them.

Doing so, the network can be adapted to the new problem and often achieve better performance.

(Works, but we leave it to students to try it out!)

Conclusion (Week 4)

- Encoding images as matrices and tensors
- The convolution operation, in full (kernel, padding, stride, dilation)
- Image processing using convolutions
- Building a convolutional layer and a Convolution Neural Network.
- On the performance of Convolution vs. Linear
- More operations on images (pooling, batchnorm, dropout)
- Data augmentation
- Some milestone models and their key ideas (residuals, parallelizing, dynamic scaling, etc.)
- A ramp to 50.035 Computer Vision on Term 7!
- Transfer learning and fine-tuning

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [Krizhevsky2012] A. **Krizhevsky**, I. **Sutskever**, and G. **Hinton**, “ImageNet Classification with Deep Convolutional Neural Networks”, 2012.
- [Simonyan2014] K. **Simonyan** and A. **Zisserman** “Very Deep Convolutional Networks for Large-Scale Image Recognition”, 2014.
- [He2015] K. **He**, X. Zhang, S. **Ren**, and J. **Sun** “Deep Residual Learning for Image Recognition”, 2015.

Learn more about these topics

Out of class, supporting papers, for those of you who are curious.

- [Huang2016] G. Huang, Z. Liu, L. van der Maaten, K. Q. Weinberger, “Densely Connected Convolutional Networks”, 2016.
- [Szegedy2014] C. **Szegedy** et al., “Going deeper with convolutions”, 2014.
- [Tan2019] M. Tan and Q. V. **Le**, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks”, 2019.
- [Yosinski2014] J. Yosinski et al., “How transferable are features in deep neural networks?” 2014.

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Karen Simonyan: Fromer researcher** at **Facebook AI,** now **Co-Founder** and **Chief Scientist** at **Inflection AI**.
  https://scholar.google.com/citations?user=L7lMQkQAAAAJ&hl=fr
  https://www.robots.ox.ac.uk/~karen/
- **Andrew Zisserman: Professor** at the **University of Oxford** and **Principal Researcher** at **DeepMind**.
  https://scholar.google.com/citations?user=UZ5wscMAAAAJ&hl=en
  https://www.robots.ox.ac.uk/~az/

Learn more about these topics

Tracking important names (Track their works and follow them on Scholar, Twitter, or whatever works for you!)

- **Shaoqing Ren:** Working at NIO and **University of Science and Technology in China (USTC)**. https://scholar.google.com.hk/citations?user=AUhj438AAAAJ&hl=zh-CN
- **Jian Sun:** Worked as **Microsoft**, now **Director of Research** at **MEGVII Technology**. https://scholar.google.com/citations?user=ALVSZAYAAAAJ&hl=en http://www.jiansun.org/
- **Quov V. Le: Research Scientist** at **Google Brain**. https://scholar.google.com/citations?user=vfT6-XIAAAAJ&hl=en

Learn more about these topics

Some extra (easy) reading and videos for those of you who are curious.

- [VisualisingConv] A good article explaining Convolution on images with animated GIFs to visualise operations. https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1
- [ExplainConv] A good guide on how Convolution works, from Mathematics to images convolutions (will typically explain the reason for flipping kernels!) https://betterexplained.com/articles/intuitive-convolution/
- [MIT2022] The CNN lecture at MIT (high-level, but should be ok now?) https://www.youtube.com/watch?v=uapdILWYTzE